JMLR: Workshop and Conference Proceedings 36:166–181, 2014 BIGMINE 2014

Frequent Subgraph Discovery in Large Attributed Streaming Graphs

Abhik Ray ABHIK.RAY@WSU.EDU
Lawrence B. Holder HOLDER@WSU.EDU
Washington State University, School of EECS, Pullman, Washington 99164-2752
Sutanay Choudhury SUTANAY.CHOUDHURY@PNNL.GOV
Pacific Northwest National Laboratory, 902 Battelle Blvd, Richland, WA 99354

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

The problem of finding frequent subgraphs in large dynamic graphs has so far only considered a dynamic graph as being represented by a series of static snapshots taken at various points in time. This representation of a dynamic graph does not lend itself well to real time processing of real world graphs like social networks or internet traffic which consist of a stream of nodes and edges. In this paper we propose an algorithm that discovers the frequent subgraphs present in a graph represented by a stream of labeled nodes and edges. Our algorithm is efficient and is easily tuned by the user to produce interesting patterns from various kinds of graph data. In our model, updates to the graph arrive in the form of batches which contain new nodes and edges. Our algorithm continuously reports the frequent subgraphs that are estimated to be found in the entire graph as each batch arrives. We evaluate our system using five large dynamic graph datasets: the Hetrec 2011 challenge data, Twitter, DBLP and two synthetic. We evaluate our approach against two popular large graph miners, i.e., SUBDUE and GERM. Our experimental results show that we can find the same frequent subgraphs as a non-incremental approach applied to snapshot graphs, and in less time.

Keywords: Dynamic graph mining, Frequent subgraph mining.

1. Introduction

One of the important unsupervised data mining tasks is finding frequent patterns in datasets. Frequent patterns are patterns that appear in the form of sets of items, subsets or substructures that have a number of distinct copies embedded in the data with frequency above a certain user-defined threshold. While frequent pattern discovery is a computationally difficult problem, it is has applications in other data mining tasks such as associative classification, clustering, cube computation and analysis, gradient mining and multi-dimensional discriminant analysis (Han et al., 2007). Frequent pattern mining also has major applications in creating indices for efficient search, mining spatiotemporal and multimedia data, mining data streams, web mining, software bug mining and system caching. There have been many extensions to the frequent pattern mining problem for datasets with uncertain data (Aggarwal et al., 2009), sequences (Wang and Han, 2004), and graph data (Kuramochi and Karypis, 2005), (Yan and Han, 2002), (Cook and Holder, 1994).

Mining graph data has become important with the proliferation of sources that produce graph data such as social networks, citation networks, civic utility networks, networks derived from movie data, and internet trace data. One of the important characteristics of these sources of real world graph data is that they are rich with information at the nodes and edges, as well as being dynamic in nature. When looked at from a graph perspective, the dynamics are manifest in three different ways:

1. Edge / Node additions: New nodes and edges are being added to the network over time.

2. Edge / Node modifications: The attributes of the nodes and edges already present in the graph are modified over time. We restrict our graphs to containing labels on the nodes and edges, i.e., we do not allow unlabeled nodes or edges.

3. Edge / Node deletions: Nodes and edges previously present are being removed from the network.

Moreover these dynamics can be represented in two different ways in a graph: (a) by representing a dynamic graph by a series of snapshots of static graphs, and (b) by representing a dynamic graph as a stream of node and edge updates to the graph. Recent trends in big data have, however, necessitated a migration to the latter form of representation for real world dynamic graph data.

In this paper we assume that our graph data has streaming updates in the form of node and edge additions. We do not consider the case where nodes and edges are also being deleted (or modified) as that would require decrements (or decrements and increments) to the counts of frequent subgraphs already discovered. This would be in involved procedure as we would have to go back over the list of frequent subgraph instances, drop the ones whose components had been deleted, and then recompute the support. We also assume the graph is attributed, i.e., nodes and edges each have a label.

We explain the scenario in which our algorithm operates as follows. We assume the graph grows in batches of updates, where each update consists of new nodes and edges that are being added to the network. We do not allow the presence of multi-edges. Our proposed algorithm estimates the frequent subgraphs present in the entire graph as each batch of updates is added to the graph. A crucial requirement for our algorithm is that it reports the frequent subgraphs present in a timely manner. The contribution of this paper is thus an algorithm that discovers frequent subgraphs in large graphs with streaming updates to the graph.

In addition to some of the general motivations behind frequent patterns mentioned above, frequent subgraphs also have the added importance of being characteristic of a certain graph. This means that we can use frequent subgraphs to monitor the health of a network as it evolves. This is important for a cyber-security analyst who is monitoring a large cyber-network and may be alerted to a potential attack due to the change in the frequent subgraphs. Frequent subgraph sets can also be used as generative models for large networks (Laxman et al., 2007).

Much of the previous work in the area of frequent subgraph mining has focused in the area of mining static single large graphs or graph transactions. Recent forays into extending frequent subgraph mining for the streaming scenario have focused on streams of

graph transactions (Bifet et al., 2011). Frequent subgraph mining is a hard problem to solve because of the involvement of graph isomorphism and subgraph isomorphism which are in NP and NP-Complete respectively. Finding FSGs (Frequent Subgraphs) in single large graphs is further complicated by the need to find the maximal number of non-overlapping instances of a candidate FSG in order to maintain the anti-monotonicity property of FSGs. We draw inspiration from the multitude of papers that have been written in the area of finding frequent subgraphs in graph transactions. One of the important properties of FSG's in graph transactions is that a subgraph is only counted as being present or absent in a particular transaction. This avoids having to find multiple occurrences of the subgraph in the transaction, and especially determine the size of the maximal set of the non-overlapping occurrences. Thus in our paper we propose a method to convert the problem of finding FSGs in a single large dynamic graph with streaming updates to a streaming graph transaction scenario. We realize that in order to identify which frequent subgraph instances are being changed, we need to only look at those regions of the graph that are being subject to change by addition of new edges and nodes. Thus we propose a method of sampling a region around the graph that has just been changed in order to form a graph transaction. With this set of graph transactions we can use any graph transaction miner to find the set of frequent subgraphs. We also keep track of the frequent subgraphs that are part of the global graph. We run experiments with two synthetic graphs and three real world graphs to evaluate our approach.

The rest of the paper is organized as follows. In the next section, we cover the related work in the area of mining dynamic graphs as well as frequent subgraph mining. In section 3 we define some of the terms related to frequent subgraph mining that we use through the rest of the paper. We also describe our problem statement. In section 4, we describe the intuition behind our proposed approach called StreamFSM, as well as provide a pseudocode version of the algorithm. In section 5.1, we discuss the datasets that we have used in this paper. Then in section 5.2, we present the evaluation and the experimental results. Finally in Section 6, we conclude the paper as well as describe future work.

2. Related Work

Previous research efforts have mostly been concentrated on developing frequent subgraph mining algorithms for static graphs. Frequent subgraph discovery algorithms can be categorized into either complete or heuristic discovery algorithms. Complete algorithms like SIGRAM (Kuramochi and Karypis, 2005) find all the frequent subgraphs that appear no less than a user specified threshold value. Heuristic algorithms like SUBDUE (Cook and Holder, 1994) and GRAMI (Elseidy et al., 2014) discover only a subset of all frequent subgraphs. The focus of the SUBDUE algorithm however was in discovering the subgraph which compressed the input graph the most. However, parameters of this algorithm could be tuned to make it output frequent subgraphs as well. SUBDUE is a heuristic discovery algorithm. SIGRAM is a complete discovery algorithm which finds frequent subgraphs from a large sparse graph.

The first frequent substructure based algorithm was designed by Inokuchi et al. (Inokuchi et al., 2000) and was inspired by the Apriori algorithm for frequent itemset mining (Agrawal and Srikant, 1994). Designed for the graph transaction scenario, it was called AGM and the

basic idea behind it was to join two size k frequent graphs to generate size (k + 1) graph candidates, and then check the frequency of these candidates separately. The algorithm FSG proposed by Kuramochi and Karypis (Kuramochi and Karypis, 2004) also used the Apriori technique. The problem with the Apriori technique is that the candidate generation takes significant time and space. Algorithms like gSpan (Yan and Han, 2002), MOFA (Borgelt and Berthold, 2002), FFSM (Huan et al., 2003), SPIN (Huan et al., 2004) and GASTON (Nijssen and Kok, 2004) were developed to avoid the overhead of the candidate generation approach. They use a pattern growth technique which attempts to grow the pattern from a single pattern directly.

Some work has also been done on subgraph mining for dynamic graphs or streaming graphs. Bifet et al. (Bifet et al., 2011) compared several sliding window approaches to mining streaming graph transactions for closed frequent subgraphs using a core set of closed frequent subgraphs as a compressed representation of all the closed frequent subgraphs discovered in the past. Aggarwal et al. (Aggarwal et al., 2010) propose two algorithms for finding dense subgraphs in large graphs with streaming updates. However they assume that the updates to the graph come in the form of edge disjoint subgraphs. Wackersreuther et al. (Wackersreuther et al., 2010) proposed a method for finding frequent subgraphs in dynamic networks, where a dynamic network is basically the union of time based snapshots of a dynamic graph. Our work is distinguished from all these works as we attempt to find subgraphs in a large graph which has streaming updates.

Berlingerio et al. (Berlingerio et al., 2009) devise a way to find graph evolution rules, which are patterns that obey certain minimum support and confidence in evolving graphs. They propose an algorithm called GERM that finds evolution rules from the frequent patterns present in the graph. However, in their approach they assume that they have the entire dynamic graph in the form of snapshots, where each snapshot represents the graph at a certain point of time. This is distinguished by our work where we do not have the entire graph all at once, and only see batches of updates to the graph. One of the challenges that they faced in their work, which was similar to ours, was that the presence of high degree nodes in real world graphs greatly increases the processing time of the transaction mining component. We take the same approach as theirs.

Frequent subgraphs have been applied in solving related problems in dynamic networks. Lahiri and Berger-Wolff (Lahiri and Berger-Wolf, 2007) apply a slightly modified concept of FSG, and use it as an interestingness criterion to extract regions of graphs from a stream of graph transactions (which represent snapshots of a dynamic graph), which they then use to predict interactions in the whole network. The definition of frequent subgraph is slightly different in their work as they constrain vertex labels to be unique. They have converted the problem of FSG into a supervised learning problem, which does not really discover new and unseen FSGs in later stages. In our scenario, however, we continuously discover frequent subgraphs as batches stream in. Unlike the approach taken by Lahiri and Berger-Wolff, our approach can detect changes to the set of frequent subgraphs in the graph. In (Zhu et al., 2011), the authors propose SpiderMine to probabilistically find the top k-large frequent subgraph patterns from a single large networks. Their focus is more on finding larger frequent subgraphs patterns. Our work on the other hand focuses more on finding frequent subgraphs in dynamic graphs, and our approach is not exclusively focused on larger patterns.

3. Technical Background

In this section we define the notation and terms related to our problem, and thus formally define the problem of finding frequent subgraphs. The batch-number refers to the batch in which an edge first appears. It is important to note that we do not allow multi-edges, even if the multi-edges have different labels. Keeping this in mind, we define the following terms:

1. **Dynamic graph**: A dynamic graph is defined as a graph $G_D = (V, E, L_V, L_E, T)$ where V is the set of vertices, E is the set of edges, L_V is the set of vertex labels, L_E is the set of edge labels, and T is the set of discrete batch numbers. For a particular batch of updates to G_D, the value of T remains the same. Therefore a batch of updates to G_D can be defined as $U_B = (V_b, E_b, L_{V_b}, L_{E_b})$, i.e., every batch of updates contains new vertices, and new edges between existing or new vertices. Thus, we can say $G_D = \bigcup U_B$.

2. **Subgraph pattern**: A subgraph pattern of a dynamic graph is a defined as $G_s = (V_s, E_s, L_{v_s}, L_{e_s})$ that is an subgraph of a dynamic graph $G_D = (V, E, L_V, L_E, T)$, where subgraph isomorphism holds if the labels match, without matching the batch-numbers T.

3. **Frequent subgraph pattern**: A subgraph pattern G_{s_f}, such that $count\left(G_{s_f}\right) \geq \alpha$, where α is a user-specified frequency threshold, in a dynamic graph G_D at time T, is said to be a frequent subgraph pattern in the graph at time T.

Problem Definition: With these definitions in mind we formally define the problem of finding frequent subgraphs in a dynamic graph with streaming updates as that of estimating all G_{s_f} for G_D after every batch of updates are applied to G_D.

4. Finding Frequent Subgraphs in Large Graphs with Streaming Updates

4.1. Complete Algorithm

Input: Set of edge updates E_{t+1}, frequency threshold α
Output: Set of frequent subgraphs
Begin
C = set of instances for all possible subgraphs of the large graph
E_{t+1} = set of edge addition updates
for *every* $e \in E_{t+1}$ **do**
  for *every* $c \in C$ **do**
    $c = c + (e \in E_{t+1}$,s.t. e is incident on vertices in $c)$, where c can be the empty set
    Add c to C
  end
  Output c_f from C where (max. number of non-overlapping instances of c_f) $\geq \alpha$
end

Algorithm 1: Complete algorithm for finding frequent subgraphs from set of edge updates

Algorithm 1 describes a complete algorithm to find the set of frequent subgraphs that are present in a large graph with the updates at time $t+1$ in E_{t+1}. In this algorithm, for every instance of every possible subgraph, incident edges in E_{t+1} are added to these subgraph instances if the edges are incident on any of the vertices in the instance. The newly created subgraph instances after the additions then form the new version of C. The frequencies of every subgraph are calculated from C using a maximum independent set computation to obtain the maximum number of non-overlapping instances of the subgraph. The frequencies that are greater than or equal to the frequency threshold α are output as frequent subgraphs.

While this approach is complete, it is also computationally inefficient due to the presence of maximum independent set counting, canonical label computation (or graph isomorphism tests) to find all instances of a subgraph, and incidence testing for an edge in E_{t+1} with all $c \in C$. In the next subsection, we describe an efficient algorithm that uses certain heuristics to reduce the computational inefficiency of the frequent subgraph mining process.

4.2. Our Approach: StreamFSM

In StreamFSM, every time a new batch of updates comes in, we first add this batch to the graph. Then using each edge as an anchor point we extract a sample of the neighborhood around this edge. Any change made by the new edge will be made in this neighborhood. However, sampling is not trivial, because no matter how large a neighborhood we sample, there may still be some of the graph one hop away that has not been sampled. And this portion of the graph could be forming an instance of a frequent subgraph with this edge. In our approach we use a rather simple sampling scheme, where we select a number of the nodes that are one hop away from each endpoint of this edge and then include all edges present between these nodes. The rationale behind this sampling approach is that any changes to the set of frequent subgraphs due to the addition of new edges will be restricted to the locality of these new edges. Selecting the nodes, however, in our approach is dependent on the domain. Domains that have a high frequency of large star-shaped patterns, where the edges in the star have similar labels, require some pruning. In other domains we may be able to select all the nodes in the immediate neighborhood of the edge. Once we have sampled a portion of that edge's neighborhood, we mark each edge in this neighborhood as 'extracted'. This will ensure that they are not extracted as part of a different edge's neighborhood, and the extracted neighborhoods remain edge disjoint. These extracted neighborhoods now become graph transactions, which are input to a graph transaction miner like gSpan (Yan and Han, 2002). This sampling scheme of course affects both the accuracy and the execution time. Selecting a higher number of neighbors increases the running time and the accuracy, while selecting a lower number of neighbors does the reverse. The graph transaction miner will find the frequent subgraphs, their frequencies as well as their canonical labels. We use the canonical labels as keys to store the subgraphs and their current frequencies in a dictionary data structure.

We repeat the procedure detailed in the above paragraph for every batch of node and edge updates to the graph. The dictionary data structure is updated with the counts of the subgraphs that are frequent for each batch of updates. Subgraphs that have been discovered previously have their counts incremented by their frequency in the set of graph transactions

Input: Set of node/edge updates U
Data: Freq. threshold f, No. of max neighbors M, Extract Bit Reset Flag X, Graph transaction frequency threshold f_t
Output: Set of frequent subgraphs per batch F_{U_i}
Begin
$G = \emptyset$ //Main graph
$C = \emptyset$ //Map of enumerated subgraphs and counts
for *every* $U_i \in U$ **do**
  $T = \emptyset$ //Used to store the set of graph transactions corresponding to U_i
  $G = G \cup U_i$
  for *every edge* $e \in U_i$ **do**
    $T_j = e$ //Used to store the extracted sample
    Mark e as *extracted*
    $E = E \cup (M$ edges incident on v_1 ($v_1 \in e$) and not already marked as *extracted*)
    $E = E \cup (M$ edges incident on v_2 ($v_2 \in e$) and not already marked as *extracted*)
    $E = E \cup$ (all edges present between all $v \in E$ in G with X not marked as *extracted*)
    $T_j = T_j \cup E$
    $T = T \cup T_j$
    Mark all edges e, s.t. $e \in G$ and e was selected to be in T_j as *extracted*
  end
  if $X ==$ *true* **then**
    reset extract bits for all edges in G
  end
  $F_T =$ Subgraphs in T that are frequent with low frequency threshold f_t found using a graph transaction miner
  Update C with the contents of F_T
  $F_{U_i} = c \in C$, s.t., $count(c) \geq f$
  Output F_{U_i}
end

Algorithm 2: StreamFSM algorithm

derived from the current batch of updates. Newly discovered subgraphs have their counts initialized in the dictionary by their currently discovered frequency. After every batch is processed the subgraphs that are frequent so far are reported.

There are two important points to note at this stage. First, in the single large graph setting the frequency of a subgraph is the number of non-overlapping instances of that subgraph in the large graph. However, in the graph transaction setting, frequency is defined as the number of transactions that contain at least one instance of that subgraph. Hence it is difficult to translate the notion of a subgraph being 'frequent' in a single large graph to that subgraph being 'frequent' in a set of transactions that only reflect certain regions of the graph. Second, certain subgraphs may become frequent over time as the graph evolves. These subgraphs would be frequent for the overall graph but not be frequent in any one set of transactions. In order to get around these issues, we use a very low frequency threshold for our graph transaction based frequent subgraph miner. While that increases our space requirements, it ensures that we do not miss subgraphs whose frequency increases gradually. We present the pseudo code of the StreamFSM algorithm in Algorithm 2.

In the following lines we explain the pseudo code of the algorithm. First, we initialize the graph G to the empty graph and C, the map of enumerated subgraphs to empty. The map of frequent subgraphs is indexed by the canonical label of the subgraph. U is a set containing batches of updates. For every batch of updates U_i in U, we first initialize the set of extracted graph transactions T to empty. We then add the current batch of updates U_i to the graph G. For every edge e in the current batch of updates U_i, we extract a region of the graph around this edge. We do this by extracting a 1-hop neighborhood around both the endpoints of e. However we restrict this neighborhood to contain only a user defined number of edges, M, around each vertex so that the resulting graph is smaller and does not contain a large star-shape pattern (as star shapes significantly increase the time required for processing of this transaction (Berlingerio et al., 2009)). After we have added these edges along with the edge present in the update we also add any edges that might be present between the vertices in this extracted region in the original graph. Once we have extracted an edge, we mark it as *extracted*, so that no edge is extracted twice. This extracted region is a small connected subgraph sample around the edge in the update. We then add this sample to the list of transactions T. After we have done the above with all the edges in the current batch of updates, we have a list of graph transactions. Depending on the value of the extract flag, the algorithm will either reset the extract bits after a batch is processed or not. We then use a frequent subgraph discovery algorithm (like gSpan) for graph transactions to find the set of frequent subgraph patterns present in the set of graph transactions with a very low frequency threshold, f_t. While the frequency threshold, f_t, for the graph transactions is a user-defined parameter, we aim to come up with an approach to automatically determine this value. This set of frequent subgraphs is stored in C, the set of candidate frequent subgraphs. We output the subgraphs in C that have frequency above the user specified frequency α. This process is repeated for every U_i, and C is updated to reflect the current counts of the subgraphs.

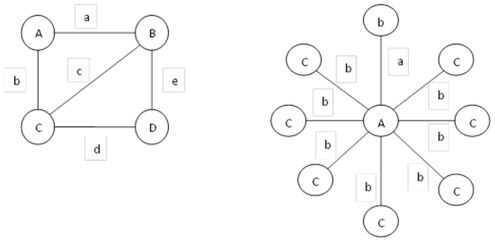

Figure 1: Known substructures embedded in (left) Artificial Graph 1, and (right) Artificial Graph 2.

5. Evaluation

5.1. Datasets

1. *Artificial Graph 1*: We created an artificial graph with 10,000 vertices and 15,000 edges where the substructure in Figure 1 (left) comprises 83% of the graph, and the rest of the edges randomly connect the instances of the substructure. The graph is then divided into 15 parts containing approximately 1000 edges each. Each part represents a batch of edge and vertex updates to the graph. The artificial subgraph generator generates the frequent subgraphs first and the connecting edges later, hence the partitioning does constrain the batches to containing entire frequent subgraphs with some subgraphs falling on the boundary of the partitions. The motivation behind this artificial graph is that it contains frequent subgraphs which are cyclical in nature. We use this to test that our algorithm can discover cyclical patterns.

2. *Artificial Graph 2*: We created an artificial graph with 10,000 vertices and 16,911 edges where the substructure in Figure 1 (right) comprises 83% of the graph, and the rest of the edges randomly connect the instances of the substructure. The graph is then divided into 17 parts containing approximately 1000 edges each. Each part represents a batch of edge and vertex updates to the graph. The partitioning has similar constraints as above. The motivation behind this artificial graph is that the dominant frequent pattern is a star-shaped pattern with very little diversity in terms of edge and node labels. Graphs which have a majority of star-shaped patterns with low label diversity make frequent subgraph discovery computationally expensive.

3. *Twitter*: The raw Twitter dataset was created by streaming Twitter data using keywords related to narcotics from January 2012 to February 2013. We created a graph by connecting users who had communicated via tweets. The graph contains a total of 45,962 vertices and 57,949 edges. The node labels are only of type 'user', and the edge type can be 'retweet', 'at', or 'mention'. This graph was created with low diversity in terms of node and/or edge labels as that makes frequent subgraph mining more computationally expensive.

4. *HETREC*: The Hetrec 2011 (Kantador, Brusilovsky and Kuflik, 2011) dataset uses the MovieLens 10M [1] and connects the movies of the MovieLens dataset with their Internet Movie Database [2] and Rotten Tomatoes website[3] pages. This brings a wealth of information into the dataset about every movie like actor names, director names, countries, genres and more. From this dataset we create a graph with only movie, actor and director information. A node in this graph can have a label of either 'movie', 'director' or 'actor'. While the labels are not unique, each node describes a unique movie, actor or director. The actual name of the node can be traced using the node identifier in a separate file which stores the movie, director and actor names. An edge from a movie to a director would have a label of 'directed-by' and an edge from a movie to an actor would have a label 'acted-by'. The entire Hetrec dataset contains data for 98 years. The resulting graph has 108,451 vertices and 241,897 edges. We divide this data up into 98 batches, where each batch represents a year. Each update to this graph comes in the form of a structure similar to Figure 1b, where the central node represents a movie. The exterior nodes contain a single node representing the 'director', and multiple nodes representing the 'actors'.

5. *DBLP*: The DBLP dataset is sourced from the DBLP citation network (DBLP) within the period of 1959 2009. We create a graph where the nodes have labels of either 'author' or one of 7,301 possible labels which indicate the publication venue. The edge labels are either 'article-author' or 'author-author' indicating links between an author and an authored article or an author and a co-author. We however only take the first 100,000 edges and related nodes. The resulting graph has 42,975 vertices and 100,000 edges.

5.2. Experiments

We evaluate our proposed approach using the following questions:

1. In a scenario where we are getting continuous batches of updates, can we report the current set of frequent subgraphs in a timely manner?
2. Is our approach capable of processing the data stream at speeds higher than the stream-rate?
3. Is our approach accurate?
4. How does our approach compare against the state-of-the-art frequent subgraph miners for large graphs?
5. How relevant are the patterns found?

In order to answer these questions, we evaluate our approach against the five datasets discussed in Section 5.1. Our experimental settings are as follows: We implement our algorithm in C++. We use two different hardware configurations in two different venues

1. http://www.grouplens.org
2. http://www.imdb.com
3. http://www.rottentomatoes.com

to expedite our experiments. We use a Linux 2.6.32 with 32 cores, AMD Opteron(TM) Processor 6272 with 1400 GHz clock speed and 64 GB memory to run experiments with the DBLP dataset. We use a dual core Intel Pentium 4 with 3.40 GHz and 3.5 GB memory for the rest of the experiments. The disparity between the two configurations is because we sought to expedite our experiments by using these two different resources at our disposal.

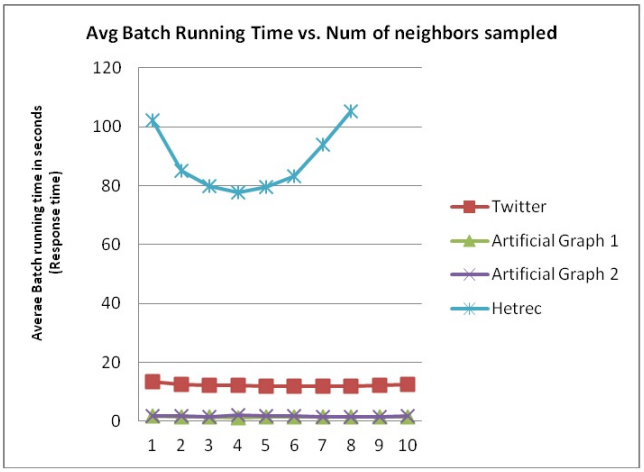

Figure 2: Average batch running time vs. Number of neighbors sampled

In order to answer the first question, we ran StreamFSM with varying values for the neighborhood sample parameter from 1-10 using four datasets. We measured the average response time per batch in seconds, where the average response time is defined as the time required to process a single batch of updates. We plot the average response time per batch against the neighborhood sample parameter in Figure 2.

From Figure 2, we can see that the average response time remains roughly the same for all the datasets except in the case of the HETREC dataset, where the running time decreases and then increases gradually. We have currently not investigated the reason behind the behavior of the HETREC dataset, but we believe that at low number of neighbor sampling, we get more transactions which increases the running time. As the number of transactions increase, we reach an optimal point with respect to the number of transactions and size of each transaction. For all the others the constant running time happens because once a particular edge is sampled, its extract bit is set and it is never sampled again. Thus the size of the sampled transactions does not grow as the graph grows keeping the average batch response times the same regardless of batch size.

In order to answer the second question, we measure the total running time for all five datasets and compare the total time versus the total time interval in which the graph data was collected. The total running time of the graphs are shown in Figures 3(left) and 3(right). In Figure 3(left) we show the total running time when the extract bit for an edge is never reset. In Figure 3(right) we show the plot for when we reset the extract bit for all the edges after every batch is processed. This ensures that while the transactions created in the same batch are edge disjoint, the transactions created across batches may have overlapping edges. Since the neighborhood sampling for a particular batch has access to the current accumulated graph to begin with, the transactions are larger. This results in a gradual

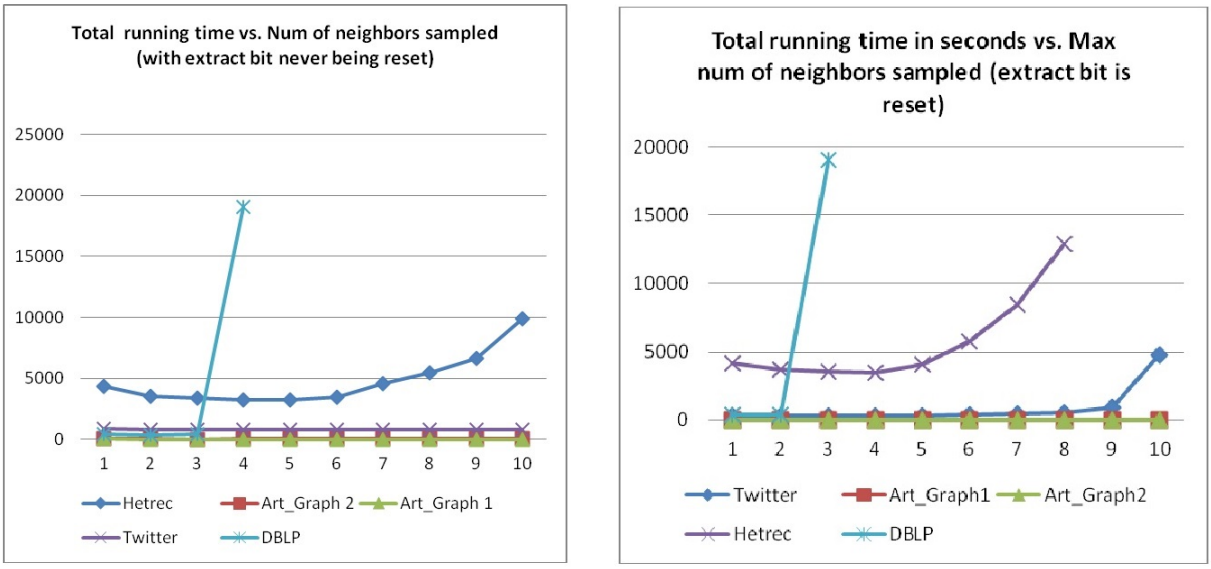

Figure 3: Total running time (in secs) vs. Number of sampled neighbors (left) extract bit not reset, (right) extract bit reset after every batch.

increase in execution time with increase in the number of sampled neighbors, as shown in Figure 3(right).

The comparison between actual times and running time can only be done for the real world datasets since the artificial graphs do not have any timing information. We perform these comparisons with the Twitter and Hetrec dataset by listing the maximum running time versus the evolution time of the network. We list these comparisons in Table 1. By the results in Table 1, we can see that the StreamFSM algorithm can indeed process the stream at rates much higher than the stream-rate. It also shows the limits of the algorithm's processing speed. Of course, if we were to encounter a Twitter stream, with a stream-rate larger than our processing ability we would have to choose between timeliness and accuracy.

Table 1: Maximum running time vs. Evolution Time

Dataset	Evolution Time	Maximum Running Time
Twitter	1 year	4,817 seconds
Hetrec	98 years	12, 939 seconds

We measure accuracy, thus answering the third question, by verifying that for the two artificial graphs the embedded patterns are discovered by the end of the stream of updates. Looking at the output of the frequent subgraph miner, the embedded subgraphs are indeed discovered as frequent subgraphs, with the frequency threshold set to 500, by the last batch of updates. For Artificial Graph 2, however, since the patterns are in the form of a star shaped structure, the number of neighbors parameter directly affects the accuracy of the frequent subgraph miner in discovering the entire star shaped graph pattern.

We also compare the average running times of our algorithm versus the running times taken by two publicly available large graph miners, GERM (Berlingerio et al., 2009) and

Table 2: Comparison with GERM and SUBDUE using Twitter

StreamFSM	GERM	SUBDUE
Finished in 4817 sec with max neighbors set to 10 and found 23 freq subgraphs.	Did not finish in more than 8 hrs found 9 freq subgraphs.	Finishes in 1305 seconds, but only gives 2 edge freq subgraphs.

SUBDUE (Cook and Holder, 1994) on the Twitter dataset. We list these in Table 2. As we see our algorithm outperforms GERM and SUBDUE on this dataset. This dataset is designed to be a disadvantage for frequent subgraph miners as it contains only one type of node label, and two types of edge labels.

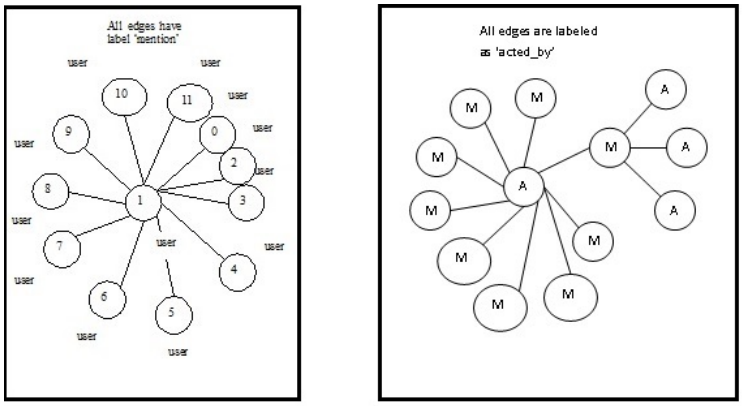

Figure 4: Frequent subgraph from (left) Twitter (right) Hetrec

We picked some of the patterns from the output of the frequent subgraph miner with the frequency threshold set to 500 that seemed to be interesting in terms of both frequency and size in order to discuss their relevance to the domains. The patterns from Twitter and HETREC are shown in Figure 4. The pattern in Figure 4 (left) can be described as follows. It contains a single user who mentions 11 other users in one or more tweets, and can thus be considered to have a link with the other users. This kind of a pattern can be considered for a social network like Twitter. Of course, this pattern is not a general characteristic of Twitter as a whole, but can definitely be considered to be a characteristic of this particular dataset. The pattern in Figure 4 (right) from HETREC, contains actors denoted by 'A', and movies denoted by 'M' connected by a link labeled 'acted_by'. The pattern shows a single actor who has acted in 9 movies, and is connected by a single movie to three other actors. As we can see, these patterns are quite relevant to the domain of the graph. Finding patterns that we expect to see is another way that we determine that our algorithm does indeed find the correct patterns.

6. Conclusions

In this paper we propose an algorithm called StreamFSM that is capable of continuously finding the current set of frequent subgraphs in a dynamic labeled graph. Our algorithm is

capable of doing this, without having to recompute all the frequent subgraphs from scratch, by only looking at the regions in the graph that have been changed due to the current batch of updates. We evaluate our algorithm on the 2 artificial graphs and 3 real world graphs and show that our algorithm is capable of processing the data streams at speeds higher than the stream rate, as well as give accurate results. We also compare our approach with the state-of-the-art in frequent subgraph miners for static graphs and show that our algorithm outperforms them in terms of interestingness of results and execution time taken together. The drawback of our algorithm is mainly in terms of several parameters that have to be tuned in order to get the optimal performance in terms of time and accuracy/interestingness of results. Also we assume that we have access to the entire graph as the graph grows. This assumption will not work in a real world streaming scenario. We plan to address these drawbacks by coming up with a automated sampling and parameter selection strategy as well as a windowing mechanism for the global graph in our future work. The datasets and code for this project will be made available at our website [4].

Acknowledgments

The authors would like to thank Dr. Courtney Corley at Pacific Northwest National Laboratory for providing the dataset from Twitter. The first author would also like to thank Dr. Medha Bhadkamkar at Symantec Research Labs for her invaluable help with Latex. This material is based upon work supported by the National Science Foundation under Grant No. 1318913.

References

Charu C. Aggarwal, Yan Li, Jianyong Wang, and Jing Wang. Frequent pattern mining with uncertain data. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 29–38, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557030. URL http://doi.acm.org/10.1145/1557019.1557030.

Charu C. Aggarwal, Yao Li, Philip S. Yu, and Ruoming Jin. On dense pattern mining in graph streams. *Proc. VLDB Endow.*, 3(1-2):975–984, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920964. URL http://dx.doi.org/10.14778/1920841.1920964.

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL http://dl.acm.org/citation.cfm?id=645920.672836.

Michele Berlingerio, Francesco Bonchi, Björn Bringmann, and Aristides Gionis. Mining graph evolution rules. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ECML PKDD '09, pages 115–

4. https://sites.google.com/site/rayabhikwsu/publications-3

130, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04179-2. doi: 10.1007/978-3-642-04180-8_25. URL http://dx.doi.org/10.1007/978-3-642-04180-8_25.

Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Mining frequent closed graphs on evolving data streams. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 591–599, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0813-7. doi: 10.1145/2020408.2020501. URL http://doi.acm.org/10.1145/2020408.2020501.

Christian Borgelt and Michael R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 51–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4. URL http://dl.acm.org/citation.cfm?id=844380.844706.

Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Int. Res.*, 1(1):231–255, February 1994. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1618595.1618605.

DBLP. Dblp. URL dblp.uni-trier.de/xml/.

Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 549–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1978-4. URL http://dl.acm.org/citation.cfm?id=951949.952101.

Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: Mining maximal frequent subgraphs from graph databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 581–586, New York, NY, USA, 2004. ACM. ISBN 1-58113-888-1. doi: 10.1145/1014052.1014123. URL http://doi.acm.org/10.1145/1014052.1014123.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '00, pages 13–23, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-41066-X. URL http://dl.acm.org/citation.cfm?id=645804.669817.

Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1038–1051, September 2004. ISSN 1041-4347. doi: 10.1109/TKDE.2004.33. URL http://dx.doi.org/10.1109/TKDE.2004.33.

Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov.*, 11(3):243–271, November 2005. ISSN 1384-5810. doi: 10.1007/s10618-005-0003-9. URL http://dx.doi.org/10.1007/s10618-005-0003-9.

M. Lahiri and T.Y. Berger-Wolf. Structure prediction in temporal networks using frequent subgraphs. In *Computational Intelligence and Data Mining, 2007. CIDM 2007. IEEE Symposium on*, pages 35–42, March 2007. doi: 10.1109/CIDM.2007.368850.

S. Laxman, P. Naldurg, R. Sripada, and R. Venkatesan. Connections between mining frequent itemsets and learning generative models. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 571–576, Oct 2007. doi: 10.1109/ICDM.2007.83.

Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 647–652, New York, NY, USA, 2004. ACM. ISBN 1-58113-888-1. doi: 10.1145/1014052.1014134. URL http://doi.acm.org/10.1145/1014052.1014134.

Bianca Wackersreuther, Peter Wackersreuther, Annahita Oswald, Christian Böhm, and Karsten M. Borgwardt. Frequent subgraph discovery in dynamic networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 155–162, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0214-2. doi: 10.1145/1830252.1830272. URL http://doi.acm.org/10.1145/1830252.1830272.

Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 79–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2065-0. URL http://dl.acm.org/citation.cfm?id=977401.978142.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4. URL http://dl.acm.org/citation.cfm?id=844380.844811.

Feida Zhu, Qiang Qu, David Lo, Xifeng Yan, Jiawei Han, and Philip S. Yu. Mining top-k large structural patterns in a massive network. *PVLDB*, 4(11):807–818, 2011.