Discovering Frequent Topological Structures from Graph Datasets

R. Jin C. Wang D. Polshakov S. Parthasarathy G. Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{jinr,wachao,polshako,srini,agrawal}@cse.ohio-state.edu

ABSTRACT

The problem of finding frequent patterns from graph-based datasets is an important one that finds applications in drug discovery, protein structure analysis, XML querying, and social network analysis among others. In this paper we propose a framework to mine frequent large-scale structures, formally defined as frequent *topological structures*, from graph datasets. Key elements of our framework include, fast algorithms for discovering frequent topological patterns based on the well known notion of a topological minor, algorithms for specifying and pushing constraints deep into the mining process for discovering constrained topological patterns, and mechanisms for specifying approximate matches when discovering frequent topological patterns in noisy datasets. We demonstrate the viability and scalability of the proposed algorithms on real and synthetic datasets and also discuss the use of the framework to discover meaningful topological structures from protein structure data.

Categorization and Subject Descriptions: H.2.8 [Database Applications]: Data Mining

General Terms: Algorithms

Keywords: Graph mining, topological minor, frequent graph pattern

1. INTRODUCTION

Recently, there has been a lot of interest in mining frequent patterns from *structured* or *semi-structured* datasets, and a majority of research in this area has focused on developing efficient algorithms for mining frequently occurring (connected) subgraphs [4, 6, 13, 8]. However, in many real world applications, such as biology, social networks, and telecommunication, *large-scale structures*, which provide high-level topological information of graphs, can be very important to provide key insights into the underlying datasets. For instance, the discovery of non-local or tertiary structural information is an important problem in protein structure analysis. Similarly, in the analysis of social or communication networks, the direct connection between a pair of nodes is often not the focus, instead, the patterns where several nodes are connected through a set of independent paths are of greater interest. However, such frequent large-scale structures can be very hard to discover using current frequent subgraph mining approaches [4, 6, 13, 8]. This is not only because the subgraphs sharing these kind of structures can be infrequent (i.e. the traditional anti-monotone property does not hold), but also because the individual subgraphs do not adequately represent such structures.

The main contribution of this paper is a framework to mine frequent large-scale structures from graphs. Our work is inspired by a well-established mathematical concept, *topological minor* [1]. A topological minor of a graph is an abstraction that focuses on its structural information. Intuitively, such an abstraction is achieved by replacing or contracting *independent paths* in a subgraph with individual edges. We develop an efficient algorithm to enumerate frequent topological structures from graph datasets.

An important notion in our framework is that of a *relabeling function*. Since often real datasets can be best represented as labeled graphs when we replace independent paths in a subgraph with edges, the information labels on such paths are lost. However, in many applications, summarized information about the contracted paths can be useful to *categorize* these topological structures. Our framework supports this notion through user-defined *relabeling functions* to recover some degree of information loss from the contracted paths. Such a function maps an entire labeled path to a single edge label. In other words, an edge label carries the desired information about its corresponding contracted path.

To the best of our knowledge, our work is the first to focus on the problem of mining frequent (large-scale) topological structures. Overall, our framework is also very flexible. It can be used for *approximate pattern mining*, where the support for a frequent pattern does not depend on the exact matches, but instead relies on some form of a *fuzzy matching* [2, 7, 10]. The topological structures together with relabeling functions provide a powerful mechanism to express various forms of fuzzy matches.

2. TOPOLOGICAL STRUCTURES AND RELABELING FUNCTIONS

We begin with some basic notations. The vertex set of a graph G is referred to as $V(G)$, and its edge set as $E(G)$. A *path* P in a graph G is a sequence of vertices $v_1, v_2, \cdots, v_k$, where $v_i \in V(G)$ and $\{v_i, v_{i+1}\} \in E(G)$. The vertices v_1 and v_k are linked by P and are called its *ends*, and $v_2, v_3, \cdots, v_{k-1}$ are the *inner* vertices of P. Also, we define the number of inner vertices in a path as its *length*. In particular, a group of paths are *independent* if none of the paths have an inner vertex on another path. Any of such paths is called as an *independent path*. Note that the independent paths are the key tools to study topological structures of a graph.

KDD'05, August 21–24, 2005, Chicago, Illinois, USA.

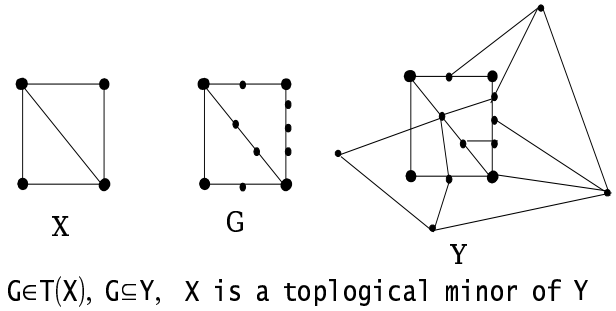


Figure 1: Topological Minor

2.1 Topological Minors and Topological Structures

Informally, a *topological minor* of a graph is obtained by contracting the independent paths of one of its subgraphs into edges. For example, in Figure 1, X is a topological minor of Y since X can be obtained by contracting the independent paths of G, which is a subgraph of Y. Clearly, contracting independent paths helps simplify a (sub)graph without compromising its topological information [1].

The formal definition of the *topological minor* of a graph is as follows. A *subdivision operation* of a graph X, is to replace the edges of X with independent paths. A *subdivision graph* of X is a graph obtained by performing a subdivision-operation of X. For example, in Figure 1, the graph G is a subdivision graph of X. Note that the subdivision operation is basically an "inverse" of the path contraction operation. Further, the topological space of X, $T(X)$, is the collection of all its subdivisions graphs. If X has a subdivision graph G ($G \in T(X)$) and G is a subgraph of another graph Y, then X is a *topological minor* of Y. The vertices of X which corresponds to the original vertices of Y are called *branch vertices*.

Topological structures of a graph are derived from topological minors. Given two parameters, l and $h, 0 \leq l \leq h$, an (l, h)-*subdivision* of a graph X, involves replacing *all* edges of X with independent paths whose lengths are between l and h. An (l, h) *subdivision graph* of X is a graph obtained by performing an (l, h)-subdivision operation of X. For example, in Figure 1, G is a $(0, 3)$-subdivision graph of X. Similarly, we can define the (l, h)-topological space of X, $T_{l,h}(X)$, to be the collection of all its (l, h)-subdivisions graphs. If X has an (l, h)-subdivision graph G ($G \in T_{l,h}(X)$) and G is a subgraph of another graph Y, then X is a (l, h)-*topological minor*, or a topological structure of Y. Therefore, in Figure 1, X is a $(0, 3)$-topological minor of Y. Note that the purpose of introducing the definition of topological structures of a graph is to control the compression ratio between a graph and its subdivision graph.

2.2 Labeled Graphs

Thus far, our discussion has focused on unlabeled graphs. One is often more interested in *labeled* graphs. We begin with the informal discussion of the topological structures on a labeled graph. Intuitively, the way to simplify a labeled graph is to remove all the inner vertices and edges of its independent labeled paths, and then connect their remaining labeled ends with an unlabeled edge. In Subsection 2.3, we will describe how to use *relabeling functions* to add labels to these contracted edges. Clearly, the main difference between the topological structures on labeled graphs and on unlabeled graphs is that the vertex labels for the ends of contracted paths are still preserved.

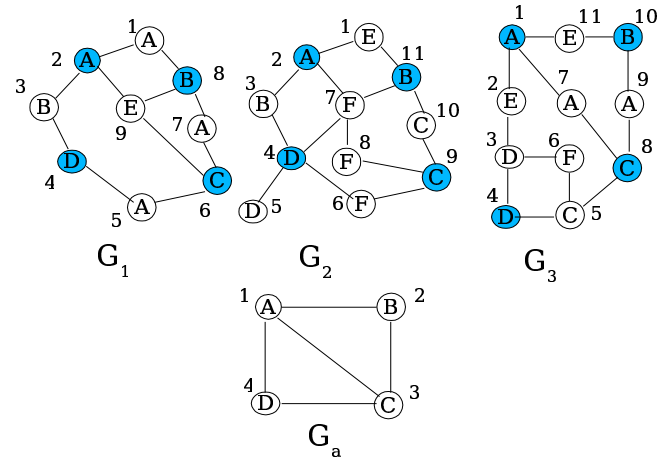


Figure 2: Running Example

Here, we will mainly focus on the vertex labeled graphs, where each vertex has a label. Note that our results and methods can be easily extended to (edge) labeled graphs. Given two parameters, l and h, the main difference between an (l, h)-topological minor on labeled graph and unlabeled graph is the subdivision operation. An (l, h)-*subdivision operation* of a vertex labeled graph X, involves replacing all edges of X with independent paths satisfying the following conditions: 1) the path lengths are between l and h, 2) the vertices (and edges) in the paths are labeled, and 3) the ends of these paths share the same vertex label as the corresponding ends of their original edges. The other concepts, i.e., the (l, h)-subdivision graph, the (l, h)-topological space, and (l, h)-topological minors, are the same as in unlabeled graphs.

Assume we have a collection of graphs, denoted as D. Given two parameters l and h, and a graph G, the number of graphs in D which have G as a (l, h)-topological minor (also topological structure) is referred to as the *support* of G.

DEFINITION 1. *Given a collection of graphs, two parameters l and h, and a threshold θ, a (l, h)-topological minor whose support is greater than or equal to θ is called*

For example, in Figure 2, for $l = 1$ and $h = 2$, the support of the graph G_a is 3 in the dataset composing of G_1, G_2, G_3, however, for $l = 0$ and $h = 1$, the support of the graph G_a is only 1.

2.3 Relabeling Functions

Consider a path $p = (v_0, v_1, \cdots, v_k)$. Normally, when it is contracted in a topological structure, the only information left is its ends, v_0 and v_k, with their vertex labels. Relabeling functions can preserve important additional information from these contracted paths, in the form of labels for the corresponding edges in the topological structure.

Formally, a relabeling function $f : \mathcal{P} \rightarrow \mathcal{L}$ can be defined as a map from the set of all possible paths $\mathcal{P}$ to the new edge-label set for the topological structure $\mathcal{L}$. A common type of relabeling functions is derived from the length of each independent path. For example, we can use the length of a contracted path to label its corresponding edge. Formally, for a given path $p = (v_0, v_1, \cdots, v_k)$, $f(p) = k - 1$. Note that in this way, the edges in the topological structures become labeled.

Because of the space limitation, we will omit further discussion of *relabeling function*, including its implementation and its application for *approximate pattern mining*. We refer the interested readers to our technical report [5] for details.

3. ALGORITHM FOR MINING TOPOLOGICAL STRUCTURES

Frequent topological structure mining is a generalization of frequent graph mining. Specifically, frequent sub-graphs for a vertex-labeled graph dataset can be mined as a special case of frequent topological structures: the $(0, 0)$-topological minors. It should also be noted that frequent topological structures are also graphs. Therefore, mining frequent topological structures shares some similarities with mining frequent graphs.

However, mining frequent topological structures is also quite different from graph mining. Given two parameters l and h, the support of a topological structure G depends on the definition of (l, h)-topological minor. Specifically, if G is a (l, h)-topological minor of a graph D_i in the graph dataset, we need to know if there is a subgraph H of D_i and H is a (l, h)-subdivision graph of G. This potentially involves not only the subgraph isomorphism testing, but also the (l, h)-subdivision operation. In particular, counting support of topological structures is one of key issues in efficiently mining frequent topological structures, which we document here.

3.1 Counting Support for Topological Structures

To tackle this problem, we use an incremental approach. Consider a topological structure G' that can be extended from another topological structure G by adding a new edge e, denoted as $G' = G \cup \{e\}$. To test if G' is a topological structure of a graph H, our approach utilizes the information derived from G. In particular, such reuse is based on a uniform representation for a topological structure G and its corresponding subgraph in H. In the following, we first establish such representation, and then discuss the details of how we count the support of a topological structure.

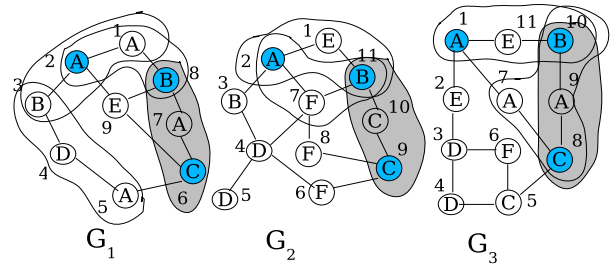
Decomposition-based Representation: Given l and h, let G be an (l, h)-topological minor of H. This implies that there exists a subgraph Y of H, where Y is a (l, h)-subdivision graph of G by a subdivision operation. To facilitate our discussion, we denote the subgraph Y together with an (l, h)-subdivision operation as an *occurrence* of G. Here, Y is isomorphic to the graph obtained by performing the subdivision operation on G. In the following, we consider how we can express the occurrences of G explicitly.

We first decompose G as a collection of edges, i.e., $G = \{e_1\} \cup \{e_2\} \cdots \cup \{e_k\}$. Based on the definition of the subdivision operation, each edge e_i corresponds to an independent path in Y, denoted as $\vec{e_i}$. Therefore, we can also decompose Y as a collection of independent paths, i.e., $\{\vec{e_1}\} \cup \{\vec{e_2}\} \cdots \cup \{\vec{e_k}\}$. We denote this decomposition as $\vec{Y}$. Clearly, the above decomposition of Y can be used to represent an occurrence of G in H. For example, in Figure 3(a), we have $\{(2, 1, 8), (8, 7, 6)\}$ of G_1 to be an occurrence of the topological structure, $G' = \{(A, B), (B, C)\}$.

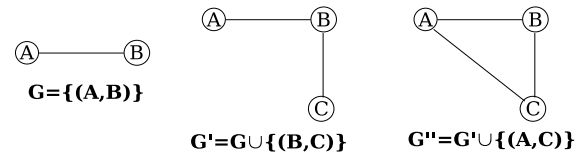
The decomposition can be further represented in a very concise format. Consider $G \cup \{e\}$ which is also a (l, h)-topological minor of H. Let $S_{G,H} = \{\vec{Y_1}, \vec{Y_2}, \cdots, \vec{Y_m}\}$ be all the occurrences of G in H. We have the following lemma.

LEMMA 1. *The occurrences of $G \cup \{e\}$ can be represented as $\vec{Y'_1} \cup \{\vec{e}\}, \cdots, \vec{Y'_n} \cup \{\vec{e}\}$, Y'_i is called the parent occurrence of $\vec{Y'_i} \cup \{\vec{e}\}$.*

Given a topological structure G', we can decompose it as $G \cup \{e\}$, where G is called a parent of G'. For example, in Figure 3(b), we have $G' = G \cup \{(B, C)\}$, where $G = \{(A, B)\}$. Lemma 1 suggests that occurrences of G' can be partially represented by the occurrences of its parent. Naturally, for each topological structure, we can build an *occurrence list* to concisely record all of its occurrences in the graph dataset by using the occurrence list of its parent. Note that a topological structure can have many parents. However, we only need one of its parents to build its occurrence list. The question of which one of these parents is chosen will be addressed in Subsection 3.2).



(a) Occurrence Decomposition



(b) Topological Structure Decomposition

G ={(A,B)}	G'= G∪{(B,C)}	G''= G'∪{(A,C)}
0:{G_1, 0, (2,1,8)}	0:{G_1,0,(8,7,6)}	0:{G_1,0,(2,9,6)}
1:{G_1,0, (2,9,8)}	1:{G_1,0,(8,9,6)}	
2:{G_1,0, (1,2,3)}	2:{G_1,1,(8,7,6)}	
3:{G_1,0, (5,4,3)}		
4:{G_2,0, (2, 1, 11)}	3: (G_2,4,(11,10,9)}	1:{G_2,3,(2,7,8,9)}
5:{G_2,0, (2,7, 11)}	4: (G_2,5,(11,10,9)}	
6:{G_3,0, (1,11,10)}	5:(G_3,6,(10,9,8)}	2:{G_3,5,(1,7,8)}
7:{G_3,0, (7,8,9,10)}		
...	...	...

(c) Occurrence Lists

Figure 3: Decomposition and Occurrence Lists

The concise representation of each occurrence in the occurrence list for a topological structure $G \cup \{e\}$ is as follows. Each occurrence has a unique ID in the occurrence list, and the detailed information is a triple, (α, β, δ). Here, α is the index of the graph in the dataset D where this occurrence appears, β is the occurrence ID of this occurrence's parent, and δ is an independent path, $\vec{e}$, corresponding to the edge e. For instance, Figure 3(c), illustrates a portion of the occurrence lists for three $(1, 2)$-topological structures, G, G', and G''.

Building the Occurrence Lists: Clearly, the support of a topological structure can be easily derived from its occurrence list. Therefore, the problem of efficiently counting the support of a potential frequent topological structure boils down to building its occurrence list efficiently. However, the naive solution can be very costly. For example, suppose we already have the occurrence list for G and try to build the occurrence lists for $G \cup \{e\}$ and $G \cup \{e'\}$, where e and e' are adjacent to the same vertex v in G. The naive method will build the occurrence lists for them independently. Specifically, for each of them, we need to go through all the occurrences of G to find out *all* the independent paths corresponding to edge e or e' (path contraction). This, however, involves a lot of repetitive work, since each time we have to find *all* the independent paths starting from the branch vertex corresponding to v in each occurrence. Note that similar problems also need to be addressed in frequent subgraph mining algorithms [8].

In order to build the occurrence lists efficiently for the topological structures, we want to minimize the number of times the *finding independent paths* (discussed later) operation needs to be invoked.

We also build occurrence lists in parallel when we invoke such an operation. To formally discuss our approach, we first introduce some notation.

Let us consider generating new frequent topological structures by extending an existing frequent topological structure G with a new edge. We classify these new edges in two categories: *inner* edges or *outer* edges. An inner edge connects two dis-adjacent vertices in the graph G, and an outer edge adds a new vertex into $V(G)$, and connects an existing vertex in $V(G)$ with this new vertex. For a topological structure G, we denote $[G]_{inner}$ to be the set of all inner edges of G, and $[G]_{outer}$ to be the set of all outer edges of G. We use $[G]_{io}$ to represent the union of $[G]_{inner}$ and $[G]_{outer}$. The significance of these two sets $[G]_{outer}$ and $[G]_{inner}$ is that they record all the potential extensions of G. Finally, for an extended graph $G \cup \{e\}$ from G, we denote its occurrence list as $e.occurrencelist$ or $(G \cup \{e\}).occurrencelist$.

The basic idea of our approach is as follows. For each topological structure G, we will maintain the occurrence list for each extended graph $G \cup \{e\}$ where $e \in [G]_{io}$. We will show an optimization in next subsection to reduce the number of recorded occurrence lists. Here, we consider how we can build these lists for $G \cup \{e\}$. If e is an inner edge, we can have $[G \cup \{e\}]_{io} \subseteq [G]_{io}$. Therefore, we need to simply copy the occurrence lists for the edges in $[G]_{io}$. Note that this is not a real copy since not all occurrences for $G \cup \{e'\}$, $e' \neq e, e' \in [G]_{io}$ can be extended to $G \cup \{e\} \cup \{e'\}$. Essentially, this copy is a *Join* operation, which will be discussed later. Further, if e is an outer edge, the new vertex generated by e will be likely to bring some new outer edges. Also, the existing outer edges of G may become inner edges for $G \cup \{e\}$. In this case, we will not only need to copy these occurrence lists from G, but also need to build the occurrence lists for all the new outer edges adjacent to the new vertex.

Finding Independent Paths: The sketch of the algorithm for finding all independent paths (*IndependentPath* subroutine) for an occurrence $\vec{Y}$ starting from a branch vertex s is as follows. Let G be the graph where this occurrence $\vec{Y}$ appears. We perform a depth-first search (DFS) to enumerate these paths. There are two important issues we need to deal with. The first involves maintaining the *independent* property, and the second involves bounding the length of each path, specifically, the number of inner vertices, between l and h. To deal with the first issue, we color the vertices in the occurrence of G. Then, as we traverse the graph G starting from the branch vertex s, we keep coloring the visited vertices. If we meet any colored vertex, we need to trace back since the path has become not independent. When we found an independent path (the number of inner vertices) bounded by l and h, we will record this path. Note that the tracing back operation is associated with un-coloring the visited vertex.

Key Operations: In the following, we formally introduce the two key operations mentioned earlier, which are the *Join* operation and the *ExtendOuterEdges* operation. The two operations are sketched in Figure 4. Assume G is generated by adding an outer edge e on its parent. The procedure *ExtendOuterEdges* will scan the entire list of occurrences of G (the first *foreach* loop in *ExtendOuterEdges*). For each occurrence, let $p.to$ be its branch vertex corresponding to the newly added vertex for G. This procedure will find all the independent paths beginning from this branch vertex (the second *foreach* loop in *ExtendOuterEdges*). Specifically, such functionality is achieved by the subroutine *IndependentPath* just introduced. Each independent path generated above corresponds to a new outer edge for the topological structure G, and the occurrence lists for these new outer edges are built by adding these independent paths (implemented by *insertOccurrence*). Finally, *ExtendOuterEdges* will return all the new edges which are frequent with respect to the given support level.

The new topological structure, $G \cup \{e\}$, will inherit more information from its parent G through the procedure *Join*. The *Join* operation will filter the occurrence lists for each edge in $[G]_{io}$ to generate all the inner edges. It will also filter all the outer edges adjacent with the vertices in $V(G)$ for $G \cup \{e\}$ (implemented by the nested *foreach* loops in *Join*). The essential part of the *Join* operation is to test if, after extending the new edge e, the paths in the occurrences are still independent. This is done by the routine (*Independent* invoked from *Join*). For brevity, the details of its implementation are omitted.

```
ExtendOuterEdges(Graph T)
 {* T is a Topological Structure *}
 E ← ∅;
 foreach (occ ∈ T.occurrencelist)
      G ← Graph(D, occ.tid);
      foreach (path p ∈ IndependentPath(G, occ, occ.δ.to))
           e ← Edge(p.from, p.to);
           if (e ∉ E)
                E ← E ∪ {e};
           InsertOccurrence(e, G, p);
 foreach (e ∈ E)
      if ( not Frequent(T ∪ {e}))
           E ← E − e;
 returnE;


Join(EdgeSet E1, Edge e2)
 E ← ∅;
 foreach (e1 ∈ E1)
      e.occurrencelist ← ∅;
      foreach ((l1, l2) : l1 ∈ e1.occurrencelist and
                          l2 ∈ e2.occurrencelist and
                          l1.parentID == l2.parentID)
           if (Independent(l1.path, l2.path))
                InsertOccurrence(e, l1.path);
      if (Frequent(e))
           E ← E ∪ {e};
 returnE;
```

Figure 4: Support Counting Procedures for Mining Topological Structures

3.2 Vertical Mining Approach

Our approach mines frequent topological structures in two phases. In the first phase, we mine all the frequent topological structures which are trees, and are referred to as *frequent tree-topological structures*. In the second phase, for each tree-topological structure T, we mine *frequent graph-topological structures* which have T as their spanning tree. The tree-topological structures are graphs without cycles, and the graph-topological structures are graphs with at least one cycle. Note, this two-phased procedure has been proposed and used for efficiently mining frequent subgraphs [13, 3]. Our algorithm is sketched in Figure 5. The mining procedure *VTreeTS* corresponds to the first phase, and the mining procedure *VGraphTS* corresponds to the second phase. To generate frequent tree-topological structures, for each tree T, we use the mechanisms introduced in GASTON [8] to determine which edges in $[T]_{outer}$ are *valid* extensions. The valid extensions can also help to enumerate all frequent tree-topological structures without replication. Specifically, the procedure *ValidExtension* (invoked by *VTreeTS* in the *foreach*

loop) provides the above mechanism. The frequent graph-topological structures are enumerated by adding a subset of inner edges in $[T]_{inner}$ to each frequent tree-topological structure T. In our algorithm, the procedure *CanonicalExtension* (invoked by *VGraphTS* in the *foreach* loop) applies hashing and graph isomorphism test to avoid duplicating graph-topological structures.

TSMiner(Dataset D, Support θ, Bound l, h)
$\{* \; Find\ Frequent\ Single-Edge\ Topological\ Structures *\}$
$E \leftarrow FrequentEdgeTS(D, \theta, l, h)$;
foreach $(e \in E)$
  $VTreeTS(e)$;

VTreeTS(Tree T)
$\{* \; New\ Outer\ Edges\ of\ T *\}$
$E \leftarrow ExtendOuterEdges(T)$;
$[T]_{outer} \leftarrow [T]_{outer} \cup E$;
$\{* Tree\ Topological\ Structure\ Growing *\}$
foreach $(e : e \in [T]_{outer}$ **and**
  $ValidExtension(T \cup \{e\}))$
  $T_e \leftarrow T \cup \{e\}$;
  $[T_e]_{io} \leftarrow Join([T]_{io}, e)$;
  $VTreeTS(T_e)$;
$\{* Enumerating\ Graph\ Topological\ Structures *\}$
$VGraphTS(T)$;

VGraphTS(Graph G)
foreach $(e : e \in [G]_{inner}$ **and**
  $CanonicalExtension(G \cup \{e\})$
  $G_e \leftarrow G \cup \{e\}$;
  $[G_e]_{inner} \leftarrow Join([G]_{inner}, e)$;
  $VGraphTS(G_e)$;

Figure 5: Algorithm Framework for Mining Topological Structures

4. CASE STUDY: MEMBRANE PROTEIN STRUCTURE ANALYSIS

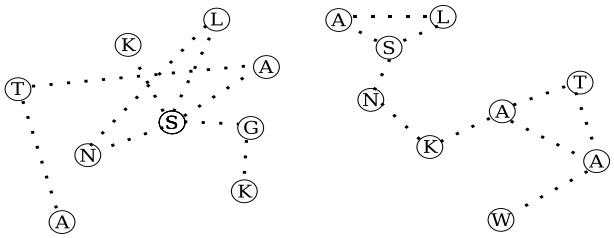
Discovery of lipids binding sites has been long known as a very challenging, but important, task for the biologists [9]. In this study, we use our new tool to search potential *protein-lipid binding sites* in an important class of proteins - membrane proteins, which are believed to account for approximately 20-30% of all protein sequences.

The dataset we use is derived from the protein data bank (PDB). We use a set of six membrane proteins known to bind with cardiolipins (CL): 1KB1, 1KQF, 1M3X, 1OKC, 1V54, and 1OGV. Amino acids as nodes in the graph (20 labels) and edges between nodes are drawn if two amino acids are within 3.5 Å . There are known to be 20 naturally occurring amino acids and these serve as node labels. In order to find the structural motifs that can serve as binding site for a CL head group, we used only the relevant parts of proteins that are known to be local to CL molecule. Such a structure typically contains around $\sim 30 - 35$ amino acids (number of nodes per graph). Note that several membrane proteins we use contain more than one CL molecule. Therefore, the total number of CL binding regions that we used to find protein-lipid binding sites is 10 (number of graphs).

Table 1 summarizes the results on mining this dataset using our tool. Note that *TSMiner* at $l = 0$ and $h = 0$ is simply a connected subgraph mining tool (same results as with Gaston). For this parameter setting, one can only find patterns till the support level is 3, and the largest one found contains at most 6 vertexes. However, upon varying the value of the parameters, we find large triangles with support 5 and 6, along with large rectangles, and topological structures containing 5 or more vertexes. At support 3, with relaxed l and h, we found a number of large topological structures, containing more than 9 vertexes, and 9 edges. Figure 6 shows two such large topological structures discovered by our toolkit. Also, such large patterns cannot be found by *MotifMiner* [10, 11]. The topological structures consist largely of polar (N, T, S), charged (K) and aromatic (W) residues which is in agreement with recent advances in the understanding of such proteins within the biophysics community[9]. The structure we find is larger than any known motifs for CL binding sites in such proteins and also seems to partially span the membrane bridging components of the protein which seems quite novel according to domain experts.

Parameters			No. of Large Topological Structures		
Support	l	h	Path	Tree	Graph
6	0	4	11 ($\|V\| = 3$)	0	1 ($\|E\| = 3, \|V\| = 3$)
5	0	3	1 ($\|V\| = 5$)	4 ($\|V\| = 4$)	4 ($\|V\| = 3, \|E\| = 3$)
5	1	2	17 ($\|V\| = 3$)	0	1 ($\|V\| = 3, \|E\| = 3$)
4	0	0	0 ($\|V\| > 2$)	0	0
4	0	1	11($\|V\| \geq 4$)	5 ($\|V\| \geq 4$)	2 ($\|V\| = 4, \|E\| = 4$)
4	1	2	27($\|V\| \geq 4$)	2 ($\|V\| \geq 4$)	1 ($\|V\| = 4, \|E\| = 4$)
4	0	2	24($\|V\| \geq 5$)	10 ($\|V\| \geq 5$)	10 ($\|V\| \geq 4, \|E\| \geq 4$)
3	0	0	1 ($\|V\| = 6$)	1 ($\|V\| = 6$)	0
3	0	1	20 ($\|V\| \geq 8$)	34 ($\|V\| = 9$)	19 ($\|V\| \geq 9, \|E\| \geq 9$)
3	1	2	12 ($\|V\| = 7$)	19 ($\|V\| = 8$)	20 ($\|V\| \geq 7, \|E\| \geq 7$)

Table 1: Number of Large Patterns Discovered by *TSMiner*



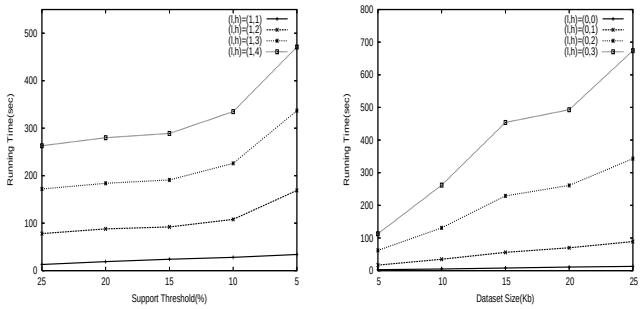
(a) frequent motif found in 1kqf (b) frequent motif found in 1m3x

Figure 6: Frequent Topological Structures Discovered by *TSMiner*

5. EXPERIMENTAL RESULTS

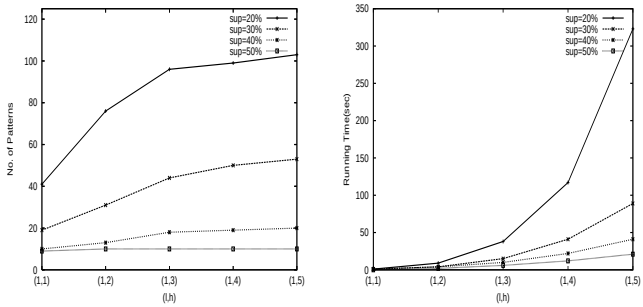
In this section, we will study the performance of our new algorithm, *TSMiner*, focusing on the following two issues: the scalability of the algorithm, how the parameters, l, h, and the support level θ, affect the performance. A more complete experimental evaluation is documented in our technical report [5]. We have implemented *TSMiner* in C++. The evaluation studies were conducted on a 2.66 GHz Pentium 4 machine with 1GB main memory, running Linux Mandrake 10.1.

Datasets: Our experiments used both synthetic and real datasets, containing vertex labeled graphs, i.e., the edge labels were not considered. The synthetic datasets were generated from the graph generator provided by Kuramochi and Karypis at the University of Minnesota. In our experiments, we fixed the average number of edges, $T = 20$, the total number of potentially frequent subgraphs, $L = 200$, the average number of edges in each potentially frequent subgraph, $I = 5$, and we vary V, the total of vertex labels, to be between 5 and 20. The real dataset was originally used for the Pre-



(a) (b)

Figure 7: (a)Varying Support (D10kV20) (b) Varying Dataset Size (D*kV20, Sup=20%)



(a) (b)

Figure 8: Chemical340 (a)No. of Patterns(Varying Support) (b)Running Time(Varying Support)

dictive Toxicology Evaluation Challenge [12]. It contains a total of 340 chemical compounds. For each compound, the atoms correspond to the vertices of the graph, and the bonds between the atoms are mapped to the edges of the graph. For simplicity, we refer this dataset as *Chemical340*.

Scalability: For the scalability study, we rely on the synthetic datasets. In Figure 7(a), we vary the support threshold from high to low, and run our algorithm on datasets containing $10,000$ graphs. As we would expect, as the support level reduces, the running time increases. Also, we can observe that as h increases (l kept the same), the running time increases. This is also expected as the number of (potential) frequent topological patterns increases as we relax the condition on the length of the independent paths. From Figures 7(b), we see that *TSMiner* scales reasonably well (close to linear) as we increase the size of the dataset. Note that the *TSMiner* with parameters $l = 0$, $h = 0$ is essentially a frequent connected subgraph mining tool for vertex labeled graphs. For such cases, we did a comparison with the state-of-art subgraph mining tool *gSpan* [13]. Our results show that our implementation is slower by a factor of 1.6. We believe this is a reasonable result, given that we offer additional functionality and do not specifically optimize for subgraph mining.

Impact of Varying l and h: In this study, we are interested in the number of patterns being generated by our new algorithm and its running time respect to the parameters l and h. Figure 8 (a) and (b) show the total number of patterns being discovered and the running time of TSMiner at different support levels, as we increase h and keep l to be 1 on the real dataset *Chemical340*.

6. CONCLUSIONS

In this paper, we have presented a novel framework for mining topological patterns in graph datasets. Based on the well known notion of a topological minor, we have designed efficient algorithms for mining such patterns. Additionally, our framework supports the notion of a user-defined relabeling function, which can be used to specify constraints and fuzzy matching criteria [5]. We have demonstrated the effectiveness and scalability of the proposed algorithms on real and synthetic datasets. We have also reported on a case study where the framework has been used to identify topological structures from membrane protein structure data.

7. ACKNOWLEDGMENTS

This work is funded in part by NSF grants CCF-0234273, CCF-0130437, CNS-0203846, CAREER IIS-0347662, and DOE grant DE-FG02-04ER25611. We thank Martin Caffrey for pointing us to the problem domain for the case study.

8. REFERENCES

[1] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000.

[2] H. Hofer, C. Borgelt, and M. R. Berthold. Large scale mining of molecular fragments with wildcards. In *Advances in Intelligent Data Analysis V*, pages 380–389, 2003.

[3] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.

[4] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Mach. Learn.*, 50(3):321–354, 2003.

[5] Ruoming Jin, Chao Wang, Dimitrii Polshakov, Srinivasan Parthasarathy, and Gagan Agrawal. Discovering frequent topological structures from graph datasets. Technical report, CSE, Ohio State University.

[6] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

[7] Thorsen Meinl, Christian Borgelt, Michael R. Berthold, and Michael Philippsen. Mining fragments with fuzzy chains in molecular databases. In *Second International Workshop on Mining Graphs, Trees and Sequences (MGTS2004)*, 2004.

[8] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.

[9] H Palsdottir and C Hunte. Lipids in membrane protein structures. *BBA*, 1666:2–18, 2004.

[10] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. *ICDM*, pages 362–369, 2002.

[11] D. Polshakov, K. Marsolo, and S. Parthasarathy. Mining 3d-motifs using phisical-chemical constraints: application to cardiolipin binding sites. In *ISMB*, 2005.

[12] A. Srinivasan, R.D. King, S.H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *IJCAI*, pages 1–6, 1997.

[13] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM*, page 721, 2002.