Graph Compression by Identifying Recurring Subgraphs

Muhammad Ejaz Ahmed*, JeongHoon Lee*, Inhyuk Na*, Sam Son*, Wook-Shin Han**

*Dept. of Creative IT Engineering, POSTECH
**Dept. of Creative IT Engineering/Dept. of Computer Science and Engineering, POSTECH

e-mail: ejaz629@gmail.com, jhlee@dlab.postech.ac.kr, ina@dlab.postech.ac.kr, sosson97@gmail.com, wshan@postech.ac.kr

Abstract

Current graph mining algorithms suffer from performance issues when querying patterns are increasingly massive network graphs. However, from our observation most data graphs inherently contain recurring semantic subgraphs/substructures. Most graph mining algorithms treat them as independent subgraphs and perform computations on them redundantly, which results in performance degradation when processing massive graphs. In this paper, we propose an algorithm which exploits these inherent recurring subgraphs/substructures to reduce graph sizes so that redundant computations performed by the traditional graph mining algorithms are reduced. Experimental results show that our graph compression approach achieves up to 69% reduction in graph sizes over the real datasets. Moreover, the required time to construct the compressed graphs is also reasonably reduced.

1. Introduction

The emergence of bulky graph datasets poses new challenges for graph data mining. For such scenarios, the target graphs are often too large which may severely limit the applicability of current pattern mining algorithms [1]. By analyzing graph structures from various graph databases such as chemoinformatics and scientific graph databases, we observed recurring subgraphs within a single graph. For instance, Fig. 1 shows such recurring subgraphs, denoted by dotted eclipse, in graphs from AIDS and NASA datasets.

In such a scenario, if current pattern mining algorithms are employed, they are required to perform extensive duplicate computations due to these recurring subgraph structures which would result in huge memory and computational resource requirements. As a result, current graph mining algorithms, under certain conditions, fail due to massive computational requirements, e.g., gSpan [1], a popular frequent pattern mining algorithm, fails (when the support is set to 10%) on the NASA dataset containing 36,790 graphs. Meizi et al. in [2] proposed an algorithm PatternTree_iso which reduces edges and nodes so as to reduce graph sizes to speedup traditional mining algorithms operations. In [3], authors proposed a framework ‘Summarize-Mine’ that focuses on data space reduction within transactions to achieve lossless compression by mining randomized summaries for multiple iterations. Akhil et al. in [4] proposed a super node-based approach to compress data graphs by using statistically significant subgraphs.

Motivated by the problems mentioned above, we intend to find recurring subgraphs in a graph so as to compress the data graphs for efficient processing of graph mining algorithms. Once in-graph subgraphs/substructures are discovered, they can be used to simplify the data by replacing the instance of the subgraph/substructure with a
Algorithm 1: MineGraph

Input: A graph \( G = (V, E) \)

Output: A set of connected patterns from \( G \)

1. Count distinct labels of nodes in \( G \)
2. For each node in \( G \)
   if (node’s occurrences count > threshold)
      add that node to list NodesList
   end for-loop
3. While (NodesList is not empty)
   startNode[i] = popped node from NodesList
   perform BFS traversal from startNode[i]
   if (node traversed in NodesList)
      add node to the i\(^{th}\) connected subgraph
      decrement occurrences count for that node
   end-loop
   if (occurrence count is 0)
      remove the node from NodesList
   end-loop
Return the set of connected patterns

pointer to the newly discovered concept. However, followings are few challenges in completing the task: 1) Identifying recurring subgraphs/substructures is itself a computational expensive and an NP hard problem. 2) While compressing data graph, it is to be make sure that data is not lost, i.e., loose less compression. 3) How to process queries over the compressed graphs? Compressing data graphs has many applications including efficient storage of semi-structured databases, efficient indexing, and web information management. However, our goal is to compress data graphs for efficient processing of graphs over current graph mining algorithms.

Our contributions in this study follow as: We propose graph mining algorithm which discover maximally connected components in a graph. Given a graph \( G = (V, E) \) a subgraph \( S = (V', E') \) is a maximally connected component if \( S \) is connected, and for all vertices \( u \) such that \( u \in V \) and \( u \not\in V' \) there is no vertex \( v \in V' \) for which \((u,v) \in E \). By finding such components in a graph, we replace them with a super node to obtain the compressed version of the graph, resulting in significantly reduced graph sizes and almost no redundant computations.

The organization of the paper goes as: In Section 2, we discuss the proposed graph mining algorithm, Section 3 details about simulation results, and Section 4 concludes this study.

2. Algorithm

Given a graph, our objective is to find maximally connected components (subgraphs/substructures) of different sizes. Algorithm 1 enumerate steps for finding maximally connected components in a graph. In the first step, each distinct label’s occurrences of the input graph are counted. If an occurrence of a node’s label is greater than a threshold, the node is assigned to NodesList vector, in step 2. In the third step, a node is popped-out and acts as the start node for the breadth-first-search (BFS) graph traversal process. Note that the start node serves as the start of the i\(^{th}\) subgraph/substructure of the input graph \( G \). During the BFS traversal, if the traversed neighbor nodes exist in the NodesList, their corresponding occurrence count is decremented and the node is added in the i\(^{th}\) subgraph/substructure. When the occurrence count of a node becomes zero, it is removed from the NodesList vector. This process continues until no neighbors of the BFS traversal exist in the NodesList. At this point, the i\(^{th}\) maximal connected component is obtained. Similarly, the next node from the NodesList is popped-out in step 3, which serves as a start of the i+1\(^{th}\) subgraph and the BFS traversal is repeated to obtain the (i+1)\(^{th}\) subgraph. Step 3 of algorithm is repeated until all the nodes from the NodesList are popped-out, and at this point, we obtain variable-sized maximally connected components of \( G \). The output of Algo.
Algorithm 2: Label subgraphs
Input: A set of subgraphs
Output: labels for each pattern
1. Group same-sized subgraphs
2. For Each Group
   if (two subgraphs are identical)
      assign them same label
   else
      assign them different labels
end for-loop
Return labels for subgraphs

Algorithm 3: Cluster Subgraphs
Input: A set of subgraphs $s_{ij}$
Output: Clustered subgraphs
1. $JC$=Calculate $JS(s_{ij}, s_{ij+1})$ using (1) among all
   subgraphs and assign it to the vector $JC$
2. Sort $JC$ in descending order
3. Get $s_{ij}$ and $s_{ij+1}$ corresponding to $JC[0]$
4. Assign label=1 to $s_{ij}$ and $s_{ij+1}$
5. Mark both the subgraphs ($s_{ij}$ and $s_{ij+1}$)
6. for $i=1$ to $|JSs|
   Get subgraphs $s_{ij}$, $s_{ij+1}$ corresponding to $JC[i]$
   if ($s_{ij}$ and $s_{ij+1}$ are not marked)
      Assign label=label+1 to $s_{ij}$ and $s_{ij+1}$
      Mark $s_{ij}$ and $s_{ij+1}$
   if ($s_{ij}$ or $s_{ij+1}$ are marked)
      assign $s_{ij}$ and $s_{ij+1}$ value of the label
   if (all the subgraphs are marked)
      break
return labelled subgraphs

1 is depicted in Fig. 3.

The next step after obtaining connected subgraphs from
Algo. 1 is to assign labels to subgraphs based on their
similarity. Algo. 2 enumerates steps for labeling the obtained
connected subgraphs. The first step is to assign the
same-sized subgraphs to the same group; i.e. the obtained
subgraphs are of sizes 5, 6, and 7. Labels for subgraphs are
shown in Fig. 3.

The subgraphs with the same labels could be potentially
compressed, and only one instance of them would suffice.
To do that, we use the concept of super node, where super
node is a subgraph with the number of instances equal to
the number of identical labels, e.g., in Fig. 3, label 1 is
assigned to two identical subgraphs, hence they could be
merged together, and the nodes which are not common
among those subgraphs could be pointed. This merge will
reduce the number of nodes (compressing), and at the same
time results in efficient computation for graph mining
algorithms avoiding redundant processing.

From Fig. 2, second step of our approach is to see if
further reduction in graph size is possible? Notice that
subgraphs with labels 3 and 4, in Fig. 3, could further be
compressed due to large similarity among them. For that, we
intend to further compress the graph using Jaccard similarity
index based clustering. Given two subgraphs, we find
similarity among them via extension of Jaccard equation as:

$$JS(s_{ij}, s_{ij+1}) = \frac{2 \times \text{common nodes}}{\text{total nodes} + \text{nodes saf } s_{ij}/\text{nodes saf } s_{ij+1}}$$

To further, compress data graphs such as subgraphs with
labels 3 and 4, we calculate Jaccard similarity coefficient,
using (1), for subgraphs $s_{ij}$ and $s_{ij+1}$ Based on Jaccard
similarity, we perform clustering to extract common patterns
and compress the data graph further. The steps for clustering
are presented in Algorithm 3. The $JS$ calculated among all
the combinations of subgraphs are stored in the Jaccard
coefficient ($JC$) vector (line 1 of Algo. 3), which is further
sorted in descending order (line 2). The first value in $JC$
vector and the corresponding subgraphs are assigned label 1
and marked (line 3-5). Similarly, next value from $JC$ vector
and its corresponding subgraphs are obtained, if they are not
marked, they are assigned incremented value of the label
variable and marked. In this way all subgraphs are marked
and labelled. The subgraphs with identical labels belongs to
the same cluster and they have common pattern replicating
in all subgraphs in that cluster. Next, we extract those
patterns so as to further compress to reduce the size of the
data graph. From Fig. 3, subgraphs with labels 2, 3, and 4,
are similar, i.e., they have a common subgraph pattern. After
applying Algo. 3, these three subgraphs comes in the same
cluster, as shown in Fig. 4, where the common subgraphs
among the given subgraphs are identified and compressed.

To obtain common subgraphs, common nodes attributes are
obtained. The obtained common nodes are checked if they
are connected and that subgraph is replicated in all the
subgraphs belonging to a cluster, e.g., in Fig. 4, the common
pattern is nodes with labels: HLMNK which is replicated in
all the subgraphs.

Fig. 5 shows the compressed data graph of Fig. 3. Two
redundant subgraphs are obtained with number of instances 2
and 3, as shown in maroon color in Fig. 5. Note that the
compressed data graph size is just 18 nodes, where original
graph size, in Fig. 3, is 34. To enforce consistency,
neighbors are added in the compressed graphs.
Fig. 6 Graph size reduction, original graph size vs compressed.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Nodes</th>
<th>Edges</th>
<th>Max. degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIDS</td>
<td>254,156</td>
<td>274,513</td>
<td>11</td>
</tr>
<tr>
<td>NASA</td>
<td>1,223,194</td>
<td>1,186,404</td>
<td>245</td>
</tr>
</tbody>
</table>

Table 1: Datasets description.

3. Simulation Results

In this section, we provide simulation results for our algorithm by testing on the following real world datasets: 1) NASA, it consist of 36,790 tree-structured graphs where the degrees of some nodes are as big as up to 245, and graph sizes vary from tens to approx. 900 nodes per graph, and 2) AIDS dataset, it consist of 10,000 sparse graphs. For simulations, we used PC with Intel Quad-Core i5-6500 @3.20GHz. The main memory is 8GB. The proposed algorithm is analyzed on the following two metrics: 1) Compressed graph size vs the original graph size. 2) Time to construct the compressed graph.

Fig. 6 shows the performance of reducing graph sizes using our proposed approach on both NASA and AIDS datasets. It is evident that the proposed algorithm significantly reduces graph sizes as the number of nodes in graphs increases. The proposed algorithm achieves maximum 69.7% reduced nodes and edges, which in turn means redundant computations in query processing in reduced up to 69.7%. Another encouraging insight that could be obtained from Fig. 6 is that with the increase in graph sizes, the compression algorithm performance also increases and compress graphs more efficiently. The compression efficiency for different sized graphs vary from 41% to 69%.

Fig. 7 shows the compressed graph construction time over NASA and AIDS datasets. It is observed that with varying graph sizes, the time to construct compressed graph is within a reasonable range. It takes around 3 seconds at most when the graph size is 869 nodes. An interesting point is that the time cost increases linearly as the graph size increases.

4. Conclusion

In this paper, we propose a novel algorithm for accelerating graph mining algorithms over massive graphs. The main contribution of our proposed algorithm is to best utilize recurring subgraphs or containment correlations among data graphs so as to avoid massive redundant computations. Current graph mining algorithms could be integrated over proposed framework to speedup their query processing performance. Extensive experimental results on real data sets show significant performance gain in compressing data graphs by exploiting recurring subgraphs within data graphs.

References