Analyzing Frequent and Constrained Subgraph Mining

Pragati Unde
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
pxu3154@cs.rit.edu

Abstract—Due to the rapid increase in data usage in scientific methods and web communication, there has been an increase in building graph-based application. Graph mining is becoming not only important but also needs to be efficient enough. The Graph is the representation of data used in fields like cheminformatics, computer network, etc. Frequent graph pattern mining is a mining technique used to discover meaningful subgraph which can be useful for further detailed analysis. Frequent subgraph mining is used to discover frequent instances of subgraph in the dataset. But there are many challenges such as frequent graph mining produces many subgraph patterns, type of graph dataset.To overcome these problems, we need to develop a framework which handles these two graphs along with some constraints on graph mining pattern.

In this paper, I focus on the SkinnyMine algorithm which was proposed by Feida et al. [1] in which all l-long δ-skinny patterns are mined based on the concept of canonical diameter, which guarantees both completeness and uniqueness of target pattern.

Index Terms—Subgraph Mining; Graph and tree search strategies; SkinnyMine; Canonical Diameter.

I. INTRODUCTION

There is research going on subgraph mining for last 15 years. Frequent graph pattern mining is one of the areas of subgraph mining. There are many direct and indirect uses of subgraph mining. The direct usage is mining conserved sub-networks, program flow analysis, anomaly detection analysis, information diffusion analysis of social networks and indirect usage of Subgraph mining patterns are clustering, classification, prediction etc.

There are typically two graphs based approaches which are used for subgraph mining, namely apriori based subgraph pattern and pattern growth approach.

In apriori based subgraph pattern we will first extract all frequent subgraph of size k. Then we will find candidates of size k+1 by joining candidate of size k edges. The pattern must share common subgraph of k-2 edges as shown in the Figure 1. Both graphs on the left side are structurally same as they have the same number of edges and nodes. This is the NP-Complete Problem. Pattern growth approach uses traditional way, i.e it uses the depth first search algorithm for candidate generation and recursively grow the pattern. Then similar edges are removed using pruning. There are many algorithms like Subdue, gSpan, Gaston, and Mofa are proposed which uses pattern growth approach for frequent subgraph mining. For example: gSpan is the state-of-the-art subgraph mining algorithm which is used to identify frequent substructure without candidate generation. Each graph will be given a unique minimum DFS code which will then be ordered in a lexicographical order. A Depth-first Search based on this lexicographical order, will be applied to mine the subgraph patterns efficiently. But all these methods produce many subgraph patterns. As research progresses on this, [2] introduces mining closed frequent graph patterns and other methods like SPIN and MARGIN, used for maximal frequent subgraph mining [3], [4], [5]. They reduce the number of patterns in the graph, but the number of resultant sub graph increases rapidly. So, this limitation is reduced by Origami [6] and Ring [7], but they failed to find graph patterns such as statistically significant subgraph pattern and discriminating patterns.

However, all algorithms failed to produce constrained mining patterns. In this paper, we tried to produce the constraint patterns using “Skinny Pattern”. Nowadays, the importance of skinny patterns is highly increased in the field of social media and network analysis. A
“Skinny” pattern, formally called a l-long δ-skinny pattern, is essentially a graph pattern with a long backbone of length l from which short twigs no longer than branch out [8]. These patterns can be used in the Mobile Data Mining and Information Diffusion Analysis in Social media to find user activities, user social community which offers useful tasks such as recommendations, clustering, scheduling etc.

In this paper, we are going to review some state-of-the-arts algorithms in section II. In section III, we define the problem formulation for SkinnyMine algorithm. In Section IV, we have introduced the approach for SkinnyMine algorithm. The implementation details of the algorithm are described in Section IV. Experimental results are present in the section V. Finally, we conclude this paper in Section VI.

II. BACKGROUND

In this section, we studied two of the algorithms thoroughly and tried to analyze the similarities and dissimilarities between them.

A. SkinnyMine Algorithm

In paper [1], they propose the direct mining framework to find constraint frequent patterns using a SkinnyMine algorithm in which all the ℓ-long δ-skinny patterns are mined using the key concept of canonical diameter, which are used in mobile data mining, information diffusion, adoption propagation etc. The canonical diameter guarantees the unique, complete and distinct target pattern. SkinnyMine is two stage algorithms,

1) Mining Canonical Diameters and
2) Growing Canonical Diameters to Skinny Patterns using Loop Invariant 1 means when adding each of the pattern P with canonical diameter L, L must remain the canonical diameter to ensure the unique generation of each skinny pattern.

To maintain the canonical diameter, they used three constraints [1]:

1) Diameter should not be increased,
2) Canonical Diameter still realizes the shortest distance between vH (Head of the diameter) And vT (Tail of the diameter).
3) The canonical diameter is less than the newly generated diameter.

Author also proposed direct mining framework with two properties of reducibility and continuity for qualified constraints. There are two stages in this framework,

1) Minimal Constraint - satisfying Pattern Generation.
2) Constraint - preserving pattern growth.

They used synthetic and real data for evaluation of their approach.

B. RESLING Algorithm

On the other hand, in paper [8], Author proposed a generic framework called RESLING (Representative subgraph Sampling) to mine top-K representative subgraph patterns for both types of databases, transactional as well as single graph dataset. The main challenge in subgraph mining is to search the pattern in exponential search space. The reduction of the search space is the key focus for other algorithm’s. But, there are still two issues, namely the result set size and information redundancy. These two issues are resolved by Resling algorithm.

RESLING Algorithm: Given a graph database, first convert the exponential search space into the EDIT MAP (EMAP) in which structurally similar patterns are placed near. The EMP build the search space into an edge weighted undirected graph where each node is subgraph and each an edge are called edits which is either a deletion of an edge or addition of an edge. Connectivity, Proximity and Size are three properties of EMAP. Diversified ranking is used to find the top-K representative pattern. In this step vertex-reinforced random walk (VRRW) which is like PageRank algorithm, but it is a time-variant random walk process and the space saving algorithm (SSA) is used. In SSA, neighborhoods of top-M (where M is selected based on main-memory capacity) most frequent nodes are stored, which changes with the frequency of the node that changes in every transition of VRRW. Resling avoids two-step process by doing both operations in a single and in integrated fashion. The two-step process is, first subgraph patterns are mined, and then representative patterns are identified. The evaluation of this algorithm shows that RESLING is up to 20 times better in its representative power and 2 orders of magnitude faster [8].

Considering the above two algorithms, we have decided to implement the SkinnyMine algorithm which is used to mine the constraint satisfying patterns.

III. PROBLEM FORMULATION

This section is referred from the original paper [1]. As a convention, the vertex set of a graph G is denoted by V(G) and the edge set is denoted by E(G). The size of a graph P is defined by the number of edges of P, written as | P |. In this paper, a graph G = (V(G), E(G)) is associated with a label function ℓG: V(G) → Σ, where Σ is a label set. Our method can also be applied to graphs with edge labels.

A. Definition 1: (Labeled Graph Isomorphism)

[1] Two labeled graphs G and G’ are isomorphic if there exists a bijection f : V(G) → V(G’), such that ∀u
\begin{enumerate}
\item Given a graph \( G \), a path \( L \) of \( G \) is represented as a sequence of vertices \( L = [v_i, v_2, \ldots, v_k] \) where \( i, 1, 2, \ldots, k \) are their physical vertex IDs and \((v_{i,1}, v_{i+1}) \in E(G)\), \( 1 \leq m < k \). By default, all paths in this paper are simple paths, i.e., all vertices are distinct. We assume there is a lexicographic order among all labels in \( \Sigma \), and for any two labels \( \ell_1, \ell_2 \in \Sigma \), denote as \( L_1 \prec L_2 \) if \( L_1 \) is lexicographically smaller than \( L_2 \) \cite{1}.

\item \textbf{(Lexicographical Path Order)}

For two labeled paths \( L_1 = [v_1, v_2, \ldots, v_n] \) and \( L_2 = [v_1, v_2, \ldots, v_n] \) of a graph \( G \) with label function \( \ell \), we say \( L_1 \) is lexicographically smaller than \( L_2 \), denoted as \( L_1 \prec L_2 \) \cite{1}.

\item \textbf{(Canonical Diameter)}

\[ L \] is defined as \( D \) be the set of all simple paths of length same with \( \ell \). Given a graph \( G \), a path \( L \) is canonical diameter of a current pattern \( P \) be \( L = [v_H, v_2, v_3, \ldots, v_{1}] \) with length \( |L| = n \). Let \( v_H \) and \( v_T \) denote the head and tail of the diameter respectively. Figure 2 shows an example graph \( G \) with canonical diameter \( L = [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8] \), the head \( v_H = v_1 \) and the tail \( v_T = v_8 \). Another path in the example is \( L' = [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_{19}] \), which is same as length \( L_G \) but it is lexicographically larger by definition 2.

\item \textbf{(Vertex Level)}

\[ L \] is same as length \( \ell \) \cite{1}. A graph \( G \) is called \( \delta \)-skinny if its canonical diameter \( L \) is of length \( \ell \), i.e., \(|L| = \ell \), and \( G \) is \( \delta \)-skinny.

\item \textbf{2) Edge formed using two \( i \)-level vertices.}

As shown in the Figure 2, we obtain each pattern after every iteration. Canonical diameter considers as level 0 pattern. Then at each iteration after that we will add the level edges.

\item \textbf{1) Edge formed using one \( i \)-level vertex and one \( i \)-level vertex.}

We maintain Loop Invariant I for each iteration, which guarantees the unique pattern generation. The Loop Invariant I is defined in the paper \cite{1} as follows.

\textbf{Loop Invariant I}: When growing current pattern \( P \) with canonical diameter \( L \), \( L \) must remain the canonical diameter after each edge extension.

To maintain the canonical diameter, we could just find all shortest paths whenever we add new edge, but that is not the efficient way. As described earlier, \( v_H \) and \( v_T \) are the head and tail of the canonical diameter respectively. So, when we grow \( P' \) by adding edge from pattern \( P \), to maintain \( L \) as the same canonical diameter of \( P' \) is to maintain following three constraints. These constraints are taken from the paper \cite{1}.

\item \textbf{A. Mining Canonical Diameters}

Given a graph \( G \), a frequent threshold \( \sigma \) and diameter constraint \( \ell^* = \ell \) we mine all frequent path of length \( \ell \) denote this set as \( S_0 \). Each path in this set is canonical diameters of length \( \ell \). Mining canonical diameter is again two step process.

\textbf{1) Step I:} First, we mine all the frequent paths whose lengths are powers of 2 and less than \( \ell^* \), i.e., \( 2^0, 2^1, 2^2, \ldots, 2^k \). For calculating the paths of length \( 2^0 \), we concatenate two current frequent paths of length \( 2^{k-1} \) with the initial set being the set of all frequent edges of length 20. Then incrementing edge at one step, this approach provides the most efficient way to find all the paths with length of powers of 2, and we denote the set as \( S_1 \) \cite{1}.

\textbf{2) Step II:} In this step, we will merge two paths which obtained from \( S_1 \) to find all frequent paths of length \( \ell \) where \( \ell \) is not a power of 2. In this case, instead of concatenation, we merge two paths that are partially overlapping. We will get a unique path because \cite{1} we have defined \( v_H \) and \( v_T \) as the head and tail of \( L \), which are unique unless \( L \) is self-isomorphic. It follows that, if \( L \) is obtained by merging two paths, they must be the prefix of \( L \) containing \( v_H \) and the suffix of \( L \) containing \( v_T \), both of which are of length \( 2^k \).

\item \textbf{B. Growing Canonical Diameters to Skinny Patterns.}

Using the Canonical Diameter \( L \) where \(|L| = \ell \), we grow the \( L \) till we reach the delta. In each iteration, we add one edge at a time. We are adding two types of edges.

\textbf{1) Edge formed using one \( i \)-level vertex and one \( i \)-level vertex.}

\textbf{2) Edge formed using two \( i \)-level vertices.}

As shown in the Figure 2, we obtain each pattern after every iteration. Canonical diameter considers as level 0 pattern. Then at each iteration after that we will add the level edges.

We maintain Loop Invariant I for each iteration, which guarantees the unique pattern generation. The Loop Invariant I is defined in the paper \cite{1} as follows.

\textbf{Loop Invariant I}: When growing current pattern \( P \) with canonical diameter \( L \), \( L \) must remain the canonical diameter after each edge extension.

To maintain the canonical diameter, we could just find all shortest paths whenever we add new edge, but that is not the efficient way. As described earlier, \( v_H \) and \( v_T \) are the head and tail of the canonical diameter respectively. So, when we grow \( P' \) by adding edge from pattern \( P \), to maintain \( L \) as the same canonical diameter of \( P' \) is to maintain following three constraints. These constraints are taken from the paper \cite{1}.

\end{enumerate}
1) **Constraint I: Diameter should not be increased.**
   In this diameter of P, $D(P)$ should be less than the diameter of $P'$ $D'(P')$ such that the diameter length is not more than the canonical diameter $L$.

2) **Constraint II: $L$ still realizes the shortest distance between $v_H$ and $v_T$.**
   Distance between $v_H$ and $v_T$ should be equal to the length of the diameter $\text{Dist}(v_H, v_T) = |L|$ and $D(P) \geq D'(P')$ such that it should not create a short path between $v_H$ and $v_T$.

3) **Constraint III: $L < L'$ for any newly generated diameter $L'$ of the same length.**
   Canonical diameter should be lexicographically smaller than the new diameter, which is created after adding an edge.

All the above constraints are independent of each other. So, they should be satisfied when generating new pattern $P'$ from P. In Figure 2, the dotted red lines depict the constraint violation for that edge. $E(f_8, e_{16})$ create diameter, which is $[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$ of length 8 which is longer than the length of $L$. Similarly for $E(c_{12}, c_{20})$ and $E(p_{22}, r_{25})$ which violates constraint I. Adding $E(a_{13}, e_7)$ and $E(b_9, b_{10})$ violates the constraint II as it creates the diameter of length 6 which is less than the length of $L$ ($\ell' = 7$). When we add $E(e_7, g_{26})$, it creates the diameter of the same length, but it is lexicographically larger than the canonical diameter. The new diameter $[v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_{26}]$ is greater because the vertex $g$ is greater than the vertex $f$. We will explain in detail, how we maintained these constraints in the next section.

### C. Constraint Maintenance

For constraint maintenance, we just need to check the following conditions based on the distances between the new vertex $u$ and head of the diameter $v_H$ and tail of diameter $v_T$. We just need to locally update the distance between new vertex and head/tail of diameter. In our case, for new edge $(u, v)$, $v$ is the vertex on the diameter and $u$ is the new vertex at i-level. So, for each vertex $u$, we calculate $D_{uH}$ and $D_{uT}$ where $D_{uH}$ is the shortest distance between vertex $u$ and $v_H$ and $D_{uT}$ is the shortest distance between vertex $u$ and $v_T$. Also, we calculate the distance between the vertex $v$ and head and tail of the canonical diameter, namely $D_{vH}$ and $D_{vT}$ where $D_{vH}$ is the shortest distance between vertex $v$ and $v_H$ and $D_{vT}$ is the shortest distance between vertex $v$ and $v_T$. So, using this distance we will check all constraints. All these constraints are necessary and sufficient as explained in the paper [1]. The proof for each constraint is given in the paper [1].

1) **Constraint I:** When we add a new edge $(u, v)$, specifically vertex $u$ as the new vertex, we just need to check that. The distance between vertex $u$ and $v_H$ is less than or equal to the diameter length as well as the
distance between the vertex \( u \) and \( v_T \) is less than or equal to diameter length.

\[
D^u_H \leq \mathcal{D}(P) \quad \text{and} \quad D^u_T \leq \mathcal{D}(P)
\]

so that the diameter length is not longer than the canonical diameter.

2) **Constraint II**: To maintain the distance between \( v_H \) and \( v_T \) should not be less than the canonical diameter when we add new edge is to check the addition of \( D^u_H \) and \( D^u_T \). This addition should be greater than or equal to length of the diameter.

\[
D^u_H + D^u_T \geq \mathcal{D}(P)
\]

3) **Constraint III**: To maintain constraint III, we need to check whether \( L < L' \) in two cases only otherwise there is no need to check this constraint and we can add that edge into the pattern \( P \).

1) if \( v \in V(P) \), \( u \not\in V(P) \) and \( \max(D^v_H, D^v_T) = \mathcal{D}(P) - 1 \)
2) if \( u, v \in V(P) \) and either \( D^u_H + D^v_T = \mathcal{D}(P) - 1 \) or \( D^v_H + D^u_T = \mathcal{D}(P) - 1 \)

if the above condition satisfy then we will check the lexicographical order of each diameter.

V. IMPLEMENTATION

In this section, we explained the SkinnyMine algorithm in detail. We used JAVA and Neo4j technology for our implementation. The work-flow of the SkinnyMine algorithm is as shown in the figure 3 As we described SkinnyMine in section IV, the pseudo code for this is in the Algorithm 1. The two subroutine DiamMine() and LevelGrow() are described in Algorithm 2 and Algorithm 3 respectively.

The input to DiamMine() subroutine is the length of the diameter, a graph \( G \) and support threshold \( \sigma \) which is the frequency of the no of edges in the graph. In step 1, we set all frequent paths in set \( S_0 \). Step 2 to 12 are used to find all frequent paths each of length is powers of 2. We get this result by concatenating two paths progressively. CheckConcat() will check if we can concatenate the two paths using end vertices of each path. For example In Figure 2 Consider “abbc” and “cade” are two frequent paths in which ‘c’ is the common node. So, we concatenate this two edges and we get “abbcade” as a resultant edge. In step 7, we are checking if the resultant set is empty or not if it is empty then we will assign the previous set of edges to the current \( S_i \) and decrement the counter i followed by exiting from while loop using break condition. Then In step 11, after concatenating two edges we will check if the resulting edge is frequent or not using cypher query.

In step 3, we are finding the canonical diameter, which is of length \( \delta \) using DiamMine() subroutine. Next steps will be explained as we progresses.

The input to DiamMine() subroutine is the length of the diameter, a graph \( G \) and support threshold \( \sigma \) which is the frequency of the no of edges in the graph. In step 1, we set all frequent paths in set \( S_0 \). Step 2 to 12 are used to find all frequent paths each of length is powers of 2. We get this result by concatenating two paths progressively. CheckConcat() will check if we can concatenate the two paths using end vertices of each path. For example In Figure 2 Consider “abbc” and “cade” are two frequent paths in which ‘c’ is the common node. So, we concatenate this two edges and we get “abbcade” as a resultant edge. In step 7, we are checking if the resultant set is empty or not if it is empty then we will assign the previous set of edges to the current \( S_i \) and decrement the counter i followed by exiting from while loop using break condition. Then In step 11, after concatenating two edges we will check if the resulting edge is frequent or not using cypher query.

If the path is frequent then we will add this path to our set \( S^i \) until \( 2^i \) is less than \( \ell^* \). In step 13 we will get all frequent path of length \( 2^i \). From step 14 to 20, we find all paths of length \( \ell^* \) where \( \ell^* \) is not a power of 2. We get this path by merging two paths from the set which is obtained in step 13. Note that, here we are merging two paths which are partially overlapped using CheckMerge() function. After merging, we get set of all simple paths i.e. canonical diameters of length \( \ell^* \).
Algorithm 2: DiamMine [1]

**Input**: input graph G, length constraint \( \ell^* \), support threshold \( \sigma \)

**Output**: Set of all qualified frequent paths \( S \).

1. \( S \leftarrow \emptyset, i \leftarrow 0, S_0 \leftarrow \) all frequent edges;
2. **Do until** \( 2^i > \ell^* \):
   1. \( i = i + 1 \)
   2. **For** each path \( L' \) in \( S^{i-1} \):
      1. \( \hat{L} \leftarrow \text{CheckMerge}( L', L'' ); \)
      2. **If** \( S_i \) is empty
         1. \( S_i \leftarrow S_i^{-1} ; \)
         2. \( i = i - 1 ; \)
         3. **break**;
      3. **If** \( \hat{L} \) is frequent
         1. \( S_i \leftarrow S_i \cup \{ \hat{L} \} ; \)
   3. \( T \leftarrow S_i \)
4. **If** \( \ell^* > 2^i \)
   1. **For** each path \( L' \) in \( T \):
      1. **For** each path \( L'' \) in \( S^{i-1} \):
         1. \( \hat{L} \leftarrow \text{CheckConcat}( L', L'' ); \)
         2. **If** \( \hat{L} \) is frequent and \( | \hat{L} | = \ell^* \)
            1. \( S \leftarrow S \cup \{ \hat{L} \} ; \)
   5. **Else** \( S \leftarrow T ; \)
5. **Return** \( S \);

The CheckConcat() function will check the end values of paths i.e the last node value from \( L' \) and the first node from \( L'' \). If both values are the same, then this function will return true, otherwise it will return false.

The CheckMerge() function will check the common subpath( or common nodes) from two paths \( L' \) and \( L'' \) respectively, and will merge the uncommon path to the first path. For example In Figure 2 Consider “abca” and “caef” are two frequent paths in which ‘ca’ is the common paths or nodes. So, we merge these two paths and we get “abcaef” as a resultant edge. Also, we will check the length of the resultant path, whether it is equal to \( \ell^* \). If it is equal, then we will add this path into the final set, otherwise we will not add the path.

In the next step of Algorithm 1, i.e. from step 4 to 11 \( \delta \) skinny patterns are added level by level using levelGrow() subroutine.

LevelGrow() subroutine gives us the skinny pattern based on the constraint. Level 0 consists of all the edges belonging to canonical diameter. Onwards, it will check if all constraints are satisfied with the new edge added to the graph. checkConstraint subroutine checks for all three constraint as explained in the above section. If the new edge satisfies these constraints, new edge will be added to graph and resultant pattern will be checking in input graph for it existence, if present in target graph then it will be added to the final result.

As shown in figure 2, we will have three patterns once this subroutine is executed. Edges highlighted with red will not satisfy the constraints, thus those edges will be removed from target graph.

VI. EXPERIMENTS AND RESULTS

The analysis of the algorithm is performed on the Windows 10 machine which had 8GB RAM and 500 hard disk. Neo4j is using the memory cache to store the dataset. So, to improve the performance of the system we configured the page cache memory to 512 MB. Also the string block size and array block size is set to 60 and 300 respectively.

We performed testing on several protein datasets of size 8, 12, 16, 32, 64, 128, 256 and compare the performance. The Table I describe the time taken for the execution of the algorithm for each dataset with respect to Stage I and Stage II. We observed that for some of the datasets and if we put less input value then the execution time is more as compared to the other datasets. This happened because the graph contains more edges for the respective skinny value. For example, Dataset of size 64 and input value 6-long 2-skinny, the time taken was more compared to the input value 8-long 3-skinny.

As shown in the Figure 5. As the graph size increases the execution time is also increases. But, execution time
TABLE I: Analysis of SkinnyMine Algorithm on Protein Datasets

<table>
<thead>
<tr>
<th>Stage I (ℓ-long)</th>
<th></th>
<th>Stage II (δ-skinny)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>26</td>
<td>53</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td>54</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 4: Runtime: Stage I-DiaMine vs Stage II- Level-Grow [V]

also depends on the input parameters given. For example: if the input is given as 5-long 2-skinny then it will take less as compared to the input 8-long 2-skinny. This is clearest in the below Figure 4. We can see that as the diameter length increases the execution time of Stage I is also increasing. The stage I (Algorithm 2) execution is not depend on the graph data size, it totally depends on the input length of ℓ. But, the Stage II (Algorithm 3) execution takes less time compared to Stage I. The red line in the figure 4 indicates the stage II execution. It is very clear from the graph that if we do not find the pattern then it will not execute the Stage II. Also, if the input, δ skinny pattern is increased then the time increases. We can conclude that this algorithm is totally based on the input ℓ-long δ-skinny.

VII. CONCLUSION

We have successfully implemented the SkinnyMine algorithm which is based on the concept of canonical diameter. Canonical diameter is of length ℓ-long, from which we find δ-skinny patterns by adding valid edges on each iteration using satisfiability test of three constraints. This algorithm provides unique and efficient patterns based on the input given as ℓ-long δ-skinny. We test this algorithm on a single graph. We can also use this algorithm on the transactional graph database.

ACKNOWLEDGMENT

I would like to thank Dr. Carlos Rivero, Assistant Professor at Rochester Institute of Technology, for guiding me throughout in the process of completing this project. He introduced me to the concept of graph mining and motivated me in implementing SkinnyMine algorithm. I would also like to thank Dr. Reynold Bailey, Associate Professor and Undergraduate Program Coordinator at Rochester Institute of Technology, for giving the inputs on designing, presenting and documenting the project.

REFERENCES


APPENDIX

Fig. 6: Input Graph for 6-long 2-skinny

Fig. 7: Output Graph for 6-long 2-skinny

Fig. 8: Input Graph for 6-long 3-skinny

Fig. 9: Output Graph for 6-long 3-skinny

Fig. 10: Input Graph for 7-long 3-skinny

Fig. 11: Output Graph for 7-long 3-skinny