Incremental Frequent Subgraph Mining on Large Evolving Graphs

Article *in* IEEE Transactions on Knowledge and Data Engineering · August 2017

DOI: 10.1109/TKDE.2017.2743075

6 authors, including:

Ehab Abdelhamid
Imperial College London
5 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Mustafa Canim
IBM
29 PUBLICATIONS **355** CITATIONS
SEE PROFILE

Mohammad Sadoghi
University of California, Davis
60 PUBLICATIONS **495** CITATIONS
SEE PROFILE

Panos Kalnis
King Abdullah University of Science and Techn…
132 PUBLICATIONS **4,378** CITATIONS
SEE PROFILE

Incremental Frequent Subgraph Mining on Large Evolving Graphs

Ehab Abdelhamid Mustafa Canim Mohammad Sadoghi
Bishwaranjan Bhattacharjee Yuan-Chi Chang Panos Kalnis

Abstract—Frequent subgraph mining is a core graph operation used in many domains, such as graph data management and knowledge exploration, bioinformatics and security. Most existing techniques target static graphs. However, modern applications, such as social networks, utilize large evolving graphs. Mining these graphs using existing techniques is infeasible, due to the high computational cost. In this paper, we propose *IncGM+*, a fast incremental approach for continuous frequent subgraph mining on a single large evolving graph. We adapt the notion of "fringe" to the graph context, that is the set of subgraphs on the border between frequent and infrequent subgraphs. *IncGM+* maintains fringe subgraphs and exploits them to prune the search space. To boost the efficiency, we propose an efficient index structure to maintain selected embeddings with minimal memory overhead. These embeddings are utilized to avoid redundant expensive subgraph isomorphism operations. Moreover, the proposed system supports batch updates. Using large real-world graphs, we experimentally verify that *IncGM+* outperforms existing methods by up to three orders of magnitude, scales to much larger graphs and consumes less memory.

Index Terms—Graph algorithms, Data mining, Indexing

1 Introduction

The goal of Frequent Subgraph Mining (FSM) is to find all subgraphs that have support larger than or equal to a user-defined threshold τ. Besides being crucial for graph analysis, FSM is a basic building block of many applications in multi-disciplinary domains, such as graph indexing [1], clustering [2], classification [3], protein functionality prediction [4], privacy-preservation [5], and image processing [6]. Most previous work assumes a static database of many small graphs [7], [8], [9], [10], [11]. Recent work [12], [13] focuses on the case of a single large static graph. This setting is considered a more general case since a database of small graphs can be viewed as one large graph with many disconnected components.

Emerging graph-based applications nowadays manage continuously evolving graphs. Examples include social networks, where friendships (i.e., graph edges) are established and dissolved over time; web graphs, where pages and links are constantly updated; protein-to-protein interaction networks, where knowledge in biomedical databases is frequently updated; or the GDELT project [1], where knowledge graphs containing billions of entries are updated every fifteen minutes with the new events happening around the globe. A straightforward approach to continuous frequent subgraph mining in evolving graphs is to run an FSM algorithm from scratch after every graph update; we call this *FullRecomp*. In a typical FSM iteration, candidate subgraphs are evaluated, frequent ones are extended and those extensions are evaluated. This process repeats until no more frequent subgraphs are found. This involves numerous NP-complete subgraph isomorphism computations. One iteration of the mining task on a graph with a few millions edges can take hours to finish on a typical machine [12], rendering *FullRecomp* infeasible in practice. Due to the high computational complexity, existing approaches on evolving graphs either target the simpler case of a stream of small graphs [14], or produce approximate results [15].

There are many applications that can benefit from an incremental FSM solution. Many graph-based security applications utilize FSM techniques. Typically, these applications are expected to be working in real-time. For example, nuclear smuggling is a world wide dangerous threat. Mining nuclear smuggling data is crucial for preventing such threat [16]. Based on FSM, a set of characteristic patterns of nuclear smuggling events are mined. Future activities that follow such patterns typically require further investigation. Since the smuggling data is updated rapidly, an incremental mining approach is important for the efficient utilization of such data. Moreover, graph-based anomaly detection techniques are used to prevent large-scale security threats [17]. In this setting, an anomalous subgraph is either part of or missing from a non-anomalous subgraph. Non-anomalous subgraphs are repetitive subgraphs that are discovered by utilizing subgraph mining techniques. Detecting anomalies and suspicious activities in real-time is crucial for securing systems against modern and sophisticated threats. On the other hand, having a batch-based solution could fail to detect potential threats.

Many organizations use the configuration management database (CMDB). This database, which can be represented as a graph [18], is important to describe the IT infrastructure entities and their interrelationships. A CMDB is considered an important

- *E. Abdelhamid and P. Kalnis are with the Computer, Electrical and Mathematical Science and Engineering Division, King Abdullah University of Science and Technology, Saudi Arabia. E-mail: ehab.abdelhamid, panos.kalnis@kaust.edu.sa.*
- *M. Canim, B. Bhattacharjee and Y.-C. Chang are with the IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA. E-mail: mustafa, bhatta, yuanchi@us.ibm.com.*
- *M. Sadoghi is with the Computer Science Department, University of California, Davis, 2063 Kemper Hall, Davis, CA, 95616, USA. E-mail: msadoghi@ucdavis.edu.*

1. http://www.gdeltproject.org/

information resource about the largely undocumented IT practices of a large organization, and thus mining the CDMB graph for frequent subgraphs can reveal the infrastructure patterns. Once mined, these patterns are used to set or modify IT policies. The CMDB is usually large (hundreds of thousands of records), and changes rapidly and on regular basis. The extracted frequent subgraphs should reflect the up-to-date changes to the database. Failing to cope with such rapid increase, or simply waiting for a large batch of changes will negatively affect the decision outcomes. Another application is query processing on graph databases, FSM is commonly used for building indexes to improve the performance [1]. Creating such indexes requires a lot of time, especially for large graphs. In order to index dynamic graphs, there should be an efficient solution to incrementally update the index, instead of building it from scratch. An outdated index can drastically impact the advantages of using indices, as such, real-time index update is crucial.

In this work, we propose an exact solution for continuous FSM on a single large evolving graph. The proposed solution is based on an incremental approach. Our problem resembles frequent itemset mining over a stream of transactions. Setting aside various approximations [19], [20], there are numerous exact incremental methods [21], [22], [23], [24], [25]. Many of them, such as the well-known MOMENT system [26], are based on variations of the idea of a "fringe" of itemsets. In the context of frequent itemset mining, a fringe is the set of itemsets on the border between frequent and infrequent ones. After the arrival of a new transaction in the stream, only the fringe needs to be updated, reducing significantly the cost. The same approach is also applicable on graphs. We implemented *MomentFSM*, an adaptation of MOMENT to graphs. Our experiments reveal that *MomentFSM* is too expensive in practice both in terms of memory and computational cost. *MomentFSM* needs to process each subgraph in the fringe, and for each one, it stores all of its embeddings. Compared to *MomentFSM*, our approach utilizes the fringe, but it processes fringe subgraphs only when needed. Furthermore, it materializes a minimal number of embeddings for each subgraph.

In this paper we propose *IncGM+*, a fast incremental system for the continuous FSM problem on a single evolving graph. *IncGM+* is orders of magnitude faster than competitors relying on existing methods, scales to much larger graphs and consumes limited memory. *IncGM+* is based on the fringe concept, but it introduces a number of novel contributions that collectively result in superior performance: (*i*) Instead of storing all embeddings for each subgraph in the fringe, it stores only enough embeddings to prove that the subgraph is either frequent or infrequent. (*ii*) It utilizes a novel index structure that efficiently maintains the stored embeddings. This index is dynamically modified to reflect the updated graph while keeping the memory overhead minimal. (*iii*) *IncGM+* introduces a set of heuristics that significantly improve the overall performance by reordering the execution of the required subgraph isomorphism operations. These heuristics are based on information collected while processing past graph updates, such as the list of subgraph nodes that can lead to quicker decisions, and input graph nodes that can be postponed to avoid useless processing. (*iv*) Finally, to cope with large number of updates, *IncGM+* is extended to support batch processing; a batch of graph updates are grouped and novel pruning techniques are used to reduce the incremental mining cost.

In summary, our contributions are:

- We propose a novel continuous FSM technique for single large evolving graphs, and introduce the fringe concept to the graph mining domain.
- We develop *IncGM+*, an incremental method for continuous FSM; it employs novel techniques that collectively decrease computational cost and memory requirements.
- We conduct extensive experiments on large real-world datasets. *IncGM+* is up to 3 orders of magnitude faster than its competitors, consumes less memory and scales to much larger graphs.

The rest of the paper is organized as follows. Section 2 discusses preliminary concepts and formalizes the problem. Section 3 describes the details of our approach, whereas Section 4 discusses our batching techniques. Section 5 presents the experimental evaluation. Section 6 surveys the related work and Section 7 concludes the paper.

2 PRELIMINARIES

Static graphs are well known, most existing FSM techniques focus on mining these graphs.

Definition 1 *A* static graph $G = (V, E, L)$ *consists of a set of nodes* V*, a set of edges* $E \subseteq V \times V$ *and a function* L *that assigns labels to nodes and edges, where* V*,* E *and* L *are static.*

There has been a recent focus on evolving graphs due to the nature of emerging applications, such as social network, which supports different dynamic operations like: adding friends, removing followers or modifying information about users.

Definition 2 *An* evolving graph $G_D = (V_D, E_D, L_D)$ *consists of a set of nodes* V_D*, a set of edges* $E_D \subseteq V_D \times V_D$ *and a function* L_D *that assigns node labels. Over time,* G_D *is changed by node additions or deletions, edge additions or deletions, and modifications to the labeling function* L_D*.*

An important task in graphs is to find matches of one graph in another graph, which is called *subgraph isomorphism*. Each match resulting from the isomorphism of a subgraph S to a graph G, is called an *embedding* of S in G. Note that, two edges are said to be of the same class if they are isomorphic to each other (i.e., having the same node labels and edge label).

For a subgraph S to be frequent in an input graph G, it has to have support larger than or equal to a user-defined threshold τ. Let S_1 be a subgraph of S_2 and G be the input graph; a support metric is called "anti-monotonic" if the support of S_1 is greater than or equal to that of S_2 in G. Anti-monotonic support metrics allow FSM algorithms to prune the search space by the a-priori principle. A naive support metric is the count of embeddings of S in G; however, this metric is not anti-monotonic. Figure 1.a shows an example; let S_1 be a subgraph containing a single node labeled 'A', which has the following 7 embeddings: $\{u_1, u_{21}, u_{23}, u_{17}, u_8, u_{11}, u_{14}\}$. Let S_2 be subgraph 'A' —— 'B' (i.e., a supergraph of S_1). The set of embeddings of S_2 is: $\{(u_1,u_2), (u_{21},u_{19}), (u_{17},u_{18}), (u_{17},u_{16}), (u_{23},u_{18}), (u_{14},u_{12}), (u_{11},u_{12}), (u_8,u_9)\}$. S_2 has 8 embeddings, which is more than those of S_1; therefore, the used metric is not anti-monotonic. Several anti-monotonic support metrics have been proposed for mining a single graph such as MIS [13], HO [27] and MNI [28]. MNI is the most efficient, since the computation of MIS and HO is NP-complete, while MNI is linear to the number of embeddings. Hence, we adopt MNI in this work, which is defined as follows:

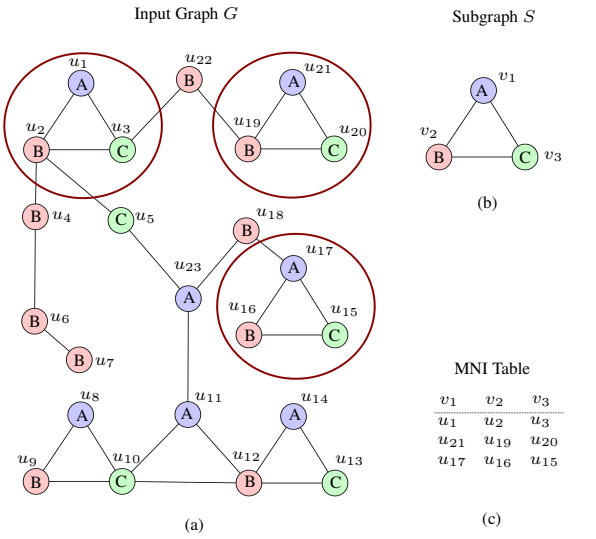


Fig. 1. (a) Input graph G (b) A Subgraph S (c) the *MNI* table of S embeddings in G when $\tau = 3$

Definition 3 *Given an input graph $G = (V, E)$ and a subgraph $S = (V_s, G_s)$. Let $F(v) = \{u \mid u \in V, u$ is a valid mapping of v in at least one embedding of S in $G\}_{\neq}$. The minimum image based support (MNI) of S in G, denoted by $Supp(S, G)$, is defined as $Supp(S, G) = min\{t|t = |F(v)|\forall v \in V_s\}$.*

The *MNI* metric builds an MNI Table. This table consists of a number of columns, each column represents one $v \in S$ and is populated by $F(v)$. The *MNI*-based frequency is calculated as the length of the smallest $F(v)$. Figure 1 shows an example of computing $Supp(S, G)$; the support of S (Figure 1.b) in G (Figure 1.a) based on the MNI metric. Assuming $\tau = 3$, for a subgraph S to be frequent, each of its $F(v)$ has to contain at least three distinct nodes. Given the three embeddings highlighted with circles, $F(v_1)$: $\{u_1, u_{21}, u_{17}\}$, $F(v_2)$: $\{u_2, u_{19}, u_{16}\}$ and $F(v_3)$: $\{u_3, u_{20}, u_{15}\}$. Figure 1.c shows the resulting MNI Table after considering the highlighted three embeddings. S is reported as a frequent subgraph since all columns have size three. Note that only three embeddings are enough to satisfy τ and report S as frequent, regardless of the actual number of embeddings. Assuming $\tau = 6$, six distinct valid assignments are found for v_1: $\{u_1, u_{21}, u_{17}, u_{14}, u_{11}, u_8\}$. However, for v_2 only five distinct nodes are found: $\{u_2, u_{19}, u_{16}, u_{12}, u_9\}$; therefore S is infrequent. For this case, $F(v_2)$, which is the reason for S to be infrequent, is called an *invalid column*.

The goal of FSM in a static graph is to find the set of frequent subgraphs. Utilizing the *MNI* metric, the result set of FSM is:

Definition 4 *Given a static graph G and support threshold τ, the FSM result set R is defined as: $R = \{Sub_1, \ldots, Sub_n\}$, where each $Sub_i \in R$ is a subgraph that has $Supp(Sub_i, G)$ greater than or equal to τ, and there is no subgraph $Sub_k \notin R$ with $Supp(Sub_k, G)$ greater than or equal to τ.*

Figure 2 illustrates the search space for a typical FSM task. Each element (circle) represents a subgraph that exists at least once in the input graph. The elements at the bottom represent subgraphs with one edge. As we move up, each subgraph is extended by one edge. The topmost element represents the input graph (the largest possible subgraph in the search space). The number of elements at each level increases as we move up. This is because the possible number of edge combinations increases as subgraphs get larger. Once a certain level is reached, the input graph constrains these extensions and the number of elements decreases for next levels. Figure 2 shows that the search space is divided into two sets; the set R of frequent subgraphs (solid circles) and the set of infrequent subgraphs (striped and empty circles). Two subsets are of great importance; the maximal frequent subgraphs (*MFS*), which is a subset of R, and the minimal infrequent subgraphs (*MIFS*). *MFS* is a compressed representation of all frequent subgraphs, and it is defined as follows:

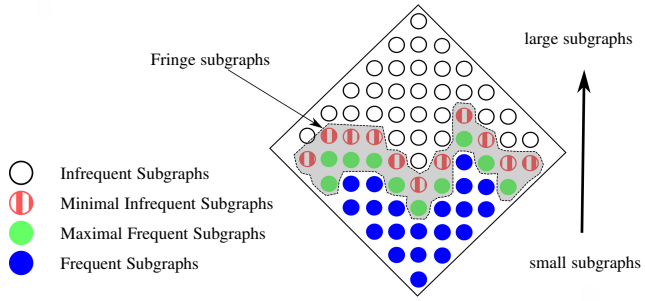


Fig. 2. FSM search space starting from small subgraphs at the bottom to larger subgraphs towards the top

Definition 5 *MFS is the set of all maximal frequent subgraphs such that $S_i \in$ MFS, if and only if S_i is frequent and there does not exist other $S_j \in R$, where S_i is subgraph of S_j.*

MFS is an efficient representation of R; any frequent subgraph either belongs to *MFS* or is a subgraph of an element in *MFS*. As shown in Figure 2, the number of elements in *MFS* is much smaller than those in R. Thus, focusing on *MFS* rather than R allows for improved performance. Another interesting set is the set of minimal infrequent subgraphs (*MIFS*):

Definition 6 *MIFS is the set of all minimal infrequent subgraphs such that for every $S_i \in$ MIFS, S_i is infrequent and there is no other $S_j \notin R$, where S_j is subgraph of S_i*

The set of infrequent subgraphs is huge; *MIFS* is a feasible representation of this set. Other infrequent subgraphs can be constructed by extending elements from *MIFS*.

In the evolving graph setting, the goal of FSM is to continuously report the result set while the input graph is updated. In this setting, FSM is defined as follows:

Problem 1 *Given an evolving graph G_D and a minimum support threshold τ, the problem of* frequent subgraph mining in evolving graph *G_D is to continuously report the result set $R_t = \{Sub_1, \ldots, Sub_n\}$, where each $Sub_i \in R_t$ has $Supp(Sub_i, G_D) \geq \tau$ after graph updates at time t.*

Dynamic graph updates can be considered as a stream of edge and node updates. Updates are either node/edge additions, deletions or label modifications. In the following we focus only on edge additions and deletions, since all other types of updates can be supported by edge additions and deletions. For example a node/edge label update can be represented as node/edge removal then insertion of a new node/edge with the new label. This update can be done efficiently by using batch updates (Section 4).

Figure 3 illustrates an example of an evolving graph at three

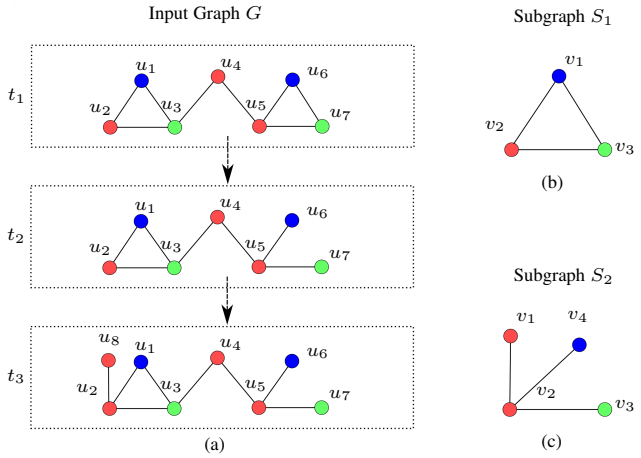


Fig. 3. (a) Evolving graph G at different points in time (b) Subgraph S_1 (c) Subgraph S_2

points in time: t_1, t_2 and t_3. Given an input evolving graph G, assume $\tau = 2$ and we are interested in evaluating two subgraphs S_1 and S_2. At time t_1 the number of matches of S_1 is two, whereas S_2 has only one match; hence, S_1 is frequent and S_2 is infrequent. Advancing to time t_2, edge u_6 —— u_7 is deleted and the number of embeddings of S_1 becomes one, while the number of embeddings of S_2 does not change. As a result, both S_1 and S_2 are infrequent. At time t_3, edge u_2 —— u_8 is added, increasing the number of matches of S_2 to two; thus, S_2 becomes frequent.

3 Incremental Graph Mining

In this section, we propose *IncGM+*, an incremental FSM solution for evolving graphs. *IncGM+* employs three novel techniques to improve the efficiency of the mining task. First, it prunes the search space by focusing on a set of carefully selected subgraphs (fringe subgraphs). After each graph update, only these subgraphs are evaluated. Other elements of the search space are evaluated only if needed. Second, *IncGM+* maintains a minimal number of embeddings for each of the selected subgraphs. These embeddings are then used to enhance or to avoid the evaluation of these subgraphs. Finally, *IncGM+* utilizes information collected during past iterations to improve the efficiency of next iterations. We use *IncGM* to refer to the incremental solution that only uses the fringe-based search space pruning.

3.1 Search Space Pruning

IncGM utilizes the "fringe" concept for incremental search space evaluation. It employs the fringe subgraphs; subgraphs that belong either to *MIFS* or *MFS*. An example of the fringe subgraphs is shown in Figure 2. The search space is significantly pruned by focusing on the fringe subgraphs. Since they are the most sensitive to graph updates, other subgraphs are evaluated only when fringe subgraphs are affected by recent graph updates. Compared to continuous itemset mining [26], evaluating a subgraph (i.e., finding whether it is frequent or not), requires significant overhead to find its embeddings. In order to alleviate this overhead, it is important to avoid unnecessary subgraph evaluations. For example, it is useless to evaluate a subgraph that is not expected to be affected by recent updates. In the following, we show two propositions that can be exploited to avoid such unnecessary overhead.

Proposition 1 *Adding an edge to the input graph results in increasing the support of one or more subgraphs. Thus, after an edge addition at time $t+1$, the only difference (if exists) between the result set R_t and R_{t+1} is the addition of one or more frequent subgraphs to R_{t+1}.*

PROOF: Given G_t is the input graph at time t, and a frequent subgraph $S \in$ *MFS*. We assume S becomes infrequent after an edge addition at time $t+1$ and we obtain a contradiction. For S to become infrequent, it requires one or more of the nodes in its *MNI* table to be removed. Removing a node u from the *MNI* table requires an embedding of S having u as part of it to disappear from the input graph after the recent update at time $t+1$. An embedding that exits in G_t vanishes at G_{t+1} if one of its edges or nodes are removed from G_{t+1}. This contradicts with the fact that the update is edge addition. □

Proposition 2 *Removing an edge from the input graph results in decreasing the support of one or more subgraphs. Thus, after an edge deletion at time $t+1$, the only difference (if exists) between the result set R_t and R_{t+1} is the removal of one or more frequent subgraphs from R_{t+1}.*

PROOF: Given G_t is the input graph at time t, and there exists an infrequent subgraph $S \in$ *MIFS*. We assume S becomes frequent after an edge deletion at time $t+1$ and we obtain a contradiction. For S to become frequent, it requires one or more distinct nodes added to its *MNI* table. Adding a node u to the *MNI* table requires an embedding of S having u as part of it to appear in the input graph after the recent update at time $t+1$. A new embedding appears in G_{t+1} only if one (or more) of its missing edges/nodes are added to G_{t+1}. This contradicts with the fact that the update is edge deletion. □

Based on the above propositions, only elements of *MIFS* need to be evaluated after edge additions, and only elements of *MFS* require evaluation after edge deletions. Given a graph update, algorithm 1 shows how *IncGM* utilizes the fringe subgraphs to prune the search space. The fringe is populated during an initial FSM step (Lines from 3 to 10). This step follows a typical FSM process. It starts with the set of distinct graph edges as the initial set of candidates (Line 3). Each candidate is evaluated (Line 6). This evaluation is done by calling EVALUATE. This function decides whether S is frequent or not by searching for enough embeddings of S in G to satisfy τ. Those found to be frequent are extended and added to $candids$ (Line 9). This process repeats until $candids$ has no more elements. The next part is to continuously process graph updates (Lines 11 to 25), each iteration deals with one update U. *IncGM* evaluates a fringe subgraph S only if it is infrequent and U is edge addition (line 15), or S is frequent and U is edge deletion (line 21). Since *IncGM* does not maintain previously found embeddings, EVALUATE incurs significant overhead while searching for embeddings from scratch. When a subgraph changes its status (e.g., a frequent subgraph becomes infrequent), the fringe is updated by calling UPDATEFRINGE (Lines 19 and 25). For a subgraph S_{FREQ} that is recently found to be frequent, UPDATEFRINGE updates the fringe by: (1) Adding S_{FREQ} to *MFS*, (2) Removing S_{FREQ} from *MIFS*, and (3) extending S_{FREQ} by joining it with other frequent subgraphs of the same size [13]. The extended subgraphs are added to *MIFS* and recursively evaluated. For a new infrequent subgraph S_{INFREQ}, UPDATEFRINGE updates the fringe by: (1)

Input: G the input graph, τ support threshold, $updates$ graph updates
Output: $fringe.MFS$
1 $fringe.MFS \leftarrow \phi$
2 $fringe.MIFS \leftarrow \phi$
3 $candids \leftarrow$ distinct edges in G
4 **while** *candids has more elements* **do**
5 $C \leftarrow$ GETNEXTCANDIDATE($candids$)
6 $isFreq \leftarrow$ EVALUATE(G, τ, C)
7 **if** $isFreq = true$ **then**
8 $ext \leftarrow$ EXTEND(C)
9 $candids \leftarrow candids \cup ext$
10 UPDATEFRINGE($fringe, C$)
11 **foreach** $U \in updates$ **do**
12 **if** U *was not seen before* **then**
13 Add U to $fringe.MIFS$
14 **if** U *is edge addition* **then**
15 **foreach** $S \in MIFS$ **do**
16 **if** SUBGRAPH(U,S) **then**
17 $isFreq \leftarrow$ EVALUATE(G, τ, S)
18 **if** $isFreq = true$ **then**
19 UPDATEFRINGE($fringe, S$)
20 **else**
21 **foreach** $S \in MFS$ **do**
22 **if** SUBGRAPH(U,S) **then**
23 $isFreq \leftarrow$ EVALUATE(G, τ, S)
24 **if** $isFreq = false$ **then**
25 UPDATEFRINGE($fringe, S$)
26 **return** $fringe.MFS$

Algorithm 1: FRINGE-BASED INCREMENTAL FSM

Adding S_{INFREQ} to *MIFS*, (2) Removing S_{INFREQ} from *MFS*, and (3) Adding decompositions of S_{INFREQ} to *MFS*. Each decomposition is created by removing one edge from S_{INFREQ}. Finally, the added decompositions are recursively evaluated. Further pruning is done in lines 16 and 22; S is evaluated only if the updated edge U can be mapped to an edges in S. This pruning is possible since there is no way to affect the support of a subgraph S by an edge update that is not contained in S. Line 12 contains an important step for the correctness of the algorithm: Any new edge update, which was not seen before, is added to *MIFS*. Without this step, *IncGM* will not consider this edge or any of its supergraphs for evaluation, even when this edge becomes frequent.

Correctness of *IncGM*: Assume there exists a frequent subgraph S that is not recognized as frequent by Algorithm 1. According to the anti-monotone property, each edge of S must be frequent. Algorithm 1 recognizes all frequent edges by: past iterations (when it is already frequent), current iteration (when it was previously infrequent) or using line 12 (when it was not seen before). Moreover, UPDATEFRINGE guarantees that for a frequent edge, all of its frequent extensions are discovered. Hence, Algorithm 1 will identify S as frequent, which contradicts with the first assumption. Moreover, each subgraph belonging to *MFS* is guaranteed to be frequent since it is evaluated after every graph update.

By utilizing the discussed propositions and optimizations, the search space is significantly pruned. However, algorithm 1 suffers from considerable overhead caused by EVALUATE. The performance can be improved in two ways. First, by optimizing EVALUATE. Second, by limiting the number of times EVALUATE is called. These improvements are discussed next.

TABLE 1
Comparison of the average number of embeddings per frequent and infrequent subgraphs.

Dataset/τ	Citeseer/120	Patents/18k
Avg(*MFS*)	235	52437
Avg(*MIFS*)	77	21

3.2 Embeddings-based optimization

Most of the overhead incurred by EVALUATE is spent on finding embeddings from scratch. Minimizing this overhead can be achieved by maintaining a list of previously found embeddings. Consequently, finding embeddings from scratch is avoided. *IncGM+* is proposed to benefit from maintaining a carefully selected set of embeddings to improve the efficiency. *IncGM+* does not store embeddings of smaller subgraphs in order to find embeddings of larger ones. Instead, it stores embeddings of a subgraph in order to optimize its evaluation in future iterations. Maintaining these embeddings minimizes the overhead of searching for them repeatedly from scratch after each graph update. In Figure 3 when $e = u_2 \text{—} u_8$ is added to the input graph, SEARCHLIMITED(S_2, e) searches for possible local embeddings that contain both u_2 and u_8. Although this search is conducted from scratch, it is done efficiently since it is restricted to matches that contain both u_2 and u_8. The new embedding $\{u_8$,u_2, u_3, $u_1\}$ is created as a result of adding $e = u_2 \text{—} u_8$. Support is then calculated using the newly found embedding as well as the already existing embedding found in previous iterations (i.e., $\{u_4$,u_5, u_7, $u_6\}$). Finally, the list of embeddings corresponding to S_2 is updated with the new one so that it can be used to optimize future evaluations.

Storing all embeddings is prohibitively expensive, since the number of embeddings grows exponentially with the graph size. A reasonable solution is to store a minimal number of embeddings that is small enough to fit in the available memory. *IncGM+* limits memory consumption by adopting the following guidelines:

- For each subgraph $S \in MFS$, *IncGM+* stores minimal number of embeddings that make $Supp(S, G) \geq \tau$, other embeddings are not maintained. Each embedding contributes to at least one distinct cell in the MNI_{tbl}, and in many cases a single embedding corresponds to more than one cell. Consequently, for a subgraph S, the upper bound for the number of embeddings that are needed to satisfy τ is: $\tau \cdot |S|$, where $|S|$ is the number of nodes in subgraph S (i.e., the number of *MNI* columns). Recall the example of Figure 1 when $\tau = 3$. Only 3 embeddings were required to satisfy τ.
- For each subgraph $S \in MIFS$, *IncGM+* stores up to τ embeddings for each of its *MNI* columns. All subgraphs in *MIFS* are infrequent (i.e., each has support $< \tau$). Hence, every $S \in MIFS$ has at least one *MNI* column with a number of nodes less than τ. In practice, the number of stored embeddings for an infrequent subgraph S is: $T \cdot |S|$, where $T < \tau$.

Based on the above guidelines, the number of stored embeddings for both *MFS* and *MIFS* is bounded by τ. Table 1 shows the results of an empirical study to measure the average number of embeddings for subgraphs belonging to either *MIFS* or *MFS*. The experiment follows the proposed approach for maintaining embeddings. As suggested by the given analysis, the number of

embeddings are bounded. Moreover, the average number of embeddings for each infrequent subgraph is much less than the average number of embeddings for each frequent subgraph. Although being bounded, maintaining these embedding as a simple list is inefficient. For efficient maintenance of those embeddings, we propose the *Fast Embeddings Lookup Store* (*FELS*). *FELS* allows efficient addition, removal and *MNI*-based support computation. More details about *FELS* are discussed in Section 3.3.

Algorithm 2 shows how *IncGM+* exploits the materialized embeddings. When an edge is added (Line 3), *IncGM+* searches only for new embeddings instead of searching for all embeddings from scratch. At line 5, SEARCHLIMITED finds new embeddings by applying subgraph isomorphism starting with the added edge. Those newly found embeddings are added to the *FELS* object associated with S (Line 8). Then, S is checked for being frequent (Line 10). This check is efficiently done by utilizing all embeddings stored in its corresponding *FELS* object. Finally, the fringe is updated accordingly (Line 10). Note that, without maintaining a list of embeddings, *IncGM+* would need to use EVALUATE, which searches for embeddings from scratch. SEARCHLIMITED employs the following optimization: In some cases, the local area around an added edge is dense and contains a large number of embeddings. Only in such scenario, searching the local area for all embeddings poses extra overhead compared with EVALUATE, which is designed to efficiently fill the *MNI* table [12]. To accommodate for such situation, while searching the local area for new embeddings, if the number of found embeddings exceeds a preset limit, SEARCHLIMITED stops and returns null, and the algorithm falls back to the normal evaluation method (Line 6). This is the only case where EVALUATE is needed for edge additions.

Calling EVALUATE after edge deletion is almost avoided by utilizing the stored embeddings. After an edge is deleted, some of the maintained embeddings will vanish and need to be removed from the list of embeddings associated with a subgraph S (Line 13). But in many cases, the deleted edge does not affect any of the stored embeddings, especially when the input graph is large, and the stored embeddings represent a small portion of the graph. In such cases, $Supp(S, G_D)$ (i.e., support of S in G_D) is not affected and therefore evaluation of S is not needed. If an edge deletion results in the removal of any stored embeddings, then *MNI* is computed using the remaining ones. If based on the currently maintained embeddings, the computed *MNI* value still satisfies τ, then there is no need to do further processing (Line 15). Otherwise, EVALUATE is used to find more embeddings (Line 16).

Algorithm 2 treats edge additions and edge deletions differently. For edge additions, only elements in *MIFS* are processed. While, for edge deletions, elements in both *MFS* and *MIFS* are processed. The following discussion highlights the reasons for this difference. Edge additions are more expensive since new embeddings are to be found. While for edge deletions, obsolete embeddings are removed from the embeddings lists, these removals are efficiently done by our novel data structure (*FELS*). Due to its efficiency, edge removal is not postponed and is immediately applied to the two sets: *MIFS* and *MFS* (Line 13). While for edge addition, in order to minimize the processing overhead and memory consumption, embeddings are only added to subgraphs belonging to *MIFS*. As a result, not all existing embeddings of subgraphs in *MFS* are maintained. Thus, it is possible for a subgraph to be frequent even if its maintained embeddings cannot satisfy τ after an edge deletion (line 15). For such case, calling EVALUATE is required to look for other embeddings that were not discovered before (Line 16).

```
Input: G the input graph, τ support threshold, U Dynamic updates
1  if U was not seen before then Add U to MIFS
2  FRINGE← MINE(G_D, τ)
3  if U is edge addition then
4      foreach S ∈MIFS do
5          EMBEDS← SEARCHLIMITED(S, U)
6          if EMBEDS is null then EVALUATE(G, τ, S)
7          else
8              FELSUPDATE(EMBEDS)
9              S.freq ← MNI(S)
10         if S changes status then UPDATEFRINGE(FRINGE, S)
11 else
12     S_All← MIFS ∪ MFS
13     REMOVEEMBEDS(S_All, U)
14     foreach S ∈MFS do
15         if MNI(S) < τ then
16             EVALUATE(G, τ, S)
17             if S changes status then
                   UPDATEFRINGE(FRINGE, S)
```

Algorithm 2: EMBEDDINGS-BASED FSM

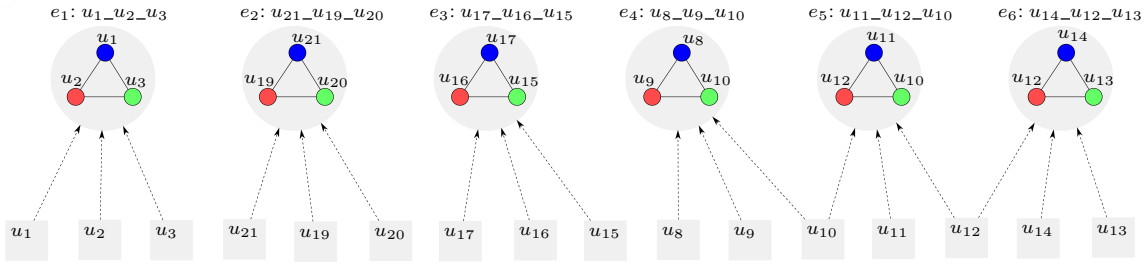
Correctness of *IncGM+*: Decisions regarding infrequent fringe subgraphs are based on either full support evaluation (calling EVALUATE) or retrieval of the complete list of new embeddings. Thus, these decisions are guaranteed to be correct. As for frequent fringe subgraphs, only a minimal number of embeddings is maintained. When a graph is updated with an edge addition or the update is edge deletion that does not affect any of the existing embeddings, then there is no effect on the frequent subgraphs. If edge deletion affects any of the maintained embeddings, then full support evaluation is used to guarantee correctness.

3.3 Fast Embeddings Lookup Store (*FELS*)

We propose *FELS*, an efficient store for maintaining a list of embeddings. It supports fast access and update, and it is used to efficiently compute the MNI-based support value.

Components: Each *FELS* object corresponds to a subgraph S and has three components: 1- A Hash table of embeddings of S, 2- Inverted index from graph nodes to embeddings, and 3- An *MNI* table. Each embedding is hashed by its unique key, this key is created by concatenating the embedding node IDs ordered according to their corresponding S nodes IDs. The inverted index is used to lookup embeddings given their nodes. *FELS* utilizes the *MNI* table to compute the updated *MNI*-based support value. Each cell in the *MNI* table corresponds to a node, the number of embeddings containing this node is attached to each cell.

Figure 4 illustrates the (*FELS*) object corresponding to subgraph S from Figure 1. This object maintains 6 embeddings, namely: $\{e_1, e_2, e_3, e_4, e_5, e_6\}$, each one having a unique key. For example, embedding e_2 has key: "$u_{21}\_u_{19}\_u_{20}$". The inverted index in Figure 4.a contains 16 distinct graph nodes, each node indexes the embeddings it is contained in. For example, e_2 is indexed by u_{21}, u_{19} and u_{20}. Some nodes may index more than one embedding, such as u_{12} which indexes two embeddings e_5 and e_6. Figure 4.b shows the *MNI* table, each column corresponds to a specific node $\in S$ and is populated with distinct matching nodes $\in G$. There is a counter value attached to each cell representing the number of embeddings indexed by the node



a. Inverted index: nodes to embeddings

v_1		v_2		v_3	
Node ID	Counter	Node ID	Counter	Node ID	Counter
u_1	1	u_2	1	u_3	1
u_{21}	1	u_{19}	1	u_{20}	1
u_{17}	1	u_{16}	1	u_{15}	1
u_8	1	u_9	1	u_{10}	2
u_{11}	1	u_{12}	2	u_{13}	1
u_{14}	1				
6		5		5	

b. *MNI* Table

Fig. 4. Fast Embeddings Lookup Store (FELS). (a) Embeddings inverted index and (b) *MNI* table

corresponding to this cell. For example, node u_{12} has a value 2 as it indexes two embeddings: e_5 and e_6.

Operations on Embeddings: *FELS* supports efficient addition and removal of embeddings. For a new embedding, its key is used to check whether it already exists. *FELS* does not allow multiple entries for the same embedding. Adding a new embedding involves adding it to the embeddings list, adding its nodes to the inverted index, and populating the *MNI* table with its nodes. In figure 4, assume e_6 is a new embedding. It is added to the list of embeddings, its nodes: $\{u_{12}$,u_{14},$u_{13}\}$ are added to the inverted index and the *MNI* table is populated with those nodes. If a node does not exist in its corresponding column, such as u_{14}, then a new entry is created for this node with a counter value set to 1. For a node that already exists, such as u_{12}, the counter associated with its entry is incremented (e.g., the value associated with u_{12} becomes 2. The removal of an embedding involves deleting it from the list of embeddings, all pointers to it are removed from the inverted index and its corresponding *MNI* entries are either decremented or removed. Assume e_6 is removed from the example *FELS* in figure 4. Its entry will be deleted, the inverted index pointers from u_{12}, u_{14} and u_{13} are deleted and consequently u_{14} and u_{13} are removed from the index. Moreover, the entries for u_{14} and u_{13} in the *MNI* table are deleted and the value associated with u_{12} is decremented and becomes 1.

MNI Computation: *FELS* utilizes the stored embeddings to compute the *MNI*-based support value. This is done by checking the length of each *MNI* column and reporting the minimum length. For example, given $\tau = 5$ in Figure 4, S is frequent because its support value based on the *MNI* table is 5. Suppose that edge u_{11} —— u_{12} is deleted from the input graph. Then embedding e_5 becomes obsolete and is removed from the inverted index and the *MNI* table. The new set of embeddings becomes: $\{e_1$,e_2,e_3,e_4,$e_6\}$. By consulting the *MNI* table, all of its columns become of length 5. Thus, the support value is still 5. This happens because u_{12} and u_{10} entries in the *MNI* table both had an attached value of 2 (two embeddings indexed by each node). Since the embedding (u_{11}, u_{12}, u_{10}) is removed, the counter attached to each node is decremented. Thus, u_{11} is removed, while u_{12} and u_{10} both remain in the *MNI* table.

3.4 Reordering

The execution order of the support evaluation step affects the performance significantly. The question is, how to select the best performing execution order? Since such knowledge cannot be obtained in advance, *IncGM* exploits information collected during past iterations to decide better ordering. It employs the following two heuristic-based ordering techniques:

1- Graph nodes reordering: Given an input graph G and a subgraph S, an invalid node is a node that belongs to G and cannot be part of an embedding of S in G. Checking the validity of these nodes usually consumes significant overhead. To enhance the performance, a list of invalid nodes is maintained during previous iterations. Then, while evaluating the support of S in subsequent iterations, invalid nodes are postponed for the hope that other nodes can satisfy τ. As such, a significant amount of computation associated with invalid nodes is avoided.

2- *MNI* column reordering: A subgraph is infrequent if it has at least one invalid column. Infrequent subgraphs usually stay infrequent by having the same invalid column in future iterations. In order to exploit this observation in future evaluations, *IncGM* starts by checking invalid columns. As such, the redundant overhead of checking *MNI* column other than the invalid ones for infrequent subgraphs is avoided. There are some cases where the invalid column is not known in advance, such as when a subgraph has never been checked before. Such subgraph is an outcome of joining two frequent subgraphs or decomposing an infrequent subgraph. For these new subgraphs, the last known invalid column of a source subgraph is used as the invalid column.

4 Batching

For practical applications with heavy workloads, batching can be used to speedup processing. It allows expensive support computations to be aggregated for improved efficiency. The proposed batching approach consists of two parts; updates grouping and subgraphs pruning.

Updates grouping. Updates grouping involves three simple steps. First, repeated similar updates are grouped and processed once. Second, edges that cancel each other are ignored (e.g., edge is added then deleted within the same batch). Third, apply grouping optimization, which is to group updates by the affected subgraphs. The idea is to identify the list of unique subgraphs that might be affected by the given batch of updates and add them to the *ToBeChecked* list. Then, the elements in *ToBeChecked* are evaluated instead of evaluating each edge separately.

Figure 5 shows an example of how important it is to use the grouping optimization. The set of minimal infrequent subgraphs is shown in Figure 5.a. Without loss of generality, we consider a batch of edge additions only (Figure 5.b). Figure 5.c shows

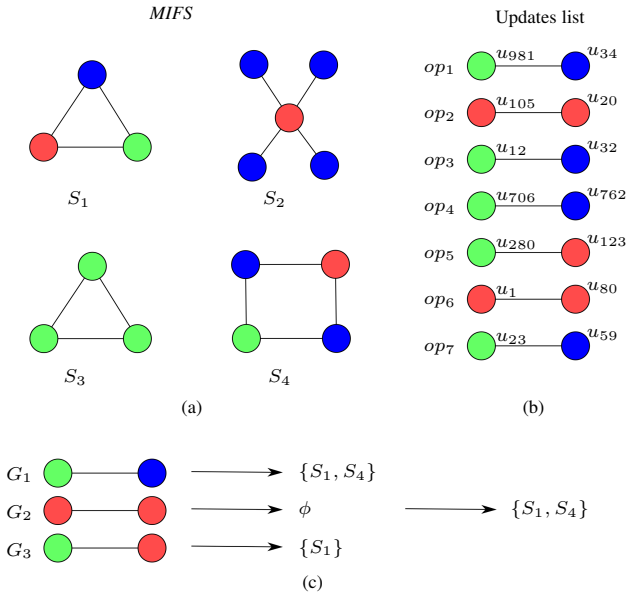


Fig. 5. Batching Steps (a) *MIFS*, (b) Updates list, and (c) Updates grouping

three groups of updates, each group contains the set of subgraphs that might be affected. Finally, the union of subgraphs contained in these groups are added to the *ToBeChecked* list. If grouping optimization was not performed, we would end up with executing nine support computations. Instead, when grouping optimization is used, only two support computations are required.

Subgraphs Pruning. *IncGM+* benefits from the relationship among subgraphs in *ToBeChecked*. A child/parent relationship is defined among these subgraphs. A subgraph S_1 is a child of another subgraph S_2, if S_1 is infrequent and it is a supergraph of S_2. Also, a subgraph S_2 is a parent of S_1 if S_2 is a frequent subgraph and it is a subgraph of S_1. The following propositions highlight interesting properties of these relationships.

Proposition 3 *Given S_1, $S_2 \in$ ToBeChecked. If a subgraph S_1 is a child of a subgraph S_2, and S_2 is checked and found to be infrequent, then S_1 can be safely removed from ToBeChecked.*

PROOF: Given that S_2 is an infrequent subgraph, and S_2 is subgraph of S_1. By the Apriori principle, S_1 must be infrequent. Being infrequent and a supergraph of an infrequent subgraph, S_1 cannot belong to the set of minimal infrequent subgraphs (*MIFS*). As such, there is no need to maintain and process S_1. □

Proposition 4 *Given S_1, $S_2 \in$ ToBeChecked. If a subgraph S_1 is a parent of a subgraph S_2, and S_2 is checked and proved to be frequent, then S_1 can be safely removed from ToBeChecked.*

The proof of this proposition follows similar steps to proposition 3.

Based on the above propositions, no need to evaluate a frequent subgraph if one of its children is found to be frequent. Also, no need to evaluate an infrequent subgraph if one of its parents is found to be infrequent. The question now is: Which subgraphs to start evaluating in order to maximize the benefits of using these propositions? It is better to start evaluating subgraphs that are about to change their status before other subgraphs. For example, for an infrequent subgraph S_1 which is a child of a frequent subgraph S_2, if it is known that S_1 will become frequent after applying the current batch of updates, then it is better to start evaluating S_1 so that processing of S_2 is avoided. Unfortunately, such information is not known in advance. Thus, *IncGM+* utilizes a heuristic-based solution to predict a good ordering of subgraphs. This ordering is based on a simple algorithm and a scoring function. The scoring function gives lower scores to subgraphs that are expected to change their status. The algorithm works as follow: First, all edge deletions are processed on all subgraphs $\in$ *ToBeChecked*. Second, the *ToBeChecked* list is shortened by removing all subgraphs that are still frequent after the first step. Finally, *ToBeChecked* is sorted in an ascending order according to the following scoring function:

$$Score(S) = |\hat{Supp}(S, G_D) - \tau|$$

$$\hat{Supp}(S, G_D) = (\alpha_S + \#Edges * \beta_S)$$

$Score(S)$ is an estimate of the difference between support of S after current graph updates ($Supp(S, G_D)$) and the required threshold τ. A small score value indicates more chances for a subgraph S to change its status. An estimate, $\hat{Supp}(S, G_D)$, is used for calculating expected support since the new exact support is unknown before graph updates evaluation. α_S is support of S before current batch of update, $\#Edges$ is the number of edge addition in current batch and β_S is the expected increase in the support of S for each edge addition. β_S is estimated based on statistics collected during previous iterations; β_S=median(L_S), where L_S is an ordered list of support increments per each previous edge addition. The final step is to order *ToBeChecked* according to $Score(S)$, then evaluate each of its elements. After evaluating each candidate, parents of a frequent subgraph are removed from *ToBeChecked* as well as children of an infrequent subgraph. New subgraphs that result from extending existing subgraphs are appended to the end of *ToBeChecked*.

5 Experimental Evaluation

This section experimentally compares the proposed incremental approaches; *IncGM* and *IncGM+* against competitors relying on existing techniques. *IncGM* represents our incremental approach based on fringe pruning (Subsection 3.1). *IncGM+* extends *IncGM* by maintaining a minimal number of embeddings and utilizing the ordering optimizations (Subsections 3.2, 3.3 and 3.4).

Competitors: Since there is no prior work on the problem of incremental FSM, we implemented three baseline competitors; *FullRecomp*, *MomentFSM* and *GastonFSM*:

FullRecomp executes a full FSM iteration from scratch after each graph update. For our experiments, *FullRecomp* employs GraMi [12], the state-of-the-art FSM technique.

MomentFSM borrows ideas from *Moment* [26]; a well known incremental frequent itemset mining solution. *MomentFSM* maintains a fringe of subgraphs that lay on the boundary between frequent and infrequent subgraphs. It stores all embeddings for each fringe subgraph. At each graph update, all fringe subgraphs are re-evaluated and their embeddings are updated accordingly.

GastonFSM is an extension of *MomentFSM* that utilizes the proposed approaches by the Gaston system [29] for generating and maintaining the list of embeddings. Generation of new embeddings is based on extending existing embeddings. Consequently, the performance of embeddings generation is improved. As for embedding storage, each embedding maintains a pointer to its predecessor, resulting in a chain of embeddings. As such, redundant

TABLE 2
Datasets and their characteristics

Dataset	Density	Nodes	Edges	Distinct node labels
Twitter	Dense	11,316,811	85,331,846	25
Patents	Dense	2,923,922	13,968,441	418
Yahoo	Dense	99304	903113	5576
CiteSeer	Medium	3,312	4,732	6

embedding nodes are shared among several embeddings, resulting in less memory consumption. On the other hand, it is required to traverse the chain of embedding in order to get full information about a single embedding,

System specs: All experiments are conducted using a machine with 2.67GHz Intel Xeon processor and 192GB of RAM. The machine runs Linux Ubuntu 12. Our systems and the mentioned competitors are implemented in Java. Notice that, we used a machine with large memory to be able to run *MomentFSM* and *GastonFSM*, while both *IncGM* and *IncGM+* can run on a machine with much lower memory.

Datasets: We use four real directed graphs in our experiments. Table 2 summarizes the characteristics of these graphs. We describe next the details of each graph.

Twitter [2]: This graph models the Twitter social network. A node represents a user and edges represent users following other users. The original graph is unlabeled, so each node is assigned a random label from a pool of 25 labels based on Gaussian distribution.

Patents [30]: This graph models U.S. patents' citations and includes all citations of patents granted between 1975 and 1999. Each node represents a patent and is labeled with the patent class. Each edge represents a citation and is assigned the grant date of the patent it is originating from.

Yahoo [3]: This graph is part of the *Yahoo! Webscope project (G5 v1.0)* which represents a network of *Yahoo! Messenger* user communications over 28 days. Each node represents a user and is labeled with his address zip code. Each edge represents a communication between two users. A time-stamp is attached to each edge stating the first communication time and date.

CiteSeer [4]: This graph is a citation network. Each node is labeled with a Computer Science area and each edge represents a citation.

Workloads: Three workloads are generated for each dataset, namely edge additions, edge deletions and an equal mix of additions and deletions.

Twitter: Twitter does not have timestamps; therefore, edge additions and deletions are randomly chosen. We use 2500 edge updates for each workload.

Patents: Edges are ordered according to their grant date. For edge additions, the experiment proceeds with the original graph without the latest 50K edges. Then, these edges are added ordered by their timestamps. For edge deletions, the whole graph is used. Then, the latest 50K edges are deleted following the chronological order. For the mixed workload, the original graph is used without the latest 25K edges. Then the latest 25K edges are added while the oldest 25K edges are deleted.

Yahoo: Since this dataset has real timestamps, workloads follows the chronological order as discussed for the Patents dataset. The size of each workload is 50K updates.

CiteSeer: The workload follows a similar approach to the Twitter dataset for the mixed workload. Since Citeseer is a small graph, the deletions workload starts with the original graph extended with 2500 random edges, then they are removed in a random order. As for additions, the whole Citeseer graph is used then extended with random edges. Each workload contains 2500 updates.

In the rest of this section, we compare the different approaches in terms of efficiency and memory consumption. Finally, the proposed batching technique is evaluated.

5.1 Efficiency

This experiment compares the efficiency of *FullRecomp*, *MomentFSM*, *GastonFSM*, *IncGM* and *IncGM+*. For each system and for each workload, efficiency is calculated as the time required for graph loading and mining for the whole workload. Figure 6 shows the efficiency results. The y-axis shows the elapsed time in log scale, while the x-axis represents the support threshold. For Twitter, Patents and Yahoo, it is not possible to process the whole workload using *FullRecomp* within reasonable time. Thus, we estimate the elapsed time by measuring the time for processing a small subset of the workload then extrapolating the total time to process the whole workload. For edge additions, *IncGM+* shows at least two orders of magnitude improvement compared to *FullRecomp*. This improvement is the result of using the fringe subgraphs to prune the search space. As for the deletions workload, improvement is more than three orders of magnitude. The deletions workload affects a smaller subset of the fringe compared to the additions workload. Consequently, deletion updates are processed faster. Both *MomentFSM* and *GastonFSM* show good performance for smaller graphs and higher τ values (i.e, Figure 6.B.1 for $\tau = 160$). For larger graphs and lower τ, they cannot even complete the task. For example, *MomentFSM* and *GastonFSM* consume the available memory and cannot finish any task for Twitter. This happens because both systems need to store all embeddings for the fringe subgraphs. For larger graphs and lower τ values, the number of embeddings becomes enormous, storing these embeddings requires storage that exceeds the available memory. Although *GastonFSM* efficiently generates new embeddings by extending old ones, Figure 6 shows a significant degradation in performance of *GastonFSM* compared to *MomentFSM* in most cases. Such poor performance is the result of the excessive use of pointers to retrieve embeddings information. Yahoo dataset is an exception; the performance is comparable between *MomentFSM* and *GastonFSM*. Additionally, *GastonFSM* is able to finish difficult tasks that *MomentFSM* cannot finish within the given memory budget (For example, Figure 6.A.4). *GastonFSM* stores intermediate embeddings in a compressed format. Therefore, it can operate within a smaller memory budget compared to *MomentFSM*.

Compared to *IncGM*, Figure 6 shows the benefits of the proposed embeddings-based optimization implemented in *IncGM+*. The improvement is particularly notable for both Citeseer and Patents. For higher τ values, the average size of frequent subgraphs found in Twitter and Yahoo is rather small. Maintaining these small embeddings almost equals the overhead of searching for embeddings from scratch. Hence, the embeddings-based optimization gives minimal improvement. For lower τ values, larger subgraphs are found, and the cost of searching for embeddings becomes significant. Thus, utilizing already found embeddings starts paying off. For example, at Figure 6.C.3, there is a considerable improvement when $\tau = 110k$.

2. http://socialcomputing.asu.edu/datasets/Twitter
3. http://webscope.sandbox.yahoo.com
4. http://linqs.cs.umd.edu/projects/projects/lbc

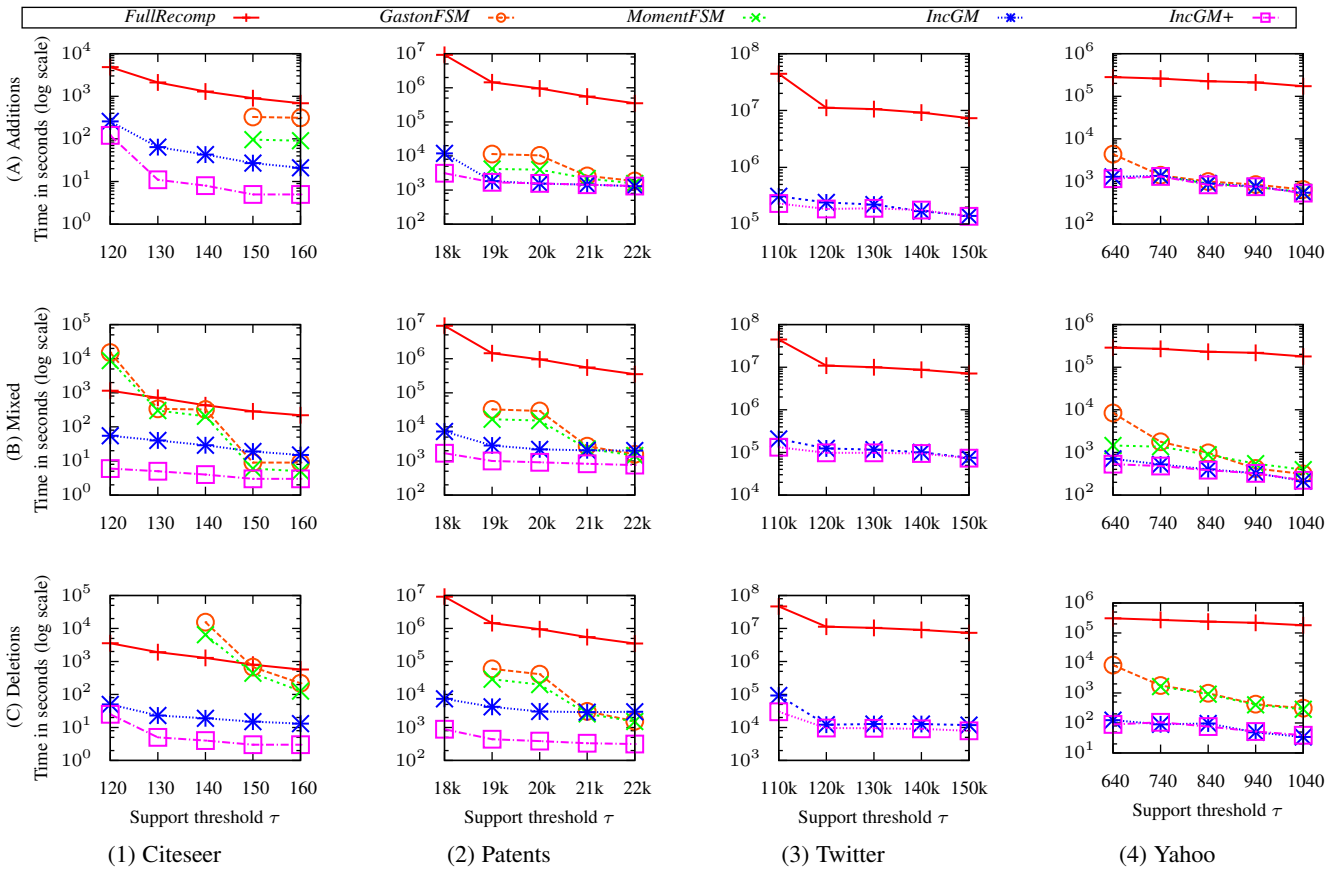


Fig. 6. Performance comparison: *FullRecomp*, *MomentFSM*, *IncGM*, and *IncGM+*. Time represents the graph loading time plus the time required for processing all graph updates.

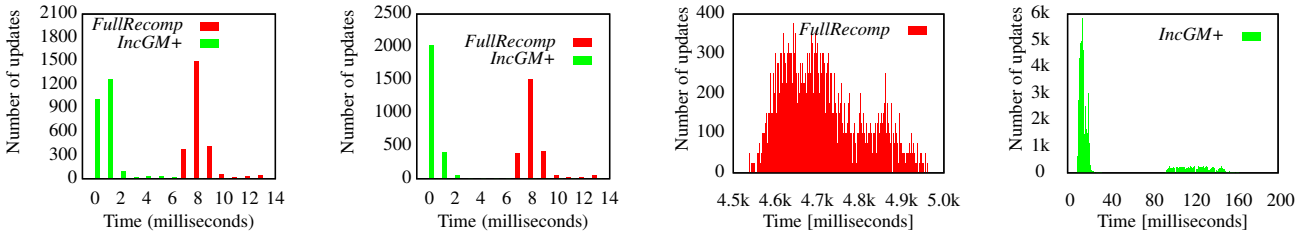


Fig. 7. Updates Histogram

By utilizing *IncGM+*, different regions of the search space are pruned according to the edge being updated. Consequently, the overhead of processing graph updates varies. On the contrary, processing overhead is always larger for *FullRecomp* and does not vary significantly for different edge updates. Figure 7 shows the histograms representing how much time is spent per edge update. The x-axis shows the time required to finish an update, and the y-axis shows the number of updates that are completed within a particular time. Figures 7.a and 7.b show the results for the Citeseer dataset when $\tau = 120$ for the additions and deletions workloads. For *IncGM+*, update times are clustered towards the left, most updates require minimal time. For *FullRecomp*, most updates require larger processing time, and there is no significant deviation of the overhead among the different updates. For the Yahoo dataset when $\tau = 640$, Figures 7.c and 7.d show the histograms of *FullRecomp* and *IncGM+* using the additions workload. Note that different scales are used since there is a significant overhead difference between the two approaches. For *IncGM+*, there are two clusters, the cluster of updates that do not require any significant processing (on left of Figure 7.d), and the cluster of updates that require more processing as a result of searching for new embeddings (on right of Figure 7.d). For *FullRecomp*, compared to *IncGM+*, updates require on average 100x more time, and deviation among the different updates is insignificant.

The number of times an update occurs on the fringe (e.g., a frequent subgraph becomes infrequent) depends on several factors; the input graph, the used τ and the type of graph updates. The following results show how these factors affect the frequency of fringe updates. Table 3 shows the average number of graph updates that result in one fringe update for Citeseer, the used frequency

TABLE 3
Average number of graph updates per one fringe update on Citeseer when using 4500 graph updates and varying τ.

τ	90	100	110	120
#updates	70	90	164	209

TABLE 4
Comparing the memory consumption in MB between MomentFSM and *GastonFSM*.

System	Citeseer/150	Citeseer/160	Patents/19k	Patents/20k
MomentFSM	703	670	58,249	50,761
GastonFSM	587	565	38,984	34,058

threshold (τ) is varied. When τ is small, *IncGM+* becomes more sensitive to graph updates and more fringe subgraphs are affected by the updates. For Patents, when deleting 9M edges and using τ = 500, changes to the fringe are happening less frequently. On average, one fringe update happens for every 280,000 graph updates. When changing the update type to mixed edge additions and deletions, the fringe becomes stable and does not change.

5.2 Memory overhead

The following experiments measure the memory overhead. Figure 8 shows the number of subgraphs stored in *MFS* and *MIFS* for the different datasets and different τ values. It is clear that the size of each set is not extremely large, and they do not rapidly grow as τ decreases. Moreover, the size of *MFS* is always smaller than the size of *MIFS*. Another interesting observation is that the size of *MIFS* is affected by the number of distinct labels in the input graph. *MIFS* gets larger as the number of distinct labels increases. For example, Citeseer has the smallest number of distinct labels, and it has the smallest *MIFS*. Also, Yahoo has the largest number of distinct label and, accordingly, has the largest *MIFS*.

Figure 9 shows the memory consumption of each system. *FullRecomp* does not store any intermediate results, so it consumes the least amount of memory. *MomentFSM* represents systems that store intermediate embeddings. Thus, *MomentFSM* consumes much memory (e.g., Figure 9.2) and even crashes due to exceeding the memory limit (e.g., Figure 9.3). For this, *MomentFSM* cannot be considered a feasible solution. We conduct another experiment to investigate the possibility of utilizing *GastonFSM*'s compressed representation of embeddings. Table 4 shows the memory consumption of *MomentFSM* and *GastonFSM* using the additions workload. As shown in table 4, *GastonFSM* cuts down the memory consumption. However, when experimenting with lower τ values (i.e., Citeseer with $\tau = 140$), *GastonFSM* crashes due to insufficient memory, similar to the behavior of *MomentFSM*.

Figure 9 also highlights the memory consumption of our proposed techniques. In comparison with *FullRecomp*, *IncGM* does not excessively consume much memory to maintain the fringe. The maximum increase in memory consumption appears for the smallest graph (around 4X the memory of *FullRecomp*). While for larger graphs, the increase is relatively smaller (Figures. 9.2-9.4). Although *IncGM+* maintains a list of embeddings, its extra memory usage is insignificant compared to *IncGM*. This is due to our approach that carefully maintains a limited number of embeddings. Note that Figure 9 shows memory in log scale, small differences are difficult to notice. Numbers show some minor differences between the memory consumption of *IncGM* and *IncGM+*. For example, when mining the Patents graph at $\tau = 22K$, *IncGM+* consumes 36MB more memory, and for $\tau = 19K$ the difference becomes 112MB. Also, for Yahoo, the memory overhead ranges from 17MB for $tau = 640$ to almost 1MB for $tau = 1040$. The extra memory is used for storing invalid nodes as well as embeddings. This extra overhead is minimal compared to the overall memory required to store the input graphs. There is a correlation between the extra memory overhead and the efficiency results of Figure 6. As more memory is consumed, the performance difference between *IncGM* and *IncGM+* becomes more significant. In Figure 6.A.2, for τ greater than 18k, there is no improvement in the efficiency of *IncGM+*. This is because no embeddings are utilized when τ is greater than 18k which is confirmed by the memory consumption (Figure 9.2). But when τ =18k, differences between *IncGM* and *IncGM+* in both memory consumption and efficiency become obvious.

5.3 Batching

Next experiments answer the following three questions: What is the effect of batching on performance? How the incremental batching is different compared to *FullRecomp* batching? and what are the benefits of the proposed batching optimizations?.

Figure 10.a answers the first question. This experiment is conducted on Twitter using the additions workload. Three settings are used based on *IncGM+*; the first setting is to process edge by edge (Inc1), the second is to batch 100 updates (Inc100) and the third is to batch 500 updates (Inc500). Each bar represents the overall processing time for each corresponding batch size. As shown in the figure, batching improves the overall performance as the batch size increases. For example, Inc500 outperforms Inc1 by more than an order of magnitude.

Figures 10.b,c and d answer the second question; comparing batching that is based on *IncGM+* and batching using *FullRecomp*. Figures 10.b and 10.c compares batching on *FullRecomp*, *IncGM* and *IncGM+* using a fixed batch size of 1000 updates. Figures 10.b shows results for the a workload of size 100K edge additions, and Figures 10.c shows the results of 100K edge deletions. It is clear that both incremental approaches outperform *FullRecomp* in both settings. The difference is more significant in the deletions workload and as the problem becomes more expensive (i.e., using lower τ value). Figure 10.d shows the speedup of incremental batching compared to *FullRecomp* batching when batch size is changed while τ is fixed. Both systems process additions workload on Patents using $\tau = 18k$. Starting with a batch size of 100, the incremental approach is two orders of magnitude faster. After a certain threshold (i.e., batch size=10k), the performance of *FullRecomp* becomes closer to the incremental approach. The reasoning behind this observation is that as the batch size gets larger, more parts of the search space are processed and, therefore, less pruning can be applied. Similar observations are also reported in incremental approaches used in other domains [31], [32].

The last experiment answers the third question regarding the effect of the proposed batching optimization techniques, namely grouping and subgraph pruning optimizations. Table 5 shows the results for Citeseer (τ=120 and batch size = 100) and Patents (τ=18K and batch size = 1000), where (NoBatching) refers to the edge-by-edge processing, (Opt1) is when only grouping optimization is enabled and (Opt2) is when both grouping and subgraph pruning optimizations are enabled. As shown in table 5, significant improvement is achieved by applying Opt1 and Opt2.

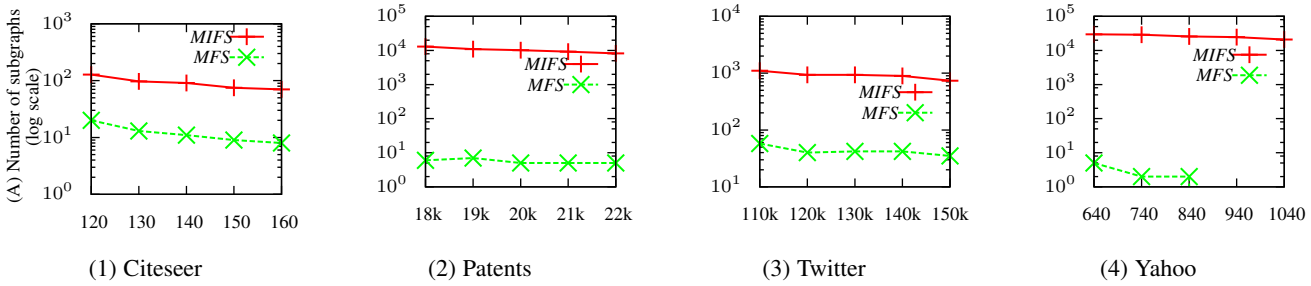


Fig. 8. The fringe size: the number of materialized subgraphs in *MFS* and *MIFS*.

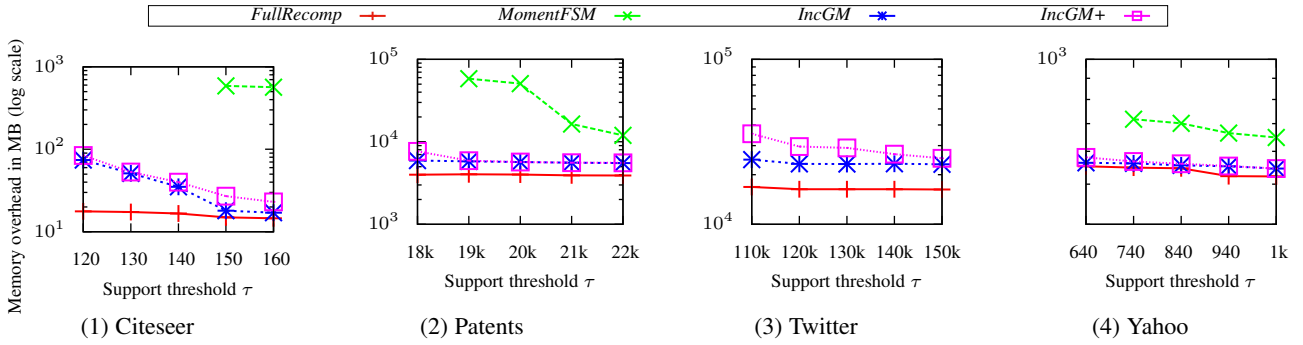


Fig. 9. Memory overhead.

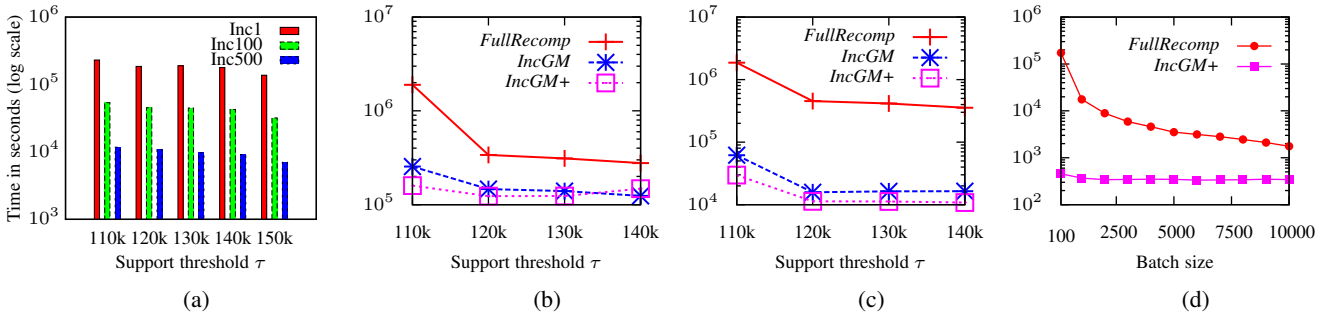


Fig. 10. Batching evaluation. Y-axis represents the time required to process the given workload in seconds. (a) When using different batch sizes compared to edge by edge updates on Twitter. (b) 100K edge additions using a batch size of 1000 edges on Twitter. (c) 100K edge deletions using a batch size of 1000 edges on Twitter. (d) Effect of different batch size on Patents when $\tau = 10K$.

TABLE 5
Comparing the performance of the different batching optimization techniques.

Dataset/τ	Citeseer/120	Patents/18k
NoBatching	121	3125
Opt1	86	488
Opt2	18	396

6 Related Work

There are several research directions, discussed below, that are related to the problem of FSM in dynamic graphs.

Frequent Subgraph Mining: FSM is known under two settings. The first is the *transactional mining* where the goal is to discover frequent subgraphs on a dataset of many, usually small, graphs [7], [8], [9], [10], [11]. In the other setting, *single graph mining*, a single graph is used [12], [13], [28]. The main difference between the two settings is in the definition of a support metric. In transactional mining, the support of a subgraph S is simply defined as the number of graphs containing S. On the contrary, several sophisticated anti-monotone support metrics were proposed for the single graph setting [13], [27], [28]. Most of the existing solutions for both settings focus on static graphs.

subgraph matching FSM relies on evaluating the support of candidate subgraphs using one of the expensive subgraph matching algorithms. In static graphs, the first practical solution utilizes a backtracking technique [33]. Since then, several performance enhancements were proposed [34], [35], [36]. Subgraph matching becomes even more challenging due to the emerging use of dynamic graphs. Thus, exact [37] and approximate [31], [38] dynamic indexes were proposed to enhance the performance. Moreover, an approximate distributed solution is proposed to scale to massive graphs [39]. Compared to previous work, our system targets exact results. Also, using generic indexes is impractical for

mining large graphs when a large volume of updates is expected.

Incremental Frequent Itemsets Mining The problem of frequent itemset mining over a stream got much attention in the literature, including approximate [19], [20], [40] and exact solutions [21], [22], [23], [24], [25], [26]. For this work, we are interested in exact solutions. *Moment* [26] is an exact solution that mines closed frequent itemsets. It utilizes a fringe composed of four types of itemsets. One type represents the frequent closed itemsets while the other three represent the boundary between frequent closed itemsets and other itemsets. StreamGen [41] follows a similar approach but it mines frequent itemset generators. An itemset generator is an itemset that does not have a subset having the same frequency. INSTANT [24] and INSTANT+ [22] are exact solutions that support only insertion updates and mine maximal frequent itemsets. CFI-Stream [21] focuses on mining closed frequent itemsets. NewMoment [23] and TMoment [25] extend *Moment* and maintain a set of all frequent closed itemsets as well as all 1-itemsets, thus they are inapplicable to the FSM setting. Although they are similar, it is not straightforward to employ itemset mining techniques in the FSM setting.

The Fringe Concept: When there exists a lattice of objects, it is beneficial to maintain a fringe of a smaller set of objects separating two (or more) distinct subsets of the lattice. For association rules mining in a dynamic setting [42], negative borders are maintained. Then, a full scan of the database is required only if the negative border is changed. In sequence mining, the work in [43] follows a similar approach by utilizing a fringe. As for graph processing, algorithms like Dijkstra [44] and minimum spanning tree make use of the "fringe vertices"; those vertices that are adjacent to already visited vertices.

Dynamic Graph Mining: Different problem definitions have been proposed under the umbrella of dynamic graph mining. An approximate technique for mining nodes that frequently co-occur in a stream of small subgraphs is proposed in [45]. Another definition of dynamic graph mining is adopted by [46], [47], where a snapshot is taken from the input graph. Each edge is labeled with a sequence of labels representing dynamic changes in the edge label over time. This snapshot is then mined to find frequent structures that share similar sequences of edge labels. Another work [48] tries to answer the following question: "Given that a subgraph is currently frequent, which subgraph is expected to be frequent in the future?". For the transactional setting, *IncGraphMiner* [14] mines closed frequent subgraphs. In this setting, the unit of update is a graph. *StreamFSM* [15] is an approximate solution for dynamic FSM. It is based on mining a sampled area around each graph update. This approach supports only batches of addition updates. None of the above techniques tackles the same problem that we propose in this work.

7 Conclusions

In this paper, we study the problem of frequent subgraph mining on evolving graphs. We highlight the importance of the problem and show that current solutions are inapplicable. A novel solution is proposed, it is based on utilizing information collected during previous iterations, such as which parts of the search space to maintain, which nodes of the input graph to postpone, and which subgraph nodes can give quicker results. Such information is exploited to enhance the performance of next iterations. Furthermore, a novel index structure is proposed to improve the efficiency of frequency evaluation. Finally, we discuss how batching can be utilized to further improve the performance. Through extensive experiments on large real-world datasets, we show that *IncGM+* outperforms state-of-the-art static FSM algorithms by up to 3 orders of magnitude. Possible directions for future studies include an investigation of how to scale to larger graphs using parallel computation platforms and approximate solutions.

References

[1] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proc. of SIGMOD*, 2004.

[2] M. Seeland, T. Girschick, F. Buchwald, and S. Kramer, "Online structural graph clustering using frequent subgraph mining," in *Machine Learning and Knowledge Discovery in Databases*. Springer, 2010.

[3] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis, "Frequent substructure-based approaches for classifying chemical compounds," *IEEE KDE*, vol. 17, 2005.

[4] Y.-R. Cho and A. Zhang, "Predicting protein function by frequent functional association pattern mining in protein interaction networks," *IEEE TITB*, vol. 14, no. 1, 2010.

[5] L. Zou, L. Chen, and M. T. Özsu, "K-automorphism: A general framework for privacy preserving network publication," *PVLDB*, vol. 2, no. 1, 2009.

[6] W.-T. Chu and M.-H. Tsai, "Visual pattern discovery for architecture image classification and product image search," in *Proc. of ICMR*, 2012.

[7] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: Mining maximal frequent subgraphs from graph databases," in *Proc. of SIGKDD*, 2004.

[8] S. Ranu and A. K. Singh, "GraphSig: A scalable approach to mining significant subgraphs in large graph databases," in *Proc. of ICDE*, 2009.

[9] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining significant graph patterns by leap search," in *Proc. of SIGMOD*, 2008.

[10] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *Proc. of ICDM*, 2002.

[11] X. Yan and J. Han, "CloseGraph: mining closed frequent graph patterns," in *Proc. of SIGKDD*, 2003.

[12] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *PVLDB*, vol. 7, no. 7, 2014.

[13] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *Data Mining and Knowledge Discovery*, vol. 11, no. 3, 2005.

[14] A. Bifet, G. Holmes, B. Pfahringer, and R. Gavaldà, "Mining frequent closed graphs on evolving data streams," in *Proc. of SIGKDD*, 2011.

[15] A. Ray, L. B. Holder, and S. Choudhury, "Frequent subgraph discovery in large attributed streaming graphs," in *BigMine*, 2014.

[16] D. Cook, L. Holder, S. Thompson, P. Whitney, and L. Chilton, "Graph-based analysis of nuclear smuggling data," *Journal of Applied Security Research*, vol. 4, no. 4, 2009.

[17] W. Eberle, L. Holder, and D. Cook, "Identifying threats using graph-based anomaly detection," in *Machine Learning in Cyber Trust*. Springer, 2009.

[18] P. Anchuri, M. J. Zaki, O. Barkol, R. Bergman, Y. Felder, S. Golan, and A. Sityon, "Graph mining for discovering infrastructure patterns in configuration management databases," *Knowledge and information systems*, vol. 33, no. 3, 2012.

[19] D. Lee and W. Lee, "Finding maximal frequent itemsets over online data streams adaptively," in *Proc. of ICDM*, 2005.

[20] J. X. Yu, Z. Chong, H. Lu, and A. Zhou, "False positive or false negative: mining frequent itemsets from high speed transactional data streams," in *Proc. of VLDB*, 2004.

[21] N. Jiang and L. Gruenwald, "Cfi-stream: mining closed frequent itemsets in data streams," in *Proc. of SIGKDD*, 2006.

[22] H. Li, N. Zhang, and Z. Chen, "A simple but effective maximal frequent itemset mining algorithm over streams," *Journal of Software*, vol. 7, no. 1, 2012.

[23] H.-F. Li, C.-C. Ho, and S.-Y. Lee, "Incremental updates of closed frequent itemsets over continuous data streams," *Expert Systems with Applications*, vol. 36, no. 2, 2009.

[24] G. Mao, X. Wu, X. Zhu, G. Chen, and C. Liu, "Mining maximal frequent item sets from data streams," *Journal of Information Science*, 2007.

[25] F. Nori, M. Deypir, and M. H. Sadreddini, "A sliding window based algorithm for frequent closed itemset mining over data streams," *Journal of Systems and Software*, vol. 86, no. 3, 2013.

[26] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Moment: Maintaining closed frequent itemsets over a stream sliding window," in *Proc. of ICDM*, 2004.

[27] M. Fiedler and C. Borgelt, "Subgraph support in a single large graph," in *Proc. of ICDMW*, 2007.
[28] B. Bringmann and S. Nijssen, "What is frequent in a single graph?" in *Proc. of PAKDD*, 2008.
[29] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Proc. of SIGKDD*, 2004.
[30] R. Zafarani and H. Liu, "Social computing data repository at ASU," 2009. [Online]. Available: http://socialcomputing.asu.edu
[31] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *Proc. of SIGMOD*, 2011.
[32] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy, "Distributed-core view materialization and maintenance for large dynamic graphs," *TKDE*, vol. 26, no. 10, 2014.
[33] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of ACM*, vol. 23, 1976.
[34] J. J. McGregor, "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms," *Information Sciences*, vol. 19, 1979.
[35] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *Proc. of SIGMOD*, 2008.
[36] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, 2012.
[37] J. Yang and W. Jin, "Br-index: An indexing structure for subgraph matching in very large dynamic graphs," in *Proc. of SSDBM*, 2011.
[38] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *Proc. of ICDE*, 2009.
[39] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *Proc. of ICDE*, 2014.
[40] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," *Next generation data mining*, 2003.
[41] C. Gao and J. Wang, "Efficient itemset generator discovery over a stream sliding window," in *Proc.of CIKM*. ACM, 2009.
[42] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka, "An efficient algorithm for the incremental updation of association rules in large databases." in *Proc. of SIGKDD*, 1997.
[43] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas, "Incremental and interactive sequence mining," in *Proc. of CIKM*, 1999.
[44] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, 1959.
[45] C. C. Aggarwal, Y. Li, P. S. Yu, and R. Jin, "On dense pattern mining in graph streams," *PVLDB*, vol. 3, no. 1-2, 2010.
[46] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm, and K. M. Borgwardt, "Frequent subgraph discovery in dynamic networks," in *Proc. of MLG*, 2010.
[47] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther, "Pattern mining in frequent dynamic subgraphs," in *Proc. of ICDM*, 2006.
[48] P. Desikan and J. Srivastava, "Mining temporally evolving graphs," in *Proc. of the WEBKDD Workshop*, vol. 22, 2004.

Ehab Abdelhamid received his B.S. degree in computer science from Cairo University, Egypt. He is currently working toward the Ph.D. degree in computer science at KAUST, Saudi Arabia. His primary research interest lies in mining massive graphs using single-threaded and distributed environments. Before starting his Ph.D., he was working on text mining and information retrieval.

Mustafa Canim received his B.S. degree in computer science from Bilkent University, Ankara, Turkey, and Ph.D. degree in computer science from the University of Texas at Dallas, Richardson, Texas, USA. He is currently a Research Staff Member at IBM T.J.Watson Research Center, Yorktown Heights, New York, USA. Before joining the Ph.D. program, he was a Senior Researcher with the Database Research Group of TUBITAK Research Center. His research interests include big data analytics, cognitive computing, graph mining and social network analysis.

Mohammad Sadoghi is a professor of Computer Science at the University of California, Davis. Previously, he was an Assistant Professor at Purdue University. Prior to joining academia, he was a Research Staff Member at IBM T.J. Watson Research Center for nearly four years. He received his Ph.D. from the Computer Science Department at the University of Toronto in 2013. His research spans all facets of massive-scale data management that now demand a careful re-examination in light of Big Data. Yet his ultimate vision lies in fundamentally rethinking the foundation of relational databases for future hardware and computing platforms by re-imagining the query, transaction, and storage models in order to sustain the unprecedented scale of data proliferation and heterogeneity observed in the Big Data era. He has over 45 publications in the leading database conferences and journals and has filed over 30 U.S. patents. He has served as the PC Chair (Industry Track) for ACM DEBS'17 and co-chaired a new workshop, entitled Active, at both ICDE and Middleware.

Bishwaranjan Bhattacharjee is a Senior Technical Staff Member (STSM) working in the Database Research Group at IBM T.J. Watson Research Center. His interests are in new research directions in data management and its applications. In particular he is interested in scalable database processing, clustering and indexing techniques, query processing and optimization, compression, information integration, access control and privacy protection and data management in new hardware. Prior to joining IBM Research, he was associated with the Database Technology Group (DBT) at IBM Toronto Labs, Canada and the Advanced Numerical Research and Analysis Group (ANURAG) of the Ministry of Defence, Government of India, India.

Yuan-Chi Chang is a Research Staff Member and Manager in Software Research at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, USA. His research interest is in the areas of data management, including data server, integration, analytics and big data. He received his Ph.D. and M.S. degrees from University of California, Berkeley and B.S. degree from National Taiwan University. He is a senior member of IEEE and a member of ACM.

Panos Kalnis is professor and chair of the Computer Science program in the King Abdullah Univ. of Science and Technology (KAUST). In 2009 he was visiting assistant professor in the CS Dept., Stanford University. Before that, he was assistant professor in the CS Dept., National University of Singapore (NUS). From 2013 to 2015 he was associate editor for TKDE. Currently, he serves on the editorial board of the VLDB Journal and the Data Science and Engineering Journal. He received his Diploma from the Computer Engineering and Informatics Dept., Univ. of Patras, Greece in 1998 and his PhD from the Computer Science Dept., Hong Kong Univ. of Science and Technology (HKUST) in 2002. His research interests include Big Data, Cloud Computing, Parallel and Distributed Systems, Large Graphs and Long Sequences.