Online Analysis of Community Evolution in Data Streams

Charu C. Aggarwal, Philip S. Yu
IBM T. J. Watson Research Center
{ charu, psyu }@us.ibm.com

Abstract

This paper discusses the problem of online change detection in a large set of interacting entities. Such trends include the gradual formation and dissolution of different communities of interaction. Our results are focussed on the case where the interacting entities are received in the form of a *fast data stream* of interactions. In such cases, a user may wish to perform repeated *exploratory querying* of the data for different kinds of user-defined parameters. This is difficult to perform in a fast data stream because of the one-pass constraints on the computations. We propose an online analytical processing framework which separates out online data summarization from offline exploratory querying. The result is a method which provides the ability to perform exploratory querying without compromising on the quality of the results. The algorithms are tested over large sets of graph data streams with varying levels of evolution.

1 Introduction

The data stream has gained importance in recent years because of the greater ease in data collection methodologies resulting from advances in hardware technology. This has resulted in a host of papers on the extension of data mining techniques to the case of data streams [1, 2, 3, 8, 13]. In this paper, we discuss the problem of detecting patterns of interaction among a set of entities in a stream environment. Examples of such entities could be a set of businesses which interact with one another, sets of co-authors in a dynamic bibliography data base, or it could be the hyperlinks from web pages. In each of these cases, the interaction among different entities can rapidly evolve over time.

A convenient way to model the entity interaction relationships is to view them as graphs in which the nodes correspond to entities and the edges correspond to the interactions among the nodes. The weights on these edges represent the level of the interaction between the different participants. For example, in the case when the nodes represent interacting entities in a business environment, the weights on the edges among these entities could represent the volume of business transactions. A *community of interaction* is defined to be a set of entities with a high degree of interaction among the participants.

The problem of finding communities in dynamic and evolving graphs been discussed in [6, 7, 9, 10, 11, 12, 14, 15, 16]. Since most of the current techniques are designed for applications such as the web, they usually assume a *gradually evolving* model for the interaction. Such techniques are not very useful for a fast stream environment in which the entities and their underlying relationships may quickly evolve over time. In addition, it is important to provide a user the *exploratory capability* to query for communities over different time horizons. Since individual points in the data streams cannot be processed more than once, we propose a framework which separates out the *offline exploratory* algorithms from the online stream processing part. The online stream processing framework creates summaries of the data which can then be further processed for exploratory querying. This paper is focussed on an Online Analytical Processing (OLAP) approach for providing offline exploratory capabilities to users in performing *change detection* across communities of interest over different time horizons.

Some examples of exploratory queries in which a user may be interested are as follows:
(1) Find the communities with substantial increase in interaction level in the interval $(t-h, t)$. We refer to such communities as *expanding communities*.
(2) Find the communities with substantial decrease in interaction level in the interval $(t-h, t)$ We refer to such communities as *contracting communities*.
(3) Find the communities with the most stable interaction level in the interval $(t-h, t)$.

We note that the process of finding an emerging or contracting community needs to be carefully designed in order to normalize for the behavior of the community evolution over different time horizons. For example, consider a data stream in which two entities n_1 and n_2 share a high level of interaction in the period $(t-h, t)$. This alone does not mean that the interaction level between n_1 and n_2 is stable especially if these entities

had a even higher level of interaction in the previous period $(t - 2 \cdot h, t - h)$. Thus, a careful model needs to be constructed which tracks the behavior of the interaction graph over different time horizons in order to understand the nature of the change.

This paper is organized as follows. In the next section, we discuss notations for modelling the interaction among different entities. We also discuss methods for modelling the change in interaction among different entities. In section 4, we discuss methods to accumulate the statistics of the aggregate interaction among different entities. We also discuss methods to derive the change in interaction from these aggregate statistics. In section 5, we present the empirical results. Section 6 contains the conclusions and summary.

1.1 Contributions of this Paper This paper discusses new algorithms for online analysis of community detection in data streams. The paper discusses an OLAP-style framework in which the online preprocessing of the data stream is separated from the offline querying of the stream. Thus, the user can have the flexibility to query these summaries in an interactive way in order to find detailed information about the communities in the most relevant horizons. We design an innovative clustering algorithm which can determine clusters of interactions with the most significant change. This includes information about the disposition of the communities in terms of their expansion or contraction. Thus, the paper discusses a general framework for online analysis of the data stream.

2 Online Summarization of Graphical Data Stream

In this section, we will discuss the overall interaction model among the different entities. We will also discuss the process of online summarization of the data stream. This interaction model is stored as a graph $G = (N, A)$, in which N denotes the set of nodes, and A denotes the set of edges. Each node $i \in N$ corresponds to an entity. The edge set A consists of edges (i, j), such that i and j are nodes drawn from N. Each edge (i, j) represents an interaction between the entities i and j. Each edge (i, j) also has a weight $w_{ij}(t)$ associated with it. This weight corresponds to the number of interactions between the entities i and j. For example, when the interaction model represents a bibliography database, the nodes could represent the authors and the weights on the edges could represent the number of publications on which the corresponding authors occur together as co-authors. As new publications are added to the database the corresponding weights on the individual edges are modified. It is also possible for new nodes to be added to the data as new authors are added to the original mix. In this particular example, the weight on each edge increases by one, each time a new co-authorship relation is added to the database. However, in many applications such as those involving business interaction, this weight added in each iteration can be arbitrary, and in some cases even negative.

In order to model the corresponding stream for this interaction model, we assume that a current graph $G(t) = (N(t), A(t))$ exists which represents the history of interactions at time t. At time $(t + 1)$ new additions may occur to the graph $G(t)$. Subsequently, each new arrival to the stream contains two elements:

- An edge (i, j) corresponding to the two entities between whom the interaction has taken place.
- An *incremental* weight $\delta w_{ij}(t)$ illustrating the additional interaction which has taken place between entities i and j at time t.

We refer to the above pair of elements as representative of an *interaction event*. We note that the nodes i, j, or the edge (i, j) may not be present in $N(t)$ and $A(t)$ respectively. In such a case, the node set $N(t)$ and edge set $A(t)$ need to be modified to construct $N(t + 1)$ and $A(t + 1)$ respectively. In the event that a given edge does not exist to begin with, the original weight of (i, j) in $G(t)$ is assumed to be zero. Also, in such a case, the value of the edge set $A(t + 1)$ is augmented as follows:

$$A(t + 1) = A(t) \cup \{(i, j)\} \tag{2.1}$$

In the event that either the nodes i or j are not present in $N(t)$, the corresponding node set needs to be augmented with the new node(s). Furthermore, the weight of the edge (i, j) needs to be modified. If the edge (i, j) is new, then the weight of edge (i, j) in $G(t + 1)$ is set to δw_{ij}. Otherwise, we add the incremental weight δw_{ij} to the current weight of edge (i, j) in $G(t)$. Therefore, we ave:

$$w_{ij}(t + 1) = w_{ij}(t) + \delta w_{ij}(t) \tag{2.2}$$

We assume that the set of interaction events received at time t are denoted by $\mathcal{E}(t)$. In each iteration, the stream maintenance algorithm adds the interaction events in $\mathcal{E}(t)$ to $G(t)$ in order to create $G(t + 1)$. We refer to the process of adding the events in $\mathcal{E}(t)$ to $G(t)$ by the $\oplus$ operation. Therefore, we have:

$$G(t + 1) = G(t) \oplus \mathcal{E}(t) \tag{2.3}$$

At each given moment in time, we maintain the current graph of interactions $G(t)$ in main memory. In addition, we periodically store the graph of interactions

on disk. We note that the amount of disk storage available may often be limited. Therefore, it is desirable to store the graph of interactions in an efficient way during the course of the stream arrival. We will refer to each storage of the graph of interactions at a particular moment as a *frame*. Let us assume that the storage limitation for the number of frames is denoted by S. In this case, one possibility is to store the last S frames at uniform intervals of t'. The value of S is determined by the storage space available. However, this is not a very effective solution, since it means that a history of larger than $S \cdot t'$ cannot be recalled.

One solution to this problem is to recognize that frames which are more stale need not be stored at the same frequency as more recent frames. Let t_c be the current time, and t_{min} be the minimum granularity at which 0th tier snapshots are stored. We divide the set of S frames into $\theta = \log_2(t_c/t_{min})$ tiers. The ith tier contains snapshots which are separated by a distance of $t_{min} \cdot 2^{i-1}$. For each tier, we store the last S/θ frames. This ensures that the total storage requirement continues to be S. Whenever it is desirable to access the state of the interaction graph for the time t, we simply have to find the frame which is temporally closest to t. The graph from this temporally closest frame is utilized in order to approximate the interaction graph at time t. The tiered nature of the storage process ensures that it is possible to approximate recent frames to the same degree of (percentage) accuracy than less recent frames. While this means that the (absolute) approximation of stale frames is greater, this is quite satisfactory fir a number of real scenarios. We make the following observations:

LEMMA 2.1. *Let h be a user-specified time window, and t_c be the current time. Then a snapshot exists at time t_s, such that $h/(1+\theta/S) \leq t_c - t_s \leq (1+\theta/S) \cdot h$.*

Proof. This is an extension of the result in [3]. The proof is similar.

In order to understand the effectiveness of this simple methodology, let us consider a simple example in which we store (a modest number of) $S = 100,000$ frames for a stream over 10 years, in which the minimum granularity of storage t_{min} is 1 second. We have intentionally chosen an extreme example (in terms of the time period of the stream) together with a modest storage capability in order to show the effectiveness of the approximation. In this case, the number of tiers is given by $\theta = \log_2(10 * 365 * 24 * 3600) \approx= 29$. By substituting in Lemma 2.1, we see that it is possible to find a snapshot which is between 99.97% and 100.03% of the user specified value.

In order to improve the efficiency of edge storage further, we need to recognize the fact that large portions of the graph continue to be identical over time. Therefore, it is inefficient for the stream generation process to store the entire graph on disk in each iteration. Rather, we store only incremental portions of the graph on the disk. Specifically, let us consider the storage of the graph $G(t)$ for the ith tier at time t. Let the last time at which an $(i+1)$th tier snapshot was stored be denoted by t'. (If no snapshot of tier $(i+1)$ exists, then the value of t' is 0. We assume that $G(0)$ is the null graph.) Then, we store the graph $F(t) = G(t) - G(t')$ at time t. We note that the graph $F(t)$ contains far fewer edges than the original graph $G(t)$. Therefore, it is more efficient to store $F(t)$ rather than $G(t)$. Another observation is that a snapshot for the ith tier can be reconstructed by summing the snapshots for all tiers larger than i.

LEMMA 2.2. *We assume that the highest tier is defined by m. Let t_i be the time at which the snapshot for tier i is stored. Let t_{i+1}, t_{i+2} ... t_m be the last time stamps of tiers $(i+1) \ldots m$ (before t_i) at which the snapshots are stored. Then the current graph $G(t_i)$ at the time stamp t_i is defined as follows:*

$$G(t_i) = \sum_{j=i}^{m} F(t_j) \tag{2.4}$$

Proof. This result can be proved easily by induction. We note that the definition of $F(\cdot)$ implies that:

$$\begin{aligned} G(t_i) - G(t_{i+1}) &= F(t_i) \\ G(t_{i+1}) - G(t_{i+2}) &= F(t_{i+1}) \\ &\ldots \\ G(t_{m-1}) - G(t_m) &= F(t_{m-1}) \\ G(t_m) - 0 &= F(t_m) \end{aligned}$$

By summing the above equations, we obtain the desired result.

The above result implies that the graph at a snapshot for a particular tier can be reconstructed by summing it with the snapshots at higher tiers. Since there are at most $\theta = \log_2(S/t_{min})$ tiers, it implies that the graph at a given time can be reconstructed quite efficiently in a practical setting.

3 Offline Constructon and Processing Of Differential Graphs

In this section, we will discuss the offline process of generating differential graphs and their application to

the evolution detection process. The differential graph is generated over a specific time horizon (t_1, t_2) over which the user would like to test the behavior of the data stream. The differential graph is defined over the interval (t_1, t_2) and is defined as a fraction of the interactions over that interval by which the level of interaction has changed during the interval (t_1, t_2). In order to generate the differential graph, we first construct the *normalized graph* at the times t_1 and t_2. The normalized graph $G(t) = (N(t), A(t))$ at time t is denoted by $\overline{G(t)}$, and contains exactly the same node and edge set, but with different weights. Let $W(t) = \sum_{(i,j) \in A} w_{ij}(t)$ be the sum of the weights over all edges in the graph $G(t)$. Then, the normalized weight $\overline{w_{ij}(t)}$ is defined as $w_{ij}(t)/W(t)$. We note that the normalized graph basically comprises the fraction of interactions over each edge.

Let t_1' be the last snapshot stored just before time t_1 and t_2' be the snapshot stored just before time t_2. The first step is to construct the graphs $G(t_1')$ and $G(t_2')$ at time periods t_1' and t_2' by adding the snapshots at the corresponding tiers as defined by Lemma 2.2. Then we construct the normalized graph from the graphs at times t_1' and t_2'. The differential graph is constructed from the normalized graph by subtracting out the corresponding edge weights in the original normalized graphs. Therefore, the differential graph $\Delta G(t_1', t_2')$ basically contains the same nodes and edges as $\overline{G(t_2')}$, except that the differential weight $\Delta w_{ij}(t_1', t_2')$ on the edge (i, j) is defined as follows:

$$\Delta w_{ij}(t_1', t_2') = \overline{w_{ij}(t_2')} - \overline{w_{ij}(t_1')} \tag{3.5}$$

In the event that an edge (i, j) does not exist in the graph $\overline{G(t_1')}$, the value of $w_{ij}(t_1')$ is assumed to be zero. We note that because of the normalization process, the differential weights on many of the edges may be negative. These correspond to edges over which the interaction has reduced significantly during the evolution process. For instance, in our example corresponding to a publication database, when the number of jointly authored publications reduces over time, the corresponding weights in the differential graph are also negative.

Once the differential graph has been constructed, we would like to find clusters of nodes which show a high level of evolution. It is a tricky issue to determine the subgraphs which have a high level of evolution. A natural solution would be find the clustered subgraphs with high weight edges. However, in a given subgraph, some of the edges may have high positive weight while others may have high negative weight. Therefore, such subgraphs correspond to the entity relationships with high evolution, but they do not necessarily correspond to entity relationships with the greatest increase or decrease in interaction level. In order to find the community of interaction with the greatest increase in interaction level, we need to find subgraphs such that most interactions within that subgraph have either a high positive or high negative weight. This is a much more difficult problem than the pure vanilla problem of finding clusters within the subgraph $\Delta G(t_1', t_2')$.

Algorithm *FindEvolvingCommunities*(Graph: (N, A),
  EdgeWeights: $\Delta w_{ij}(t_1, t_2)$, NumberOfClusters: k);
begin
  Randomly sample nodes $n_1 \ldots n_k$ as seeds;
  Let B be the bias vector of length $|N|$;
  Set each position in B to 0;
  while not(*termination_criterion*) **do**
    begin
    $(N_1 \ldots N_k)$ =*AssignNodes*(N, B, $\{n_1 \ldots n_k\}$);
    B =*FindBias*$(N_1 \ldots N_k)$;
    $(N_1' \ldots N_k')$ = *RemoveNodes*$(N_1 \ldots N_k)$;
    { Assume that the removed nodes are null partitions }
    $(n_1 \ldots n_k)$ =*RecenterNodes*$(N_1 \ldots N_k)$;
    end
end

Figure 1: Finding Evolving Communities

3.1 Finding Evolving Communities In this section, we will define the algorithm for finding evolution clusters in the interaction graph based on the user defined horizon. The process of finding the most effective clusters is greatly complicated by the fact that some of the edges correspond to an increase in the evolution level, whereas other edges correspond to a decrease in the evolution level. The edges corresponding to an increase in interaction level are referred to as the positive edges, whereas those corresponding to a reduction in the interaction level are referred to as the negative edges.

We design an algorithm which can effectively find subgraphs of positive or negative edges by using a partitioning approach which tracks the positive and negative subgraphs in a dynamic way. In this approach, we use a set of seed nodes $\{n_1 \ldots n_k\}$ in order to create the clusters. Associated with each seed n_i is a partition of the data which is denoted by N_i. We also have a bias vector B which contains $|N|$ entries of $+1$, 0, or -1. The bias vector is an indicator of the nature of the edges in the corresponding cluster. The algorithm utilizes an iterative approach in which the clusters are constructed around these seed nodes. The overall algorithm is illustrated in Figure 1. As illustrated in Figure 1, we first pick the k seeds randomly. Next, we enter an iterative loop in which we refine the initial seeds by performing the following steps:

- We assign each nodes to one of the seeds. The process of assignment of an entity node to a given seed node is quite tricky because of the fact that we would like a given subgraph to represent either increasing or decreasing communities. Therefore, we need to choose effective algorithms which can compute the distances for each community in a different way. We will discuss this process in detail slightly later. We note that the process of assignment is sensitive to the nature of the *bias* in the node. The bias in the node could represent the fact that the cluster seeded by that node is likely to become one of the following: (1) An Expanding Community (2) A Contracting Community (3) Neutral Bias (Initial State). Therefore, we associate a *bias bit* with each seed node. This bias bit takes on the values of $+1$, or -1, depending upon whether the node has the tendency to belong to an expanding or contracting community. In the event that the bias is neutral, the value of that bit is set to 0. We note that an expanding community corresponds to positive edges, whereas a contracting community corresponds to negative edges. The process of assignment results in k node sets which are denoted by $N_1 \ldots N_k$. In the assignment process,we compute the distance of each seed node to the different entities. Each entity is assigned to its closest seed node. The algorithm for assignment of entities to seed nodes is denoted by *AssignNodes* in Figure 1.

- Once the assignment step has been performed, we re-assess the bias of that seed node. This is denoted by *FindBias* in Figure 1. The overall approach in finding the bias is to determine whether the interactions in the community attached to that seed node represent expansion or contraction. We will discuss the details of this algorithm slightly later. The bias bit vector B is returned by this procedure.

- Some of the seed nodes may not represent a coherent community of interaction. These seed nodes may be removed by the community detection algorithm. This process is achieved by the algorithm *RemoveNodes*. The new set of nodes is denoted by $N_1' \ldots N_k'$. We note that each N_i' is either N_i or null depending upon whether or not that node was removed by the algorithm.

- The final step is to re-center the seeds within their particular subgraph. The re-centering process essentially reassigns the seed node in a given subgraph N_i' to a more central point in it. In the event that N_i' is a null partition, the recentering process simply picks a random node in the graph as the corresponding seed. Once the recentering process is performed, we can perform the next iteration of the algorithm. The recentering procedure is denoted by *RecenterNodes* in Figure 1. The new set of seeds $n_1 \ldots n_k$ are returned by this algorithm.

3.2 Subroutines for Determination of the Communities In the afore-mentioned discussion, we described the overall procedure for finding the communities of interaction. In this section, we will discuss the details of the subroutines which are required for determination of these communities. We will first discuss the procedure *AssignNodes* of Figure 1. The process of assigning the nodes to each of the centroids requires the use of the bias information stored in the bias nodes. Note that if a seed node has positive bias, then it needs to be ensured that the other nodes which are assigned to this seed node are related to it by positive interactions. The opposite is true if the bias on the seed node is negative. Finally, in the case of nodes with neutral bias, we simply need to find a path with high absolute interaction level. In this case, the positivity or negativity of the sign matters less than the corresponding absolute value. In order to achieve this goal, we define a *bias sensitive* distance function $f(n_1, n_2, b)$ between two nodes n_1 and n_2 for the bias bit b. For a given path P in the graph $\Delta G(t_1, t_2)$, we define the average interaction level $w(P)$ as the sum of the interaction levels on P divided by the number of edges on P. Therefore, we have:

$$w(P) = \sum_{(i,j) \in P} \Delta w_{ij}(t_1, t_2)/|P| \tag{3.6}$$

We note that a path with high average *positive* weight corresponds to a set of edges with increasing level of interaction. This can also be considered an expanding community. The opposite is true of a path with high average *negative* weight, which corresponds to a contracting community. Therefore, the value of $w(P)$ is equal to the weight of the path divided by the number of edges on that path. We also define the average *absolute* average interaction level $w^+(P)$ as follows:

$$w^+(P) = \sum_{(i,j) \in P} |\Delta w_{ij}(t_1, t_2)|/|P| \tag{3.7}$$

Note that the absolute interaction level does not have any bias towards a positive or negative level of interaction. Once we have set up the definitions of the path weights, we can also define the value of the interaction function $f(n_1, n_2, b)$. This interaction function is defined over all possible pairs of nodes (n_1, n_2).

$f(n_1, n_2, b)$ = Most positive value of $w(P)$ $\forall$ paths

P between n_1 and n_2 if $b = 1$
Modulus of most negative value of $w(P)$
$\forall$ P between n_1 and n_2 if $b = -1$
Largest value of $w^+(P)$ $\forall$ paths
P between n_1 and n_2 if $b = 0$

We note that the above interaction function is defined on the basis of the sum of the interaction values over a given path. In some cases, this interaction function can provide skewed results when the path lengths are long. This could result in less effective partitioning of the communities. A different interaction function is defined as the minimum interaction on the path between two entities. However, the bias of the corresponding centroid on that path is used in order to define the interaction function. This minimum interaction $w(P)$ is defined as follows:

$$w(P) = \begin{cases} \min_{(i,j)\in P}\max\{\Delta w_{ij}(t_1,t_2),0\} & \text{if } b = 1 \\ \max_{(i,j)\in P}\min\{\Delta w_{ij}(t_1,t_2),0\} & \text{if } b = -1 \\ \min_{(i,j)\in P}|\Delta w_{ij}(t_1,t_2)| & \text{if } b = 0 \end{cases}$$

We note that the above mentioned function simply finds the minimum (absolute) weight edge of the corresponding sign (depending on the bias) between the two nodes. The corresponding interaction function $f(n_1, n_2, b)$ is defined in the same way as earlier. Henceforth, we will refer to the two above-defined functions as the average-interaction function and minimum interaction function respectively. In the latter case, the interaction distance corresponds to the interaction on the weakest link between the two nodes. As our experimental results will show, we found the minimum function to be slightly more robust than the average interaction function.

During the assignment phase, we calculate the value of the function $f(n_i, n, b)$ from each node n_i to the seed node n using the bias bit b. Each node n_i is assigned to the seed node n with the largest *absolute* value of the above-mentioned function. This process ensures that nodes are assigned to seeds according to their corresponding bias. The process of computation of the interaction function will be discussed in some detail slightly later.

Next, we determine the bias of each seed node in the procedure *FindBias*. In order to do so, we calculate the bias-index of the community defined by that seed node. The bias index of the community N_i is denoted by $\mathcal{I}(N_i)$, and is defined as the edge-weight fraction of the expanding portion of N_i. In order to do, so we divide the positive edge weights in the community by the total absolute edge weight in the same community. Therefore, we have:

$$\mathcal{I}(N_i) = \frac{\sum_{(p,q)\in N_i} \max\{0, \Delta w_{pq}(t_1,t_2)\}}{\sum_{(p,q)\in N_i} |\Delta w_{pq}(t_1,t_2)|} \quad (3.8)$$

We note that the bias index is 1 when all the edges in the community corresponding to increasing interaction, and is 0 when all the edges correspond to reducing interaction. Therefore, we define a threshold $t \in (0, 0.5)$. If the value of $\mathcal{I}(N_i)$ is less than t then the bias bit is set to -1. Similarly, if the value of $\mathcal{I}(N_i)$ is larger than $1-t$, the bias bit is set to 1. Otherwise, the bias bit is set to zero.

Once the bias bits for the nodes have been set, we remove those seeds which have very few nodes associated with them. Such nodes usually do not correspond to a coherent community of interaction. This procedure is referred to as *RemoveNodes*. Thus, each set of nodes N_i is replaced by either itself or a null set of nodes. In order to implement this step, we use a minimum threshold on the number of nodes in a given partition. This threshold is denoted by mn_t. All partitions with less than mn_t entities are removed from consideration, and replaced by the null set.

The last step is to recenter the nodes within their corresponding partition. This denoted by the procedure *RecenterNodes* in Figure 1. The process of recentering the nodes requires us to use a process in which the central points of subgraphs are determined. In order to recenter the nodes, we determine the node which minimizes the maximum distance of any node in the cluster. This is achieved by computing the distance of all points in the cluster starting at each node, and finding the minimax distance over these different values. The process of recentering helps to adjust the centers of the nodes in each iteration such that the process of partitioning the community sets becomes more effective over time.

3.3 Approximating Interaction Distances Among Nodes The only remaining issue is to discuss the methodology for determining the interaction distances among nodes. We would like our algorithm to be general enough to find the maximum interaction distance for general functions. It is important to understand that the problem of finding the maximum interaction distance between two nodes is NP-hard.

OBSERVATION 3.1. *The problem of determining the maximum interaction distance between two nodes is NP-hard for arbitrary interaction functions.*

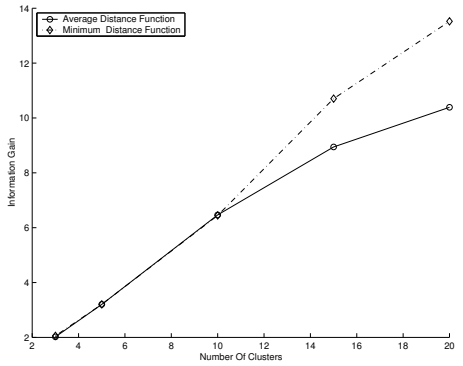

Figure 2: Information Gain with Number Of Clusters (C4.I5.D500)

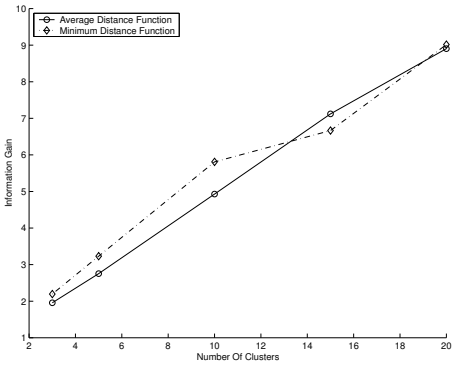

Figure 3: Information Gain with Number Of Clusters (C5.I6.D500)

This observation is easily verified by observing that the problem of finding the longest path in a graph is NP-hard [5]. Since the particular case of picking the interaction function as the edge weight is NP-hard, the general problem is NP-hard as well.

However, it is possible to approximate the interaction distance of a maximum length using dynamic programming. Let wn_{ij}^t be the maximum interaction distance between two nodes using at most t nodes on that path. Let P_{ik}^t be the maximum length path between i and k using t edges. Let $PS_{ikj}^{\oplus}$ denote the path obtained by concatenating P_{ik}^t with the edge (k, j). Then, we define wn_{ij}^t recursively as follows:

$$wn_{ij}^0 = 0;$$
$$wn_{ij}^{t+1} = \max_k\{wn_{ij}^t, w(PS_{ikj}^{\oplus})\}$$

We note that this dynamic programming algorithm does not always lead to an optimal solution in the presence of cycles in the graph [5]. However, for small values of t, it approximates the optimal solution well. This is because cycles are less likely to be present in paths of smaller length. It is also important to understand that if two entities are joined only by paths containing a large number of edges, then such pairs of entities should not be regarded as belonging to the same community. Therefore, we imposed a threshold *maxthresh* on the maximum length of the (shortest interaction distance) path between two nodes for them to belong to the same community. For a pair of nodes in which the corresponding path length was exceeded, the value of the interaction distance is set to 0.

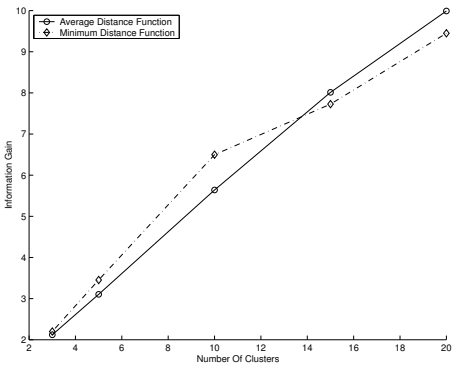

Figure 4: Information Gain with Number Of Clusters (C6.I7.D500)

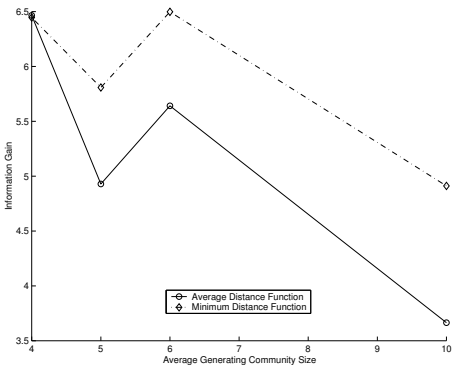

Figure 5: Information Gain with Increasing Community Set Size

4 Empirical Results

For the purpose of testing, we generated a number of graphs with strong correlations of interaction among the different nodes. In general, it was desirable to generate

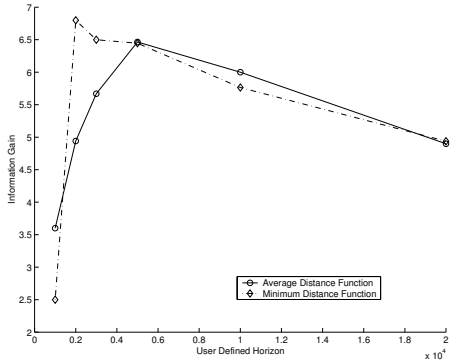

Figure 6: Information Gain with Increasing User Specified Horizon (C4.I5.D500)

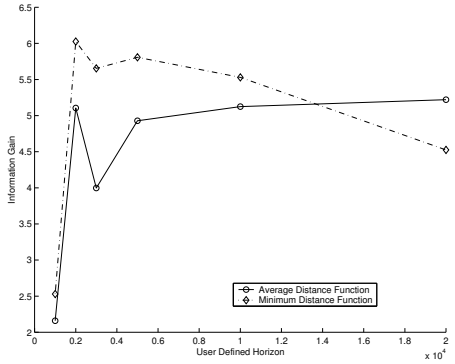

Figure 7: Information Gain with Increasing User Specified Horizon (C5.I6.D500)

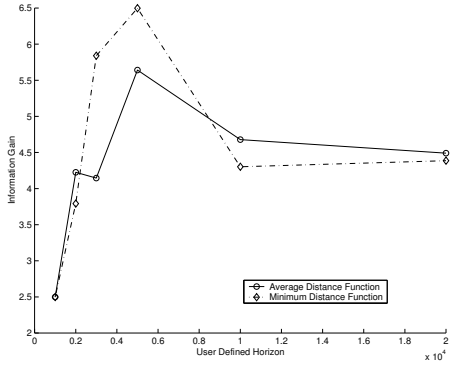

Figure 8: Information Gain with Increasing User Specified Horizon (C6.I7.D500)

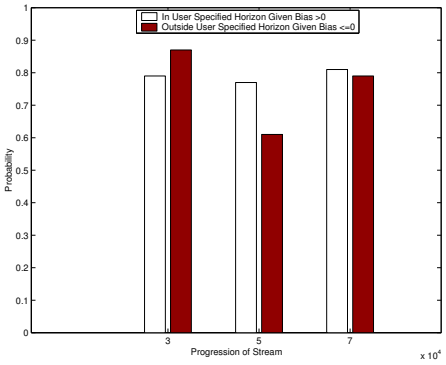

Figure 9: Effectiveness of finding expanding or contracting communities

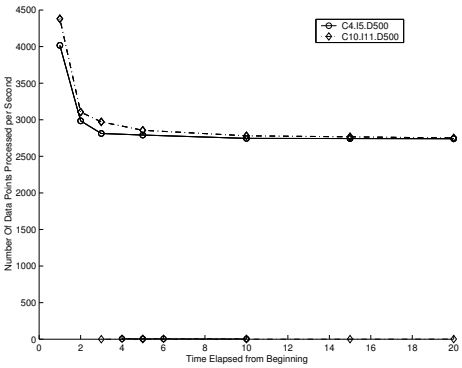

Figure 10: Stream Processing Rate

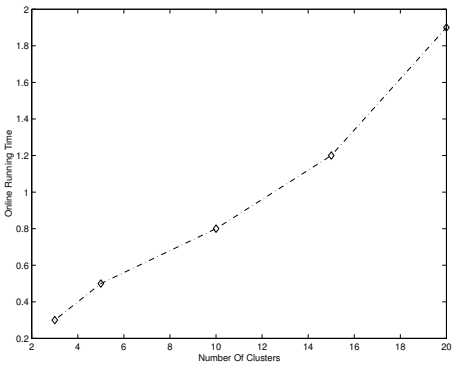

Figure 11: Interaction Times for Clustering

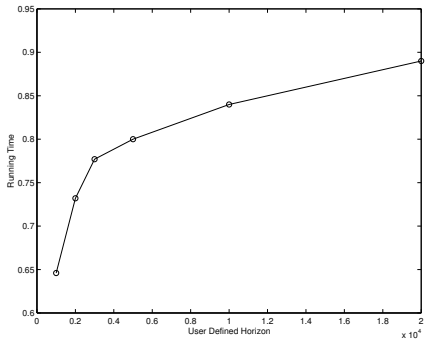

Figure 12: Interaction Time for Clustering

graphs in which the nodes showed strong interactions with each other, as well as variations in behavior over time. In order to achieve this goal, we used a technique which draws its motivation from the frequent itemset generation methodology discussed in [4]. However, the generation method was modified in order to suit the community detection problem.

The first step was to generate $L = 2000$ maximal "potential communities" over the nodes in the graph. These potential communities capture the tendencies of particular groups of entities to interact with one another. We first picked the size of a maximal potentially large community as a random variable from a poisson distribution with mean μ_L. Each successive community was generated by picking half of its items from the current community, and generating the other half randomly. This method ensures that large communities often have common entities. Each community I has a weight w_I associated with it, which is chosen from an exponential distribution with unit mean.

The maximal potential communities were then used in order to generate the *community sets* of interaction. First, the size S_T of a *current community set* was chosen as a poisson random variable with mean μ_T. Each current community set was generated by assigning maximal potentially large communities to it in succession. The community to be assigned to a community set was chosen by rolling an L sided weighted die depending upon the weight w_I assigned to the corresponding community I. If a community did not fit exactly, it was assigned to the current community set half the time, and moved to the next community set the rest of the time. In order to capture the fact that all members in a potentially large community may not always participate in the current interaction, we added some noise to the process by corrupting some of the added communities. For each community I, we decide a noise level $n_I \in (0, 1)$. We generated a geometric random variable G with parameter n_I. While adding a potentially large community to a community set, we dropped $\min\{G, |I|\}$ random nodes from the set. The noise level n_I for each community I was chosen from a normal distribution with mean 0.5 and variance 0.1.

We shall also briefly describe the symbols that we have used in order to annotate the data. The three primary factors which vary are the average community set size μ_T, the size of an average maximal potentially large community μ_L, and the number of community sets being considered. A data set having $\mu_T = 10$, $\mu_L = 4$, and 100 community sets is denoted by C10.I4.D100.

Next, we generate the edges of the graph stream from these community sets. For this purpose, we use two parameters which are referred to as the *epochsize* and the *maxrange*. The generation of the graph edges evolves over time and is divided into different periods which are referred to as an epoch. Within a given epoch, the edges of the graph are generated from the same distribution. At any given epoch, a set of *maxrange* contiguous community sets are used to generate the interactions among different entities. Within a given epoch, we generated *epochsize* such interactions. Each of these interactions was generated by picking two nodes from the community set, and assigning a random interaction level to this edge. Unless otherwise mentioned, we used a value of $maxrange = 21$ and $epochsize = 500$.

The interaction stream was processed in an online fashion to generate the community graph $G(t)$, which were stored using our tiered scheme. The stored summary information could then be utilized by a user to perform interactive clustering over multiple horizons. This is a more useful approach because of the potentially exploratory nature of user queries. The process of matching the output clusters to the original community sets was much more difficult since the original community sets had considerable overlaps with each other. Therefore, a given node could fall into multiple community sets, and could not be meaningfully matched with the original clusters. However, it is possible to calculate the level of correlation between the input and the output clusters using a measure called *information gain*. This information gain is defined by analyzing the level of fragmentation of the community sets among the different clusters. Ideally, we would like to have a situation in which the community sets do not get fragmented among the different clusters. Consider a community set T, which gets distributed among the k clusters such that the community i contains a fraction $p_i(T)$ of the entities in that cluster. Then, we define the gini index of

community set T as follows:

$$Gini(T) = \sum_{i=1}^{k} p_i(T)^2 \tag{4.9}$$

We note that if the community set T occurs in only one cluster, then the value of $Gini(T)$ is 1 unit. On the other hand, when the community gets divided among the different clusters equally, the value of this function is $1/k$. Therefore, the information gain for the community set T is defined as the ratio of $Gini(T)$ to $1/k$.

$$InformationGain(T) = k \cdot Gini(T) \tag{4.10}$$

In order to calculate the overall information gain within a particular range, we need to use all the generating community sets for a given horizon. Let this generating community set be denoted by $\mathcal{G}$. Then, the information gain $\mathcal{IN}(\mathcal{G})$ for the community set collection $\mathcal{G}$ is defined as follows:

$$\mathcal{IN}(\mathcal{G}) = \sum_{T \in \mathcal{G}} InformationGain(T)/|T| \tag{4.11}$$

The value of $\mathcal{IN}(\mathcal{G})$ defines an intuitive measure for the information gain of all community sets in $\mathcal{G}$. A value of $\mathcal{IN}(\mathcal{G})$ which is significantly larger than 1 defines a clustering which is able to distinguish between the different community sets $\mathcal{G}$ very well. The use of information gain as a clustering measure is specially effective in situations in which there is considerable overlap between different communities. In order to test the quality of the clustering process, we computed the information gain for all the generating community sets within the currently specified user horizon.

We tested the information gain for a variety of different data sets and ranges of input/output parameters. The data sets from which the streams were generated were C4.I5.D500, C5.I6.D500, and C6.I7.D500. In the first set of results of Figures 2, 3, and 4 we have illustrated the information gain variation with the number of input clusters. The number of clusters are plotted on the X-axis, whereas the information gain is plotted on the Y-axis. The horizon was fixed at 5000 for each case. Furthermore, for each case we have illustrated the results for the use of the average distance function as well as the minimum value distance function. The general observation was that both distance functions were roughly competitive, though the minimum distance function provided superior results. A closer examination of the clusters revealed that the use of the average distance function resulted in some clusters with poor locality behavior. Such clusters did not contribute significantly to the value of the information gain. For all cases, the information gain was significantly larger than 1, and increased with the number of input clusters. The increase in information gain with the number of input clusters is expected since a larger number of input clusters helps separate out the different overlapping community sets much more effectively.

Another observation from the results of Figures 2, 3, and 4 is that the information gain for the data in Figure 2 was much higher than the information gain in either of the other two data sets. The data in Figure 2 corresponds to the case in which the average community sizes were much smaller. In such cases, there is lower overlap among the different communities and it is possible to find more informative clusters. This also results in a higher level of information gain. In order to test the behavior with changing community size more explicitly, we varied the size of the generating community set. Specifically, we used the generating community set Cx.I(x+1).D500. The results are illustrated in Figure 5. The value of x was made to vary on the X-axis. It is clear that the information gain generally reduces with increasing community size. As in previous cases, the minimum value distance function provided more effective results than the average distance function.

We also tested the variation of the information gain with the user specified horizon. The results are illustrated in Figures 6, 7, and 8 respectively. The number of clusters were fixed at 10 in each case. The user specified horizon is plotted on the X-axis, whereas the information gain is plotted on the Y-axis. One of the interesting observations was that in each case, the information gain peaked at approximately the same value of the user specified horizon. The exact value of this user specified horizon was determined by the rate of evolution of the data stream. This rate of evolution was in turn determined by parameters such as *maxrange* and *epochsize*. In particular, the value of *epochsize* (which was fixed at 500) corresponds to the horizon in which a clear evolution of the stream can be determined. This is because within a given epoch, the base community sets do not change, but their frequency evolves over time. Correspondingly, the most distinct clusters can also be determined for this choice of the horizon. The graphs in Figures 6, 7, and 8 illustrate this trend in which the information gain peaks at approximately 500, but can also be a little different in some cases. For example, in the case of Figure 7, the clustering process does not peak at any point with the use of the average distance function. This is because of the randomness inherent in the clustering process itself. In each case, the information gain was significantly higher than the break-even value of 1. Furthermore, the behavior was quite similar over different data sets and robust over the

entire range of horizons tested. This suggest that the technique is able to find interesting and useful clusters over different values of the horizon.

While the information gain is an effective measure of the clustering process, when there are a very large number of overlapping clusters, we can get a more intuitive idea of the nature of the community clusters by using a smaller number of generating community sets with very large epoch sizes. Therefore, we used the generating community set C10.I11.D25, with $maxrange = 2$ and $epochsize = 10,000$. This effectively means that in a given epoch, 10,000 edges are generated using two overlapping communities. Then, we utilized user horizons of size 20,000 to test how the different clusters correspond to expanding or contracting communities. In order to provide a measure of an entity's interaction, we compared its level of interaction in the user specified horizon (t_h, t) to that in the previous horizon $(t - 2 \cdot h, t - \cdot h)$. For a given clustering, we separated out the nodes with positive bias from the nodes with negative or neutral bias. For each kind of node, we tested the probability whether or not its interactions in $(t - 2 \cdot h, t)$ belonged to the currently specified horizon $(t - h, t)$. The tests were performed at three different points in the stream processing. The results are illustrated in Figure 9. It is clear that a positive bias corresponds to a very high probability that nodes are present in the user specified horizon and vice-versa. This is consistent with the aim of finding expanding or contracting communities. We note that a perfect classification is not possible since in some cases, entities were present both inside and outside the horizon. Such entities could belong to either expanding or contracting communities depending upon the relative frequency in either. However, in most cases, the conditional probability of an interacting entity (not) belonging to a given horizon, given the bias bit is greater (less) than zero, is greater than 80%.Therefore, the positive and negative bias correspond to the expected phenomenon of expanding or contracting communities. It also illustrates that by using the bias bit of a cluster, it is possible to identify expanding or contracting communities in a real application.

To summarize, the following properties of the community detection algorithm were observed:

(1) The minimum value distance function provided more robust results than the average distance function. This is especially because of the more robust computation of the minimum distance function in the case of larger clusters.

(2) The clustering process was more robust when it was able to separate out the different (overlapping) clusters. Larger community sets resulted in greater level of overlap among different clusters. Therefore, the information gain was much higher when the community sets were smaller.

(3) A larger number of clusters resulted in a higher level of information gain. This is because the number of overlapping community sets were very large and increasing the number of input clusters also increased the level of information gain.

(4) The clustering process was most effective when the horizon was chosen in a way so as to match with the evolution epoch in the data generation process. Thus, the clustering process was sensitive to the level of evolution in the data stream.

(5) The bias bit could be used to identify expanding or contracting communities.

We tested the scalability of both the online stream processing component and the offline and interactive stream clustering component. The stream processing component is important in order to be able to process a high number of data points per second. In Figure 10, we have illustrated the scalability of the stream processing component. The results for the two data sets C4.I5.D500 and C10.I11.D500 are reported in terms of the number of data points processed per second. Therefore, the X-axis illustrates the progression of the data stream whereas the Y-axis illustrates the number of data points processed per second at that point in the stream progression. In each case, the initial processing rate was much higher than the steady state processing rate. This is because the initial processing needed to store very sparse graphs. Such graphs could be stored away relatively efficiently. We note that the data sets corresponding to C10.I11.D500 resulted in denser graphs than those generated from C4.I5.D500. The generation of such denser graphs also resulted in a lower processing rate per second. This is clear from the results illustrated in Figure 10. However, the difference between the two graphs is relatively minor. In each case, the data stream mining framework is able to process thousands of data points per second.

We also tested the interaction time scalability of the data stream. The scalability was tested with respect to the number of clusters as well as the value of the user-specified horizon. In Figure 11, we have illustrated the scalability of the interaction time results with respect to the number of the clusters in the data. The user-defined horizon was fixed at 5000 in this case. In each case, the results were averaged over 10 different queries. We found that the interaction time increased approximately linearly with the number of clusters in the data. The results with respect to the user-specified horizon are

illustrated in Figure 12. The number of clusters in the data were fixed at 10 in this case. In each case, the interaction time increased with the horizon, though the rate of increase was less than linear. This is because very small horizons lead to extremely sparse graphs which cluster very fast. With an increasing horizon, the differential graph grows more dense. This is because a larger number of edges can be included in a given horizon. A more dense differential graph also results in a greater amount of interaction time. However, the rate at which the interaction time increases with horizon is sublinear. This is because the denseness in the graph levels off after a certain point. This behavior also shows up in the interaction time. In each case, the interaction times turned out to be extremely small and were usually smaller than 1 or 2 seconds. This tends to indicate that the process can be efficiently used in order to perform online mining of the data streams effectively.

5 Conclusions and Summary

In this paper, we discussed a method for effective community detection of data streams. The approach in this paper finds communities in the graph data stream using an online approach in which we find the most relevant changes over a pre-defined horizon. The results show that the techniques can find the relevant communities in the data effectively even in case of considerable overlap among the different constituents. The quality of the clusters were defined in terms of an intuitive measure known as the *information gain*. The clusters in the data were shown to have a very high level of information gain compared to the breakeven behavior. The community detection approach separates out the online stream processing part from the offline community detection part which is based on user-defined parameters. Such an approach provides the maximum flexibility, since it is possible to process a high speed data stream without losing the ability to perform exploratory querying. The results show that the online processing part is very efficient and can process thousands of interactions per second. At the same time, the offline interaction component is able to process large community relationship sets in online interaction times. This provides a user with a comprehensive framework to query for changes in the community behavior in online interaction times.

References

[1] C. C. Aggarwal, *A Framework for Diagnosing Changes in Evolving Data Streams*, ACM SIGMOD Conference, (2003).

[2] C. C. Aggarwal, J. Han, J. Wang, and P. Yu, *A Framework for Clustering Evolving Data Streams*, VLDB Conference, (2003).

[3] C. C. Aggarwal, J. Han, J. Wang, and P. Yu, *On-Demand Classification of Evolving Data Streams*, ACM KDD Conference, (2004).

[4] R. Agrawal, and R. Srikant, *Fast Algorithms for Mining Association Rules*, *VLDB Conference*, (1994).

[5] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, (1992).

[6] C. Cortes, D. Pregibon, and C. Volinsky, *Communities of Interest*, Proceedings of Intelligent Data Analysis, (2001).

[7] C. Cortes, D. Pregibon, and C. Volinsky, *Computational Methods for Dynamic Graphs*, Journal of Computational and Graphical Statistics, 12, (2003), pp. 950-970.

[8] P. Domingos, and G. Hulten, *Mining High-Speed Data Streams*, ACM SIGKDD Conference, (2000).

[9] D. Gibson, J. Kleinberg, and P. Raghavan, *Inferring Web Communities from Link Topology*, Proceedings of the 9th ACM Conference on Hypertext and Hypermedia, (1998).

[10] D. Kempe, J. Kleinberg, and E. Tardos, *Maximizing the Spread of Influence Through a Social Network*, ACM KDD Conference, (2003).

[11] J. Kleinberg. *Authoritative Sources in a Hyperlinked Environment*, ACM SODA Conference, (1998).

[12] R. Kumar, J. Novak, P. Raghavan, and A. Tomkins, *On the Bursty Evolution of Blogspace*, Proceedings of the WWW Conference, (2003).

[13] G. Hulten, L. Spencer, and P. Domingos, *Mining Time Changing Data Streams*, ACM KDD Conference, (2001).

[14] S. Rajagopalan, R. Kumar, P. Raghavan, and A. Tomkins, *Trawling the Web for emerging cyber-communities*, Proceedings of the 8th WWW conference, (1999).

[15] N. Imafuji, and M. Kitsuregawa, *Finding a Web Community by Maximum Flow Algorithm with HITS Score Based Capacity*, DASFAA, (2003), pp. 101–106.

[16] M. Toyoda, and M. Kitsuregawa, *Extracting evolution of web communities from a series of web archives*, Hypertext, (2003) pp. 28–37.