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.

1. INTRODUCTION

Recently, there has been a lot of interest in mining frequent patterns from *structured datasets*, such as chemical compounds, proteins, web-logs, and XML datasets. Such patterns can effectively summarize the data, provide key insights and often serve as a preprocessing step for further analysis. Since, such datasets can often be modeled as graphs, a majority of research in this area has focused on developing efficient algorithms for mining frequently occurring (connected) subgraphs [9, 10, 18, 13].

However, in many real world applications, such as biology, social networks, and telecommunication, *large-scale structures*, which provide high-level topological information of graphs, may be equally or more important than discovering the basic components. 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. Such frequent large-scale structures can be very hard to discover using current frequent subgraph mining approaches. This is not only because the subgraphs sharing these kind of structures can be infrequent (i.e. the traditional anti-monotone property leveraged by most such algorithms does not hold), but also because the individual subgraphs are not adequately abstracted or represented.

As an example of a large-scale structure we are focusing on, consider mining a protein dataset where each protein is represented as a graph. The vertexes of each graph are protein secondary structures, and an edge is associated with two protein secondary structures if their distance in the three-dimensional space is within a certain range. A frequent large-scale topological structure in such a dataset can be as follows: three α-helices that are not direct neighbors of each other, but form a triangle in the three-dimensional space. Specifically, in the graphs for different proteins, each pair of above α-helices is connected through independent paths formed by other secondary structures, possibly including α-helices, β-sheets, or loops. The triangle information can be useful for understanding the functionalities of these proteins. For instance, two DNA-binding regulatory proteins (1ALI and 1E31), though seemingly different from the local-structure perspective, share such a α-helices triangle, and perform similar functionalities [5]. In fact, both belong to the class of zinc finger proteins. However, because this kind of structure is hidden under the pair-wise relationship, it is very unlikely to be identified using the existing frequent subgraph mining approaches. In particular, even if some subgraphs which embed the three α-helices may appear to be frequent, the triangle structure can easily be missed.

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* [4]. 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.

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. For example, we may prefer to distinguish the α-helix triangles of different sizes, and the length of each independent path connecting these α-helices can help to provide such measurement. 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. For instance, in the above example, the relabeling function can use the length of each contracted path as their corresponding edge labels. An additional benefit of the relabeling function is that it can be used to support the mining of *constrained* topological structures.

To summarize, the main contributions of this paper are as follows:

1. We introduce a novel framework for discovering frequent topological structures from graph datasets based on a vertical mining approach.
2. We study the basic properties of relabeling functions, and demonstrate their use for summarization and discovery of

constrained topological structures. Our algorithms push the constraints deep into the mining process maximizing performance gains.

3. We evaluate the scalability and quality of the proposed framework on several real and synthetic datasets. We also demonstrate the use of the framework for discovering novel and meaningful motifs in membrane protein structures.

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* [6, 12]. The topological structures together with relabeling functions provide a powerful mechanism to express various forms of fuzzy matches.

2. TOPOLOGICAL MINORS AND TOPOLOGICAL STRUCTURES

We begin with some basic notations. Let $G = (V, E)$ be a graph, where V is the set of vertices, and E is the set of edges, and $E \subseteq V \times V$. 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. A path is *simple* if its vertices are all distinct, and we only consider simple paths in this paper. 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. For simplicity, we call a path intersecting with other paths only at its ends as an *independent path*. Note that the independent paths are the key tools to study topological structures of a graph.

2.1 Topological Minors

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 [4].

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*.

2.2 Topological Structures

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

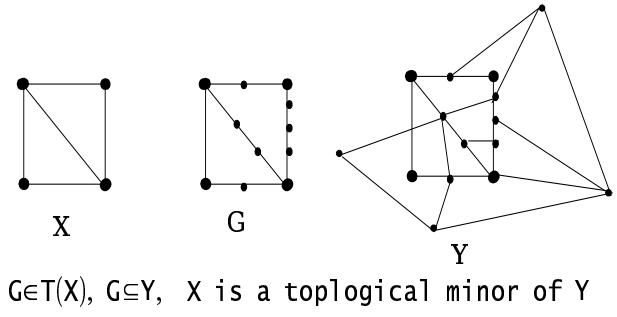


Figure 1: Topological Minor

$(0, 3)$-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.

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. In other words, when later we discover the frequent topological patterns from a graph dataset, the embeddings (subgraphs) that can contribute to the support of such a topological structure should be in a controllable size. Specifically, the following lemma describes the size difference between a graph and its subdivision graph in terms of vertex and edge number.

LEMMA 1. *If a graph G is obtained by a (l, h)-subdivision operation of X, the number of vertices of G, ($|V(G)|$), and the number of edges of G, ($|E(G)|$), are bounded as follows:*

$$|V(X)| + |E(X)| \times l \leq |V(G)| \leq |V(X)| + |E(X)| \times h$$
$$|E(X)| \times (l+1) \leq |E(G)| \leq |E(X)| \times (h+1)$$

The following two lemmas also describe important properties of topological structures of a graph, and their proofs directly follow the above definitions.

LEMMA 2. *Assume X is a (l_1, h_1)-topological minor of G, then for any l and h, where $l \leq l_1$ and $h_1 \leq h$, X is (l, h)-topological minor.*

LEMMA 3. *The number of graphs in the (l, h)-topological space of X ($|T_{l,h}(X)|$) is bounded by $(h - l + 1)^{|E(X)|}$.*

In the following, we will mainly focus on the topological structures ((l, h)-topological minors) of a graph.

2.3 Labeled Graphs

So far, our discussion has focused on unlabeled graphs. Data miners are often more interested in *labeled* graphs. In the following, we extend the concept of topological structures on labeled graphs. Note that unlabeled graphs can be treated as a special case of labeled graphs, where all the vertices and edges have the same label.

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. Later, in Section 4, we will study how to use *relabeling functions* to add labels to these edges. Clearly, the main difference between the topological structures on labeled graphs and on unlabeled graphs is that the vertex labels for the ends

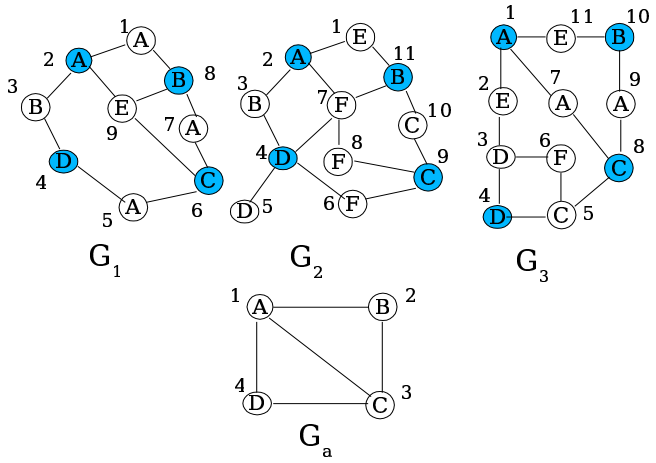


Figure 2: Running Example

of contracted paths are still preserved. Similarly, in an unlabeled graph, such simplification maintains the important topological information from the original graph.

To facilitate our formal discussion of topological structures on labeled graphs, we first define a labeled graph. Let $G = (V, E)$ be an unlabeled graph. Let L_v and L_e be two sets of labels. A vertex labeling function, $l_v : V \rightarrow L_v$, will assign a vertex v with a vertex label $l_v(v) \in L_v$. Similarly, an edge labeling function, $l_e : V \rightarrow L_e$, will assign an edge e with an edge label $l_e(e) \in L_e$. We refer to a graph G labeled by l_v and l_e as a *labeled graph*. A graph G only labeled by the vertex labeling function (l_v) is called a *vertex labeled graph*, and similarly, a graph G only labeled by the edge labeling function (l_e) is referred to as an *edge labeled graph*.

To simplify our discussion, we will mainly focus on the vertex labeled graphs. For example, all the graphs in Figure 2 are vertex labeled graphs. 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, including the (l, h)-subdivision graph, the (l, h)-topological space, and (l, h)-topological minors, are the same as in unlabeled graphs. Therefore, in Figure 2, the vertex labeled graph G_a is a $(1, 1)$-topological minor of the graph G_1, and a $(1, 2)$-topological minor to the graph G_2 and G_3.

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 a frequent topological structure.*

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.

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.

In the following, we first present our approach to efficiently counting the support for a topological structure (Subsection 3.1). Then, we show how we perform a depth-first search to enumerate all the frequent patterns using the counting approach (Subsection 3.2).

3.1 Counting Support for Topological Structures

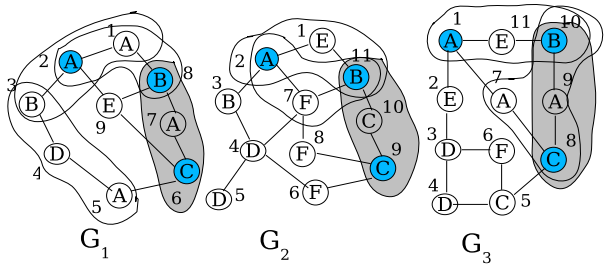
As mentioned before, compared with frequent subgraph mining, one of the main challenges for our mining algorithm is the need to handle the subdivision operation (path contraction) in addition to the subgraph isomorphism testing. To tackle this problem, we use an incremental approach. Consider a topological 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.

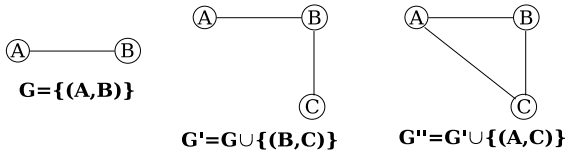
We first decompose G as a collection of edges, i.e., $G = \{e_1\} \cup \{e_2\} \cdots \cup \{e_k\}$. Based on the definition of the subdivision operation, each edge e_i corresponds to an independent path in Y, denoted as $\vec{e_i}$. Therefore, we can also decompose Y as a collection of independent paths, 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 4. *The occurrences of $G \cup \{e\}$ can be represented as*



(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

$\vec{Y_1'} \cup \{\vec{e}\}, \cdots, \vec{Y_n'} \cup \{\vec{e}\}$, *where* $Y_i' \in S_{G,H}, 1 \leq i \leq m$. 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 4 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).

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 becomes the one of how we build its occurrence list efficiently. However, the straightforward 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 straightforward 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 the similar problem also needs to be addressed in frequent subgraph mining algorithms. However, it is even more costly in our algorithm because of the high cost of finding independent paths.

In order to build the occurrence lists efficiently for the topological structures, we try to minimize the number of times the *finding independent paths* 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 for an occurrence $\vec{Y}$ starting from a branch vertex s is illustrated in Figure 4. 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 (in *IndependentPath*). 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 (the *foreach* loop in *RecursivePath*). When we found an independent path (the number of inner vertices) bounded by l and h, we will record this path. Finally, our traversal will trace back when the length of path is greater than the upper bound h. Note that the tracing back operation is associated with uncoloring the visited vertex.

```
global Set Visited, Set PathSet;
IndependentPath(Graph G, Embedding emb, Vertex s)
 Visited ← ExtractEmbeddingVertex(emb);
 PathSet ← ∅;
 RecursivePath(G, s, {s}); {*p.from = s*}
 {* return all the independent paths corresponding to an
  outer edge, bounded by l and h, and staring from s *}
 returnPathSet;

RecursivePath(Graph G, Vertex v, Path p)
 if (|p| − 2 > h){* No. of inner vertices *}
      return;
 Visited ← Visited ∪ {v};
 foreach (v' : (v', v) ∈ G and (v' ∉ Visited)
      p ← p ∪ {v'}; {*p.to = v'*}
      if (|p| − 2 ≥ l)
           PathSet ← PathSet ∪ {p};
      RecursivePath(G, v', p);
      p ← p − {v'};
 Visited ← Visited − {v};
```

Figure 4: Enumerate Independent Paths

Operation Description. In the following, we formally introduce the two key operations mentioned earlier, which are the *Join* operation and the *ExtendOuterEdge* operation. The two operations are sketched in Figure 5. 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.

A 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 deailts of its implementation are omitted.

Correctness. One of the key properties of the topological structure is that all the paths corresponding to the edges in the subdivision graph are independent. In our algorithm, we explicitly maintain the paths corresponding to the edges for a topological structure G, by two operations, *ExtendOuterEdges* and *Join*. Therefore, the correctness of our algorithm depends on whether these paths in an occurrence are independent. Formally, assume that a graph-topological structure G is generated from the following edge sequence: $\{e_0\}, \{e_1\}, \cdots, \{e_k\}$. In our algorithm, an occurrence of G can be represented by the union of the corresponding paths, i.e., $\{\vec{e_0}\}, \{\vec{e_1}\}, \cdots, \{\vec{e_k}\}$. The following lemma states that the independence property is maintained for these edges. Therefore, it implies that our algorithm can correctly generate topological structures for a graph, and henceforth, correctly discover frequent topological structures.

LEMMA 5. *The paths in any occurrence of G, i.e., $\{\vec{e_0}\}, \{\vec{e_1}\}, \cdots, \{\vec{e_k}\}$, are independent.*

Proof: By induction. □

```
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 5: 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 that the two-phase procedure has been proposed and used for efficiently mining frequent subgraphs also [18, 8].

In the first phase of our algorithm, a candidate frequent tree-topological structure can be generated by looking at edges in $[G]_{outer}$. In the second phase, a candidate frequent graph-topological structure can be generated through $[G]_{inner}$. Finally, if a topological structure G' is generated by adding a new edge e on G, $e \in [G]_{io}$, we call G as the *parent* graph of G'. Note that the above treatment is very similar to the algorithms in mining (connected) subgraphs since the frequent topological structures are also graphs.

A difficulty in enumerating frequent topological structures is that one frequent topological structure can be derived from different

parent graphs, i.e. $G_1 \cup \{e_1\} = G_2 \cup \{e_2\}$, where $G_1 \neq G_2$. Clearly, an efficient mining algorithm needs to avoid generating duplicate frequent topological structures. This requires efficient topological structure isomorphism tests. This is why we use a two-phase procedure to enumerate frequent tree and graph topological structures separately. Basically, linear-time algorithms exist for enumerating tree topological structures, and therefore, our first phase can efficiently deal with tree-isomorphism. The complicated cases which require graph isomorphism testing arise only in the second phase.

Our algorithm is sketched in Figure 6. 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 by Nijssen [13] 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 (*nauty* [11]) to avoid duplicating graph-topological structures.

The dominant computational time of our algorithm is in maintaining the edge sets, $[G]_{outer}$ and $[G]_{inner}$, for each topological structure G. Note that when G is a graph-topological structure, we only need to maintain its inner edge set. Our algorithm maintains them in an incremental manner. For a new tree-topological structure, $T \cup \{e\}$, it can inherit some of the inner and outer edges in $[T]_{io}$ through a *Join* operation (the *foreach* loop in *VTreeTS*). However, the new vertex (because of e) in the graph $T \cup \{e\}$ brings new outer edges, which do not appear in $[T]_{outer}$. In our algorithm, the procedure *ExtendOuterEdges* (invoked by *VTreeTS*) generates these new outer edges. For a new graph-topological structure, $G \cup \{e\}$, it only needs to inherit inner edges from its parent's inner edge set $[G]_{inner}$ through the *Join* operation (the *foreach* loop in *VGraphTS*).

```
VTSMining(Dataset D, Support θ, Bound l, h )
 {* Find Frequent Single − Edge Topological Structures*}
 E ← FrequentEdgeTS(D, θ, l, h);
 foreach (e ∈ E)
      VTreeTS(e);

VTreeTS(Tree T)
 {* New Outer Edges of T*}
 E ← ExtendOuterEdges(T);
 [T]outer ←[T]outer ∪ E;
 {*Tree Topological Structure Growing*}
 foreach (e : e ∈[T]outer and
          ValidExtension(T ∪ {e}))
      Te ← T ∪ {e};
      [Te]io ← Join([T]io, e);
      VTreeTS(Te);
 {*Enumerating Graph Topological Structures *}
 VGraphTS(T);


VGraphTS(Graph G)
 foreach (e : e ∈[G]inner and
          CanonicalExtension(G ∪ {e})
      Ge ← G ∪ {e};
      [Ge]inner ← Join([G]inner, e);
      VGraphTS(Ge);
```

Figure 6: Algorithm Framework for Mining Topological Structures

4. MINING TOPOLOGICAL STRUCTURES WITH RELABELING FUNCTIONS

As discussed before, topological structures of a subgraph are extracted through compressing the inner vertices and edges of their independent paths into corresponding unlabeled edges. Two paths that have a different set of inner vertices and edges can be treated as the same, as long as the labels of their ends are the same. However, in many applications, the labels for inner vertices (and the inner edges) can provide important additional information. In order to reflect such information in the topological structures, we allow users to define a *relabeling function*, which assign labels to the edge in topological structures corresponding to the path that has been contracted.

In this section, we first formally introduce relabeling functions and briefly discuss their efficient implementation in the mining process. Then, we discuss how we can use relabeling functions to perform *constraint* topological structure mining. Finally, we relate relabeling functions with *approximate pattern mining*, and present how our framework can handle *fuzzy chains* in molecular fragments [12].

4.1 Relabeling Functions and Their Implementation

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 $\mathcal{L}$. To facilitate our discussion, the set $\mathcal{L}$ always contains a *null* symbol, $\emptyset$. Note that a given path p can usually be expressed in two different formats, p and $\overline{p}$, where $\overline{p}$ is the reverse of p, i.e. $\overline{p} = (v_k, \cdots, v_1, v_0)$. Clearly, not any map between $\mathcal{P}$ and $\mathcal{L}$ is valid, because they have to be consistent with respect to both p and $\overline{p}$. Therefore, a valid relabeling function f needs to satisfy the *reverse symmetric* property, i.e. $f(p) = f(\overline{p})$, for a given path $p \in \mathcal{P}$.

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 $p = (v_0, v_1, \cdots, v_k)$, $f(p) = k - 1$. Clearly, it satisfies the reverse symmetric property. Note that in this way, the edges in the topological structures become labeled. In order to efficiently mine frequent topological structures utilizing these relabeling functions, we need to push relabeling deeply into the support counting process. In our mining algorithm, the *ExtendOuterEdge* scans these independent paths generated by the routine *IndependentPath*, and contracts these paths into corresponding edges ($e \leftarrow$ *Edge(p.from,p.to)* in Figure 5, p is an independent path). To implement a relabeling function, we need to compute a new label using the relabeling function f, $f(p)$, where p is an independent path, and then use it to label the corresponding edge, *Edge(p.from,p.to)*. In particular, if it is the null symbol $\emptyset$, we simply remove this path. Otherwise, we put this path into the occurrence list for the contracted edges with this new label $f(p)$.

	A	B	C	D	E
A	$(A\|B)^1\|D^2$	$A\|B\|C^2\|E$	$?^1\|?^2$	$\emptyset$	$D\|E\|GF$
B	$A\|B\|C^2\|E$	$(A\|B)^1\|D^2$	$?^1\|?^2$	$AB\|BC\|D$	$?^1\|?^2$
C	$?^1\|?^2$	$?^1\|?^2$	$(A\|B)^1\|D^2$	BB	$?^1\|?^2$
D	$\emptyset$	BB	$BA\|CB\|D$	$(A\|B)^1\|D^2$	$?^1\|?^2$
E	$D\|E\|FG$	$?^1\|?^2$	$?^1\|?^2$	$?^1\|?^2$	$(A\|B)^1\|D^2$

Figure 7: Constraint Condition Table

4.2 Mining Topological Structures with Constraint Conditions

In this subsection, we study a specific type of relabeling function: *constraint conditions*. Such constraint conditions can help data miners focus only on certain types of independent paths to be contracted. In this way, for the edges in a frequent topological structure, the user can have an idea of what kind of paths (subgraphs) are contributing to them. In the following, we consider a powerful mechanism to specify such constraint conditions, which is based on regular expressions. For example, the following expression

$$\{A\} : A|B|C^2|E : \{B\}$$

requires that an independent path in the graphs starting from a vertex with label A, ending with a vertex with label B, either have length one with the inner vertices labeled as A,B,E or have length two with the inner vertices both labeled with C.

Such constraint conditions can be transformed into a table format: a table C with $|L_v|$ rows and $|L_v|$ columns, where L_v is the set of all the vertex labels. (The details of the transformation procedure is omitted for simplicity.) Each row and each column corresponds a label in L_v; and each cell has a regular expression. A cell C_{l_i,l_j} specifies a path starting with a label l_i, ending with a label l_j, and with the inner path labeled as C_{l_i,l_j}, can be contracted into an edge $\{l_i, l_j\}$. Specifically, $C_{l_i,l_j} \subseteq L_v^l \cup \cdots \cup L_v^h$ since the length of each contracted path needs to be bounded by l and h. For example, Figure 7 illustrates such a table for the vertex label set $\{A, B, C, D, E\}$ in the table format. Note that an empty set ($\emptyset$) in a cell C_{l_i,l_j} suggests no path can be contracted as $\{l_i, l_j\}$; the symbol (?) represents the set L_v. Finally, the table also satisfies the *reverse symmetric* property: $C_{l_i,l_j} = \overline{C_{l_j,l_i}}$.

Mathematically, we can treat a regular-expression based condition as a type of relabeling function. Specifically, we can define the new edge-label set $\mathcal{L}$ of topological structures as $\mathcal{L} = \{1, \emptyset\}$. The symbol 1 represents that a path is acceptable by the constraint condition, and the symbol $\emptyset$ corresponds to the rejection of a path. Therefore, for a given path $p = (v_0, v_1, \cdots, v_k)$, if it satisfies the constraint condition, the relabeling function returns 1, otherwise, it returns $\emptyset$ (in other words, this path is simply removed). The detailed implementation is as follows. Basically, for each candidate path, we will use its ends to find the corresponding regular expression in the constraint table. To facilitate processing, we will map the regular expressions in the constraint table into DFAs (Deterministic Finite Automaton). Then, we will test if the path is accepted or rejected by the DFA. If it is rejected by the DFA, we will simply remove this path.

4.3 Mining Fuzzy Chains using Relabeling Functions

In the following, we study how we can use topological structure together with relabeling functions to implement one type of *approximate pattern mining*, which is mining fuzzy chains in chemical compounds [12].

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 ($\|V\| > 8$)	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*

We begin with the definitions of fuzzy chains. A chain in a chemical compound satisfies the following conditions: 1) every vertex corresponding to an atom in a chain has the same type, 2) every vertex in the chain must have exactly two edges (labeled with *single bound* type) to other vertex, and 3) a chain always consists of the maximal possible number of atoms satisfying the first two conditions and must have a minimum length of one. For a biologist or a chemist, two chains are equivalent if both chains have the same atom type and the lengths of the two chains can be different and are bounded by user-defined ranges. Since such chains do not need an exact match, we call them *fuzzy chains*.

Let us consider the length of the fuzzy chains to be between two and four atoms (the common case). Then, the frequent chemical fragments with such fuzzy chains can be mined in our framework as $(0, 4)$-topological minors with the following relabeling function. For a given independent path, 1) if the path has no inner vertex, use the original edge label as the edge label for the new edge, 2) if the path has a number of inner vertices between 2 and 4, we check the following conditions for the path to see if it satisfies the chain condition, and return the atom type in the chain to label the new edge for the true case, and 3) remove the path in other conditions.

The method discussed in Subsection 4.1 can be used to implement this relabeling function.

5. 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 [14]. 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 pa-

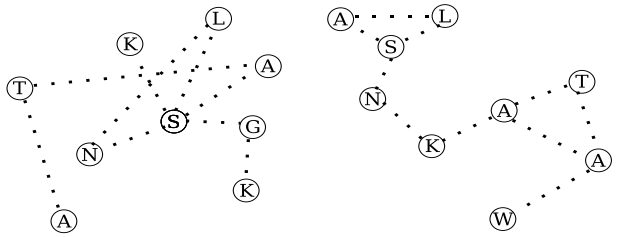


Figure 8: Frequent Topological Structures Discovered by *TSMiner*

rameter 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 8 shows two such large topological structures discovered by our toolkit. 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[14]. 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.

6. EXPERIMENTAL RESULTS

In this section, we will study the performance of our new algorithm, *TSMiner*, focusing on the following three issues: the scalability of the algorithm, how the parameters, l, h, and the support level θ, affect the performance, and how the relabeling functions affect performance. 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.

6.1 Datasets Description

Our experiments used both synthetic and real datasets, containing vertex labeled graphs, i.e., the edge labels were not considered.
Synthetic Datasets: The synthetic datasets were generated from the graph generator provided by Kuramochi and Karypis at the University of Minnesota. Though this generator was originally designed for evaluating frequent subgraph mining algorithms, we have used it to study the performance and scalability of the algorithm for mining frequent topological structures. In our experiments, the following parameters were used to generate datasets: 1) $|D|$, the total number of graphs to be generated, 2) $|T|$, the average number of edges for the generated graphs, 3) $|L|$, the total number of potentially frequent subgraphs, 4) $|I|$, the average number of edges in each potentially frequent subgraph, and 5) $|V|$, the total number of available labels for the vertices. In our experiments, we fixed $T = 20$, $L = 200$, $I = 5$, and we vary V, the total of vertex labels, to be between 5 and 20.
Chemical Compound Dataset from PTE: This dataset was originally used for the Predictive Toxicology Evaluation Challenge [17]. 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. Overall, the entire dataset contains a total of 66 vertex labels. For simplicity, we refer this dataset as *Chemical340*.

6.2 Performance Evaluation

Scalability:. For the scalability study, we rely on the synthetic datasets. Figure 9 shows the performance of *TSMiner* under different conditions. In Figure 9(a) and (c), 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 9(b) and (d), 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* [18]. 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.

Number of Patterns and Running Time with respect to 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 10 presents the experimental results on the real dataset Chemical340. Figure 10 (a) shows the number of path, tree, and graph topological structures discovered by *TSMiner* at a support of 200. The primary observations of note here are: when using traditional graph mining algorithms ($l = 0$ and $h = 0$ in our tool) no frequent graph patterns are found; upon increasing the value of h to 1, 2, and 3, we are able to identify frequent graph structures; and finally from Figure 10(b) we can see that as the value of h is increased the running running time of our tool increases as it has to evaluate more candidate patterns and the cost for generating each pattern increases (the independent paths become longer). Figure 10 (c) and (d) 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.

The Effect of Relabeling Functions. In this study, we focus on how relabeling functions impact the performance of our algorithm. We study two types of relabeling functions. The first uses the length of the contracted path to relabel the corresponding edge, and is referred to as *length-relabeling* (see Subsection 4.1). The second involves constraining each path with a regular expression, or DFA, and is referred to as *DFA-relabeling* (see Subsection 4.2).

Figure 11(a) and (b) shows the number of patterns being generated by TSMiner without length-relabeling and the corresponding running time. The result is quite interesting. Using relabeling, *more frequent patterns are being generated, however, the running time decreases significantly*. Basically, as we relax the condition for the length of independent path for a given topological structure, many occurrences with independent paths of different length maps to it. As we perform length-relabeling, the topological structures will be further categorized based on the size of its occurrences. This reduces the number of occurrences, as the condition for a subgraph being the subdivision graph of a topological structure becomes stricter. Therefore, such a relabeling function can improve the performance of *TSMiner*.

Figure 11(c) and (d) show the number of patterns being generated by *TSMiner* without DFA-relabeling and the corresponding

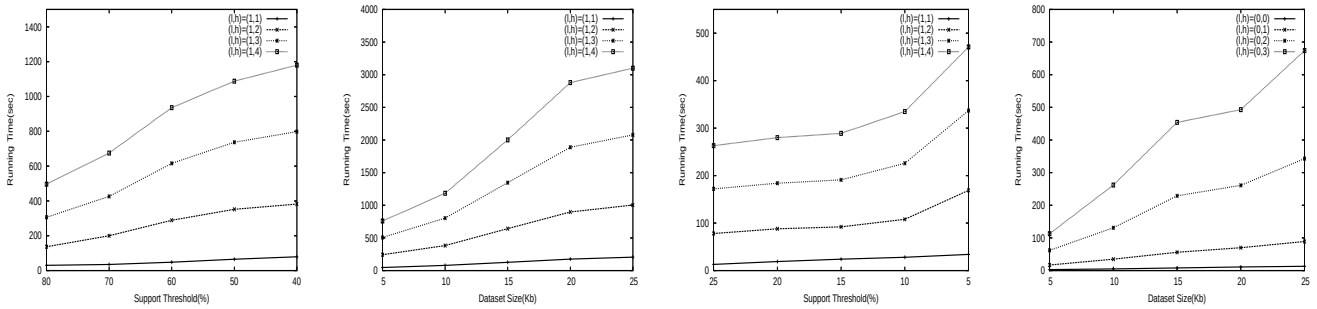


Figure 9: (a) Varying Support(D10kV5) (b) Varying Dataset Size(D*kV5, Sup=40%)) (c)Varying Support (D10kV20) (d) Varying Dataset Size (D*kV20, Sup=20%)

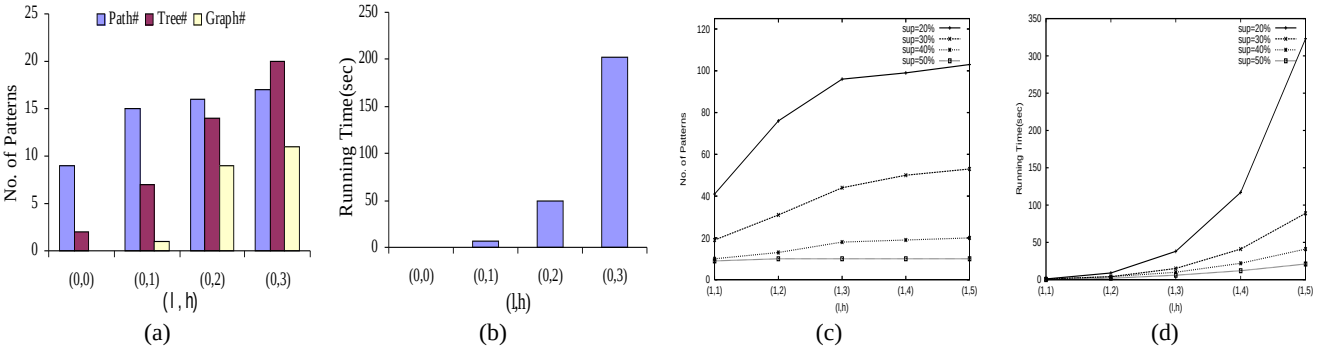


Figure 10: Chemical340 (a)No. of Patterns(Support=200) (b)Running Time(Support=200) (c)No. of Patterns(Varying Support) (d)Running Time(Varying Support)

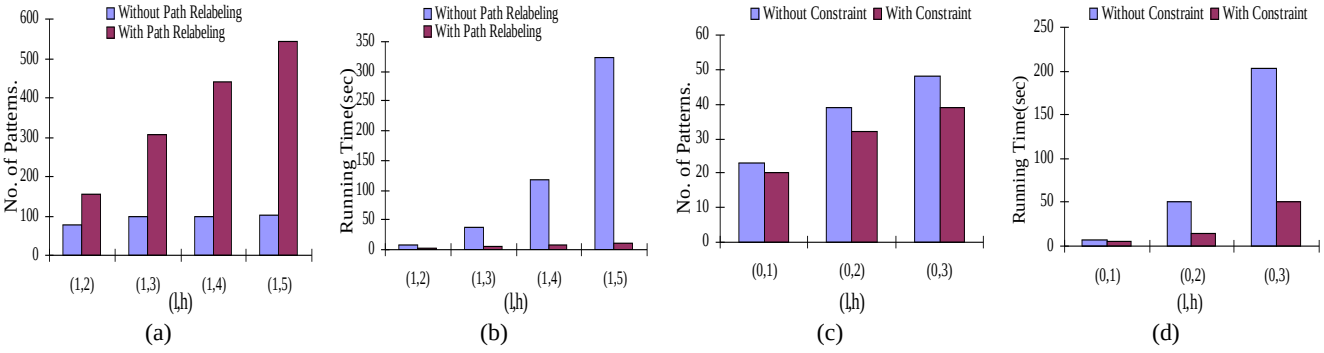


Figure 11: Relabeling with the Path Length on Chemical340 (Support=200) (a) No. of Patterns (b) Running Time; DFA Constraints on Chemical340 (Support=200) (c)No. of Patterns (d)Running Time

running times. The constraint conditions are generated as follows. We first randomly generate a group of 100 DFAs to describe the conditions of an independent path. In particular, we use a parameter r to control how likely it is that an independent path can be accepted. In our experiment, for the independent paths having length 1, 2, and 3, their possibilities to be accepted were 0.5, 0.25, and 0.125, respectively. Then, each cell in the constraint condition table (defined in Subsection 4.2) is randomly assigned with a generated DFA. As shown in the figures, the DFA-relabeling reduces the number of frequent topological patterns being generated, as well as the running time.

7. RELATED WORK

The early efforts on discovering useful patterns from graph datasets include the SUBDUE system [2] and WARMER algorithm [3]. The SUBDUE system relies on the Minimal Description Length (MDL) principle and a greedy strategy to find a subset of frequently occurring subgraphs. The WARMER algorithm combines Inductive Logical Programming (ILP) with Apriori's level-wise search strategy [1] to find a wide class of frequent substructures. However, it is well known that ILP-based approaches are still quite expensive computationally, and do not scale very well to large datasets.

Recently, frequent *subgraph* mining approach has received much attention. This approach enumerates all frequent patterns defined by a class of subgraphs. The AGM algorithm [9] was the first to be proposed in this category. It can find all frequent *induced* subgraphs in a graph dataset. A subgraph G_s of G is *induced* if the subgraph G_s contains all edges in G connecting its vertices. The more recent efforts focus on discovering all frequent *connected* subgraphs. Several efficient algorithms, such as FSG [10], gSpan [18], FFSM [7], and Gaston [13], have been proposed to mine these kind of patterns. Two different types of search strategies are used in these algorithms: apriori's level-wise strategy and Eclat's [20] depth first search strategy. The experimental results show in most of the cases, the latter is more computationally efficient, and the former is more memory efficient. The framework proposed in this paper enumerates a more *generalized* pattern in a graph dataset. The connected subgraph mining is a special case for this new type of topological structure mining. To efficiently enumerate these kind of patterns, our new algorithm, TSMiner, also uses Eclat's DFS strategy. However, the critical difference is that the new algorithm has to use with the topological minor test, which is more complicated than the subgraph isomorphism test.

Hofer *et al.* [6], as well as Parthasarathy and Coatney[15], make the observation that in many real world applications, a *fuzzy* match is needed, and not an exact match. As we demonstrate in our work, such fuzziness can be handled in our framework through the design of suitable relabeling functions.

To reduce the computational costs associated with enumerating frequent subgraphs, researchers have looked at generating *closed* [19], *maximal* [8] and *free-tree based* [16] frequent subgraph patterns. Such concepts can be naturally extended to handle frequent topological patterns as well. Further, several researchers have studied how to find efficient patterns in the tree dataset, such as an XML dataset [21]. Frequent topological patterns could be defined on tree datasets as well, and our algorithm is clearly capable of enumerating such patterns.

8. 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. 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.

9. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[2] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[3] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press., 1998.

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

[5] Leonard P. Freedman, Keith R. Yamamoto, Ben F. Luisi, and Paul B Sigler. More fingers in hand. *Cell*, 54(4):444, 1988.

[6] 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.

[7] Jun Huan, Wei Wang, Deepak Bandyopadhyay, Jack Snoeyink, Jan Prins, and Alexander Tropsha. Mining protein family-specific residue packing patterns from protein structure graphs. In *Eighth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 308–315, 2004.

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

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

[10] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, 2001.

[11] Brendan McKay. Practical graph isomorphism. *Congr. Numer.*, 30:45–87, 1981.

[12] 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.

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

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

[15] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. *IEEE International Conference on Data Mining*, pages 362–369, 2002.

[16] Ulrich Ruckert and Stefan Kramer. Frequent free tree discovery in graph data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.

[17] A. Srinivasan, R.D. King, S.H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1–6. Morgan-Kaufmann, 1997.

[18] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 721, 2002.

[19] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003.

[20] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W.Li. Parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343–373, December 1997.

[21] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2002.