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 \( \tau \). Besides being crucial for graph analysis, FSM is a basic building block of many applications in multidisciplinary domains, such as graph indexing [1], clustering [2] and 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
information resource about the largely undocumented IT practices of a large organization, and thus mining the CMDB 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. The 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 MNI 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 MNI 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 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 \( \tau \). 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: \( \{u_{11}, u_{21}, u_{23}, u_{17}, u_{46}, u_{11}, u_{14}\} \). Let \( S_2 \) be subgraph ‘A’—‘B’ (i.e., a supergraph of \( S_1 \)). The set of embeddings of \( S_2 \) is: \( \{(u_{11}, u_{22}), (u_{21}, u_{19}), (u_{17}, u_{18}), (u_{17}, u_{16}), (u_{23}, u_{19}), (u_{14}, u_{12}), (u_{11}, u_{12}), (u_{8}, u_{9})\} \). \( S_2 \) 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:

\[ \text{Support} = \frac{\text{Number of embeddings}}{\text{Total number of embeddings}} \]
Definition 3 Given an input graph \( G = (V, E) \) and a subgraph \( S = (V_s, G_s) \). Let \( F(v) = \{ u \mid u \in v, u \text{ is a valid mapping of } v \text{ in at least one embedding of } S \text{ in } G \} \). The minimum image based support (MNI) of \( S \) in \( G \), denoted by \( \text{Supp}(S, G) \), is defined as \( \text{Supp}(S, G) = \min \{ \ell \mid \ell = |F(v)| \forall v \in V_s \} \).

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 \( \text{Supp}(S, G) \); 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, \( F(v_1) = \{ u_1, u_2, u_17 \}, F(v_2) = \{ u_2, u_19, u_16 \} \) 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 \( \tau \) 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 the FSM task is defined as follows:

Definition 4 Given a static graph \( G \) and support threshold \( \tau \), the FSM result set \( R \) is defined as: \( R = \{ S_{1}, \ldots, S_{n} \} \), where each \( S_{i} \in R \) has \( \text{Supp}(S_{i}, G) \) greater than or equal to \( \tau \), and there is no subgraph \( S_{k} \notin R \) with \( \text{Supp}(S_{k}, G) \) greater than or equal to \( \tau \).

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:

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 \( S_{j} \) is subgraph of \( S_{i} \).

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 \( \tau \), the problem of frequent subgraph mining in evolving graph \( G_{D} \) is to continuously report the result set \( R_{t} = \{ S_{1}, \ldots, S_{n} \} \), where each \( S_{i} \in R_{t} \) has \( \text{Supp}(S_{i}, G_{D}) \geq \tau \) 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
useless to evaluate a subgraph that is not expected to be affected
its embeddings. In order to alleviate this overhead, it is important
whether it is frequent or not), requires significant overhead to find
continuous itemset mining [26], evaluating a subgraph (i.e., finding
focusing on the fringe subgraphs. Since they are the most sensitive
are then used to enhance or to avoid the evaluation of these
In this section, we propose
IncGM+
S
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 \textsc{evaluate}. This function decides whether \( S \) is frequent or not by searching for enough embeddings of \( S \) in \( G \) to satisfy \( \tau \). Those found to be frequent are extended and added to \textit{candids} (Line 9). This process repeats until \textit{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, \textsc{evaluate} inculs 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 \textsc{updatefringe} (Lines 19 and 25). For a subgraph \( S_{\text{freq}} \) that is recently found to be frequent, \textsc{updatefringe} updates the fringe by: (1) Adding \( S_{\text{freq}} \) to MFS, (2) Removing \( S_{\text{freq}} \) from MIFS, and (3) extending \( S_{\text{freq}} \) by joining it with other frequent subgraphs of the same size [13]. The extended subgraphs are
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2743075, IEEE Transactions on Knowledge and Data Engineering

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. XX, NO. X, MONTH XXXX 5

Input: $G$ the input graph, $\tau$ support threshold, updates graph updates

Output: fringe.MFS

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

Algorithm 1: FRINGE-BASED INCREMENTAL FSM

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

<table>
<thead>
<tr>
<th>Dataset/\tau</th>
<th>Citeseer/120</th>
<th>Patents/18k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg(MFS)</td>
<td>235</td>
<td>77</td>
</tr>
<tr>
<td>Avg(INFREQ)</td>
<td>52437</td>
<td>21</td>
</tr>
</tbody>
</table>

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.

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. $\text{IncGM+}$ is proposed to benefit from maintaining a carefully selected set of embeddings to improve the efficiency. $\text{IncGM+}$ 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 $e = u_2 \ldots u_8$ is added to the input graph, \text{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 $e = u_2 \ldots u_8$. 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. $\text{IncGM+}$ limits memory consumption by adopting the following guidelines:

- For each subgraph $S \in$ MIFS, $\text{IncGM+}$ stores minimal number of embeddings that make $\text{Supp}(S, G) \geq \tau$, other embeddings are not maintained. Each embedding contributes to at least one distinct cell in the $\text{MNI}_\text{sub}$, 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 $\tau$ is: $\tau \cdot |S|$, where $|S|$ is the number of nodes in subgraph $S$ (i.e., the number of $\text{MNI}$ columns). Recall the example of Figure 1 when $\tau = 3$. Only 3 embeddings were required to satisfy $\tau$.

- For each subgraph $S \in$ MIFS, $\text{IncGM+}$ stores up to $\tau$ embeddings for each of its $\text{MNI}$ 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 $\tau$. In practice, the number of stored embeddings for an infrequent subgraph $S$ is: $T \cdot |S|$, where $T < \tau$.

By utilizing the above guidelines, the number of stored embeddings for both MIFS and MIFS is bounded by $\tau$. Table 1 shows the results of an empirical study to measure the average number of embeddings for subgraphs either belonging to MIFS.

or MFS. The experiment utilizes 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 the stored 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+ only needs to search 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 needs to search 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 that 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 \(\tau\), 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 \(\tau\) after an edge addition (line 15). For such case, calling EVALUATE is required to look for other embeddings that were not discovered before (Line 16).

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 node. 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, embedding \(e_2\) 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

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

Algorithm 2: Embeddings-based FSM
```
to a specific node \( e \in S \) and is populated with distinct matching nodes \( e \in G \). There is a counter value attached to each cell representing the number of embeddings indexed by the node 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:** \textit{FELS} supports efficient addition and removal of embeddings. For a new embedding, its key is used to check whether it already exists. \textit{FELS} does not allow multiple entries for the same embedding. Adding a new embedding involves adding it to the embeddings list, adding its node to the inverted index, and populating the \textit{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 \textit{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 \textit{MNI} entries are either decremented or removed. Assume \( e_5 \) is removed from the example \textit{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 \textit{MNI} table are deleted and the value associated with \( u_{12} \) is decremented and becomes 1.

**MNI Computation:** \textit{FELS} utilizes the stored embeddings to compute the \textit{MNI}-based support value. This is done by checking the length of each \textit{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 \textit{MNI} table is 5. Suppose that edge \( u_{11} \rightarrow u_{12} \) is deleted from the input graph. Then embedding \( e_5 \) becomes obsolete and is removed from the inverted index and the \textit{MNI} table. The new set of embeddings becomes: \( \{e_1, e_2, e_3, e_4, e_6\} \). By consulting the \textit{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 \textit{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 \textit{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, \textit{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 \( \tau \). 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, \textit{IncGM} starts by checking invalid columns. As such, the redundant overhead of checking \textit{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. Repeated similar updates are grouped and processed once. For instance, when adding an edge \( u_i \rightarrow u_j \) more than once, only one update is considered. Second, edges that cancel each other are ignored (e.g., edge is added then deleted within the same batch). The third optimization is 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 updates batch and add them to the \textit{ToBeChecked} list. Then, the
Proposition 4

Given $S$. As such, there is no need to maintain and process $S$.

Proposition 3

Given $S$. Then, $S$ can be safely removed from ToBeChecked.

Proof: Given that $S$ is an infrequent subgraph, and $S$ is a 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 $S$, $S$ 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:

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

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

Score($S$) is an estimate of the difference between support of $S$ after current graph updates ($\text{Supp}(S,G_D)$) and the required threshold $\tau$. A small score value indicates more chances for a subgraph $S$ to change its status. An estimate, $\hat{\text{Supp}}(S,G_D)$, is used for calculating expected support since the new exact support is unknown before graph updates evaluation. $\alpha_S$ is support of $S$ before current batch of update, $\#\text{Edges}$ is the number of edge addition in current batch and $\beta_S$ is the expected increase in the support of $S$ for each edge addition. $\beta_S$ is estimated based on statistics collected during previous iterations; $\beta_S=\text{median}(L_S)$, 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

In this section, we experimentally compare 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+ is the extension of IncGM that maintains a minimal number of embeddings and utilizes 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 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:** This graph models the Twitter social network. Each node represents a user and edges represent users following other users. The original graph does not have node labels, so each node is assigned a random label from a pool of 25 labels based on Gaussian distribution.

**Patents:** 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:** 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:** This graph is a citation network. Each node is labeled with a random label from a pool of 25 labels based on Gaussian distribution.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Density</th>
<th>Nodes</th>
<th>Edges</th>
<th>Distinct node labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>Dense</td>
<td>11366811</td>
<td>85331346</td>
<td>425</td>
</tr>
<tr>
<td>Patents</td>
<td>Dense</td>
<td>29239322</td>
<td>13968441</td>
<td>418</td>
</tr>
<tr>
<td>Yahoo</td>
<td>Dense</td>
<td>993034</td>
<td>903113</td>
<td>5576</td>
</tr>
<tr>
<td>CiteSeer</td>
<td>Medium</td>
<td>3312</td>
<td>4732</td>
<td>6</td>
</tr>
</tbody>
</table>

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 and 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 \( \tau \) values (i.e., Figure 6.B.1 for \( \tau = 160 \)). For larger graphs and lower \( \tau \), 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 \( \tau \) 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, it cannot finish any task for Twitter. 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 \( \tau \) 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 $\tau$ 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 $\tau = 110k$.

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 based on how much pruning can be applied. 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
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 \( \tau \) value and the type of graph updates. We conduct an experiment to measure how these factors affect the number of fringe updates. For Citeseer, the used frequency threshold (\( \tau \)) is varied. Table 3 shows the average number of graph updates per one fringe update. When \( \tau \) is small, \textit{IncGM+} becomes more sensitive to graph updates and more fringe subgraph are affected by the updates. For Patents, when deleting 9M edges and using \( \tau = 500 \), we see less fringe changes per a graph update. 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 is 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 each set: \textit{MFS} and \textit{MIFS} for the different datasets and different \( \tau \) values. It is clear that the size of each set is not extremely large, and they do not rapidly get bigger as \( \tau \) decreases. Moreover, the size of \textit{MFS} is always smaller than the size of \textit{MIFS}. Another interesting observation is that the size of \textit{MIFS} is affected by the number of distinct labels in the input graph. \textit{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 \textit{MIFS}. Also, Yahoo has the smallest number of distinct label and, accordingly, has the largest \textit{MIFS}.

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

Figure 9 also highlights the memory consumption of our proposed techniques. In comparison with \textit{FullRecomp}, \textit{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 \textit{FullRecomp}).

While for larger graphs, the increase is relatively smaller (Figures 9.2-9.4). Although \textit{IncGM+} maintains a list of embeddings, its extra memory usage is insignificant compared to \textit{IncGM}. This is due to our approach that carefully maintains a limited number of embeddings. Note that Figure 9 shows time in log scale, small differences are difficult to notice. Numbers show some minor differences between the memory consumption of \textit{IncGM} and \textit{IncGM+}. For example, when mining the Patents graph at \( \tau = 22K \), \textit{IncGM} consumes an extra memory of 36MB, 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 \textit{IncGM} and \textit{IncGM+} becomes more significant. In Figure 6.A.2, for \( \tau \) greater than 18k, there is no improvement in the efficiency of \textit{IncGM+}. This is because no embeddings are utilized when \( \tau \) is greater than 18k which is confirmed by the memory consumption (Figure 9.2). But when \( \tau = 18k \), differences between \textit{IncGM} and \textit{IncGM+} in both memory consumption and efficiency become clearer.

### 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 \textit{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 \textit{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 \textit{IncGM+} and batching using \textit{FullRecomp}. Figures 10.b and 10.c compares batching on \textit{FullRecomp}, \textit{IncGM} and \textit{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 \textit{FullRecomp} in both settings. The difference is more significant in the deletions workload and as the problem becomes more expensive (i.e., using lower \( \tau \) value). Figure 10.d shows the speedup of incremental batching compared to \textit{FullRecomp} batching when batch size is changed while \( \tau \) is fixed. Both systems process additions workload on Patents and 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 \textit{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...
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 $\tau = 10^5$.

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

<table>
<thead>
<tr>
<th>Dataset/(\tau)</th>
<th>Citeseer/120</th>
<th>Patents/18k</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoBatching</td>
<td>121</td>
<td>3125</td>
</tr>
<tr>
<td>Opt1</td>
<td>86</td>
<td>488</td>
</tr>
<tr>
<td>Opt2</td>
<td>18</td>
<td>396</td>
</tr>
</tbody>
</table>

grouping and subgraph pruning optimizations. Table 5 shows the results for Citeseer ($\tau=120$ and batch size = 100) and Patents ($\tau=18K$ 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.

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 an appropriate anti-monotone support metric. In transactional mining, given a subgraph $S$, the support of $S$ is simply defined as the number of graphs containing $S$. On the contrary, single graph mining requires more sophisticated metrics. Several anti-monotone support metrics were proposed for the single graph setting [13], [27], [28]. Compared to our work, most of the existing solutions for both settings focus on static graphs.

subgraph matching FSM relies on evaluating the support of
candidate subgraphs. Such evaluation is performed by subgraph matching algorithms, which are known to be expensive. In static graphs, the first practical algorithm that addresses this problem 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. Exact [37] and approximate [31], [38] dynamic indexes were proposed; the goal is to build an index that is efficiently maintained as the graph is updated. In order to scale, an approximate approach that leverages an existing distributed graph framework is proposed [39]. Compared to previous work, our system targets exact results. Also, a generic index for subgraph matching requires maintenance overhead for each graph update. Such overhead is impractical for a dynamic graph mining system that is required to process large stream of updates on large graphs.

Incremental Frequent Itemsets Mining The research community has given much attention to the problem of frequent itemset mining over data streams. Approximate [19], [20], [40] and exact solutions were proposed [21], [22], [23], [24], [25], [26]. For the proposed problem, we are more 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. Following the same approach, StreamGen [41] is proposed to mine frequent itemset generators. An itemset generator is an itemset that does not have a subset having the same frequency. In StreamGen, three sets are maintained. One is for the frequent itemset generators, and the other two are for the boundary with other itemsets. INSTANT [24] and INSTANT+ [22] are exact solutions that support only insertion updates and mine maximal 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. The itemset mining setting is similar to the FSM setting, though, it is not straightforward to employ these techniques in the FSM setting.

The Fringe Concept: The fringe concept is used in the literature since a long time ago. 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. In data mining, many approaches utilize the fringe concept. 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] utilizes a fringe to improve the performance. Certain portions of the original database are processed based on the affected parts of the 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.

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

IEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. XX, NO. X, MONTH XXXX 14


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. 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 Purdue University. Previously, he was a Research Staff Member at IBM T.J. Watson Research Center. 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, i.e., cloud, by re-imaging 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 40 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 at ICDE’17 titled “Active 17; First International Workshop on Data Management on Virtualized Active Systems”.

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 Tokyo 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.