Intelligent Data Analysis 23 (2019) 103–132
DOI 10.3233/IDA-173705
IOS Press

Efficient frequent subgraph mining on large streaming graphs

Abhik Ray[a,*], Lawrence B. Holder[a] and Albert Bifet[b]
[a]*School of EECS, Washington State University, Pullman, Washington, WA, USA*
[b]*LTCI, Télécom ParisTech, Paris 75013, France*

Abstract. We propose an efficient, approximate algorithm to solve the problem of finding frequent subgraphs in large streaming graphs. The graph stream is treated as batches of labeled nodes and edges. Our proposed algorithm finds the set of frequent subgraphs as the graph evolves after each batch. The computational complexity is bounded to linear limits by looking only at the changes made by the most recent batch, and the historical set of frequent subgraphs. As a part of our approach, we also propose a novel sampling algorithm that samples regions of the graph that have been changed by the most recent update to the graph. The performance of the proposed approach is evaluated using five large graph datasets, and our approach is shown to be faster than the state of the art large graph miners while maintaining their accuracy. We also compare our sampling algorithm against a well known sampling algorithm for network motif mining, and show that our sampling algorithm is faster, and capable of discovering more types of patterns. We provide theoretical guarantees of our algorithm's accuracy using the well known Chernoff bounds, as well as an analysis of the computational complexity of our approach.

Keywords: Streaming graphs, frequent subgraph mining, chernoff bounds, graph sampling

1. Introduction

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, it has now become important to design unsupervised learning algorithms for real world graph data. Real world graph data is dynamic, sometimes evolving at very high speeds, and is rich with information at the nodes and edges. Unsupervised pattern discovery algorithms allow the discovery of important structural motifs in the data. In this paper, we address the problem of finding subgraph patterns with frequency above a user-defined threshold (formally called the frequent subgraph discovery problem).

Frequent subgraph mining has many applications. Frequent subgraph patterns define normative substructures in a graph, thereby allowing summarization of the graph. A graph can be modeled as a distribution on its component frequent subgraphs. Frequent subgraph sets can then be used as generative models for large networks [20]. Frequent subgraphs can be used to compress graphs by computing a frequency based compression metric and replacing the subgraphs with the highest values by a single node. We can use frequent subgraphs to monitor the vulnerability of a cyber-network by inspecting the change in the set of frequent subgraphs over time.

[*]Corresponding author: Abhik Ray, School of EECS, Washington State University, Pullman, Washington, WA, USA. E-mail: abhik.ray@wsu.edu.

While useful, the problem of finding frequent subgraphs computationally lies in the space of NP-Hard. This is because a computationally complete solution of frequent subgraph mining for a single large graph requires solving the maximum independent set counting problem which is NP-Hard. While this can still be tractable for static graphs, it quickly becomes intractable for dynamic graphs, especially, if we model the graph's evolution as a series of static snapshots. Given that most real world graphs densify over time [21], we assume that our graph data has streaming updates in the form of node and edge additions. Also, since most real world graphs have data associated with the nodes and edges, we allow the nodes and edges to be labeled.

We explain the scenario in which our system 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. As each batch of updates is added to the graph, our approach first samples regions of the graph being changed, and then estimates the frequent subgraphs present in the entire graph as each batch of updates is added to the graph. A crucial requirement for our approach is that it reports the frequent subgraphs present in a timely manner. The contribution of this work is an approach that samples regions of change, and discovers frequent subgraphs in large graphs with streaming updates to the graph. The sampling approach thus makes the subgraph isomorphism problem more tractable.

The rest of this paper is organized as follows. In the next section, we discuss the related work. Then in the following section, 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. After that, we describe the intuition behind our proposed approach called StreamFSM, as well as provide a pseudocode version of the algorithm. We then discuss the datasets that we have used to evaluate our approach. Then we present the evaluation and the experimental results. Finally we summarize the paper and discuss our 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 [18] find all the frequent subgraphs that appear no less than a user specified threshold value. Heuristic algorithms like SUBDUE [10] discover only a subset of all frequent subgraphs. The first work done on discovering frequent subgraphs in single large graphs was done by Cook and Holder [10] and the algorithm was called SUBDUE. The focus of the 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 developed by Kuramochi and Karypis [17] 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. [14] and was inspired by the Apriori algorithm for frequent itemset mining [32]. 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 [17] also used the Apriori technique. The problem with the Apriori technique is that the candidate generation takes significant time and space. Algorithms like gSpan [31], MOFA [7], FFSM [12], SPIN [13] and GASTON [25] 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. [6] 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. [1] 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. [27] 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. [5] 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 Berlingerio et al. [5] face, 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. They use a user-defined parameter to restrict the number of edges in a pattern. We take a similar approach to theirs: restricting the proportion of the neighborhood that is sampled by using a user-defined parameter to restrict the degree of the nodes.

Frequent subgraphs have been applied in solving related problems in dynamic networks. Lahiri and Berger-Wolff [19] apply a slightly modified concept of FSG, and use it as an interestingness criteria 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 this paper 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 [34], Zhu et al. 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 finds both large and small patterns depending on the parameters in the sampling phase. More recently, Braun et al. [8] propose a novel data structure called DS-Matrix, that allows computation of frequent patterns in a dense graph stream in a single pass. Their approach, however, requires that the possible edges and nodes be known before-hand, and cannot handle an infinite stream of nodes and edges. Our approach, unlike theirs, is an online approach which can handle an infinite stream.

Kashtan et al. [16] propose a sampling method that randomly samples n-nodes in order to extract connected subgraph samples that are of order ‘n’. Their approach consists of repeatedly sampling subgraphs of order ‘n’ until the desired sample size is reached. The concentration of different subgraphs are then estimated, which is used to find motifs in the network. The StreamSampleNeighborhood sampling scheme that we use in our approach uses a snowball sampling like scheme that is better for discovering star-shaped patterns in addition to other types of patterns. Wernicke [28] modified the sampling scheme due Kashtan et al. [16] in order to correct the sampling bias. The TIES algorithm [2] also used a sampling that is similar to ours. However, the objective of TIES is to sample a subgraph that is representative of the entire graph. In contrast, our sampling scheme samples a subset of the neighborhood around an incoming edge that is representative of the neighborhood.

Several graph analytics frameworks have been developed to support parallel/distributed processing of big data graphs [4]. Kang et al. [15] introduced PEGASUS, a big graph mining system built on top of MapReduce. PEGASUS was able to compute various properties of billion-node graphs using the distributed and scalable matrix-based capabilities of the MapReduce-Hadoop platform. Malewicz et al. [24] developed Pregel, which takes a distributed, vertex-centric approach to graph processing where each vertex can pass messages to its neighboring vertices. Low et al.'s GraphLab [23] is also a vertex-centric approach, but is targeted toward shared memory architectures. GraphLab supports an update function based on a vertex and its neighbors that can update the data associated with these vertices. Xin et al.'s GraphX [29] is built on top of the Spark distributed processing platform. GraphX takes an edge-centric approach by representing graphs as a set of edges in Spark's Resilient Distributed Dataset (RDD). Finally, Yan et al. [30] developed the Blogel framework that takes a block, or subgraph, centric approach to big graph analytics, where processing involves distributed message passing between blocks. All of these frameworks support fast processing on large graphs, but tend to be optimized for finding global graph properties and require the entire graph to be loaded into the platform (i.e., no streaming).

Still, some specific approaches have been built on top of these frameworks to solve the frequent subgraph mining problem on big data graphs. Liu et al.'s MapReduce-based Pattern Finding (MRPF) method [22] and Hill et al.'s approach [11] use multiple MapReduce passes to find subgraph pattern extensions and their frequencies. Aridhi et al.'s approach [3] also utilizes MapReduce, but after first partitioning the graph, then mapping partitions to local frequent subgraphs, and then reducing these results to a set of globally frequent subgraphs. While fast and able to handle big data graphs, these approaches still require the entire graph be available. Carefully designed streaming graph processing can overcome this limitation while still scaling to large graphs.

3. Definitions

In this section we define the notation and terms related to our problem, and thus formally define the problem of finding frequent subgraphs. 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:

Vertex In a streaming graph, a vertex is defined as a pair, $v = (L_v, T_v)$, where L_v is the label of vertex v, and T_v identifies the time unit when this vertex was introduced to the graph.

Edge In a streaming graph, an edge is defined as a 4-tuple $e = (s, t, L_e, T_e)$, where s and t are the source and destination vertices for the edge, L_e is the label of the edge, and T_e identifies the time unit when this edge was introduced to the graph.

Graph A graph is defined as a 4-tuple, $G = (V, E, L_V, L_E)$ where V is the set of vertices, E is the set of edges, L_V is the set of vertex labels, and L_E is the set of edge labels.

Now that we have defined the basic graph data type we will be using, we move on to definitions for dynamic graphs that are more specific to our application, including what defines a batch of updates and how batches of updates taken together define the streaming graph.

Batch of node/edge updates A batch of node/edge updates to a streaming graph can be defined as $U_b = (V_b, E_b, L_{V_b}, L_{E_b})$, where $b \geqslant 1$. Every batch of updates U_b may contain new vertices, and new edges between existing or new vertices. In our work, the time unit T_v and T_e for a vertex and edge respectively are set to the value of b when the vertex/edge first appears. In Fig. 1 we see a graph that has evolved over 3 batches of updates U_1, U_2, and U_3. The nodes and edges showing the time when they were introduced in the graph with T_1, T_2, and T_3 corresponding to U_1, U_2, and U_3 respectively.

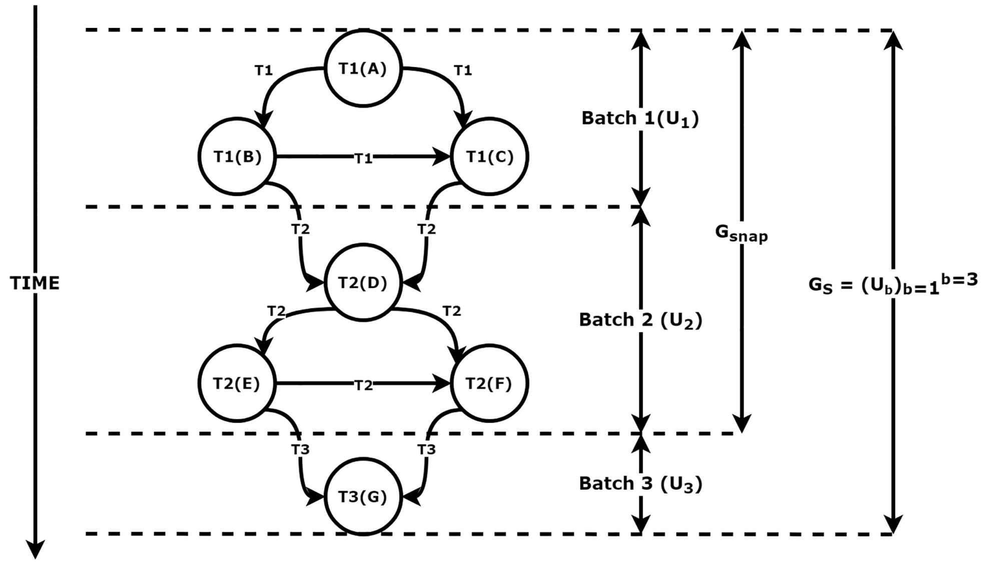


Fig. 1. Batch, graph snapshot, and graph stream.

Graph stream A graph stream G_S is defined as a sequence of batches of node/edge updates. Hence,

$$G_S = (U_b)_{b=1}^{b=n} = (V_b, E_b, L_{V_b}, L_{E_b})_{b=1}^{b=n}$$

Conceptually, every batch of updates adds new nodes and/or new edges to the graph under consideration. Our approach is designed to handle an infinitely evolving graph, hence it can theoretically accept an infinite number of batches of updates. We do not consider the situation in which nodes/edges are deleted or modified. This is because keeping track of changes to the count of a subgraph in case of deletions/modifications will require storage of every instance of a subgraph that contributes to the count. This would increase the space complexity of our approach. In Fig. 1, we see the graph stream $G_S = (U_b)_{b=1}^{b=3}$

Graph snapshot A graph snapshot G_{snap} of k batches starting with batch m, is a subsequence of k batches of updates. Hence

$$G_{snap} = (U_b)_{b=m}^{b=m+k-1} = (V_b, E_b, L_{V_b}, L_{E_b})_{b=m}^{b=m+k-1}$$

For instance, in Fig. 1, the snapshot G_{snap} of 2 batches is the subsequence $(U_b)_{b=1}^{b=2}$.

Now that we have defined graph streams and snapshots and the general structure of the graph data, we move on to defining the graph patterns that we are looking for.

Streaming subgraph pattern A subgraph pattern of a streaming graph snapshot is defined as $s_{G_{snap}} = (V_s, E_s, L_{v_s}, L_{e_s})$ that is a subgraph of a streaming graph G_{snap}, where subgraph isomorphism holds and the corresponding edge/node labels match. In Fig. 1, the vertex induced subgraphs formed by node $\{A, B, C\}$ and $\{D, E, F\}$ are isomorphic and form a subgraph pattern of G_S with frequency 2.

Frequent subgraph in a streaming graph A subgraph pattern s_{G_S} that has more than a user-specified number of non-edge-overlapping occurrences α in a graph snapshot G_{snap} in a time interval 0 to T, is said to be a frequent subgraph pattern of the streaming graph G_S at time T.

Graph canonical code A canonical code of a graph G, denoted by *Canon(G)* is an ordering of the nodes and edges of the graph such that if two graphs G_1 and G_2 are isomorphic to each other, $\mathit{Canon}(G_1) = \mathit{Canon}(G_2)$.[1]

The problem of finding frequent subgraph patterns exists in two different graph data type settings. One type is where the graph data represents a single large graph. In this scenario, frequency of a subgraph pattern is defined as the maximum number of non-overlapping instances of the pattern. It is necessary to count only the non-overlapping instances in order to maintain the downward closure property of subgraph counts. The other scenario is where the graph data represents a collection of smaller edge-disjoint graphs called graph transactions. In this scenario, the frequency of a subgraph pattern is defined as the number of graph transactions that it is present in. Multiple occurrences of a pattern are counted only once.

Set of graph transactions A set of graph transactions, or database of graphs is defined as $D_G = \{G_1, G_2, \ldots, G_n\}$ where G_i is a graph and all G_i's are edge-disjoint.

Frequent subgraphs in a set of graph transactions A frequent subgraph s_{D_G} in a set of graph transactions D_G is a subgraph appearing more than a user-defined α' number of graph transactions G_i, where $G_i \in D_G$. If s_{D_G} has more than a single embedding in any G_i, it is counted only once.

Finding frequent subgraphs in a streaming graph The problem of finding frequent subgraphs in a streaming/dynamic graph can therefore be formally defined as the problem of continuously finding the set of all frequent subgraphs $F = \{s_{G_{snap}} | \mathit{frequency}(s_{G_{snap}}) \geqslant \alpha\}$ in the graph snapshot G_{snap} after every batch of updates U_b is added to the graph G_{snap}.[2]

4. Approach

While the problem of finding frequent subgraphs in a large static graph is hard, continuously finding frequent subgraphs in graphs with streaming updates is harder. This is because the set of frequent subgraphs has to be updated as each batch of node/edge updates arrives. In most domains, such as social and cyber networks where the graphs mostly grow in size, a snapshot based approach of finding frequent subgraphs would not be efficient. This is because in a snapshot based approach we would be performing a lot of redundant computation. An incremental pattern growth based approach would be inaccurate as the frequency of the subgraphs discovered, towards the beginning of the graph's lifetime, will not be very high. Many subgraphs will gradually increase in frequency over time, and such subgraphs will be eliminated from the initial set of frequent subgraphs. Thus a complete approach would require re-computation of frequent subgraphs with every new batch of updates to the graph, and would be very inefficient.

The intuition behind our approach is that as each edge arrives in the stream of edges, it makes a change to the distribution of the frequent subgraphs only in its local neighborhood. Therefore we first propose a novel sampling algorithm for sampling the neighborhood of an incoming edge. The pseudocode for our sampling algorithm is given in Algorithm 1. This sampling is done in order to extract neighborhoods around the incoming edge, and is performed after the current batch of edges have been added to the

[1]Computing the canonical code for a graph is at least as computationally hard as computing the graph isomorphism. However, once computed, the code can be stored in memory so that recurrent isomorphism tests are reduced to string matching.

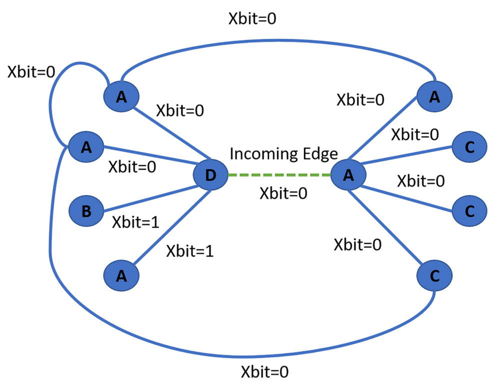
[2]Subgraph isomorphism is NP-Complete, hence our approach uses a heuristic sampling algorithm to make subgraph isomorphism more tractable.

graph. The sampling is seeded at the new edges. If a new edge e defined by (u, v) has not already been extracted, it is added to the sample. Then the sample extracts M edges each from the neighbors of u and v. After adding these edges, all edges between the nodes in the sample given by $T_{(u,v)}$ (the sampled neighbors) are added to the sample. This ensures that cyclic frequent subgraph patterns can be captured as well. The rationale behind the sampling algorithm is to thus sample a portion of an incoming edge's neighborhood that is edge-disjoint from the neighborhoods of other incoming edges.

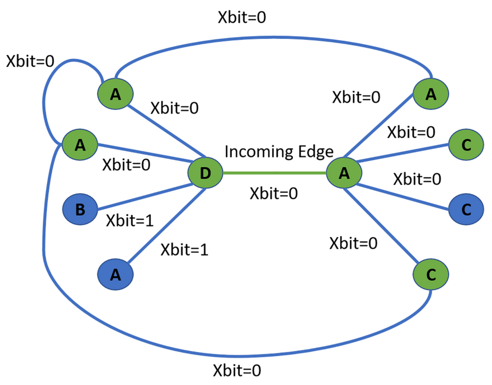
Algorithm 1: StreamSampleNeighborhood Algorithm

```
Input: Graph $G$, Edge $(u, v)$ $U$
Data: No. of max neighbors $M$
Output: Sampled neighborhood $T_{(u,v)}$ from Graph $G$
1  Begin
2  $T_{(u,v)} = \emptyset$ //Neighborhood sample
3  $counter = M$
4  for $every\ e \in neighborhood(u)\ and\ counter > 0$ do
5      if $ExtractBit(e) \neq 1$ then
6          $T_{(u,v)} = T_{(u,v)} \cup e$
7          $ExtractBit(e) = 1$
8          $counter = counter - 1$
9  $counter = M$
10 for $every\ e \in neighborhood(v)\ and\ counter > 0$ do
11     if $ExtractBit(e) \neq 1$ then
12         $T_{(u,v)} = T_{(u,v)} \cup e$
13         $ExtractBit(e) = 1$
14         $counter = counter - 1$
15 for $every\ p \in V(T_{(u,v)})$ do
16     for $every\ e \in neighborhood(p)$ do
17         if $e \notin E(T_{(u,v)})$ and ($e$ incident on $q \neq p$ where $q \in V(T_{(u,v)})$) and $ExtractBit(e) \neq 1$ then
18             $T_{(u,v)} = T_{(u,v)} \cup e$
19             $ExtractBit(e) = 1$
20 return $T_{(u,v)}$
```

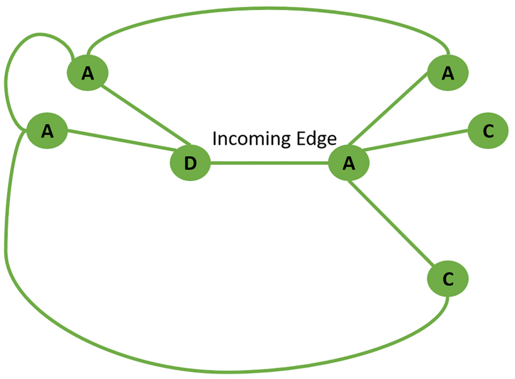
In our approach the maximum number of nodes to select, adjacent to the nodes of the incoming edge, is dependent on the domain. Domains that have a high frequency of large star-shaped patterns, where the edges in the star have similar labels, require some pruning. In other domains we may be able to select all the nodes in the immediate neighborhood of the edge. This pruning is done by tuning the variable M. M is a user-defined variable as the algorithm is domain agnostic. Domain experts may, however, use their knowledge of the domain to inform the choice of M for a particular domain. When the number of possible neighboring nodes to sample is more than M, we choose M randomly. 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. Lines 4-8 and lines 10–14 in Algorithm 1 perform this sampling



(a) New edge is added to the graph.



(b) Neighbors connected by outgoing edges with ExtractBit, *Xbit* = 0.



(c) Retain selected nodes and all edges between these nodes with *Xbit* = 0.

Fig. 2. StreamSamplingNeighborhood on sample graph with $M = 3$.

process. Figure 2a–c show the StreamSamplingNeighborhood in action. In Fig. 2a the incoming edge is added to the graph. With $M = 3$, at most 3 edges are selected with *ExtractBit* $= 0$ from both endpoints of the edge in Fig. 2b. The selection procedure in Fig. 2b selects a set of nodes. Then as Fig. 2c shows StreamSampleNeighborhood selects all edges between the set of selected nodes, and changes ExtractBit to 1 for all of them. Lines 15–19 in Algorithm 1 describe this process. The variable p iterates through the vertex set of the neighborhood sample $T_{(u,v)}$ denoted by $V(T_{(u,v)})$. The sampling algorithm selects every edge e that is incident on vertices p and q, where both $p \in V(T_{(u,v)})$ and $q \in V(T_{(u,v)})$ and $p \neq q$. The transaction created in Fig. 2c is the transaction created from this edge. This process is repeated for every unextracted edge in the current batch.

These extracted neighborhoods now become graph transactions, which are input to a graph transaction miner like gSpan [31]. 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 hash table data structure.

We repeat the procedure detailed in the above paragraph for every batch of node and edge updates to the graph. The hash table 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 derived from the current batch of updates. Newly discovered

Algorithm 2: StreamFSM algorithm

```
Input: Set of node, edge updates $U$
Data: Freq. threshold $\alpha$, No. of max neighbors $M$, Extract Bit Reset Flag $X$, Graph Transaction
      frequency $\alpha'$
Output: Set of frequent subgraphs per batch $F_{U_i}$
1  Begin
2  $G = \emptyset$ //Main graph
3  $C = \emptyset$ //Hash-map of enumerated subgraph canonical codes and counts
4  for every $U_i \in U$ do
5      $T = \emptyset$
6      $G = G \cup U_i$
7      for every edge $e \in U_i$ do
8          $T = T \cup$ StreamSampleNeighborhood(G,e,M)
9      if $X = true$ then
10         reset extract bits for all edges in G
11     $F_T$ = $\{(f, \delta) | \delta = frequency(f)$ in $T, \delta \geqslant \alpha'\}$//Subgraphs in $T$ that are frequent with low
         frequency threshold using a graph transaction miner
12     $C = C \bigcup \{(c, \delta) | c = canon(f)$ and $f \in F_T, \delta = frequency(f)\}$
13     $F_{U_i} = \{c \in C | frequency(c) \geqslant \alpha\}$
14     Output $F_{U_i}$
```

subgraphs have their counts initialized in the hash table 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 address 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 call our algorithm StreamFSM and the pseudo-code of the StreamFSM algorithm is given 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 the hash table C of enumerated subgraphs to empty. The hash table 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. While, G will eventually get too large for memory, we can use windowing mechanisms to reduce the size of 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 [5]). 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. If an edge in a batch is extracted as a part of a transaction based on another edge in the batch, then this edge is not extracted again. 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. Resetting the extract bits reduces the impact of the order of edge arrival on accuracy. 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. In Section 7, we investigate the effects of varying the frequency threshold for the graph transaction miner. This set of frequent subgraphs is stored in C, the set of candidate frequent subgraphs. Any subgraphs that have been found before have their frequency updated by their count found for the current set of transactions. In this case the union operation in Line 14 of Algorithm 2 adds δ to this count. 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. In the next section, we present a theoretical analysis on the limitations, accuracy, and runtime of our approach. We then describe the datasets that we have used to evaluate our algorithm in Section 6. Then in Section 7, we evaluate the performance of our algorithm on the discussed datasets, as well as investigate the effects of varying the values of the various parameters.

5. Analysis

Before we evaluate our algorithm empirically, we discuss certain theoretical results on the accuracy and speed of our approach. Hence, in this section we first present theorems on the structural limitations of the patterns that can be discovered. Next we derive an expression for accuracy from the Chernoff bounds, and finally we discuss the time complexity of our algorithm.

5.1. Analysis of sampling algorithm

With regards to the StreamSampleNeighborhood algorithm, we have the following theorems on the limitations of the patterns that can be discovered by our approach. We focus on finding bounds for the diameter and degree as these two metrics are representative of the size and density of a subgraph. We need these theorems to prove the structural limitations of the patterns found using the StreamFSM algorithm.

Theorem 1. The minimum possible diameter of the discovered patterns is 0.

Proof. The smallest discovered pattern can be a single vertex. Hence, the minimum diameter is 0. □

Theorem 2. The lower bound on the maximum diameter of the discovered patterns is 3.

Proof. The sampling algorithm selects the incoming edge and M edges from both end-points of the incoming edge. Hence the lower bound on the maximum diameter of the discovered patterns is 3, provided each neighbor has at least one neighbor. □

Theorem 3. The upper bound on the maximum diameter of the discovered patterns is the length of the longest Hamiltonian path on the sampled graph. Hence the maximum diameter is greater than or equal to $2 * M + 2$.

Proof. The longest Hamiltonian path covers every vertex exactly once. If a chain graph is formed from this path, the diameter of the chain will be the length of the chain. No longer path, which includes every vertex exactly once, is possible in this graph. Hence, this will be the upper bound on the maximum diameter of a pattern discovered from this sample. If the graph sample is a Hamiltonian graph, then this diameter will be $(n - 1)$, where n is the order of the sample. The value of n can be determined from the value of the parameter M by the following expression:

$$n = 2 * M + 2 \tag{1}$$

Hence the upper bound on the maximum diameter of the discovered pattern is $2 * M + 2$. □

Theorem 4. The maximum degree of a node in a subgraph pattern can be $2M + 1$.

Proof. The node that connects to every other node in the graph transaction will have a degree of $2M + 1$, which is the maximum degree possible. □

The structures that can be possibly discovered by StreamFSM are limited by the above theorems. The limitations are not too restrictive and we will be able to discover star graphs, cyclical graphs, line graphs, and chain graphs, as well as other arbitrary shapes. However, there is a restriction on the diameter of the graph and degree of the node which is due to the sampling. Also because we only sample one hop, there is a chance we will not be able to discover bushy tree patterns. If we want to discover larger shapes, then the value of M has to be set higher. This will come at a cost to run time. However, in most domains the larger graph patterns are not too frequent and hence we will not lose too much accuracy.

5.2. Accuracy bounds

We use the well known Chernoff bounds to provide accuracy bounds for the StreamFSM algorithm. There are two cases when sampling may be required. In the first case, if the batches get too large, then StreamFSM may have to drop certain edges in order to maintain reasonable processing times. Thus we have Case 1.

Case 1: When the number of edges in a batch gets too large, we have to sample the edges from the batch. We use the Chernoff bound as shown in [33] and [35] to determine how much of the edge stream we should sample. Let there be B edges in a batch, from which we sample b edges. Let the expected change in support for a frequent 1-edge subgraph due to the sampling be, $b\tau$, where τ is the support expressed as a fraction of the total number of edges in the sample. Let X be the actual change in support that was caused by sampling this batch. Then, for any positive constant, $(0 \leqslant \epsilon \leqslant 1)$, using the Chernoff bound, we have:

$$P(X \leqslant (1 - \epsilon)b\tau) \leqslant e^{(-\epsilon^2 b\tau/2)} \tag{2}$$

$$P(X \geqslant (1 + \epsilon)b\tau) \leqslant e^{(-\epsilon^2 b\tau/3)} \tag{3}$$

The Chernoff bound tells how close the actual count of an edge in the sample is to the expected count in the sample. This is defined as accuracy of the sample and is given by $(1 - \epsilon)$. This bound also gives the probability that a sample of size b will have a given accuracy. We call this the confidence of the sample, and is given by the 1 minus the expression on the right hand side of Eqs (2) and (3). The Chernoff bounds thus give us the upper and lower bounds for the confidence that the accuracy of our sample is $(1-\epsilon)$. Specifically, Eq. (2) gives the lower bound for the confidence and Eq. (3) gives the upper bound. Using Eq. (2), we get the lower bound of the confidence value as: $C = 1 - e^{(-\epsilon^2 b\tau/2)}$, where C is the confidence. Replacing $e^{(-\epsilon^2 b\tau/2)}$ with c, we have $b = -2ln(c)/(\tau\epsilon^2)$. In practice, these bounds can be too restrictive as they do not depend on the size of the available data. The other issue is that in a large graph, it is questionable whether the edges are independent of one another. In several domains, like social networks, triangle closing laws can violate this assumption. In future work we will explore the impact these factors have on the theoretical bounds.

Case 2: We extend this to the problem of determining the number of graph transactions and the size of an individual transaction to build from the graph stream in order to get an accuracy of $(1-\epsilon)$ on the count of frequent subgraphs. Let S be the size of the large graph built so far. Let us assume that we build g graph transactions of size S_g each from the graph of size S. Let X be the number of transactions that a subgraph appears in the sample, and τ be the support of the subgraph. Then, for $(0 \leqslant \epsilon \leqslant 1)$, using the Chernoff bound, we have:

$$P(X \leqslant (1-\epsilon)g\tau) \leqslant e^{(-\epsilon^2 g\tau/2)} \tag{4}$$

$$P(X \geqslant (1+\epsilon)g\tau) \leqslant e^{(-\epsilon^2 g\tau/3)} \tag{5}$$

As in Case 1 above, we get the lower bound of the confidence value from Eq. (4),

$$g = \frac{-2ln(c)}{(\tau\epsilon^2)} \tag{6}$$

where $C = 1 - c$ is the confidence, or

$$S_G = \frac{-2A_S ln(c)}{(\tau\epsilon^2)} \tag{7}$$

where S_G is the size of the total set of graph transactions in terms of edges, and A_S is the size of the individual graph transactions. We assume that inside a transaction there is only one instance of a pattern. However, since the transactions are small this is a valid assumption for the larger subgraphs. For the smaller patterns, such as single edges, we do not expect that the reduction in counts will make much difference when it comes to making a single edge frequent.

From Eq. (7), we can get the accuracy for a certain desired confidence level. The expression for that is given as,

$$\epsilon = \sqrt{\frac{-2A_S ln(c)}{\tau S_G}} \tag{8}$$

For the HETREC dataset, with $M = 5$, we get an average size of 5 for the transactions and a total of 241,648 edges in all the transactions combined. Hence we set $A_S = 5$, and $S_G = 241{,}648$. For a frequency threshold of 500, the value of τ can be computed as a fraction, by calculating the coverage of frequent subgraphs of size 5 in the HETREC graph which has a total number of 241,897 edges. τ turns out to be 0.01 (1%). We set $c = 1 - 0.9 = 0.1$ for 90% confidence. Hence by Eq. (8), we get $\epsilon = 0.09$. Therefore, accuracy $= 1 - \epsilon =$ 0.91 or 91%. As the confidence increases the accuracy will decrease. For a confidence of 95% $(c = 1 - 0.95 = 0.05)$, the accuracy is 88.87%. For a confidence value of 99% $(c = 1 - 0.99 = 0.01)$, the accuracy decreases to 86.19%. The accuracy reflects the accuracy in the count of subgraphs. When compared empirically, we see for Artificial Graph 1, StreamFSM gives the count of the embedded substructure as 2233, and the actual count of the embedded substructure is 2490. This gives us an accuracy of 89.68%. Again, as discussed in Case 1, one of the restrictions of the Chernoff bound is that the accuracy is not dependent on the size of the dataset. We also make the assumption that the sample transactions taken are independent of each other. The independence assumption might be violated if an edge is sampled as a part of more than one transaction across batches. Keeping the extract bit turned off will prevent this from happening, making the transactions more independent of each other.

5.3. Time complexity analysis

Let the number of batches be N. We first calculate the time complexity of the StreamSampleNeighborhood algorithm. For every edge $e(u,v)$ in batch U_i, StreamSampleNeighborhood has to check every outgoing edge incident on u and v separately to see if the edges have already been extracted. Once the first phase of the sampling is done, the algorithm has to check if the edges between the nodes selected in the first phase have been extracted to form the graph transaction. Hence the maximum number of comparisons is the size of the clique,

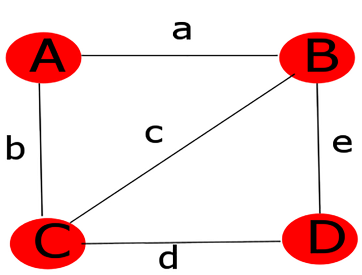
$$\textit{\#comparisons} = \frac{(d_u + d_v + 2)(d_u + d_v + 1)}{2} \tag{9}$$

where d_u and d_v are the degrees of u and v respectively. If $d \approx d_u \approx d_v$, then we have $\mathcal{O}(d^2)$. If the total number of transactions created is T, then we have $\mathcal{O}(Td^2)$. The time complexity of gSpan can be bounded by $\mathcal{O}(kFS + rF)$ [31]. Here F is the number of frequent subgraphs, S is the dataset size and is the same as T, and k is the maximum number of subgraph isomorphisms between a frequent subgraph and a sampled graph transaction. The maximum number of duplicate codes that can be generated for a frequent subgraph from other min codes[3] is given by r. For graphs with a diverse label set, $k = 1$, as the number of matches is reduced due to the label set diversity. For dense graphs with low diversity in the label sets $k > 1$. Factoring the complexity of StreamSampleNeighborhood, we have $\mathcal{O}(Td^2 + kFT + rF)$.

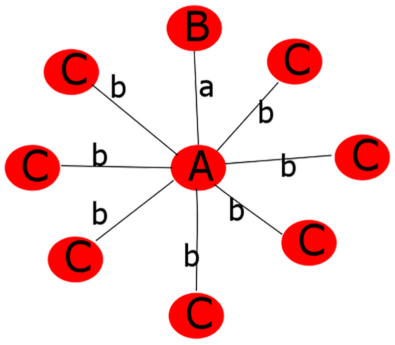
Depending on the characteristics of the graph, and the parameters set for StreamFSM, different terms in the time complexity expression would be dominant. The longer we process the graph stream, the value of T increases. For graphs with high degree hubs, the d^2 term would dominate. Depending on whether the graph sample is sparse or dense, the value of $k \approx 1$ or $k > 1$. The value of the gSpan frequency parameter will determine the value of F. The value of r can be bounded by the product of the number of nodes and edges in a frequent subgraph. The maximum number of nodes in a frequent subgraph is $(2M + 2)$, and the maximum number of edges is $\frac{(2M+2)(2M+1)}{2}$ which is the total number of edges in a clique of $(2M + 2)$ nodes. Hence r can be bounded to $\mathcal{O}(M^3)$. T can be bound to the size of the graph in edges, S, in the worst case where each transaction is an edge. Hence, we can say $T = S$. Since the size of the sampled transaction can be restricted by the value of M, we can replace d^2 with M^2. Hence we can rewrite the time complexity expression as $\mathcal{O}(SM^2 + kFS + M^3F)$. With a high label diversity ($k = 1$) we can further reduce the expression to $\mathcal{O}(SM^2 + FS + M^3F)$. Given that $S >> F$ and $S >> M^3$ for most cases, we can say that the runtime is linear with the size of the graph S. However it grows as we increase M.

There are two possible cases in terms of worst case behavior. First, considering the worst case value of $M = d_k$, where d_k is the degree of a node in a complete graph, we have $d_k^2 \approx S$. Hence in this case we have a time complexity, $\mathcal{O}(S^2 + FS + \sqrt{S}^3 F)$, that is quadratic in the size of the graph given that the S^2 term dominates the expression. In the absolute worst case, we however have exponential runtime as the value of F can be loosely bounded above by $\mathcal{O}(2^{n(n-1)/2} - 1)$, the total number of subgraphs

[3] A DFS min code is the smallest DFS code of a graph in lexicographical order.



(a)



(b)

Fig. 3. Known substructures embedded in (a) Artificial graph 1, and (b) Artificial graph 2.

possible. This happens when every possible connected subgraph is frequent. In this case, the runtime will be exponential. However, it is important to note that neither of these worst cases happen in most real world graphs. Most real world graphs are sparse and therefore far from being complete graphs. Setting the frequency threshold to 1, which is required in order to make every possible connected subgraph frequent, will also not result in useful subgraph results.

While the runtime of StreamFSM will depend on the gSpan frequency, it can also depend on the size of the batch that we are processing. Individual batches will be processed much faster than the entire graph. Hence this will allow StreamFSM to keep up with the stream-rate. While processing the graph one batch at a time, T could be bound to B (average batch size) instead of S (size of entire graph). In that case the time complexity would be $\mathcal{O}(BM^2 + FB + M^3F)$. Hence for an average batch size of B, we can handle a stream-rate which is bounded by $B/\theta(BM^2 + FB + M^3F)$. The gSpan frequency determines the value of F. Hence lower gSpan frequencies will result in higher values of F leading to longer runtimes. The size of the batch will determine the runtime. If the batch size is reduced, while the runtime will go up, accuracy might suffer as the sampling algorithm will not have access to the edges that are not part of the current batch. From our experiments in Section 7, we have seen that StreamFSM can handle a maximum of 310 edges/sec for the Twitter stream, when $M = 9$. As we have discussed in that section, if the stream exceeds this then we have to turn down the value of M in order to be able to keep up with the stream rate.

6. Datasets

This section describes the datasets that we use to evaluate the efficiency and accuracy of our approach. We use five large graph datasets. The first two are artificial graphs, and the final three are real-world datasets. The artificial graphs are created using a graph generator called Subgen that takes as input a pre-defined subgraph, the percentage coverage of the subgraph, the order and size of the desired graph, and distributions of the node and edge labels. It then creates the artificial graph with the number of instances required to meet the desired percentage coverage of the graph, and then randomly connects all the instances. The artificial graphs contain multiple occurrences of a known frequent subgraph. The size and order of the artificial graphs is chosen to ensure the correct proportion of instances of the known frequent subgraphs. We choose a large number of embeddings (83%) in order to make sure that any patterns that might be formed by the random patterns have a low frequency. The labels for the random edges are chosen from a pre-defined distribution over the nodes and edges. The pre-defined distribution mirrors the distribution of the labels in the embedded subgraph. After creating the artificial graphs, we randomize the edges of the graph in order to make sure that subsequent edges in the graph do not form instances of frequent subgraphs. We then partition the edges serially into batches as described below. The partitioning can result in breaking a frequent subgraph pattern across batches. One of the reasons for choosing a high frequency is so that the embedded subgraph pattern can have high frequency even after losing some instances to the partitioning.

1. *Artificial graph 1:* We created an artificial graph with 10,000 vertices and 15,000 edges where instances of the substructure in Fig. 3a comprise 83% of the graph, and the rest of the edges randomly connect the instances of the substructure. We choose 83% coverage so that the embedded pattern remains the dominant pattern after Subgen has added random edges. No other models of noise were added. The graph is then divided into 15 parts containing approximately 1000 edges each. The choice of 15 batches is simply to keep the number of edges close to 1000 in every batch. Each part represents a batch of edge and vertex updates to the graph.
2. *Artificial graph 2:* We created an artificial graph with 10,000 vertices and 16,911 edges where instances of the substructure in Fig. 3b comprise 83% of the graph, and the rest of the edges randomly connect the instances of the substructure. We choose 83% coverage so that the embedded pattern remains the dominant pattern after Subgen has added random edges. No other models of noise were added.The graph is then divided into 16 parts containing approximately 1000 edges each. The choice of 16 parts is simply to keep the number of edges close to 1000 in every batch. Each part represents a batch of edge and vertex updates to the graph.
3. *Twitter:* The raw Twitter[4] 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’. While this graph is comparatively small, it is complicated for frequent subgraph mining as it has very low diversity in terms of node and/or edge labels. The low label diversity makes frequent subgraph mining more computationally challenging. The original data consisting of tweets from one user to another is divided up into 67 equal batches. After removing multi-edges, the resulting streaming graph contains approximately 864 edges per batch. Again the choice of the number of batches was based on keeping close to 1000 edges per batch. A sample of the Twitter graph is shown in Fig. 4.
4. *Hetrec:* The Hetrec 2011 [9] dataset uses the MovieLens 10M[5] and connects the movies of the MovieLens dataset with their IMDB (Internet Movie Database)[6] and RottenTomatoes[7] pages. This brings a wealth of information into the dataset about every movie: actor names, director names, countries, genres, etc. 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

[4]http://www.twitter.com.
[5]http://www.grouplens.org.
[6]http://www.imdb.com.
[7]http://www.rottentomatoes.com.

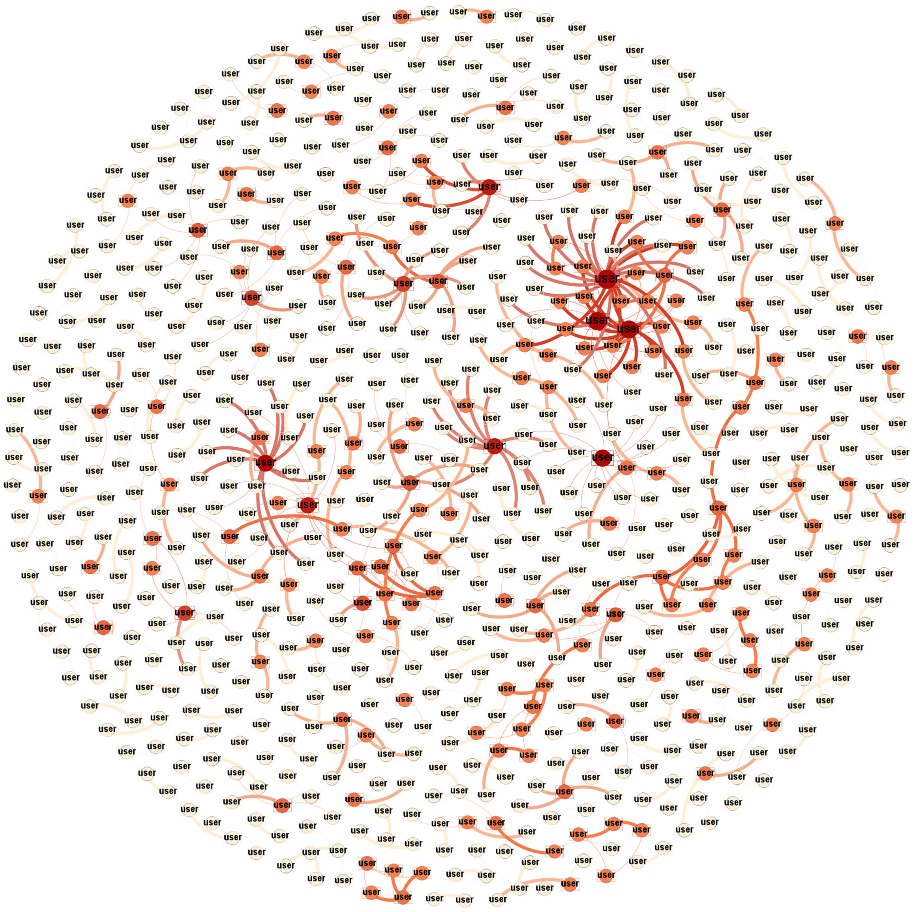


Fig. 4. Sample of Twitter graph.

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 Fig. 3b, where the central node represents a movie. The exterior nodes contain a single node representing the ‘director’, and multiple nodes representing the ‘actors’. A visualization of the Hetrec graph after the first two batches is shown in Fig. 5.

5. *ArnetMiner:* The ArnetMiner citation dataset [26] is a citation network dataset with 629,814 papers and more than 632,752 citations. We create a graph where the nodes can be either of ‘paper’, ‘author’, or ‘venue’ types. As with the HETREC dataset, the nodes represent unique papers, authors or venues. Their labels however are not unique. The edges can be either ‘paper to author’, ‘paper to venue’, and ‘paper to paper’ types. The resulting graph has 2,501,424 nodes, and 7,542,887 edges. We take the first 1.5×10^6 edges, and stream them in using 1500 batches. Each batch has 1000 edges. As with the other real world datasets, this dataset also has very low label diversity. Figure 6 shows a visualization of the the first batch of the ArnetMiner dataset. We refer to this dataset as the Arnet dataset.

The five datasets chosen allow us to evaluate our approach along several different graph types. The artificial datasets contain two different kinds of patterns, and the real world datasets contain low label diversity. A summary of the datasets in given in Table 1.

Table 1
Summary of graph datasets

Dataset	# Nodes	# Edges	# Node labels	# Edge labels
Art1	10,000	15,000	4	5
Art2	10,000	16,911	3	2
Twitter	45,962	57,949	1	3
Hetrec	108,451	241,897	3	2
ArnetMiner	739206	1,500,000	3	3

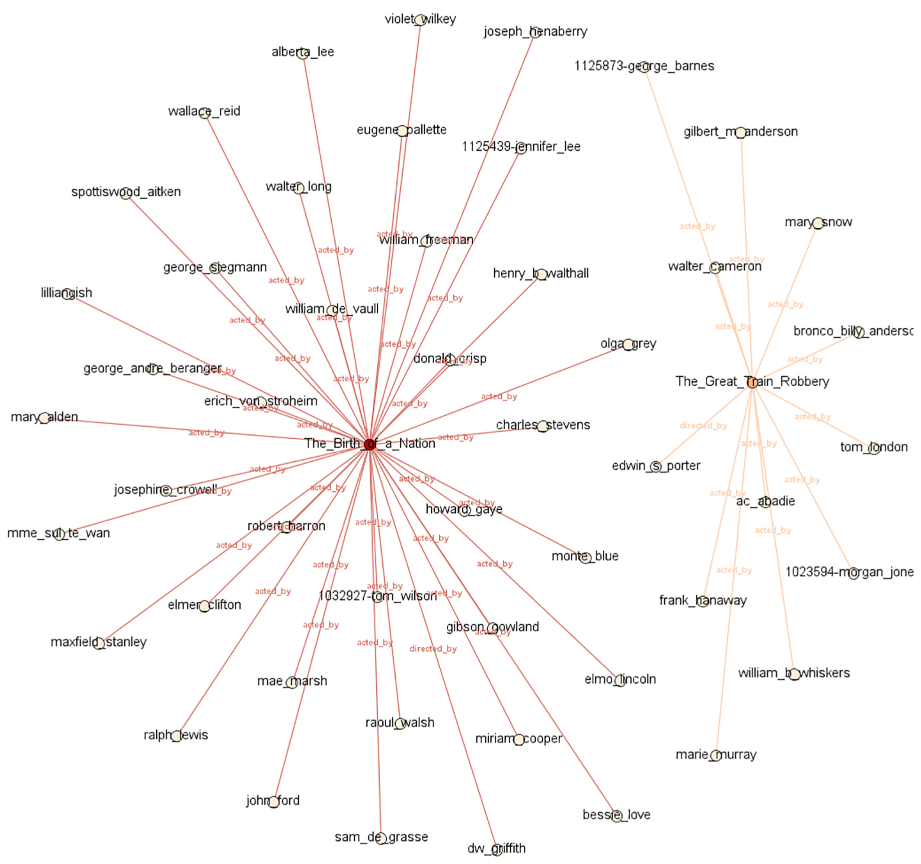


Fig. 5. Hetrec graph after first two batches with node labels showing names of actors, directors, and movies.

7. Experiments

We evaluate the StreamFSM approach based on the following questions:

1. How does the run-time vary as we vary M, the maximum number of neighbors sampled?
2. In a scenario where we are getting continuous batches of updates, can StreamFSM report the current set of frequent subgraphs in a timely manner?
3. Can StreamFSM process the data stream at speeds higher than the stream-rate?
4. Is StreamFSM accurate in terms of finding known substructures?
5. How does StreamFSM compare against the state-of-the-art large graph miners?
6. How relevant are the patterns found?

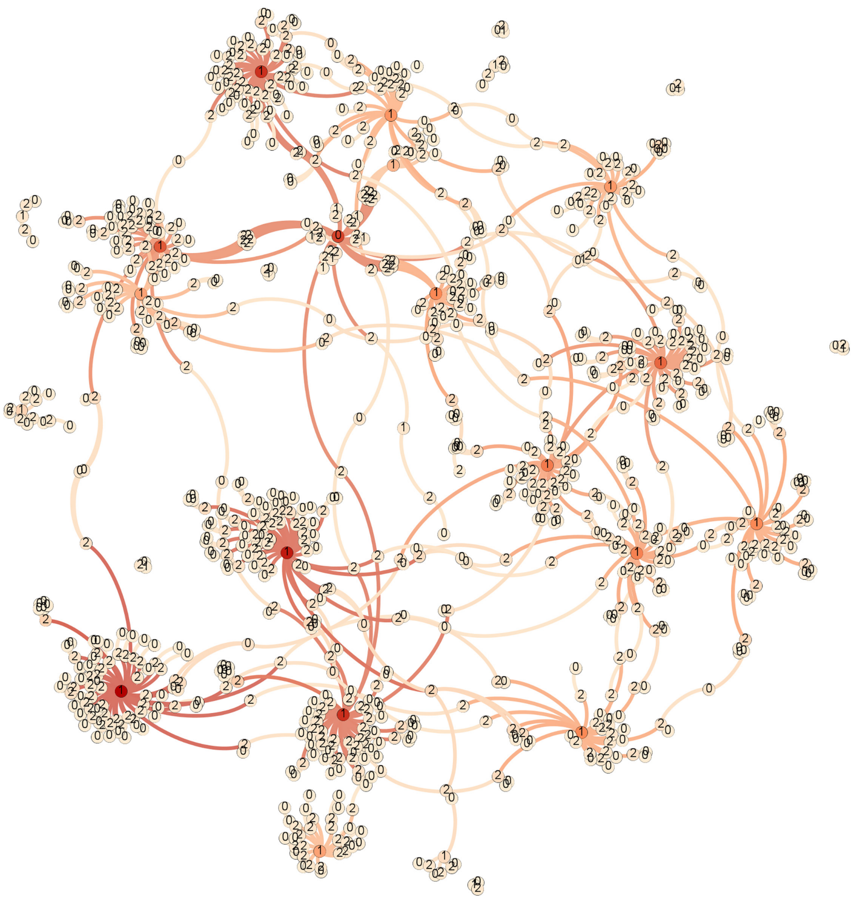


Fig. 6. Arnet graph after first batch with node labels.

7. What is the effect of varying the frequency threshold of the graph transaction miner on the run-time and accuracy?
8. How does StreamFSM perform in terms of time and memory when working on graphs that are of the scale of millions of nodes and edges?

We answer these questions using the datasets described in Section 6. Our experimental settings are as follows: We implement our algorithm and interface to gSpan in C++. We use a Ubuntu 14 system with 4th Generation Intel Core i7-4770K Processor with 3.50 GHz base frequency, 8 MB Cache, and 16 GB DDR3 SDRAM (1600 MHz) memory.

In order to evaluate the runtime as we increase M, we ran StreamFSM with varying values for the neighborhood sample parameter M from 1 to 10 using the first four datasets. We measured the total running time in seconds as the values of M were increased from 1 to 10. We ran similar experiments with both the extract bit reset feature turned on and off. Figure 7 shows the total runtime in seconds taken by the four datasets when the extract bit for every edge is reset 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 an exponential increase in execution time with increase in the number of sampled neighbors, as shown in

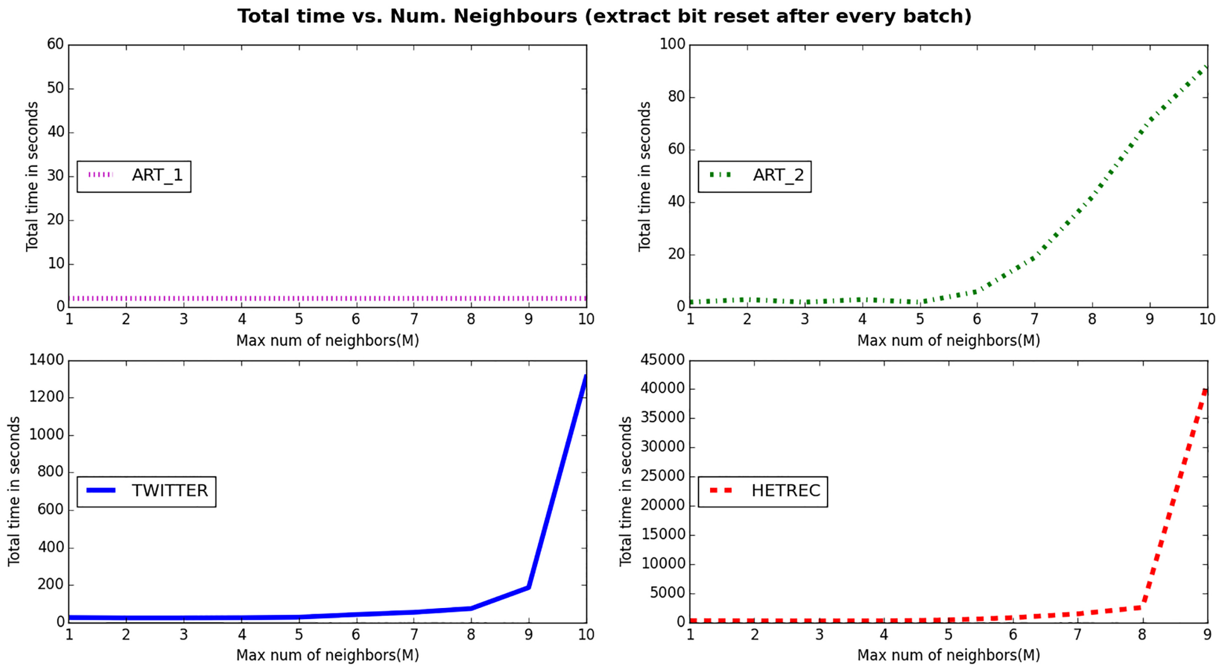


Fig. 7. Total running time (in secs) vs. number of sampled neighbors (extract bit reset after every batch).

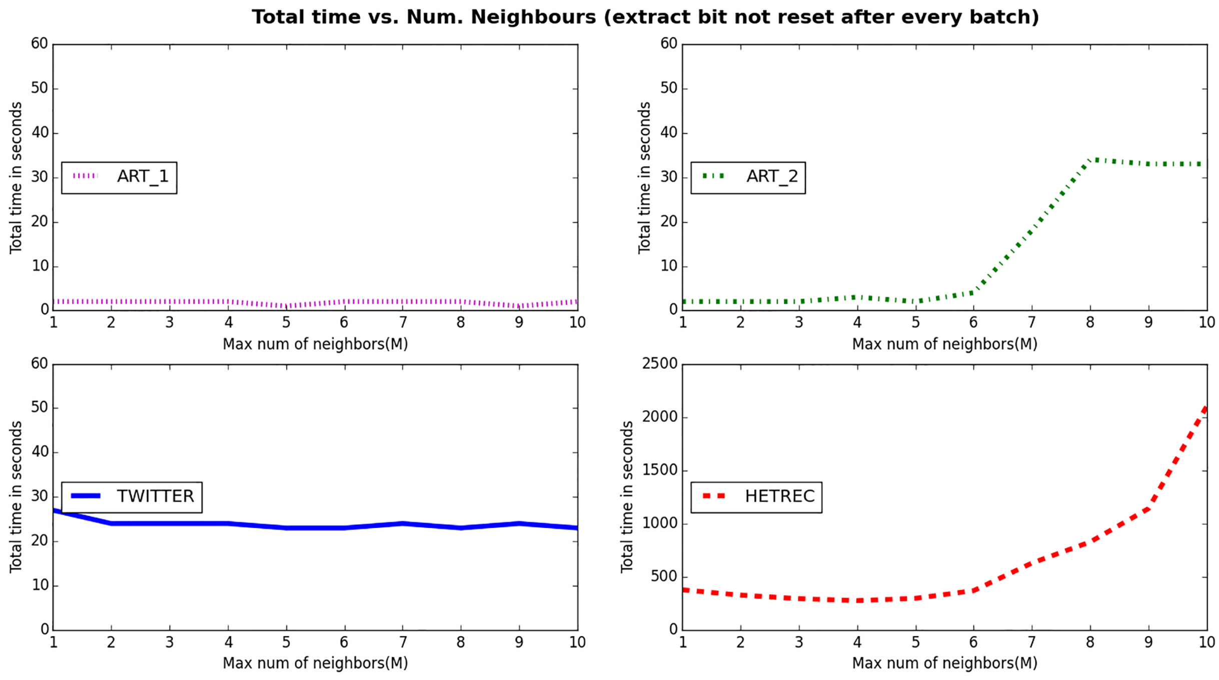


Fig. 8. Total running time (in secs) vs. number of sampled neighbors (extract bit not reset).

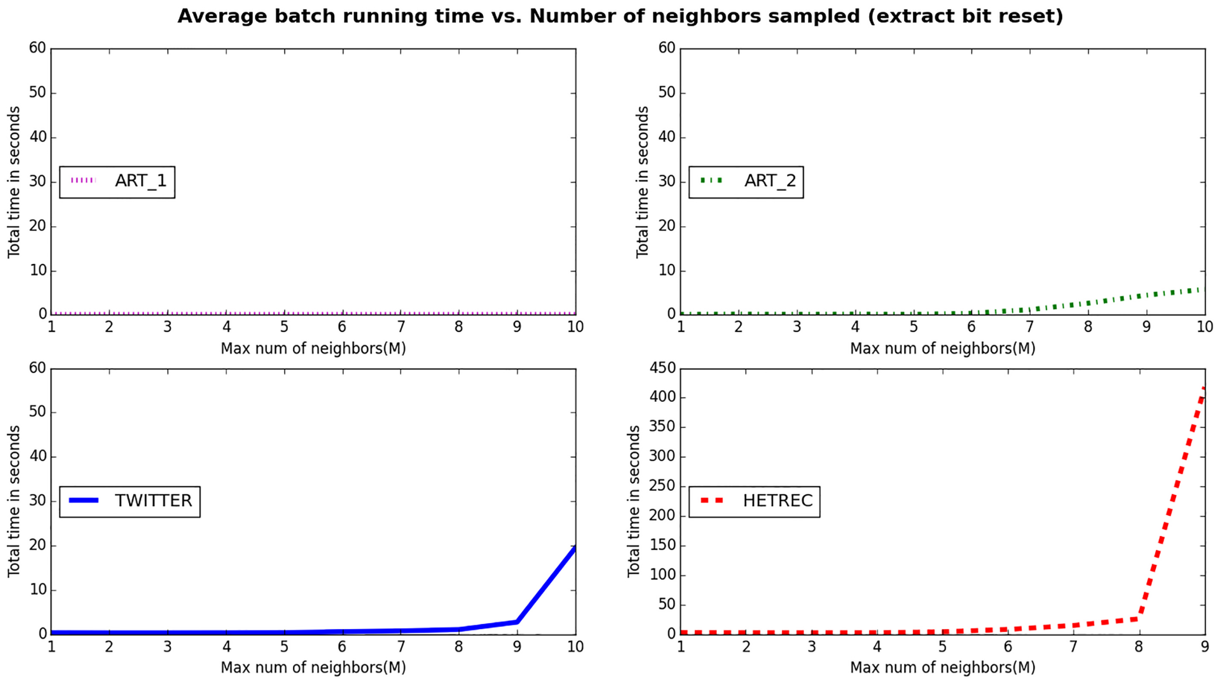


Fig. 9. Average batch running time vs. number of neighbors sampled (extract bit reset after every batch).

Fig. 7. Figure 8 shows the total runtime in seconds taken by the four datasets when the extract bit for every edge is never reset, ensuring that an edge cannot be extracted as part of a transaction more than once. As we can see this mode of the algorithm results in significant speed-up.

The second question, verifying that we can report the frequent subgraphs on time as each batch comes in, can be answered by measuring the average response time per batch in seconds, where the 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 Figs 9 and 10. From Fig. 10, 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. The reason for this is because, when M is very small the number of graph samples (or transactions) is larger, increasing the running time. As the number of transactions increase, we reach a tradeoff with respect to the number of transactions and size of each transaction, making the runtime drop. Beyond the optimal point, the samples get larger and have stars with higher degree, which again increases the runtime.

We also look at the runtime per batch as an arbitrarily long graph streams in. In order to simulate such a graph we stream the Twitter data in thrice in the form of B1, B2, . . ., B67, B1, B2, . . ., B67, B1, B2, . . ., B67. We modify the vertex numbers so that after B67 the vertex numbers in B1 are incremented by the number of vertices in the original Twitter stream. This makes the entire stream resemble one large graph. We call this new graph TwitterExtended. The parameters for StreamFSM are as follows: $M = 8$, Frequency Threshold $\alpha = 500$, and gSpan frequency 0.1. This value of M is chosen since this is the maximum with reasonable runtimes. The values of α and gSpan frequency are chosen to keep parity with the rest of the experiments. Figure 11 shows the runtime per batch. We see that in general as the graph streams in there is an overall increase as each iteration of the graph streams in.

The comparison between actual times and running time for evaluating our ability to keep up with the stream-rate can only be done for the real world datasets since the artificial graphs do not have any

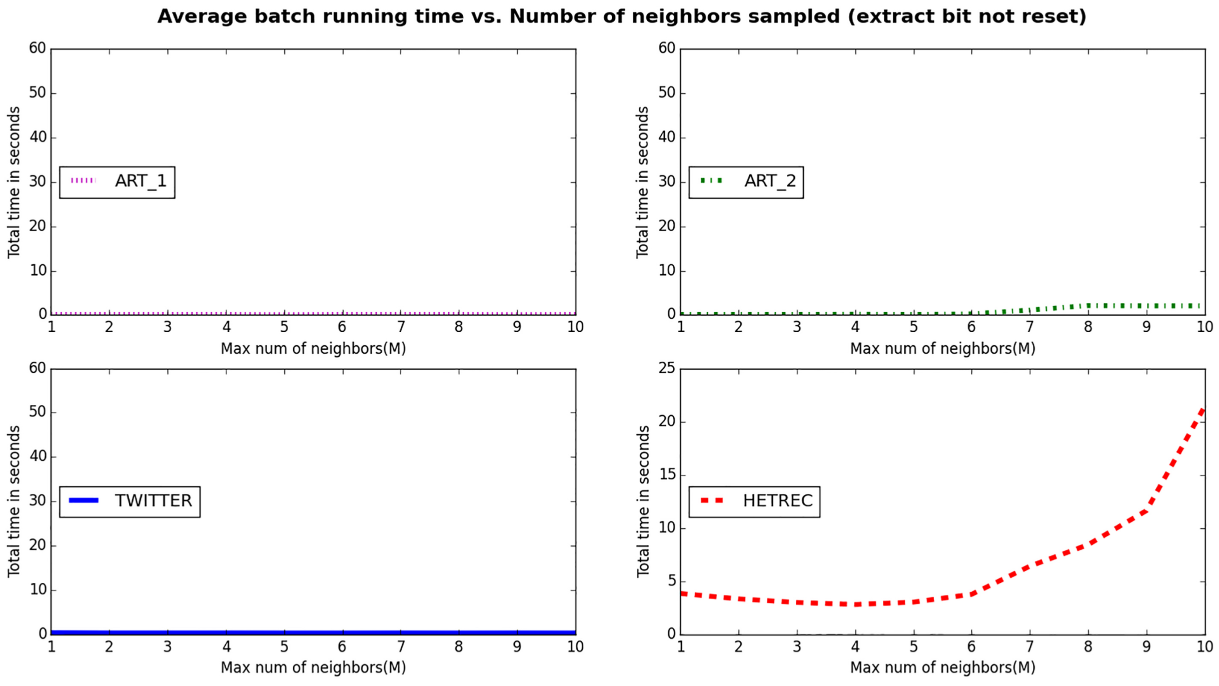


Fig. 10. Average batch running time vs. number of neighbors sampled (extract bit not reset).

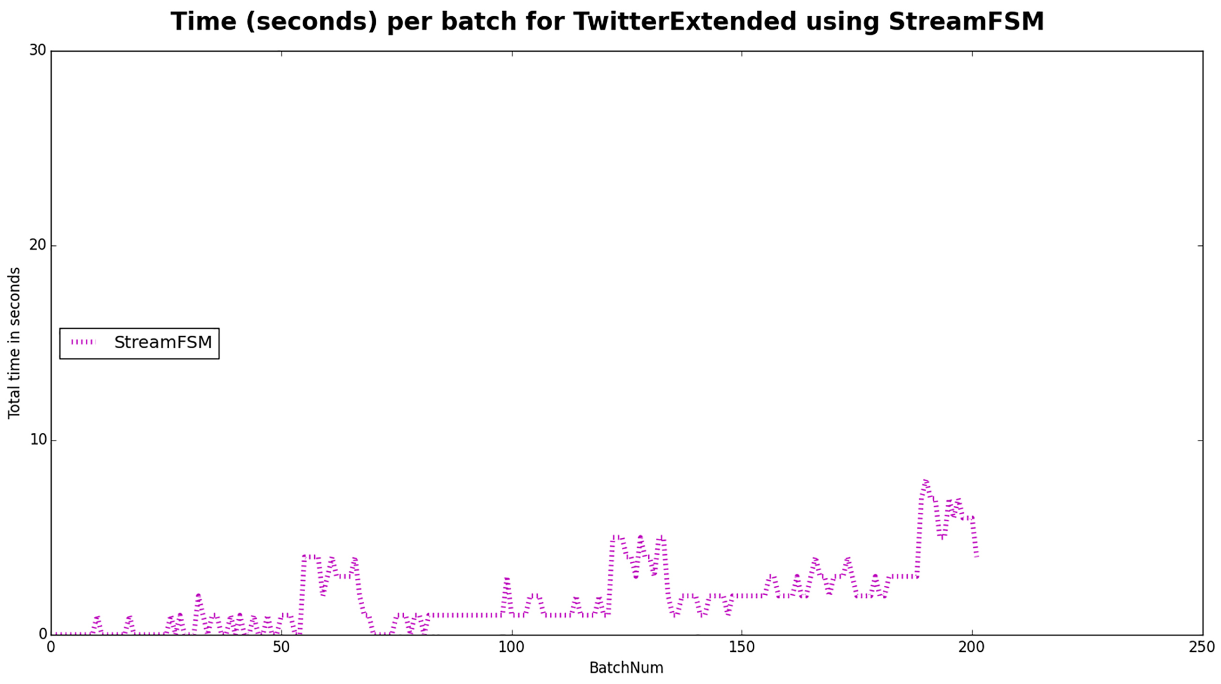


Fig. 11. Runtime per batch as TwitterExtended streams in.

Table 2
Maximum running time vs. evolution time (of graph data)

Dataset	Evolution time	Max total runtime	Actual edges/sec	Edges/sec processed
Twitter	1 year	187 seconds (9 neighbors)	0.002	310
Hetrec	98 years	40987 seconds (9 neighbors)	7.83×10^{-7}	6

Table 3
Comparison of 1-edge subgraph counts in Twitter

1 edge subgraph	Actual	StreamFSM (XBitNotReset)	StreamFSM (XBitReset)
User-retweet-user	19,492	19,485	21,689
User-at-user	4,452	688	1,719
User-mention-user	34,005	32,580	34,672

timing information. We perform these comparisons with the Twitter and Hetrec dataset by listing the total running time, when M is set to 9, versus the evolution time of the network. The evolution time of a network is the total time over which the graph data streams in. We list these comparisons in Table 2. By the results in Table 2, 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 for the two datasets. 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. When the individual batches contain more edges than we can sample, we would have to drop edges from the batch. We could use the Chernoff bounds as discussed in Section 5.2 Case 1 to do this. If the graph as a whole grows too large to fit in memory, a sliding window scheme over the graph would ameliorate the problem.

We measure accuracy, thus answering the fourth 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 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 graph. As the number of neighbors, M, affects the size of the frequent patterns, one could choose a value of M that is at least half as large as the desired frequent subgraph size. We also measure accuracy by comparing the actual counts of 1-edge subgraphs in the Twitter data. The result from this evaluation is given in Table 3. We set the value of $M = 1$ for StreamFSM, and the frequency threshold $\alpha = 500$. We measure the result when the ExtractBit is never reset (Column 3), and when the ExtractBit is reset after every batch (Column 4). Column 1 shows the 1-edge subgraph pattern in the form (VertexLabel-EdgeLabel-VertexLabel), and Column 2 shows the actual counts. We see that when the ExtractBit is never reset, StreamFSM undersamples all the 1-edge subgraph, when the ExtractBit is reset StreamFSM oversamples in some cases. We have, however, in both cases found the 1-edge subgraphs that are actually frequent.

We also compare the average running times of our algorithm versus the running times taken by two publicly available large graph miners, GERM [5] and SUBDUE [10] on the Twitter dataset. We list these in Table 4. 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. We also run SUBDUE on Artificial Graph 1 to compare the results with StreamFSM. For Artificial graph 1, SUBDUE finds 2306 instances of the embedded substructure, whereas StreamFSM (with $M = 10$, gSpan frequency 0.1) finds 2233 instances of the embedded substructure. During creation of this graph, 83% of the graph contained instances of the embedded

Table 4
Comparison with GERM and SUBDUE using Twitter

StreamFSM	GERM	SUBDUE
Finished in 1309 sec with max neighbors set to 10 and found 23 freq subgraphs.	Did not finish in more than 2.67 hrs found 9 freq subgraphs.	Finishes in 401.53 seconds, but only gives 2 edge freq subgraphs.

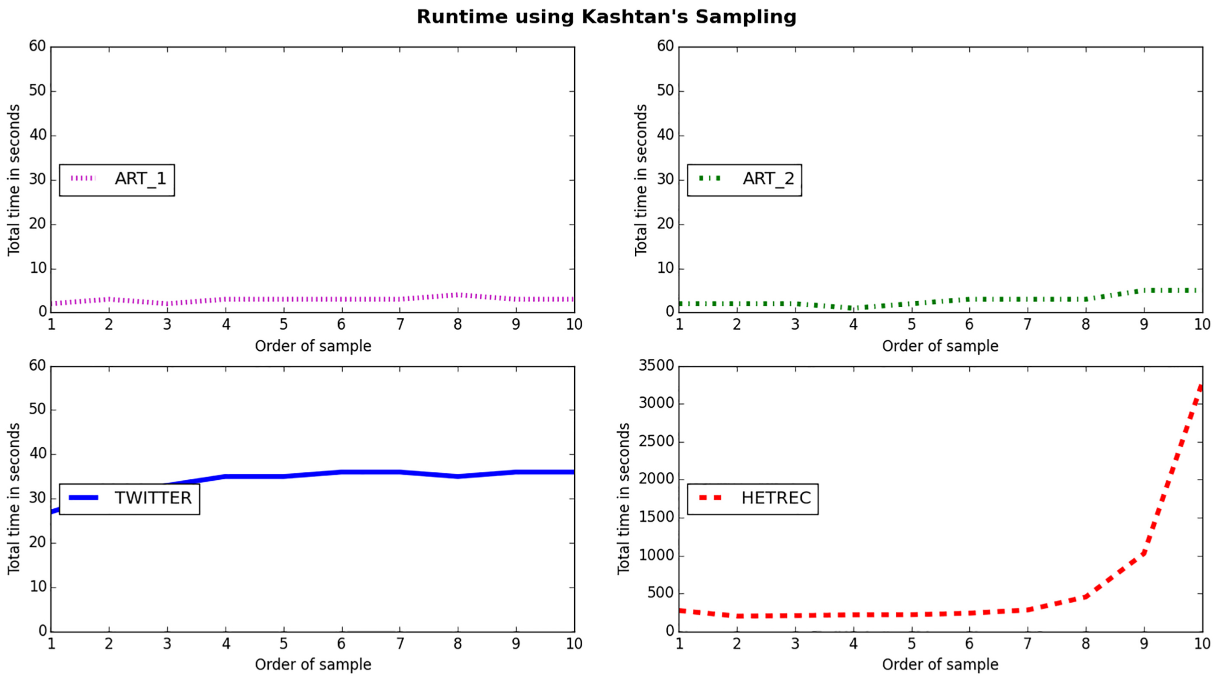


Fig. 12. Runtime while using Kashtan et al.'s sampling.

substructure resulting in 2490 instances approximately. SUBDUE reports the top 3 substructures by default, whereas StreamFSM reports 33 frequent subgraphs. For Artificial Graph 2, SUBDUE finds 1270 instances of the embedded pattern, whereas StreamFSM (with $M = 10$, gSpan frequency 0.1) finds 1171 instances of the pattern. During creation of this graph, 83% of the graph contained instances of the embedded substructure resulting in 1754 instances approximately. SUBDUE ran continuously for 5 days but did not finish it's computations. SUBDUE's result is the best substructure after those five days. StreamFSM on the other hand finds 38 substructures for Artificial Graph 2.

In order to compare the efficacy of our sampling algorithm, we compare it with the sampling scheme described by Kashtan et al. [16]. The sampling algorithm in [16] uses a random edge as seed and samples a random neighbour based region around the incoming edge. Their random neighbor sampling takes the form of a random walk with random restarts until the desired number of nodes n is achieved. After that the sample is formed from the vertex induced subgraph from the large graph. This process is repeated S_T number of times, and then the subgraph concentrations of order n subgraphs are computed. We implement that scheme as a part of StreamFSM, and use it in place of Algorithm 1. The parameter M in this case indicates the desired number of nodes in the sample. We plot the run times in Fig. 12 with the extract bit being reset after every batch. Comparing the runtime of HETREC for example in Fig. 8, we see that the sampling scheme proposed by Kashtan et al. takes more time compared to the

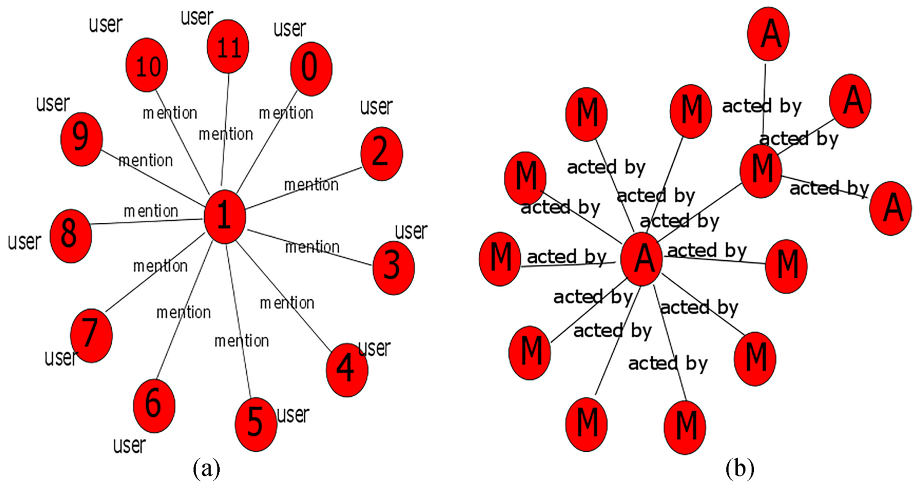


Fig. 13. Frequent subgraph from (a) Twitter (b) Hetrec.

StreamSampleNeighborhood algorithm. The accuracy of the sampling algorithm is good for Artificial Graph 1 as it finds the embedded substructure with M set to 4. However, for Artificial Graph 2 Kashtan et al.'s sampling method fails to find the embedded substructure even with M set to 10.

We picked some of the patterns from the output of the frequent subgraph miner 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 Fig. 13. The pattern in Fig. 13a 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. We find 1192 instances of this pattern. This kind of a pattern can be considered to be a characteristic pattern for a social network like Twitter. Of course, this pattern is probably not the only pattern that is characteristic of Twitter as a whole, but can definitely be considered to be a characteristic of this particular dataset, as well as one of the patterns characteristic of Twitter. The pattern in Fig. 13b 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.There are 597 instances of this pattern. One of the interesting aspects about the dataset that this pattern illuminates is that there are at least 597 actors who have worked in 9 films and collaborated with 3 other actors. As we can see, these patterns are quite relevant to the domain of the graph.

We investigate the effect of varying the gSpan (or any graph transaction miner) frequency on the runtime of the algorithm and the accuracy of the results. In general, we see that the runtime decreases as we increase the gSpan frequency, as potentially gSpan has to evaluate fewer substructures. We study the potential change in accuracy by looking at the change in the number of frequent subgraphs discovered. With the exception of the Artificial graph 1, the number of frequent subgraphs discovered decreases. This decrease would result in missing structures that might become potentially frequent. Conceptually, there is a relationship between the gSpan frequency, and the StreamFSM threshold. As the StreamFSM threshold is increased, one might be able to choose a higher gSpan frequency with less reduction in accuracy. We present the results in Figs 14 and 15.

We also use the Twitter data to investigate the number of frequent subgraphs that we get as we vary the StreamFSM frequency α from 500 to 2500 (in steps of 500) and gSpan frequency from 0.1 to 0.9 (in steps of 0.1). We show these results in Fig. 16. We see that even as the StreamFSM frequency threshold

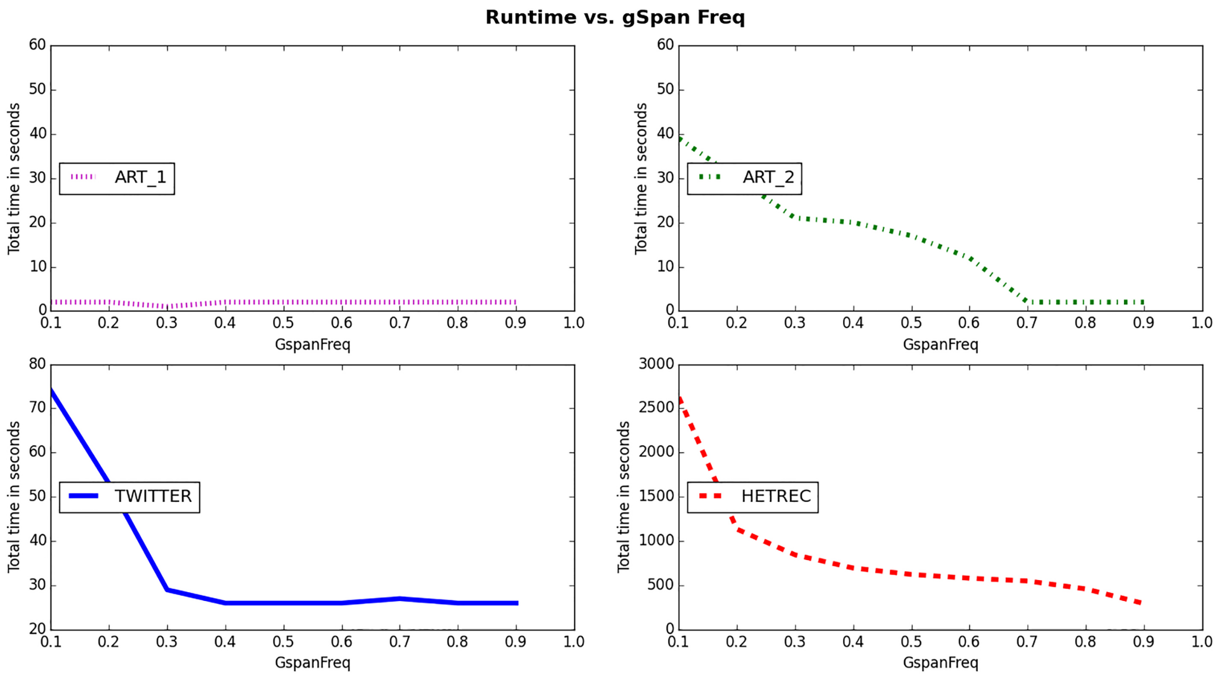


Fig. 14. Total running time vs. gSpan frequency.

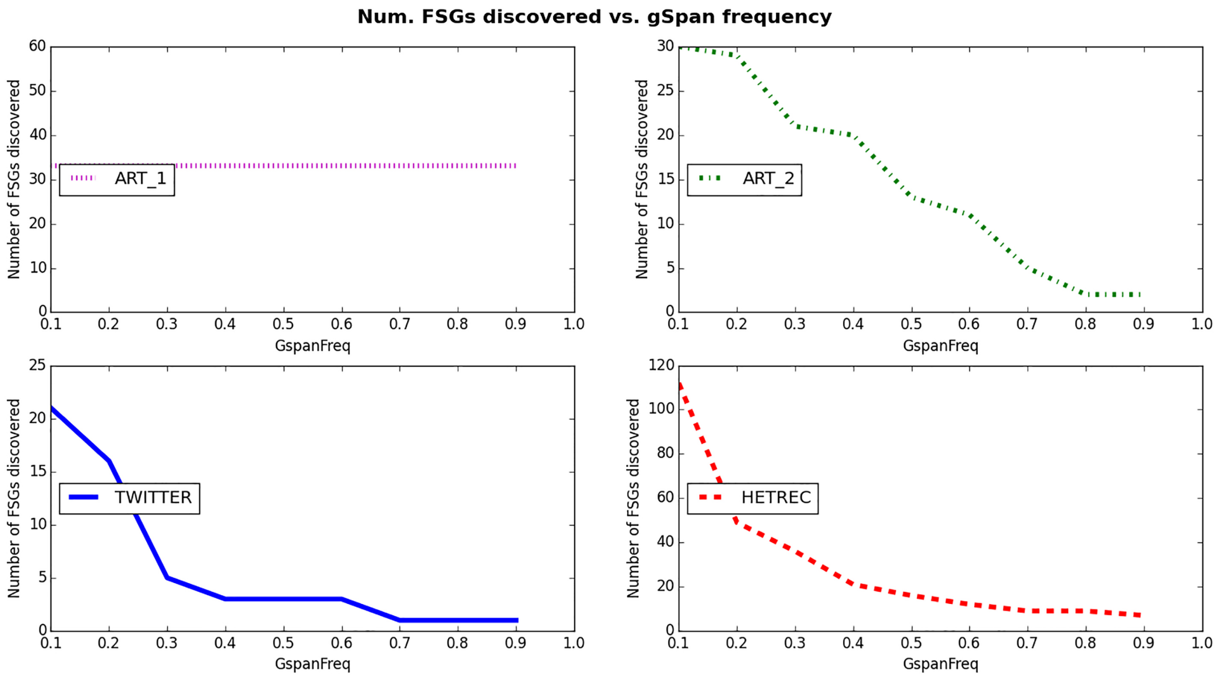


Fig. 15. Number of frequent subgraphs discovered vs. gSpan frequency.

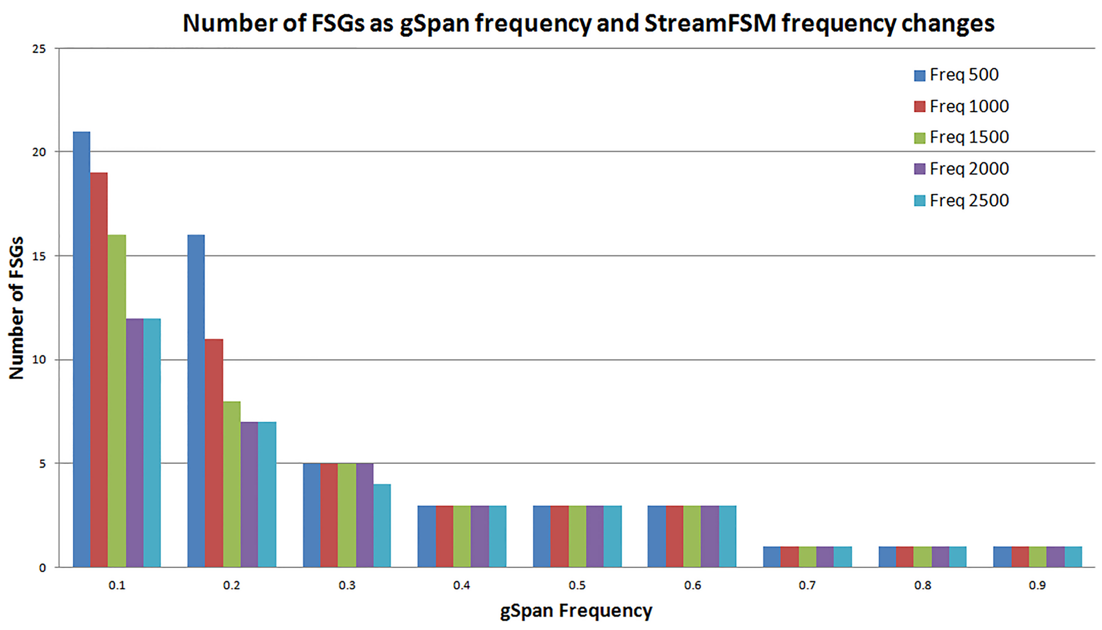


Fig. 16. Number of frequent subgraphs discovered vs. gSpan frequency as StreamFSM frequency changes.

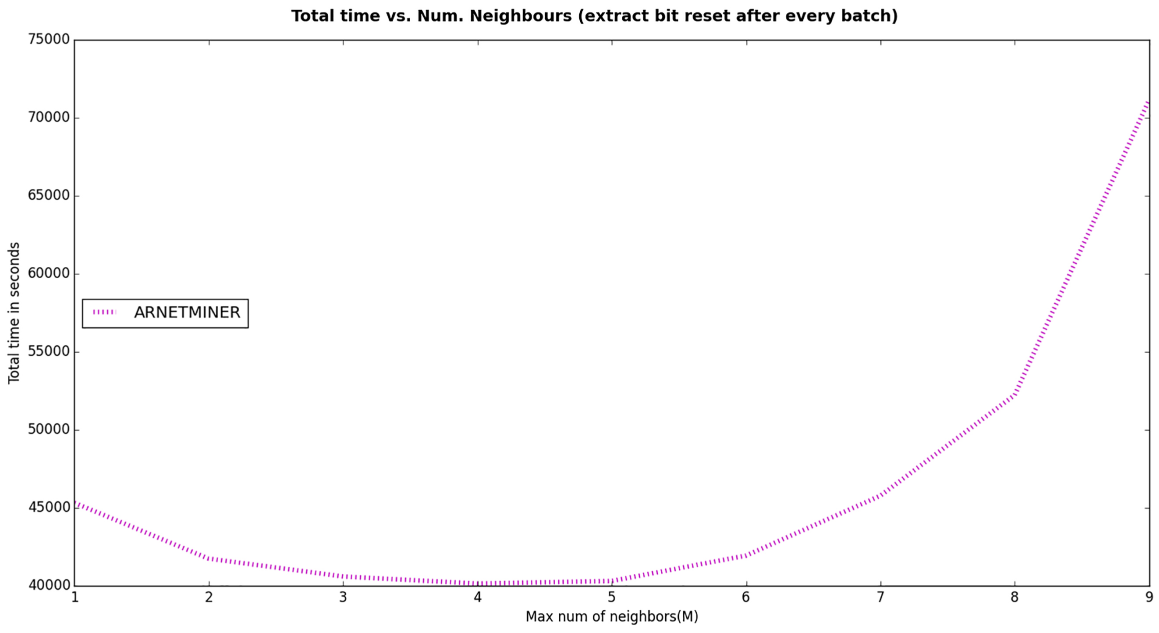


Fig. 17. Total runtime vs. number of neighbors sampled (extract bit reset after every batch).

increases, increasing the gSpan frequency results in a loss of frequent subgraphs. This reinforces what we learned from Fig. 15, i.e., tuning the gSpan parameter will always result in a trade-off between time and number of frequent subgraphs discovered.

Finally, we evaluate the time and memory required by our algorithm while processing graphs that are

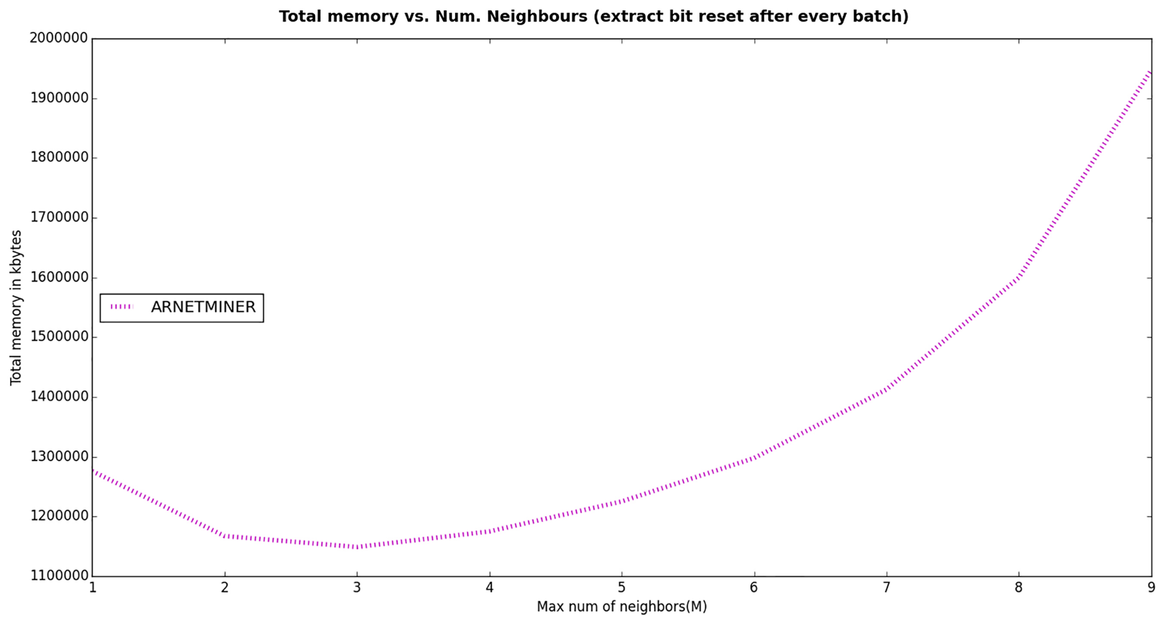


Fig. 18. Total memory vs. number of neighbors sampled (extract bit reset after every batch).

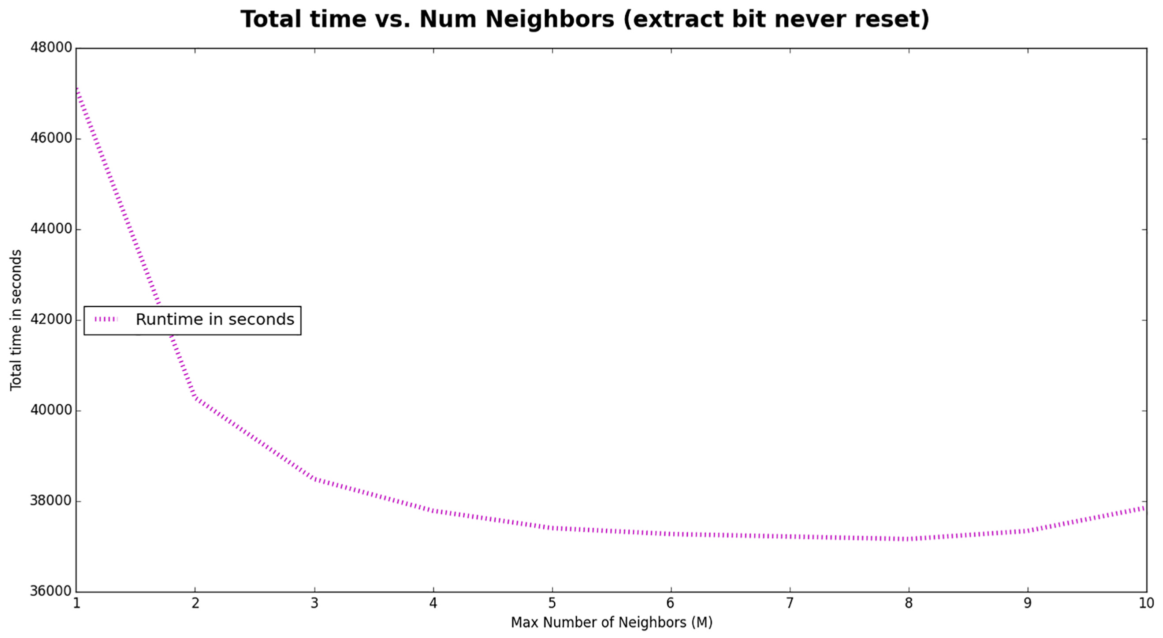


Fig. 19. Total runtime vs. number of neighbors sampled (extract bit not reset after every batch).

on the order of millions of nodes and edges. We use the ArnetMiner graph for this purpose. The results of runtime versus the value of M (maximum number of neighbors) is shown in Fig. 17. We see similar scaling of the runtime, though the magnitudes are much higher than on the previous datasets. We also plot the memory used for the ArnetMiner dataset as we increase the maximum number of neighbors as

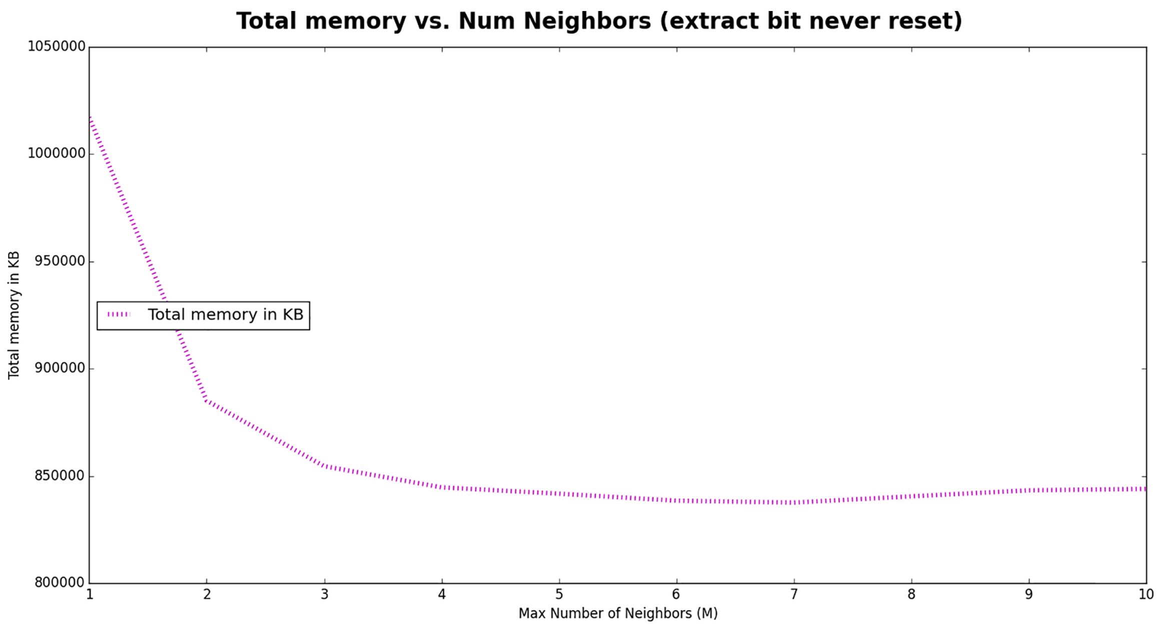


Fig. 20. Total memory vs. number of neighbors sampled (extract bit not reset after every batch).

shown in Fig. 18. The runtime and memory show an initial drop from $M = 1$ to $M = 3$ as the number of transactions created decreases as M decreases. After that, the runtime and memory start increasing exponentially due to the increase in star shaped structures in the transactions.

We also show the time and memory usage when the ExtractBit is never reset in Figs 19 and 20. We see a similar trend as when the ExtractBit is not reset, except the increase starts after $M = 8$, and after that the increase is slow.

While StreamFSM is able to process the ArnetMiner graph, it makes the assumption that the entire data can fit in memory. For an infinite stream of nodes and edges, this assumption can easily lead to the memory being filled up very quickly. It can also lead to longer runtimes. This problem can be alleviated by using a sliding window scheme on the stream of nodes and edges.

8. Summary

In this paper, we propose an algorithm called StreamFSM that is capable of continuously finding the current set of frequent subgraphs in a streaming 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 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. We also compare our sampling approach against the well known sampling approach developed by [16] and show that the runtime using this sampling algorithm is higher than our runtime, and the accuracy is worse for graphs with many star-shaped patterns. We then show that our algorithm scales similarly on very large graphs as the number of neighbors sampled increases. Finally, we analyze the algorithm and provide time complexity results as well as results on theoretical accuracy bounds using the well known Chernoff bounds.

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 our approach relies on storing the entire graph in main memory. We plan to address the first issue by developing a scheme that automatically selects and/or adapts the parameters according to the graph statistics. In order to handle the second problem, we plan to use a sliding window on the streaming graph. We are currently looking into using a fixed size sliding window and plan to extend this to adaptive sliding windows.

Acknowledgments

We would like to acknowledge Diane J. Cook and Matthew E. Taylor at Washington State University for reviewing the paper and providing many helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1646640.

References

[1] C.C. Aggarwal, Y. Li, P.S. Yu and R. Jin, On dense pattern mining in graph streams, *Proc VLDB Endow* **3**(1-2) (Sep. 2010) 975–984.

[2] N. Ahmed, J. Neville and R. Kompella, Network sampling via edge-based node selection with graph induction, 2011.

[3] S. Aridhi, L. D.'Orazio, M. Maddouri and E. Mephu Nguifo, Density-based data partitioning strategy to approximate large-scale subgraph mining, *Inf Syst* **48**(C) (Mar. 2015), 213–223.

[4] S. Aridhi and E.M. Nguifo, Big graph mining: Frameworks and techniques, *CoRR*, abs/1602.03072, 2016.

[5] M. Berlingerio, F. Bonchi, B. Bringmann and A. Gionis, Mining graph evolution rules, In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ECML PKDD '09, Berlin, Heidelberg, 2009. Springer-Verlag, pp. 115–130.

[6] A. Bifet, G. Holmes, B. Pfahringer and R. 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, New York, NY, USA, 2011, ACM, pp. 591–599.

[7] C. Borgelt and M.R. Berthold, Mining molecular fragments: Finding relevant substructures of molecules, In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, Washington, DC, USA, 2002. IEEE Computer Society, p. 51.

[8] P. Braun, J.J. Cameron, A. Cuzzocrea, F. Jiang and C.K. Leung, Effectively and Efficiently Mining Frequent Patterns from Dense Graph Streams on Disk, *Procedia Computer Science* **35** (2014), 338–347.

[9] I. Cantador, P. Brusilovsky and T. Kuflik, 2nd workshop on information heterogeneity and fusion in recommender systems (hetrec 2011). In *Proceedings of the 5th ACM conference on Recommender systems*, RecSys 2011, New York, NY, USA, ACM, 2011.

[10] D.J. Cook and L.B. Holder, Substructure discovery using minimum description length and background knowledge, *J Artif Int Res* **1**(1) (Feb. 1994), 231–255.

[11] S. Hill, B. Srichandan and R. Sunderraman, An iterative mapreduce approach to frequent subgraph mining in biological datasets, In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12, New York, NY, USA, ACM, 2012, 661–666.

[12] J. Huan, W. Wang and J. Prins, Efficient mining of frequent subgraphs in the presence of isomorphism, In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, Washington, DC, USA, 2003. IEEE Computer Society, pp. 549.

[13] J. Huan, W. Wang, J. Prins and J. 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, New York, NY, USA, ACM, 2004, pp. 581–586.

[14] A. Inokuchi, T. Washio and H. 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, London, UK, UK, 2000. Springer-Verlag, pp. 13–23.

[15] U. Kang, C.E. Tsourakakis and C. Faloutsos, Pegasus: A peta-scale graph mining system implementation and observations, In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, Washington, DC, USA, 2009. IEEE Computer Society, pp. 229–238.

[16] N. Kashtan, S. Itzkovitz, R. Milo and U. Alon, Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs, *Bioinformatics* **20**(11) (July 2004), 1746–1758.

[17] M. Kuramochi and G. Karypis, An efficient algorithm for discovering frequent subgraphs, *IEEE Trans on Knowl and Data Eng* **16**(9) (Sept. 2004), 1038–1051.

[18] M. Kuramochi and G. Karypis, Finding frequent patterns in a large sparse graph, *Data Min. Knowl. Discov.* **11**(3) (Nov. 2005), 243–271.

[19] M. Lahiri and T. Berger-Wolf, Structure prediction in temporal networks using frequent subgraphs, In *Computational Intelligence and Data Mining*, 2007. CIDM 2007. IEEE Symposium on, March 2007, pp. 35–42.

[20] 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*, Oct 2007, 571–576.

[21] J. Leskovec, J. Kleinberg and C. Faloutsos, Graph evolution: Densification and shrinking diameters, *ACM Trans Knowl Discov Data* **1**(1) (Mar. 2007).

[22] Y. Liu, X. Jiang, H. Chen, J. Ma and X. Zhang, Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network, In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, APPT '09, Berlin, Heidelberg, 2009. Springer-Verlag, pp. 341–355.

[23] Y. Low, J.E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin and J.M. Hellerstein, Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[24] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, Pregel: A system for large-scale graph processing, In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, 2010. ACM, pp. 135–146.

[25] S. Nijssen and J.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, New York, NY, USA, 2004. ACM, pp. 647–652.

[26] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang and Z. Su, Arnetminer: Extraction and mining of academic social networks, In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, New York, NY, USA, 2008. ACM, pp. 990–998.

[27] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm and K.M. Borgwardt, Frequent subgraph discovery in dynamic networks, In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, New York, NY, USA, 2010. ACM, pp. 155–162.

[28] S. Wernicke, A faster algorithm for detecting network motifs, In *Proceedings of WABI âĂŹ05, number 3692 in LNBI*, Springer-Verlag, 2005, pp. 165–177.

[29] R.S. Xin, J.E. Gonzalez, M.J. Franklin and I. Stoica, Graphx: A resilient distributed graph system on spark, In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, NY, USA, 2013. ACM, pp. 2:1–2:6.

[30] D. Yan, J. Cheng, Y. Lu and W. Ng, Blogel: A block-centric framework for distributed computation on real-world graphs, *Proc. VLDB Endow* **7**(14) (Oct. 2014), 1981–1992.

[31] X. Yan and J. Han, gspan: Graph-based substructure pattern mining, In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, Washington, DC, USA, 2002. IEEE Computer Society, p. 721.

[32] C.H. You, L.B. Holder and D.J. Cook, Learning patterns in the dynamics of biological networks, In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, New York, NY, USA, 2009. ACM, pp. 977–986.

[33] M. Zaki, S. Parthasarathy, W. Li and M. Ogihara, Evaluation of sampling for data mining of association rules, In *Research Issues in Data Engineering*, 1997. Proceedings Seventh International Workshop on, Apr 1997, 42–50.

[34] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han and P.S. Yu, Mining top-k large structural patterns in a massive network, *PVLDB* **4**(11) (2011), 807–818.

[35] R. Zou and L.B. Holder, Frequent subgraph mining on a single large graph using sampling techniques, In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, New York, NY, USA, 2010. ACM, pp. 171–178.