A Partition-Based Approach to Graph Mining

Junmei Wang  Wynne Hsu  Mong Li Lee  Chang Sheng
National University of Singapore, Singapore 117543
{wangjunm, whsu, leeml, shengcha}@comp.nus.edu.sg

Abstract

Existing graph mining algorithms typically assume that databases are relatively static and can fit into the main memory. Mining of subgraphs in a dynamic environment is currently beyond the scope of these algorithms. To bridge this gap, we first introduce a partition-based approach called PartMiner for mining graphs. The PartMiner algorithm finds the frequent subgraphs by dividing the database into smaller and more manageable units, mining frequent subgraphs on these smaller units and finally combining the results of these units to losslessly recover the complete set of subgraphs in the database. Next, we extend PartMiner to handle updates in the dynamic environment. Experimental results indicate that PartMiner is effective and scalable in finding frequent subgraphs, and outperforms existing algorithms in the presence of updates.

1 INTRODUCTION

Research on pattern discovery has progressed from mining frequent simple patterns to mining complex structured patterns, such as trees and graphs. As a general data structure, graphs can model arbitrary relations among objects. Existing graph mining algorithms [6, 8, 16, 10, 15] typically assume that the graphs in the databases are relatively static and simple, that is the number of possible labels in the graphs is small. They do not scale well for mining graphs in a dynamic environment. For example, in spatiotemporal applications, the complex relationships in spatiotemporal data can be modeled as graphs. In general, a spatiotemporal database can contain millions of different structures. Changes to the spatiotemporal databases will cause changes to the graph structures that model the relationships in the spatiotemporal data. Re-execution of the mining algorithm each time the graphs are updated is costly and may result in an exponential growth in the computational time and I/O resources. Consequently, there is an urgent need to find an algorithm that is scalable and can incrementally mine from only parts of the graph databases that have been changed.

In this paper, we propose a partition-based approach for graph mining. Our idea is to isolate the changes to a small set of subgraphs and re-execute the graph mining algorithm only on the isolated subgraphs. Instead of finding frequent subgraphs on the graphs that are typically large and complex, we recursively partition the graphs into smaller, more manageable subgraphs until these subgraphs can fit into the main memory. With this approach, existing memory-based graph mining algorithms can be utilized to discover frequent patterns in these subgraphs. The discovered patterns are joined via a merge-join operation to recover the final set of frequent patterns that exist in the original graphs. Experimental results show that PartMiner is comparable with the state-of-the-art graph mining algorithm ADIMINE [15] for static graph databases and in the presence of updates, it outperforms ADIMINE by a few orders of magnitude. Our contributions are listed as follows:

- We design a partition-based algorithm to divide the graphs into \( k \) subgraphs (\( k \) is determined by the size of main memory) with the goal of reducing the connectivity among the subgraphs while localizing most of, if not all, the updated nodes to a minimal number of subgraphs. We also develop a merge-join operation to losslessly recover the complete set of frequent subgraphs in the database from the set of subgraphs found in the partitions. We give a theoretical proof to ensure that mining of frequent subgraphs in the partitions will be equivalent to mining in the original graph database.

- We develop a partition-based graph mining algorithm, called PartMiner. PartMiner is inherently parallel in nature and makes use of the cumulative information obtained during the mining of previous subgraphs to effectively reduce the number of candidate graphs. We also extend PartMiner to handle updates in the graph database. The IncPartMiner makes use of the pruned results of the pre-updated database to eliminate the generation of unchanged candidate graphs, thus leading to tremendous savings.

The paper is organized as follows. Section 2 reviews the related work. Section 3 gives some preliminary concepts,
and we present the partition-based graph mining approach in Section 4. Section 5 reports the experimental results, and we conclude the paper in Section 6.

2 Related Work

[7] first study the techniques to partition the graphs in the database and develop the software package METIS. The algorithms in METIS are based on multilevel graph partitioning and reduce the size of the graph by collapsing vertices and edges, partition the smaller graphs, and then uncoarsen it to construct a partition for the original graph.

Research in graph mining includes [7, 6, 8, 16, 10, 15], [6, 8] introduce Apriori-like algorithms AGM and FSG to mine the complete set of frequent graphs. Both algorithms are not scalable since they require multiple scans of the databases and tend to generate many candidates during the mining process. [16, 10] devise depth-first graph mining approaches, gSpan and Gaston. These approaches are essentially memory-based and their efficiencies decrease dramatically if the graph database is too large to fit into the main memory. Recognizing this problem, [15] design an algorithm, called ADIMINE for mining graphs on large, disk-based databases. An effective index structure called ADI is proposed to facilitate major graph mining operations. However, this solution does not work well when the graph database is still evolving. This is because the ADI structure has to be rebuilt each time the graph database is being updated. Recent work [17, 5, 3, 18] also examines algorithms/methods to discover different types of patterns in graphs. As special cases of graphs, tree patterns have also been the focus of [19, 2, 13, 14].

To achieve scalability, partition-based approaches have been employed to discover frequent patterns in databases, such as association rules mining[11], classification[4, 12], clustering [1, 9], etc. The data partitioning approach involves splitting a dataset into subsets, learning/mining from one or more of the subsets, and possibly combining the results. To date, there has been no work on mining of frequent subgraphs using the partition-based method.

3 PRELIMINARY CONCEPTS

We represent a labeled graph by \( G = (V, E, L_V, L_E) \) where \( V \) is the set of vertices; \( E \) is the set of edges denoted as pair of vertices; \( L_V \) is a set of labels associated with the vertices; and \( L_E \) is a set of labels for the edges. A graph \( G \) is connected if a path exists between any two vertices in \( V \). The size of a graph is the number of edges in it, and a graph \( G \) with \( k \) edges is called a graph of size \( k \) or \( k \)-edge graph.

A graph \( G_1 \) is isomorphic to a graph \( G_2 \) if there exists a bijective function \( f : V_1 \rightarrow V_2 \) such that for any vertex \( u \in V_1 \), \( f(u) \in V_2 \) and \( L_u = L_{f(u)} \), and for any edge \((u, v) \in E_1, (f(u), f(v)) \in E_2 \) and \( L_{(u,v)} = L_{(f(u), f(v))} \).

An automorphism of a graph \( G \) is an isomorphism from \( G \) to \( G \). A subgraph isomorphism from \( G_1 \) to \( G_2 \) is an isomorphism from \( G_1 \) to a subgraph \( G_2 \), and \( G_1 \) is called a supergraph of \( G_2 \), denoted as \( G_2 \subseteq G_1 \).

A graph database is a set of tuples \((gid, G)\), where \( gid \) is a graph identifier and \( G \) is an undirected labeled graph. Given a graph database \( D \), the support of a graph \( G \) is the number of graphs in \( D \) that are supergraphs of \( G \).

To find all the frequent subgraphs in a database, we need to encode the structure of a graph such that if two graphs have identical encoding, then they are isomorphic. We use the method proposed in [16] to encode a graph. The method in [16] performs a depth-first search on a graph to order the vertices and construct the DFS-tree \( T \) of \( G \). Based on the DFS-tree \( T \), we can further order the edges in a graph \( G \) and encode \( G \) with the DFS code. Since a graph can have many different DFS-trees, [16] define the notion of the minimum DFS code, which is the minimum of all the DFS codes of a graph \( G \), to encode the graph.

Figure 1 shows a graph \( G \), and three DFS trees of \( G \), together with their corresponding DFS codes. The code \( (G, T_1) \) in Figure 1(b) is the minimum DFS code.

4 Partition-based Graph Mining

Figure 2 shows the framework of the proposed partition-based graph mining approach. It consists of two phases. In the first phase, we iteratively call a graph partitioning algorithm to partition each graph in the database into smaller subgraphs. Then we group the subgraphs into units.

The second phase applies an existing memory-based graph mining algorithm to discover the frequent subgraphs in each unit. The set of frequent subgraphs in each unit are then merged via a merge-join operation to recover the complete set of frequent subgraphs. The proposed framework can be easily extended to handle incremental mining when updates occur in the graph database (see Section 4.5).
4.1 Dividing Graph Database into Units

The motivation for the proposed partition-based graph mining approach is to effectively deal with the graphs in the presence of frequent updates. By partitioning the graphs, we can reduce the graph complexity as well as the size so that existing memory-based graph mining algorithms can be applied. However, the frequent subgraphs obtained from each unit need to be combined using a merge-join operation which is expensive. To minimize the number of units involved in the merge-join operation, it is important to minimize the connectivity (i.e. the number of connective edges) among the subgraphs in the units. Moreover, it is also important to isolate those vertices and edges that are changed frequently and localize them to a minimal number of units to reduce the number of units that need to participate in the incremental mining process.

To achieve the goal of minimizing connectivity among the units, each graph in the database must be carefully partitioned and organized into units. If we randomly partition the graphs and group them into units, then the connectivity among the subgraphs in the units will not be clear, and a merge-join operation will be needed on every pair of the units. Instead, we have adopted an approach that repeatedly bi-partitions each of the graphs in the database.

Figure 3 shows a graph $G$ which is first divided into 2 subgraphs $G_1$ and $G_2$. $G_1$ ($G_2$) is again further divided into 2 subgraphs $G_{11}$ and $G_{12}$ ($G_{21}$ and $G_{22}$). This bi-partitioning process yields a total of 4 subgraphs for $G$. By applying this bi-partitioning procedure on each of the graphs $G_i$ in the database, we will have four subgraphs $G_{ij}$, $1 \leq j \leq 4$, will be grouped into one unit $U_j$.

The bi-partitioning approach facilitates the recovery of the complete set of graphs in a database from the subgraphs in the units. In our example, we just need to combine the set of subgraphs in $U_1$ and $U_2$ to get the set of subgraphs in $U_{12}$, and the set of subgraphs in $U_3$ and $U_4$ to get the set of subgraphs in $U_{34}$. The set of subgraphs in $U_{12}$ and $U_{34}$ are subsequently combined to obtain the original graph database. This also indicates the sequence of combining the frequent subgraphs mined in each unit to obtain the final set of frequent subgraphs for the database.

We use two criteria to carry out the bi-partitioning. The first criteria is to minimize the connectivity between the subgraphs, and the second criteria is to isolate the frequently updated vertices to a subgraph. Figure 4 illustrates how a graph $G$ can be partitioned using these two criteria. Note that subgraphs should include the connective edges between the subgraphs so that we can recover the original graph later. For example, the edges $(v_1, v_2)$ and $(v_3, v_4)$ in Figure 4(a), and edges $(v_5, v_6)$ and $(v_6, v_7)$ in Figure 4(b) are connective edges.

We associate each vertex $v$ in a graph $G$ with a value $v.u.freq$ to indicate its update frequency, denoted as $v.u.freq$. The vertices of $G$ are sorted in descending order according to their update frequencies. Suppose that the vertex set $V$ of the graph $G$ is divided into two subsets $V_1$ and $V_2$, we define a weight function $w(V_1)$ to reflect the average update frequencies of the vertices in the vertex set $V_1$ and its connectivity to the vertex set $V_2$ (see equation (1)).

$$w(V_1) = \lambda_1 \frac{\sum_{v \in V_1} v.u.freq}{|V_1|} - \lambda_2 |E_{V_1,V_2}|$$

where $E_{V_1,V_2}$ is the set of connective edges $e(v_i, v_j)$ be-
between the vertex sets \( V_1 \) and \( V_2 \), i.e. \( v_i \in V_1, v_j \in V_2 \) or \( v_i \in V_2, v_j \in V_1 \). The first term in \( w(V_i) \) computes the average update frequencies of the vertices in the subset \( V_i \), and the second term counts the number of connective edges. We use two parameters \( \lambda_1 \) and \( \lambda_2 \) to set the weight between these two terms.

Figure 5 shows the graph partitioning algorithm, called GraphPart. Line 1 sorts the vertex set \( V \) of the graph \( G \) according to their update frequency. Let \( v_{c} \) be the centroid of \( V \). Then \( v_{c} \) divides \( V \) into two subsets \( V_1 \) and \( V_2 \), where \( V_1 \) contains the vertices \( v_i \in V \wedge v_i.u.freq \geq v_{c}.u.freq \), and \( V_2 \) contains vertices \( v_j \in V \wedge v_j.u.freq < v_{c}.u.freq \). For each vertex \( v_i \in V_1 \), we traverse the graph \( G \) in depth-first manner to construct the vertex subset \( V_i \), and compute the weight function \( w(V_i) \) (lines 4-12). The vertex set with the largest weight function is the final subset \( V^* \). Note that when scanning the unvisited neighbors of a vertex, the vertex with the highest frequency should be the next visited node (line 21). Finally, we obtain two subgraphs \( G_1 \) and \( G_2 \), \( G_1 \) contains all the vertices in \( V^* \), and \( G_2 \) contains all the vertices in \( V \setminus V^* \). Note that both \( G_1 \) and \( G_2 \) include the connective edges (lines 13-14).

After partitioning each graph in the database into a set of subgraphs, the next step is to group the subgraphs into units such that each unit can fit into the main memory. Figure 6 shows the algorithm to divide the database into units. We use a parameter \( k \) to indicate the number of units that the database will be divided into. The value of \( k \) is determined by the availability of the main memory and the size of the database. For each graph in the database, we repeatedly call

Algorithm GraphPart

Input: \( G \), the graph
Output: \( G_1, G_2 \), the two subgraphs of \( G \)
\begin{enumerate}
    \item \( V = \{ \text{vertices sorted according to their update frequency} \} \)
    \item \( V^* = \emptyset \)
    \item \( u(V^*) = -\infty \)
    \item \( \text{for } i = \emptyset \text{ \ where } i < \lfloor |V|/2 \rfloor \text{ \ do} \)
        \begin{enumerate}
            \item \( V_i = \emptyset \)
            \item \( \text{call DFSScan}(V, i, V_i) \)
            \item Compte \( w(V_i) \)
            \item \( \text{if } \lambda_1(w(V_i)) > \lambda_2(w(V^*)) \text{ \ do} \)
                \begin{enumerate}
                    \item \( w(V^*) = w(V_i) \)
                    \item \( V^* = V_i \)
                \end{enumerate}
        \end{enumerate}
\end{enumerate}

Procedure DFSScan
\begin{enumerate}
    \item \( \text{stack} = \emptyset, m = 0 \)
    \item \( \text{while } \text{stack} \neq \emptyset \text{ \ and } m \leq |V|/2 \text{ \ do} \)
        \begin{enumerate}
            \item \( v = \text{stack}.pop() \)
            \item \( V_i = V_i \cup \{ v \} \)
            \item \( m = m + 1 \)
        \end{enumerate}
    \item \( \text{choose the neighbor vertex } v_{n}, \text{ s.t. } v_{n}.visited = 0 \text{ \ and } \forall v_{j}, v_{j}.visited = 0 \wedge (v_{j}, v_{n}) \in \mathbb{E}, v_{j}.u.freq < v_{n}.u.freq \)
    \item \( \text{stack} = \text{stack}.push(v_{n}) \)
\end{enumerate}

Procedure DBPartition
\begin{enumerate}
    \item \( \text{DivideDBPart}(D, k) \)
    \begin{enumerate}
        \item \( D \), graph database
        \item \( k \), number of units
        \item \( D_{0,0} = D \)
        \item \( i = 1 \)
        \item \( \mathcal{I} = \{ \log_2 k \} \)
        \item \( \text{while } i \leq \mathcal{I} \text{ \ do} \)
            \begin{enumerate}
                \item \( \text{for } j = 0 \text{ \ to } 2^{i-1} \text{ \ do} \)
                    \begin{enumerate}
                        \item \( \text{DivideDBPart}(D_{i,j}, D_{i,j+1}) \)
                    \end{enumerate}
                \end{enumerate}
            \end{enumerate}
        \end{enumerate}
\end{enumerate}

Figure 6. Dividing graph database into units

Algorithm GraphPart to partition it. The subgraphs generated during the partitioning process are kept in the database \( D_{i,j}, 1 \leq i \leq \lfloor \log_2 k \rfloor, 0 \leq j \leq 2^{i-1} \). Finally, the resulting \( k \) subgraphs are distributed to \( k \) units, \( U_1, U_2, \ldots, U_k \).

4.2 Mining Frequent Subgraphs in Units

After dividing the graph database into \( k \) units such that each unit can fit into the main memory, we can now use any existing memory-based algorithms to find the frequent subgraphs in the units.

Many memory-based algorithms have been proposed to discover the frequent subgraphs. In this work, we use the Gaston algorithm [10] to find the set of frequent graphs in the units. The Gaston algorithm is based on the observation that most frequent substructures in practical graph databases are actually free trees, and employs a highly efficient strategy to enumerate the frequent free trees first.

Figure 7 gives an outline of the Gaston algorithm. Line 1 finds all the frequent edges in the database. For each frequent edge \( p \), the algorithm generates the descendants \( G' \) of \( p \) with the set of allowable extended edges \( L \) (lines 4-6). According to the types of \( G' \) and the extended edges, the algorithm will decide to find paths, trees or cyclic graphs in the database (lines 7-14).

4.3 Combining Frequent Subgraphs

At this point, we have computed the set of frequent subgraphs in the units. Now, we need to recover the complete set of frequent subgraphs in the database. We design a merge-join operation to accomplish this. We first illustrate the idea behind the merge-join operation, and then give a theoretical proof to show that the complete set of frequent subgraphs in the database can be losslessly recovered by...
Algorithm Gaston

Input: $U$, one of the units of the database

Output: $P(U)$, the set of frequent subgraphs in $U$

1. $F_1 = \{\text{frequent edges in } U\};$
2. for each $p \in F_1$
   3. $L = \{\text{allowable extended edges of } p\};$
   4. for each allowable extended edge $l \in L$
      5. $G' = \text{Adding } l \text{ to } p;$
      6. $L' = \{\text{allowable extended edges of } G'\};$
      7. if $l$ is a node refinement
         8. if $G'$ is a path
            9. find paths with $G'$ and $L'$;
            10. else
                11. find trees with $G'$ and $L'$;
         12. else
             13. find cyclic graphs with $G'$ and $L'$;
      14. }
   15. }

Figure 7. Outline of Gaston algorithm

performing the merge-join operation on the set of frequent subgraphs found in the units.

Suppose unit $U$ is partitioned into two units $U_1$ and $U_2$. Let $P(U_1)$ and $P(U_2)$ be the set of frequent subgraphs found in the units $U_1$ and $U_2$. We want to recover the set of frequent subgraphs in the unit $U$, i.e., $P(U)$. We first sort the frequent subgraphs in each unit according to their number of edges. We use $P_k(U_i)$ to denote the set of $k$-edge subgraphs in the unit $U_i$.

First, frequent 1-edge subgraphs in the units $U_1$ and $U_2$ are simply merged since they do not share any common connective edges. We denote the resulting set of 1-edge subgraphs as $P^1(U) = P^1(U_1) \cup P^1(U_2)$. Next, we merge frequent 2-edge subgraphs ($P^2(U) = P^2(U_1) \cup P^2(U_2)$) and join the 2-edge subgraphs based on the common connective edges to produce a set of candidate 3-edge subgraphs $C_3^3$.

Following, a subgraph isomorphism check is carried out to remove the infrequent 3-edge subgraphs in $C_3^3$, resulting in the set of frequent 3-edge subgraphs, i.e., $F^3$.

For frequent $k$-edge subgraphs, $k > 2$, we obtain the set of $k$-edge subgraphs by merging $P^k(U_1)$, $P^k(U_2)$ and $F^k$ together, that is $P^{k+1}(U) = P^k(U_1) \cup P^k(U_2) \cup F^k$. Then, the merge-join operation is applied to find the set of candidate $(k+1)$-edge subgraphs $C_{k+1}^{k+1}$. We obtain $C_{k+1}^{k+1}$ in three steps:

Step 1. Join the subgraphs in the set $P^k(U_1)$ with the subgraphs in the set $F^k$ to obtain the candidate set $C_1^{k+1}$.

Step 2. Join the subgraphs in the set $P^k(U_2)$ with those in the set $F^k$ to obtain the candidate set $C_2^{k+1}$.

Step 3. Join the subgraphs in the set $F^k$ with themselves to get the candidate set $C_3^{k+1}$.

The set of candidate subgraphs $C_{k+1}^{k+1} = C_1^{k+1} \cup C_2^{k+1} \cup C_3^{k+1}$. We will remove those infrequent $(k+1)$-edge subgraphs from $C_{k+1}^{k+1}$, resulting in $F_{k+1}$. This process continues until all frequent subgraphs in the original database are discovered.

Figure 8 shows an example of the merge-join operation. Figure 8(a) shows the unit $U$ with one graph $G$ and its two subgraphs $G_1$ in $U_1$ and $G_2$ in $U_2$. The process of recovering $P(G)$ from $P(G_1)$ and $P(G_2)$ is illustrated in Figure 8(b) where the left light grey region marks the set of subgraphs of $G_1$, i.e., $P(G_1)$, and the dark grey region marks the set of subgraphs of $G_2$, $P(G_2)$.

4.3.1 Proof of Completeness

In this section, we prove that the complete set of frequent subgraphs in the database $D$ can be recovered by performing the merge-join operation on the set of frequent subgraphs found in all the units. We first prove that it is possible to recover all subgraphs of a graph $G$ from its partitioned subgraphs. Then we introduce the Apriori property of subgraphs, and proceed to show that the complete set of frequent subgraphs in the original database is able to be losslessly recovered even when the mining is performed on
which share one of the common edges

Figure 9. Base Case

Figure 10. Induction step

the individual smaller units.

**Theorem 1** The set of subgraphs of a graph $G$ (i.e. $\mathcal{P}(G)$) with size of $n$, $n \geq 2$, can be losslessly recovered by recursively applying the merge-join operation on its bi-partitioned subgraphs $G_1$ and $G_2$.

**Proof:** We prove this by induction. Let $H_n$ denote the hypothesis that the set of subgraphs of a graph $G$ of size $n$ can be losslessly recovered from its partitions $G_1$ and $G_2$, $n \geq 2$

**Base Case:** $n = 2$. This is trivially true as shown in Figure 9. Figure 9(a) shows the division of the graph $G$ into two subgraphs $G_1$ and $G_2$. The process to recover the $\mathcal{P}(G)$ from $\mathcal{P}(G_1)$ and $\mathcal{P}(G_2)$ is shown in Figure 9(b).

**Induction Step:** Suppose $H_n$ is true, we want to show that $H_{n+1}$ is also true. If $H_n$ is true, we know that we are able to recover all subgraphs of a graph $G$ of size $n$. Now we have a graph $G'$ of size $n + 1$. We partition graph $G'$ into two subgraphs. Let $G_1$ denote the partition of size $n$, and let $G_2$ denote the partition of size $i - 2$, $3 < i < n$ (see Figure 10). We can recover all subgraphs from $G_1$ (because $G_1$ is of size $n$). Hence, the only missing subgraphs are those involving the edge $(v_1, v_2)$ in $G_2$. These subgraphs are formed by the joining of the subgraphs of $G_1$ and $G_2$, which share one of the common edges $(v_2, v_3), ..., (v_2, v_1)$ marked as grey in Figure 10. This step in fact is included in the merge-join operation. In other words, if $H_n$ is true, then $H_{n+1}$ must be true.

**Theorem 2 (Apriori Property)** If a graph $G$ is frequent, all of its subgraphs are frequent.

**Theorem 3** Let $D$ be a graph database that has been divided into $k$ smaller units $U_i$, $k \geq 2$, $1 \leq i \leq k$. If we know the complete set of frequent subgraphs $\mathcal{P}(U_i)$ in each unit $U_i$, $1 \leq i \leq k$, we can determine the complete set of frequent subgraphs $\mathcal{P}(D)$ in $D$.

**Proof:** Let $H_{k-1}$ denote the hypothesis that the complete set of frequent subgraphs of the unit $U$ can be losslessly recovered from the set of frequent subgraphs in its $k-1$ subunits $U_i$ ($1 \leq i \leq k - 1$).

**Base case:** $k = 2$, that is $D$ is divided into two units $U_1$ and $U_2$. Given $\mathcal{P}(U_1)$ and $\mathcal{P}(U_2)$, according to the Theorem 1 and Theorem 2, we can losslessly recover the set of frequent subgraphs in $D$, that is we can get the complete set of frequent graphs $\mathcal{P}(D)$.

**Induction step:** $k > 2$. Suppose $H_{k-1}$ is true, we want to show that $H_k$ is also true. Given the unit $U$ which is divided into $k$ subunits $U_i$ ($0 \leq 0 \leq k$). If $H_{k-1}$ is true, we know that we are able to recover the complete set of frequent subgraphs of the unit $U'$ from the set of frequent subgraphs in its $k - 1$ subunits $U_j$ ($0 \leq j \leq k - 1$). Now we have the set $\mathcal{P}(U')$ and $\mathcal{P}(U_k)$. From the Base case, we can losslessly recover $\mathcal{P}(U)$ from $\mathcal{P}(U')$ and $\mathcal{P}(U_k)$. In other words, if $H_{k-1}$ is true, then $H_k$ must be true.

From Theorem 3, we find that the completeness of the frequent subgraphs in the database $D$ depends on the completeness of the frequent subgraphs in the units. The complete set of frequent subgraphs in the units can be found by reducing the minimum support.

**4.4 PartMiner Algorithm**

Figure 11 shows the outline of the PartMiner algorithm. The algorithm takes as input the database $D$, the minimum support $\text{sup}$, and the number of units $k$, and outputs the set of frequent subgraphs in $D$. Algorithm PartMiner works in two phases. In the first phase, it divides the database $D$ into set of units with proper and manageable size(line 1). In the second phase, PartMiner first calls the algorithm Gaston to find the set of frequent subgraphs in the $k$ units with the support threshold $\text{sup}/k$ (lines 2-17). The reason that we use the decreased support for mining the units is to guarantee that the subgraphs that are frequent in the original database are also frequent in the units. After mining the units, line 14 recursively calls the procedure $\text{MergeJoin}$ to combine the results of $D_{i,j}$ and $D_{i,j+1}$ ($0 \leq i < \log_2 k, 0 \leq j < k$) together. The process continues until the set of frequent subgraphs of $D$ (i.e. $\mathcal{P}(D)$) is found.

**4.5 Handling Graph Updates**

The motivation for the proposed approach is to effectively deal with graphs with updates. It is important to isolate those vertices and edges that are changed frequently into a small set of subgraphs so that the number of subgraphs that need to participate in the incremental mining process is minimized.
Algorithm PartMiner

Input: D, graph database; s_up: minimum support; k: number of units
Output: P(D): set of frequent subgraphs in D.

/*Phase 1: dividing the database into k units*/
1: DBPartition(D, k);
/*Phase 2: combining the results of k units*/
2: l = \log_2 k;
3: i = l + 1;
4: for j = 0; j < k - 2; j++
5: Mining U_{j+1} using Gaston;
6: P(D_{i-1,j}) = MergeJoin(D_{i-1,j}, P(U_{j+1}), P(U_{j+1}), \frac{1-s_up}{2})
7: i = i - 1;
8: while (i > 0)
9: for j = 0; j < 2^l; j++
10: Mining D_{i,j} and D_{i,j+1} using Gaston;
11: S = D_{i-1,i};
12: P(S) = MergeJoin(S, P(D_{i,j}), P(D_{i,j+1}), \frac{1-s_up}{2})
13: i = i - 1;
14: }

Procedure MergeJoin

Input: S, the database.
P(S_1), set of frequent subgraphs in S_1;
P(S_2), set of frequent subgraphs in S_2;
sup, minimum support.
Output: P(S), the set of frequent subgraphs in the dataset S.
1: P^1(S) = \{ frequent 1-edge subgraphs in S \};
2: P = P(S_1) \cup P(S_2) \cup P^1(S);
3: Pruning graphs in P(S_1) and P(S_2) with P;
4: P^2(S) = P^2(S_1) \cup P^2(S_2);
5: C^3 = \text{Join}(P^3(S_1), P^3(S_2));
6: F^3 = \text{CheckFrequency}(c_3, sup);
7: for (k = 3; F^k \neq 0; k++)
8: P^k(S) = P^k(S_1) \cup P^k(S_2) \cup F^k;
9: C^{k+1} = \text{Join}(P^k(S_1), F^k);
10: C^{k+1} = \text{Join}(P^k(S_2), F^k);
11: C^{k+1} = \text{Join}(F^k, P^k(S_1));
12: C^{k+1} = \text{Join}(F^k, P^k(S_2));
13: F^{k+1} = \text{CheckFrequency}(c_{k+1}, sup);
14: }
15: P(S) = \{ P^k(S) \}

Figure 11: Outline of PartMiner algorithm

Recall that, PartMiner finds the set of frequent subgraphs by first partitioning the database into several units, then mining the set of frequent subgraphs in each of these units, and finally merging the results of the units with the merge-join operation. If we are able to isolate the updated vertices and/or edges of a graph to a small set of subgraphs, we will be able to focus only on this set of subgraphs instead of mining on the entire database each time an update occurs.

Patterns that are affected by the updates can be classified into 3 categories: (1) UF (unchange frequencies): the set of patterns whose frequencies remain unchanged; (2) FI (frequent to infrequent): the set of previous frequent patterns that have become infrequent; and (3) IF (infrequent to frequent): the set of previously infrequent patterns that have become frequent.

Algorithm PartMiner can be easily extended to discover the three sets of patterns, that is UF, IF and FI. The extension idea is as follows: When the database D is updated, for each updated unit U′ (U_i is the original unit before the updates), we re-execute the main memory algorithm to find the set of frequent subgraphs in the updated unit U′, i.e. P(U′). We then compare the set P(U′) against the set P(U_i). If they are different, we do the following:

1. We keep the subgraphs that appear in the set P(U_i) but not in the set P(U′) into the prune set P. For each subgraph in the prune set P, we check to see if it exists in any other P(U_j) (0 ≤ j ≤ k ∧ j ≠ i). If it exists, we remove it from P. Otherwise, we do nothing. Note that P keeps track of all subgraphs that may potentially change from frequent to infrequent.
2. Next, we check the set of subgraphs in the pre-updated database D, i.e. P(D) against the prune set P. For those subgraphs in P(D) that are supergraphs of some graphs in P, we remove them from P(D) and add them to the set F1. The pruned P(D) is denoted as P(D′).

When all of the updated units are checked, we perform a final merge-join operation to obtain the updated results. However, since there are some graphs whose frequencies are not changed during the updates, we do not need to check them. This can result in further optimization.

The IncPartMiner algorithm for handling updates is shown in Figure 12. Line 1 scans the updated database D′ to get the set of frequent edges (i.e. P^1(D′)). Line 2 compares the set of frequent edges in the original database D (i.e. P^1(D)) with the set P^1(D′), and adds the subgraphs that exist in P^1(D) but not in P^1(D′) into the prune set P. Next, for each unit U_i that consists of the updated vertices, we re-execute the Gaston algorithm to mine the set of frequent subgraphs (i.e. P(U′)) and compare it with the set of graphs in the unit U_i, i.e P(U_i). The potentially infrequent subgraphs are added to the prune set P (lines 3-9). Following, line 10 prunes the subgraphs in the set P(D) with the prune set P, which results in the pruned set P(D′). Then lines 11-12 further uses the pruned set P(D′) to prune the candidate graphs when carrying the merge-join operation of the updated results of the units. Finally, the three sets of subgraphs, i.e. UF, FI and IF are output (lines 13-15), where IF consists of the graphs in P(D′) but not in P(D), UF is the set of graphs in P(D′) but not in IF, and FI contains all the graphs G in P(D) such that there is a graph G′ in P that is a subgraph of G.

5 Experimental Study

In this section, we report the performance study of the proposed algorithms. The algorithms are implemented in C++. All the experiments are conducted on P4 2.8GHZ CPU, 2.5GB RAM and 73GB hard disk. The operating system is Redhat Linux 9.0.
Figure 12. Outline of IncPartMiner algorithm

We use the synthetic data generator described in [15]. The data generator takes as input five parameters $D, N, T, I,$ and $L$, whose meanings are shown in Table 1. For example, the dataset $D50kT20N20L200I5$ indicates that the dataset is made up of 50k graphs, the average number of edges in each graph is 20, and there are 20 possible labels and 200 potentially frequent kernels. The average number of edges in the frequent kernels is 5.

We further extend the data generator to handle updates in the database in 3 different ways: (1) Update the vertex/edge labels with existing or new labels. For example, updating the vertices/edges with label $l$ in the graphs to the label $l'$; (2) Add a new edge with existing or new label between two vertices $v_i$ and $v_j$ in the graph $G$. For example, adding an edge with the label $l'$ between the vertices $v_0$ and $v_1$ if there is no edge between them.; and (3) Add a new vertex $v$, and a new edge $(v, v_1)$ with the existing or new labels on the vertex $v_1$ in the graph $G$.

Table 1. Parameters of data generator

<table>
<thead>
<tr>
<th>Para.</th>
<th>Meaning</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>the total number of graphs in the data set</td>
<td>100k - 1000k</td>
</tr>
<tr>
<td>$N$</td>
<td>the number of possible labels</td>
<td>20, 30, 40, 50</td>
</tr>
<tr>
<td>$T$</td>
<td>the average number of edges in graphs</td>
<td>10, 15, 20, 25</td>
</tr>
<tr>
<td>$I$</td>
<td>the average number of edges in potentially frequent graph patterns</td>
<td>2, 3, 4, 5, 6, 7, 9</td>
</tr>
<tr>
<td>$L$</td>
<td>the number of potentially frequent kernels</td>
<td>200</td>
</tr>
</tbody>
</table>

5.1 Performance Study

In this section, we study the performance of the proposed algorithms in the static and dynamic environments. We compare them with ADIMINE [15] algorithm. In the static environment, we generate the databases by fixing the parameter $L$ with the value 200 and $k$ to 2. In the dynamic environment, we update the dataset D50kT20N20L200I5 using 3 different ways, and evaluate the IncPartMiner by varying the percentage of the graphs updated in the database from 20% to 80% with the number of the units in the database fixed at 2.

5.1.1 Effect of Partitioning Criteria

We first study the effect of using different partitioning criteria on the mining process. There are 3 ways to partition graphs: (1) Isolate the updated vertices into the same subgraphs, i.e. $\lambda_1 = 1, \lambda_2 = 0$ (Partition1); (2) Minimize the connectivity between the subgraphs, i.e. $\lambda_1 = 0, \lambda_2 = 1$ (Partition2); and (3) Isolate the updated vertices AND minimize the connectivity, i.e. $\lambda_1 = 1, \lambda_2 = 1$ (Partition3). We also use the METIS approach to partition the graphs before mining the graphs in the database.

Figure 13 shows the results. We observe that the proposed graph partitioning algorithms give a better performance compared to the METIS. Figure 13(a) shows that in static datasets, Partition2 gives the best performance. While Figure 13(b) indicates that Partition3 has the best performance in the dynamic dataset. This is because Partition3 not only reduces the connectivity among the units, but also tries to isolate the updated vertices into minimum number of units and minimizes the number of units needed to be re-mined and re-examined in the merge-join operation.

5.1.2 Varying Minimum Support

Next, we study the performance of the algorithms by varying the minimum support from 1% to 6%. Figure 14 shows the results to find the complete set of the graphs.

Figure 14(a) shows the results in static datasets. Compared to ADIMINE, we observe that PartMiner needs less time to find the complete set of frequent subgraphs when

We obtain the executable ADIMINE from the authors.
the minimum support is greater than 1.5%, and more time to find the complete set of the frequent graphs when the minimum support is less than 1.5%. This is because when the minimum support decreases, more subgraphs become frequent and the subgraphs also become more complex. As a result, PartMiner will need more time to count the frequency of the subgraphs. In contrast, the index structure of the ADIMINE is advantageous at this time.

Figure 14(b) indicates that IncPartMiner is more efficient in finding the new set of frequent graphs compared to ADIMINE and PartMiner. This is due to the pruning techniques IncPartMiner employs, which enable it to check only on those subgraphs that were infrequent but become frequent in the updated database. This results in significant savings. On the contrary, ADIMINE and PartMiner have to re-mine the database to find the subgraphs.

5.1.3 Effect of Number of Units \( k \)

We test the performance of the algorithms by varying the number of units from 2 to 6. We evaluate our algorithms both in the serial mode and the parallel mode. In the serial mode, we measure the aggregate time which is computed by adding the time spent in all the units together. In the parallel mode (with 1 CPU), the units are executed concurrently and we take the maximum of the time spent in the units.

Figure 15 shows that more time is needed to find the complete set of subgraphs in the database as the parameter \( k \) increases. Moreover, Figure 15(a) shows that running PartMiner in parallel is faster than ADIMINE to find the complete set of frequent subgraphs. Figure 15(b) indicates that IncPartMiner runs faster than ADIMINE for mining dynamic datasets both in serial mode and in parallel mode (with 1 CPU). This is due to the ability that IncPartMiner checks only on those subgraphs that were infrequent but become frequent in the updated database.

5.1.4 Scalability

We evaluate the performance of PartMiner by varying the parameters \( T \) and \( D \) of the synthetic data generator. Figure 16 shows that PartMiner scales linearly with the parameters \( T \) and \( D \), and is faster in finding the complete set of the frequent subgraphs compared to ADIMINE.

Figure 16(a) shows the results when the parameter \( T \) varies from 10 to 25. We observe that the PartMiner needs longer time to find the set of frequent graphs as \( T \) increases. This is to be expected since the frequent subgraphs in the final results tend to be more complex as \( T \) increases. The effect of varying the size of database from 50,000 to 1000,000 are shown in Figure 16(b). We observe that PartMiner scales linearly with the size of the database.
Figure 17. Effect of various types of updates

in the database from 20% to 80%. Figure 17(a) shows the results of updating the labels of the vertices/edges to the existing/new labels. Figure 17(b) show the results of adding new edges/vertices with the existing/new labels. The results confirm that IncPartMiner outperforms ADIMINE when mining frequent graphs in dynamic datasets. Moreover, we note that IncPartMiner scales linearly with the size of the graph and the number of labels in the database. This is contributed to the pruning techniques we employ. IncPartMiner only focuses on the patterns in the set $IF$. The partitioning process will handle the patterns in $UF$, and the patterns in $FI$ can be determined from the results of the pre-updated databases.

6 Conclusion

In this paper, we have presented a partition-based algorithm PartMiner for discovering the set of frequent graphs. Each graph in the database is partitioned into smaller subgraphs. PartMiner can effectively reduce the number of candidate graphs by exploring the cumulative information of the units. We also present IncPartMiner, an extended version of PartMiner to handle updates in the graph databases. The IncPartMiner uses the pruning results of the pre-updated databases to eliminate the generation of candidate graphs that remain unchanged. It only checks those subgraphs that were infrequent but tend to be frequent in the updated databases. This has led to tremendous cost savings. The experimental results also verify that PartMiner is effective and scalable in finding frequent subgraphs, and outperforms existing algorithms in the presence of updates.

References


