Efficient Processing of Streaming Graphs for Evolution-Aware Clustering

Mindi Yuan
Dept. of Electrical & Computer Engineering
University of Illinois
Urbana, IL 61801
myuan5@illinois.edu

Kun-Lung Wu
IBM T. J. Watson Research Center
1101 Kichawan Road
Yorktown Heights, NY 10598
klwu@us.ibm.com

Gabriela Jacques-Silva
IBM T. J. Watson Research Center
1101 Kichawan Road
Yorktown Heights, NY 10598
g.jacques@us.ibm.com

Yi Lu
Dept. of Electrical & Computer Engineering
University of Illinois
Urbana, IL 61801
yilu4@illinois.edu

ABSTRACT

The clustering of vertices often evolves with time in a streaming graph, where graph update events are given as a stream of edge (vertex) insertions and deletions. Although a sliding window in stream processing naturally captures some cluster evolution, it alone may not be adequate, especially if the window size is large and the clustering within the windowed stream is unstable. Prior graph clustering approaches are mostly insensitive to clustering evolution. In this paper, we present an efficient approach to processing streaming graphs for evolution-aware clustering (EAC) of vertices. We incrementally manage individual connected components as clusters subject to a constraint on the maximal cluster size. For each cluster, we keep the relative recency of edges in a sorted order and favor more recent edges in clustering. We evaluate the effectiveness of EAC and compare it with a previous state-of-the-art evolution-insensitive clustering (EIC) approach. The results show that EAC is both effective and efficient in capturing evolution in a streaming graph. Moreover, we implement EAC as a streaming graph operator on IBM’s InfoSphere Streams, a large-scale distributed middleware for stream processing, and show snapshots of the user cluster evolution in a streaming Twitter mention graph.

Categories and Subject Descriptors
H.3.3 [Information Search & Retrieval]: Clustering

Keywords
Clustering streaming graphs; evolution-aware clustering

1. INTRODUCTION

Clustering vertices of a graph based on dynamic changes in edge connections is a powerful tool to understand social graphs, e.g., recognizing user communities. In scenarios where entity relationships change over time, a graph clustering algorithm must process a stream of updates. Each update can be the insertion or deletion of an edge or a vertex in the graph. Clustering of vertices in streaming graphs can be used to find user communities in real-time.

Graph clustering usually partitions vertices of a graph into different groups based on edge weights subject to a constraint, which could be either the maximal number of vertices in a cluster or the total number of clusters. For example, vertices can be partitioned into clusters so that the sum weight of the inter-cluster edges (also called weighted cut size) is minimized.

Due to memory limitation, streaming applications typically use a sliding or tumbling window to limit the amount of data for processing. These windows maintain, for example, only the most recent updates of a graph, like the graph consisting of the last 1 million edges. As new updates continue to stream in, old updates are removed from the window.

Even though a sliding window in stream processing naturally captures some cluster evolution, it alone might not be adequate. This is particularly true if the window size is large and the clustering of the vertices within the window has already changed significantly. Prior online graph clustering methods, like [7], are mostly insensitive to the clustering evolution, because they usually ignore the emerging clusters in a graph by not giving proper weights to the more recent edges in the window.

Formally, we study the following problem. Consider an undirected weighted graph \( G = (V, E) \). Each edge is in the form of \( < v_i, v_j, w_{ij} > \), where \( v_i \) and \( v_j \) are the two endpoints of an edge and \( w_{ij} \) the associated weight. There is also a constraint on the maximum number of vertices in each cluster, i.e. the maximum cluster size (MCS), denoted by \( M \). We partition the vertices \( V \) into clusters \( C_1, C_2, ..., C_n \) so that the sum weight of the inter-cluster
edges (or called weighted cut size) is minimized and \( |C_s| \leq M, \forall s \in \{1, 2, ..., n\} \)

That is

\[
\min \sum_{C(v_i) \neq C(v_j)} u_{ij} \\
\text{s.t.} \quad |C_s| \leq M, \forall s \in \{1, 2, ..., n\} \\
C_s \cap C_t = \emptyset, \forall s \neq t, s, t \in \{1, 2, ..., n\}
\]

where \( C(v_i) \) returns the cluster ID of vertex \( v_i \). The first constraint limits the MCS and the second forces that each vertex can only participate in one cluster.

To cluster a streaming graph, we must solve the clustering problem (the above optimization problem) whenever the graph in the window changes. This can happen when an edge is added into or removed from the graph. As a large volume of graph updates can occur in a short period of time, most of the offline graph clustering algorithms, like [12], are generally inefficient, especially when the window size is large. This is because offline algorithms need to re-cluster the graph in the window from scratch each time an update is processed. As a result, an online incremental algorithm is preferred for clustering streaming graphs.

In this paper, we propose a new and efficient evolution-aware clustering approach (EAC) for a streaming graph. In EAC, we treat each connected component as a cluster and maintain these clusters incrementally upon a graph update. We assume that each cluster is constrained by a maximal number of vertices. We treat the timestamp of an edge as its weight. We keep all the edges in each cluster in a sorted order based on its recency. We favor more recent edges than older ones in our cluster merging and splitting, effectively capturing emerging cluster evolution in the streaming graph. The maintenance of recency-ordered edges within a cluster makes our algorithm incremental and efficient.

Unlike traditional clustering approaches where vertices are moved around among clusters and the cut edges are still kept for future clustering decisions, once our scheme chooses an edge to be part of the cut due to an MCS constraint violation, the cut edge is no longer considered for future clustering decisions. Because of that, it is as if this cut edge is deleted from the graph. Although similar in practice, this operation is conceptually different from a deletion of an edge that occurs when a window slides (additional windowing semantics will be defined in more details in section 3.2.1).

We evaluate the effectiveness and efficiency of our evolution-aware clustering approach against a previous state-of-the-art evolution-insensitive clustering (EIC) approach [7] using both synthetic and real-world graph data. Moreover, we implemented EAC as a stream graph operator on IBM’s InfoSphere Streams [11], a large-scale distributed middleware for stream processing, and demonstrated in practice via a graph visualization tool snapshots of the cluster evolution in a Twitter mention graph.

Our contributions can be summarized as follows:

1. We propose an evolution-aware clustering algorithm for a streaming graph whose edges are weighted by the associated timestamps.

2. We design a clustering evolution analysis scheme to quantify the clustering stableness of an underlying stream. We compare the performance of EAC with streaming graphs of different stableness and show its advantages.

3. We conduct quality and throughput experiments to compare our algorithm with EIC and METIS using synthetic and real-word data sets. The results show that EAC outperforms EIC in better clustering quality (up to 6x) and better throughput (up to 8x).

4. We implement EAC as a streaming graph operator using IBM’s InfoSphere Streams and demonstrate the evolution of vertex clustering in a Twitter graph.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the evolution-aware algorithm. Section 4 presents the clustering evolution analysis. We show the experiments in Section 5 and the practical implementation on IBM’s InfoSphere Streams in Section 6. We conclude in Section 7.

2. RELATED WORK

One popular offline clustering algorithm is METIS [12]. It is based on the multilevel graph partitioning paradigm. METIS was shown to produce high-quality (balanced) partitions. For online streaming graphs, Aggarwal et al. [3] proposed an algorithm for clustering graph streams. However, their algorithm does not deal with edge deletions, making it not applicable to clustering streaming graphs in the face of sliding windows. Stanton and Kliot [15] designed a series of natural and simple heuristics for partitioning large streaming graphs into distributed compute nodes. However, they assumed that the graph is already stored on disks, so they were able to stream the graph from disks in a certain order. In a general streaming setting, however, the orders of edges are not predictable.

In [7], Eldawy et al. investigated the problem of clustering streaming graphs, similar to the problem studied in this paper. However, their scheme, referred to as EIC, is not sensitive to cluster evolution. Although EIC can capture some evolution via the streaming window, the clustering algorithm itself is not aware of cluster evolution. In EIC, each edge is assigned with a random number. The system then considers for the clustering problem a fraction of the edges whose random numbers are below a sampling threshold. For those sampled edges, the algorithm maintains them in a sorted order based on the random numbers. When there is a violation of the MCS constraint, edges are removed one by one from the one with the largest random number. Because of its reliance on random numbers assigned, unnecessary deletions might happen to clusters whose sizes actually do not exceed the MCS constraint. The unnecessarily deleted edges need to be reinserted to their corresponding clusters. As a result, not only the deletion process takes more steps, but also additional insertions of edges are needed.

In [13], to detect meaningful temporal communities in evolving networks, Kawadia et al. presented a measure of partition distance called estrangement, which facilitates community detection when incorporated into the measurement of partition quality. They might miss some evolution if the network is highly dynamic. In [14], Lin et al. proposed a framework for analyzing communities and their evolutions through a unified process. Gupta et al. [9] proposed an evolutionary clustering method to analyze bibliographic networks. They clustered a sequence of snapshots of a graph and tried to diagnose the evolution by tracking and comparing the clustering results. The algorithm for clustering each graph snapshot is still offline.
For capturing emerging events, one can identify the densest subgraph(s) of a graph. Bahmani et al. [5] developed an algorithm to find locally dense components of a graph under a streaming model. Their model assumes that all the vertices of the graph are known and that edges arrive one by one. In [4], Angel et al. proposed an algorithm to maintain dense subgraphs under streaming edge weight updates for real-time story identification. This work also assumed that the graph is complete. Neither of these works considered windowing scenarios. Agarwal et al. [1] designed an algorithm for real-time discovery of dense clusters in highly dynamic graphs. The problem of clustering is different from just finding the densest subgraphs. In this paper, we partition the vertices of the graph into different clusters so that we can identify and follow the cluster evolution. The densest subgraph problem aims at identifying a few densest clusters above a specified threshold rather than clustering the entire graph within the window.

STINGER [6] is a general-purpose graph data structure that enables fast insertions, deletions, and updates. It assumes that the graphs are stored in the shared memory of a multicore system, and provides index structures for fast access to vertices and edges. However, it is not obvious how those indexes can be used for efficient processing of evolution-aware clustering of streaming graphs.

3. EVOLUTION-AWARE CLUSTERING

In this section, we describe our evolution-aware clustering algorithm for a windowed streaming graph.

3.1 Intuition

Consider the following example in Fig. 1. Suppose there is a soccer discussion forum during the European Championship 2012. The vertices A, B, ..., G represent 7 users. Assume A, B and D are fans of Germany, C is a fan of Italy, and E, F and G are fans of Spain. Suppose it is June 28th when there is a game between Germany and Italy and the winner will enter the final to play Spain. Assume that an edge represents an interaction between two users, such as a conversation or a reply to a message posted on the forum. New edges arrive in the order specified by their label (e.g., edge (B, C) arrives first). The first conversation happens between B and C. They are debating how one team will defeat the other. Then, the edge (A, C) comes in followed by (C, D), and (E, F), when the Spaniards cannot wait to see the final and so on. After that day, the German team lost. As a result, A, B and D might not be as active as before. C, on the contrary, is so excited about her team entering the final and she begins to talk to the Spanish fans. That interaction is represented by edges (C, E), (C, F), and (C, G). Before time 10, there are naturally two separate clusters (two clouds in Fig. 1). After time 10, C is talking to the members of the right cluster (represented by dotted lines). The fact that C is now talking to entities in the right cluster rather than the ones in the left cluster, as before, is an example of clustering evolution.

What if we want to cluster this example at timestamp 10? Suppose that the maximum number of vertices in one cluster is 4. Once there is an edge between C and E, the two clusters merge together because we use connected components to represent clusters. After that, there is only one big cluster of size 7, so we have to split the overcrowded cluster. Where should we cut the graph? If the edges have the same weight, say each edge weighs 1 unit, we should remove the edge that just came in, that is the edge between C and E. This way the cut size is 1, which is the best. Similarly, when the 11th and 12th edges come, it is still the best to cut the graph by removing the newest edges in this setting. However, the evolution here is that C wants to move to the cluster on the right. But C continues to stay with the left cluster after the 12th edge update. This means, we are not able to detect and adapt to the evolution that C is now eager to become part of the cluster containing E, F, G, rather than the cluster containing A, B, D.

A better way is to move C to the cluster on the right and allow them to form a new cluster, i.e. splitting the cluster by cutting the edges with timestamps 1, 2 and 3. How can we allow the algorithm to automatically follow the clustering evolution as it happens? Our intuition is to use edge recency as a measure of weight. Our clustering algorithm puts more weight on a new edge than an old edge. For example, if we use the ordering of arrivals or the timestamp of the edge as the edge weight, then, when the 10th edge comes in, we cut edges 1, 2 and 3, which has a total weighted cut size of 6 (compared to 10). In this way, the weighted cut size actually reflects how well an algorithm can capture the evolution.

3.2 System Design

Fig. 2 is an overview of the system architecture. The first building block is called window manager (WM), and the second one is graph manager (GM). Once a new graph update comes, it enters the system through the WM. The WM then forwards graph insertion and deletion requests to the GM. A graph query goes directly to the GM, which is responsible for answering it.

![Figure 2: The system architecture has a window manager (WM) and a graph manager (GM).](image)

3.2.1 Window Manager

Windowing is generally used in a streaming environment to limit the amount of data for processing. For certain an-
alytics, we care more about the most recent state of the graph, such as the graph of interactions occurring within the past 24 hours. In this sense, the streaming window naturally helps us capture some clustering evolution. But, it may not be adequate to capture all the clustering evolution, especially when the window size is large and the clustering within it evolves a lot.

There are generally two kinds of windows in streaming: sliding and tumbling windows. Both can be time-based or count-based. For example, a count-based tumbling window stores new updates until it reaches the maximum window size. Once the window is full, all updates are discarded and a new window starts. In this paper, we focus on count-based sliding window in our algorithm. A count-based sliding window maintains a specified window size, such as the newest 1M updates. If a new update comes and the window is full, the oldest update must be evicted from the window.

Note that it is also easy to optionally apply sampling before the window manager, as in [7]. Sampling can be used to sparsify the graph. However, applying sampling is orthogonal to the graph clustering problem and is not the focus of this paper.

### 3.2.2 Graph Manager

The graph manager maintains the cluster structures of the current graph and is responsible for answering queries. The most common query is whether or not a particular vertex is in the current graph, and if so, to which cluster it belongs. This question comes from users who would like to know about the graph and from the clustering algorithm itself for inserting an edge. Other common queries include (a) for a given vertex, what are the vertices in the same cluster? and (b) what is the total number of clusters?

Fig. 3 shows the two key data structures that the graph manager maintains for the evolution-aware clustering. The Vertex Table is a hash table used to maintain the mapping of a vertex ID to a cluster ID. The Cluster Table is another hash table used to maintain all the edges in a cluster, consisting of the edges forming a connected component of the current graph within the streaming window. The key for the Vertex Table is the vertex ID and the value is the cluster ID. We use the ID of the first vertex in the cluster as the cluster ID. In this way, two different clusters never have the same cluster ID. The key for the Cluster Table is the cluster ID and the value is a pair. The first member of the pair is the cluster size, which records the number of vertices in the cluster. The second member is a list of edges, storing all the connected edges in this cluster. This list is naturally sorted by the edge weights (timestamps). The only two operations for the list of a single cluster are (i) the addition of an edge to the end (where the weight is biggest/the edge is newest) and (ii) the removal of an edge from the front (where the weight is smallest/the edge is oldest).

This *Cluster Table* is efficient for edge insertion. The expensive part is deletion, since, after the deletion of an edge, we do not know if the other edges still forms a connected component. While there is an online method to keep track of connected components [10], the algorithm is complicated to implement and expensive to maintain. Instead, we use a very simple method for edge deletion, as described in [7]. Upon a deletion of an edge from a cluster, we delete the entire cluster and then reinsert all the edges except for the deleted one. In this way, the insertion routine automatically merges connected components.

In summary, for each insertion, the graph manager first queries the *Vertex Table* and finds the corresponding cluster or creates a new cluster for the vertices. Then, the graph manager goes to the *Cluster Table*, increases the size of the target cluster and appends the edge to the end of cluster edge list. For each deletion, the graph manager erases the entire cluster and reinserts the rest of the edges.

#### 3.3 The Algorithm

For insertion, we first look up the cluster membership of the two endpoints. There are four cases. Case 1: If both vertices are new, we create two new entries in the vertex table and use the ID of the first vertex as the cluster ID. After that, we create a new entry in the cluster table and insert the new edge. We assume that the maximum cluster size is always great than or equal to 2. Case 2: If one of the vertices is new, we create one new entry in the vertex table and assign it the cluster ID of the other vertex. After that, we append this edge to the corresponding cluster in the cluster table. Case 3: If both vertices already exist and are in the same cluster, we simply append the edge to the cluster. Case 4: If the vertices exist and are in different clusters, we merge the smaller cluster to the bigger one so that we modify as few entries in the vertex table as possible.

The insertion algorithm is shown in Algorithm 1. Note that after an edge insertion, we need to check the constraint for the maximum cluster size in Cases 2 and 4. If the constraint is violated, we need to delete the oldest edge from the cluster with the violation. Note that this deletion is conceptually different from the deletion due to the sliding out of a window. In EAC, once we choose an edge to be deleted because of an MCS constraint violation, it no longer is considered for future clustering decisions. As mentioned before, with each edge weighted by its timestamp, a clustering algorithm can better capture the evolution by minimizing the weighted cut size. By putting more weights on more recent edges and trying to keep new edges rather than the old ones upon an MCS violation, we are more likely to capture the emerging clusters and achieve high-quality clustering.

The deletion algorithm is used in two situations. The first is when there is a violation of the maximum cluster size. The second is when an edge must be evicted from a window. If the deletion is caused by the violation of the maximum cluster size, we know the cluster ID where we need to delete the edge. If the deletion is due to expiration from a streaming window, we only delete the edge from the cluster table if the edge still exists.

If the edge must be deleted from the cluster table, we use Algorithm 2. As described earlier in this Section, we initially remove the first element (the oldest) of the edge list of the target cluster and remove the list entry from the cluster table. After that, we reinsert each element from the list into
the cluster table. Note that the reinsertions of the edges from a removed cluster only recluster the edges within that cluster. They will not cause a cascade of constraint violations because none of them will connect to any other clusters existed before the deletion. The insertion algorithm automatically merges disconnected clusters. Hence, each deletion caused by an MCS violation is a step towards solving the violation and the algorithms will terminate, even though Algorithms 1 and 2 may call each other multiple times during the process. Although the description of our algorithm indicates that we erase all affected vertices from the vertex table, we do not do that in practice. Instead, we mark them with an invalid cluster ID. The reason for doing so is that deleting/inserting an entry in a hash table could be more expensive than just modifying its value, mainly considering that most of the vertices will be reinserted immediately.

Note it is hard to incrementally check the connectivity of the cluster upon an edge deletion. Thus we adopt our simple but efficient deletion algorithm. Alternatively, one can use BFS to check the connectivity of the vertices in the related clusters after deletions. But this will require a slightly more complicated data structures, since then we need to remember the neighbors of every vertex. An adjacency list or matrix can be used for this purpose depending on density of the graph. However, this modification will not only increase space usage, but also time consumption, since the insertion of an edge will then be more complex.

4. EVOLUTION ANALYSIS

Here we propose methods for analyzing clustering evolution and quantifying its intensity in a streaming graph.

4.1 Evolution of Clusters

As proposed in [2], we can understand the changes in clusters by comparing their structures at two different times. For this comparison, we need two clock times: \( t_1 \) and \( t_2 \) and a window size \( s \). Without loss of generality and for ease of analysis, suppose \( t_2 > t_1 \) and \( s \) is the window size (e.g., 24 hours). Let us use the updates arrived between \( t_1 - s \) and \( t_1 \) to construct one graph \( G_{[t_1-s,t_1]} \) and those in \([t_2 - s, t_2]\) to construct another graph \( G_{[t_2-s,t_2]} \). The changes occurring between \( t_1 \) and \( t_2 \) can be identified by observing the following: (i) How many new clusters appeared and what they are; (ii) How many clusters disappeared and what they are; (iii) How many clusters are still there, what they are and how much their sizes changed.

We can leverage EAC to do this kind of analysis. This is because EAC keeps the table for all the clusters in the current graph. More specifically, EAC can output all the clusters and their sizes both on \( t_1 \) and \( t_2 \). By comparing the two outputs, we can understand how the evolution happened between \( t_1 \) and \( t_2 \). Charu et al [2] observed that when a large fraction of clusters belong to the third case (the clusters retained from \( t_1 \) to \( t_2 \), it is a sign that the stream is stable during this period of time.

By investigating the graph clusters at different times, one can get useful information about how the clusters evolve. In order to understand the general trend in the evolution, however, we need to sample many time slots, which could be a significant extra overhead. Besides, even if all the clusters keep exactly the same from \( t_1 \) to \( t_2 \), there still could be evolution. For example, the clusters can change a lot during \([t_1, t_3]\), where \( t_3 < t_2 \), but finally they all mutate back to the original shapes at \( t_2 \). This is quite possible when the input streams are periodic. If we do not choose the sampling points carefully, we might overlook the clustering evolution.

4.2 Quantify Stream Stableness

As an alternative to the approach proposed in [2], we propose a method that looks into the stream of graph updates, as it is the updates themselves that cause the evolution. The changes in clustering results at different points in time are only the effect of the updates, i.e., these phenomena are merely the reflection of the evolving stream updates. Therefore, by monitoring the graph updates, we can better understand and capture the clustering evolution.

**Definition 1.** Assuming that there is no clustering size constraint and we simply treat a connected component as a cluster, given a graph and a new edge to be inserted to the graph, this edge is called an unstable edge, if the number of clusters changes after inserting this edge; otherwise, this edge is called a stable edge.

In this paper, a graph is the incremental streaming graph within the current window. Hence the unstableness and stableness of a graph stream is defined as follows.

\[
\text{unstableness} = \text{Ave} \left( \frac{\text{numUnstableEdgesInWindow}}{\text{WindowSize}} \right) \quad (1)
\]
5. EXPERIMENTS

We conducted a set of experiments to thoroughly evaluate EAC, including visualizing its ability to capture the emerging cluster evolution, measuring its quality using the weighted cut size, and its efficiency in terms of throughput. All the experiments were run on a single machine with Intel Xeon processor and 126 GB physical memory, which runs the Linux Red Hat 4.4 operating system. All the algorithms were implemented in C++.

We compared our algorithm with EIC and two versions of METIS: the recursive METIS (Rmetis) and the Kway METIS (Kmetis) [12]. We did not compare with any other evolutionary clustering algorithms, like the ones with sophisticated definitions of edge weights, because they are not applicable in a dynamic streaming setting, where efficiency is critical. We compared with METIS, because our problem is formulated as a cut size minimization problem, where METIS can serve as an optimum for quality measurement. We compared with EIC because it is also a streaming clustering algorithm and is good for throughput comparison. We used the METIS software package of version 5.0 with all its default settings (e.g., maximum load imbalance among the partitions is 1.03).

For the data sets, we used one synthetic data set and four real-world ones. We used the synthetic data set to visualize how the algorithms capture the emerging clustering evolution. We did not use the real data sets for this purpose because it was difficult to control the number of vertices/edges in the stream for precise placement in a canvas for a good visualization. In order to compare with EIC, the four real-world data sets were similar to those used in [7].

1. Twitter: Twitter replies obtained during a five-day period. These replies were 1% sampled, so that they were the sparsest among the four data sets. In this set, vertices denote Twitter users and edges denote the replies. There were 1.9M vertices and 1.6M edges in total.

2. Web Notre Dame (or Web for short): This data set was obtained by taking a snapshot of the web graph of the University of Notre Dame in 1999. We inserted random timestamps to this set because, originally, there was no time information recorded. There were 330K vertices and 1.5M edges in this data set.

3. Citations: This was a data set from paper citations of high energy physics in arXiv. The data covered a time range of 124 months. There were 34K vertices and 350K edges.

4. DNS requests (or DNS for short): This set contained DNS requests from the computers within the IBM Watson Research Center to domains on the web in a day. We observe that some machine requests the same IP many times a day, implying that the graph can be very dense. There were 180K vertices and 4.8M edges in it.

We designed three kinds of experiments:

- Visualization of evolution: We plotted the real cluster structures of the graph by taking snapshots during the execution of the clustering algorithms. We ran this experiment only on the synthetic data set.

- Quality experiments: For the quality metric, we measured the weighted cut size of the clustering algorithm, which reflects how well an algorithm captures the evolution. Note that it is difficult to measure the “real-time cut size” of the clustering, as the inter-cluster edges are not explicitly kept by the clustering algorithms. Instead, similar to [7], we designed a quality experiment based on a tumbling window. For a tumbling window of size $s$, we ran the clustering algorithm until $s$ edge updates arrived. Then we used the cluster structures (e.g., the cluster table in EAC) obtained up to that moment for measuring the weighted cut size by re-scanning all the edges from the tumbling window. For each edge, if both its endpoints were in the same cluster, we skipped the edge; otherwise, we accumulated its weight toward the weighted cut size. We repeated this for the rest of the data set and computed the average of the weighted cut sizes associated with different tumbling windows.

- Throughput experiments: we ran the algorithms under different data sets, and calculated how many updates they can handle in one second.

5.1 Capturing the evolution

This experiment shows a visualization of the clusters generated by METIS, EAC, and EIC when running a synthetic data set. To construct the synthetic data set, we first created two sets of vertices, each with a total number of 200 vertices. Vertices with ID 1-200 were in the first set and vertices with ID 201-400 were in the second set. Then we randomly created a sequence of 400 edges for each of the two sets. The edges were created within vertices in the same set, so that we created two different clusters. After the 800th edge, we created edges among vertices of different sets. This means that we started generating inter-cluster edges. We used an inter-cluster edge probability $p_{ic}$ to control the intensity of the cluster evolution. For each new edge to be created, it connected two vertices in different clusters with probability $p_{ic}$ and two vertices in the same cluster with probability $1 - p_{ic}$. For better visualization, an inter-cluster edge chose a vertex from the right half of the first cluster (vertex ID 101-200) and the left half from the second cluster (vertex ID 201-300).

Fig. 4 shows 3 snapshots of the clustering structures for each algorithm. METIS is on the top, EAC in the middle and EIC on the bottom. There are 3 columns: on the

\[ \text{stableness} = -\log \text{unstableness} \]
left are clustering snapshots for the initial graphs (after inserting the first 800 edges), in the middle are the clustering snapshots after 150 additional updates, and on the right are the clustering snapshots after 300 additional updates. Each clustering snapshot is plotted on a unit Cartesian coordinate system (we do not show the x and y labels for brevity). The position of each vertex in the figure is determined as follows: for each vertex, we set its x coordinate to be vertexID/400 and its y coordinate to be a random number uniformly sampled from [0, 1]. We set $p_{ic}$ to 0.9 and the maximum cluster size to 200. The red circles, blue diamonds and black squares represent different clusters. Each green dot in EAC and EIC represents isolated vertices. We do not show edges, as they would have cluttered the visualization.

With this synthetic workload, a new cluster should be forming in the middle of the figure after 300 updates. This can be clearly observed in the first row of Fig. 4, which is the result from METIS (regarded as the best). After 300 updates, we can see that EAC clearly identifies the new cluster (shown in black squares). EAC captures the emerging evolution and follows the emerging trend in the stream. We can see that EIC does not capture this change, as the two initial clusters are mixing their members and the new cluster is still not emerging. Note that because of edge deletions during clustering, there are isolated vertices in both EAC and EIC. They are represented by the green dots. Although there are more isolated vertices in EAC than in EIC, as will be shown later in Fig. 5, the weighted cut size of EAC is actually smaller than that of EIC because the cut edges in EAC are mostly older edges with much less weights.

![Figure 4: Visualization of clustering evolution.](image)

Fig. 5 shows the weighted cut sizes of EAC and EIC for this synthetic data set with different inter-cluster edge probabilities and different maximum cluster sizes. EAC is generally better than EIC in the weighted cut size (the smaller the better). This is because EAC always removes the oldest edge (edge with the least weight) first in order to capture the emerging evolution, while EIC removes edges randomly. As the inter-cluster probability decreases from 0.9 to 0.1, the weighted cut size first increases and then decreases for both algorithms. This is because when the trend of forming the new cluster is uncertain (i.e. $p_{ic} = 0.5$), it becomes a “random cut” for both algorithms. This results in the largest weighted cut sizes for both algorithms. When $p_{ic}$ is larger than 0.5, the trend is more and more obvious, and hence the weighted cut size begins to drop. When it is below 0.5, the two original clusters will be more and more stable, so the weighted cut size will also drop for both EAC and EIC.

Another way to understand the shapes of the curves is to consider the 2 extreme cases when $p_{ic} = 0$ or 1. When $p_{ic} = 0$, it means that there is no change in clustering, so the weighted cut size will be 0. When $p_{ic} = 1$, it means that after enough updates, all the vertices will come to the middle and hence the cut size will also be 0 finally. Notice that when $p < 0.2$, EIC is a little better than EAC, because EAC might have over-reacted to the changes. This difference is relatively small.

Fig. 5 also shows the results when the maximum cluster size changes. For higher values of maximum cluster size, the weighted cut size is smaller. This is because each cluster can hold more vertices, leading to less inter-cluster edges. For all the cases using this data set, EAC yields a lower weighted cut size, which is better. This shows that EAC captures the clustering evolution better.

![Figure 5: Quality experiments for the synthetic data. EAC is generally better than EIC.](image)

### 5.2 Variation of the Stream Stableness

This section describes our quality and performance experiments when real data sets have different stableness characteristics. Fig. 6 characterizes the stableness of each of our data sets according to the metric defined in Section 4.2. We show average stableness results and its standard deviation (error bars) for different window sizes. The results show that Twitter is the most unstable stream, while DNS the most stable. For all streams, the bigger the window, the more stable the stream is.

![Figure 6: Different stableness of the data sets: Twitter is the most unstable, while DNS the most stable.](image)

Fig. 7 shows the performance of EAC and EIC in terms of ratios of the weighted cut sizes and throughputs for differ-
ent streams. On the x-axis, we plot the stableness measures of the different data sets for a window of size 100K. The two dotted lines are comparisons of the ratios of weighted cut sizes. The red line with square markers represents experiments when the maximum cluster size is 1K, while the black circle-marked line represents experiments when the maximum cluster size is 2K. Because EAC generally yields a smaller weighted cut size, we use the ratio of the weighted cut size of EIC divided by that of EAC. Our results show that when the stream is less stable, the difference between EAC and EIC is bigger. EAC can be about 6 times better than EIC under the Twitter stream with MCS of 1K. When the stream is very stable, the difference is smaller. The ratio is close to 1 at that point, EIC being slightly superior to EAC for DNS data.

The two solid lines in Fig. 7 compare EAC and EIC throughputs. The blue line marked by diamonds shows the throughput ratios under 4 different streams with MCS of 1K, while the magenta one with triangle markers is that with MCS of 2K. Since EAC can process more insertions each second, we use the ratio with the throughput of EAC divided by that of EIC. When the stream is less stable, EAC can be up to 8 times better than EIC in both cases. When the stream is quite stable, EAC can still produce about 3 times better throughput. This is because of the simpler architecture and more efficient data structures used by EAC.

Thus the above implies there is an underlying relationship between the stableness and the applicable range of the EAC. EAC is more advantageous when the data is unstable. For example, EAC is more powerful in processing highly dynamic graphs, like the social network data.

5.3 Variation of the Maximal Cluster Size

In this section, we evaluate the quality and throughput of EAC when varying the maximum cluster size (MCS) constraint. We compare EAC with METIS and EIC. For all data sets, we used a window of size 100K. In the following experiments, we also include METIS for comparison. It is considered as the best offline algorithm in terms of quality. So it can serve as the best an algorithm can do.

Fig. 8 shows the quality experiments with different MCS values. Note that we used relatively smaller MCS values for the Twitter data set. This is because the data set is very sparse (the average number of edge connections per vertex is small) and a big MCS value would result in very few cut edges. As a result, it is more difficult to compare the differences among the clustering algorithms. The weighted cut size is normalized by the Kmetis results. We can see that METIS is the best, and that EAC is generally better compared to EIC, especially when the stream is more dynamic (Twitter). EAC is also closer to METIS when the stream is unstable, especially if we look at its performance under the Twitter data set. In some cases, EIC is better than EAC, like in the DNS figure when MCS is less than 4K. This is because when the graph is rather stable, EAC may overreact to the change in the stream, resulting in the removal of unnecessary edges. In those cases, a random algorithm like EIC will do a better job.

Figure 7: EAC vs EIC running data sets with different stream stableness. EAC is more advantageous when the stream is less stable.

Figure 8: Weighted cut size experiments with varying maximum cluster sizes. Results are normalized to the results of Kmetis. METIS is the best, and EAC is better than EIC.

Figure 9: Throughput in tuples/second when the maximum cluster size value changes. EAC is consistently the best.

Fig. 9 shows the throughput in tuples/second of the algorithms when varying the MCS. We can see that EAC is superior to EIC in all cases. METIS is slow because it is an offline algorithm and is not able to incrementally partition the graph after each update. When the maximum cluster size increases, the throughput tends to decrease for both EAC and EIC. In general, a higher MCS value leads to bigger cluster sizes. Thus, when there is a deletion, the number
of edges to be removed/reinserted is larger. We observe that the rate in which the throughput decreases is lower as the MCS grows. This is because there are fewer violations of the constraint as MCS increases, reducing the chance that we need to do an expensive deletion operation. When there are few MCS violations, the deletion operation is mainly triggered by an expired edge from the streaming window. This happens rather stably, making the throughput stable as well. Note that METIS is not sensitive to MCS. This is because the running time of METIS depends on the number of vertices and edges, and it is not penalized for expensive deletion operations.

Our results show that if the graph is sparser, the throughput of EAC and EIC is higher when compared to METIS. For example, EAC has a throughput on the order of $10^5$ when processing the Twitter data. For the DNS data set, which is densest among the four data sets, the throughput will drop to the order of $10^4$. When the graph is sparse, the cluster structure is simpler. The chance of an MCS constraint violation in a sparse graph is lower when compared to a denser graph. If the constraint violation occurs, the number of edges to be removed in order to satisfy the constraint is smaller. These factors result in increased throughput for EAC in sparse graphs.

6. IMPLEMENTATION OF EAC ON IBM’S INFOSPHERE STREAMS

Here we describe our implementation of EAC as a streaming graph operator on IBM’s InfoSphere Streams [11] and show a few snapshots from a graph visualization tool of the evolution of vertex clusters from a streaming graph based on a Twitter feed.

Our EAC stream operator receives three different kinds of streams. The first stream is the stream of graph edge updates, where each tuple has two vertex IDs (i.e., an edge between vertex $u$ and $v$). Events that arrive on this stream result in the execution of the EAC algorithm. The second stream has a single integer attribute $k$, which indicates a user query for the current top $k$ clusters. Events on this stream generate an output with a map with $k$ cluster ID entries pointing to a set of vertex IDs. The third stream has a single vertex ID attribute, and indicates queries for the cluster of the provided vertex. When such queries arrive to the operator, it outputs a set of vertex IDs, indicating all vertices that are in the same cluster.

The operator has two key parameters. The first parameter indicates the maximum cluster size that EAC can maintain. The second is related to the windowing configuration of the edge update stream. The window can be configured as sliding or tumbling, and its size can be defined by count (e.g., last $x$ tuples) or by time (e.g., last $x$ seconds).

To visualize the clustering evolution, we extended the EAC operator implementation to periodically output the top $k$ clusters at specific intervals. We used this implementation to visualize the evolution of a Twitter mention graph, i.e., each edge indicates that a user has mentioned another user in a tweet. In this experiment, we processed one day of a 10% Twitter data feed and filtered the tweet content based on keywords associated with a mobile device brand. The EAC operator used a window size of 100,000 edges and a maximum cluster size of 800. We then output the top 10 clusters maintained by the EAC algorithm at every 20,000 newly processed edges. We chose such a cluster size to easily force cluster merge and split operations.

Figure 10 shows the top 10 clusters at three different points of time using the GraphInsight visualization tool [8]. In the figures, each connected component represents one cluster. The colors of vertices represent their degrees. The color ranges from dark blue, for nodes with a low degree, to red, for nodes with a high degree. In snapshot 1, we highlight three different clusters. The red cluster (freeform shape with solid line) contains two main nodes, labeled 1 and 2. The black cluster contains two high degree vertices, labeled 3 and 4. The green cluster contains one main high degree vertex, labeled 5. In snapshot 2, we observe that the red cluster was subject to two actions: (i) a split, as the vertex labeled 2 is no longer in the red cluster, and (ii) a merge, as the cluster contains many of the nodes that were in the black cluster (vertices labeled 3 and 4). This is the result of a merge operation reaching the maximum cluster size constraint. In addition, we observe that the vertex labeled 2 is together with the green cluster, which contains vertex with label 5. In snapshot 3, we notice that, as new edges are processed, the red cluster merged with part of the green cluster (vertex with label 2). This operation led to a split, which resulted in the vertex labeled 3 to again be in a different cluster (black cluster).

7. CONCLUSION AND FUTURE WORK

We proposed an efficient evolution-aware clustering algorithm (EAC) for a streaming graph. We designed an efficient data structure based on hash tables and presented an evolution analysis scheme to quantify the stream stableness. In general, unstable streams tend to cause more cluster evolution. We conducted a set of quality experiments to show how well different algorithms can capture the clustering evolution. Among all the algorithms, METIS is the best. EAC shows better clustering quality than another state-of-the-art clustering algorithm for streaming graphs (EIC). We also conducted throughput experiments on different real-world datasets. Our results show that EAC has the best performance and has orders of magnitude higher throughput than METIS. We also observed that if the stream is unstable, EAC is better than EIC in both quality and throughput. We implemented EAC as a streaming graph operator on IBM’s InfoSphere Streams to demonstrate in practice the evolution of vertex clusters in a Twitter graph.

One area of future research is the distributed implementation of EAC. If the graph within the current streaming window is too large to fit into the main memory of a single machine, we can distribute the storage and computational requirements to different machines. Since the vertex table is much smaller than the cluster table, it is likely that the vertex table can fit into a single machine. As a result, only the cluster table needs to be distributed among multiple machines. The cluster table can be distributed based on a hash partitioning of the cluster IDs. Another area of future research is the study of applying different weights for capturing cluster evolution, besides edge recency.

8. ACKNOWLEDGEMENT

Research was sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program, Agreement...
Figure 10: Visualization of the top 10 clusters computed by the EAC algorithm when processing a Twitter mention/retweet graph. Snapshot 1 highlights the red cluster, containing two prominent nodes (1 and 2). As EAC processed more edges (Snapshot 2), part of the red cluster gets merged with the black cluster (anchored by nodes 1, 3 and 4), and another part gets merged with the green cluster (anchored by nodes 2 and 5). In Snapshot 3, the red cluster merges back with the vertex with label 2, resulting into vertex 3 and 5 being back to different clusters.

Number W911NF-12-C-0028. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Defense Advanced Research Projects Agency or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Mindi Yuan and Yi Lu are partially supported by NSF grant CNS-1150080.

9. REFERENCES