Finding Frequent Patterns in a Large Sparse Graph*

Michihiro Kuramochi and George Karypis

Department of Computer Science & Engineering/
Digital Technology Center/Army HPC Research Center
University of Minnesota

4-192 EE/CS Building, 200 Union St SE
Minneapolis, MN 55455

{kuram, karypis}@cs.umn.edu

Technical Report #03-039

Last updated on September 25, 2003 at 2:11 PM

Abstract

This paper presents two algorithms based on the horizontal and vertical pattern discovery paradigms that find the connected subgraphs that have a sufficient number of edge-disjoint embeddings in a single large undirected labeled sparse graph. These algorithms use three different methods to determine the number of the edge-disjoint embeddings of a subgraph that are based on approximate and exact maximum independent set computations and use it to prune infrequent subgraphs. Experimental evaluation on real datasets from various domains show that both algorithms achieve good performance, scale well to sparse input graphs with more than 100,000 vertices and around 200,000 edges, and significantly outperform previously developed algorithms.

Keywords pattern discovery, frequent subgraph, graph mining.

1 Introduction

Data mining is the process of automatically extracting new and useful knowledge hidden in large datasets. This emerging discipline is becoming increasingly important as advances in data collection have led to the explosive growth in the amount of available data.

In recent years, there has been an increased interest in developing data mining algorithms that operate on graphs. Such graphs arise naturally in a number of different application domains including network intrusion [47, 41], semantic web [4], behavioral modeling [67, 55], VLSI reverse engineering [70], link analysis [34, 40, 39, 58], and chemical compound classification [14, 43, 22, 16]. Moreover, they can be used to effectively model the structural and relational characteristics of a variety of datasets arising in other areas such as physical sciences (e.g., chemistry, fluid dynamics, astronomy, structural mechanics, and ecosystem modeling), life sciences (e.g., genomics, proteomics, pharmacogenomics, and health informatics), and home-land defense (e.g., information assurance, network intrusion, infrastructure protection, and terrorist-threat prediction/identification).

The focus of this paper is on developing algorithms for a particular data mining task, which is that of finding frequently occurring patterns in graph datasets. Frequent patterns play a critical role in many data mining tasks as they can be used among other to derive association rules [1], act as composite features for classification algorithms [14, 56, 63, 51, 22, 50, 15], cluster the (graph) transactions [1, 48, 35, 36, 49, 24], and help in determining the similarity between graphs [54, 23, 42, 59, 9, 49, 13, 60, 66]. Within the context of graphs, the most widely used definition of a pattern is that of a connected subgraph [8, 68, 32, 29, 69, 30, 44] and is the definition that we will use in this paper. However, different pattern definitions have been proposed as well [32].

There are two distinct problem formulations for frequent pattern mining in graph datasets that are referred to as the *graph-transaction setting* and the *single-graph setting*. In the graph-transaction setting, the input to the pattern mining algorithm is a set of relatively small graphs (called transactions), whereas in the single-graph setting the input data is a single large graph. The difference affects the way the frequency of the various patterns is determined. For the graph-transaction setting, the frequency of a pattern is determined by the number of graph transactions that the pattern occurs in, irrespective of how many times a pattern occurs in a partic-

*This work was supported in part by NSF CCR-9972519, EIA-9986042, ACI-9982274, ACI-0133464, and ACI-0312828; the Digital Technology Center at the University of Minnesota; and by the Army High Performance Computing Research Center (AHPCRC) under the auspices of the Department of the Army, Army Research Laboratory (ARL) under Cooperative Agreement number DAAD19-01-2-0014. The content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

ular transaction, whereas in the single-graph setting, the frequency of a pattern is based on the number of its occurrences (i.e., embeddings) in the single graph. Due to the inherent differences of the characteristics of the underlying dataset and the problem formulation, algorithms developed for the graph-transaction setting cannot be used to solve the single-graph setting, whereas the latter algorithms can be easily adapted to solve the former problem.

In recent years, a number of efficient and scalable algorithms have been developed to find patterns in the graph-transaction setting [8, 68, 32, 29, 69, 30, 44]. These algorithms are complete in the sense that they are guaranteed to discover all frequent subgraphs and were shown to scale to very large graph datasets. However, developing algorithms that are capable of finding patterns in the single-graph setting has received much less attention, despite the fact that this problem setting is more generic and applicable to a wider range of datasets and application domains than the other. Moreover, existing algorithms that are guaranteed to find all frequent patterns [21, 65] or algorithms that are heuristic, such as GBI [71] and SUBDUE [28] which tend to miss a large number of frequent patterns, are computationally expensive and do not scale to large datasets.

Developing algorithms that find the complete set of frequent patterns in the single-graph setting is the focus of this paper. We present two computationally efficient algorithms that can find subgraphs which are frequently embedded within a large sparse graph. The first algorithm, called HSIGRAM, follows a *horizontal approach* and finds the frequent subgraphs in a breadth-first fashion, whereas the second algorithm, called VSIGRAM, follows a *vertical approach* and finds the frequent subgraphs in a depth-first fashion. These algorithms incorporate efficient algorithms for candidate generation and frequency counting that allow them to scale to graphs containing over 100,000 vertices and find patterns with relatively low occurrence frequency. Our experimental evaluation on six real graphs shows that both HSIGRAM and VSIGRAM achieve reasonably good performance, scale to large graphs, and substantially outperform previously developed approaches for solving similar or simpler versions of the problem.

The rest of this paper is organized as follows. Section 2 defines the graph model that we use, reviews some graph-related definitions, and introduces the notation that is used in the paper. Section 3 surveys related research in this area. Section 4 formally defines the problem of frequent subgraph discovery and discusses the challenges associated with finding them in a computationally efficient manner. Section 5 describes in detail the HSIGRAM and VSIGRAM algorithms that we developed for solving the problem of frequent subgraph discovery from a single large sparse graph. Section 6 provides a detailed experimental evaluation of the HSIGRAM and VSIGRAM algorithms on various real datasets and compares them against existing algorithms. Finally, Section 7 provides some concluding remarks.

2 Definitions and Notation

A **graph** $G = (V, E)$ is made of two sets, the set of vertices V and the set of edges E. Each edge itself is a pair of vertices, and throughout this paper we assume that the graph is undirected, i.e., each edge is an unordered pair of vertices. Furthermore, we will assume that the graph is **labeled**. That is, each vertex and edge has a label associated with it that is drawn from a predefined set of vertex labels (L_V) and edge labels (L_E). Each vertex (or edge) of the graph is not required to have a unique label and the same label can be assigned to many vertices (or edges) in the same graph. If all the vertices and edges of the graph have the same vertex and edge label assigned to them, we will call this graph **unlabeled**.

Given a graph $G = (V, E)$, a graph $G_s = (V_s, E_s)$ is a *subgraph* of G if and only if $V_s \subseteq V$ and $E_s \subseteq E$. A graph is **connected** if there is a path between every pair of vertices in the graph. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if they are topologically identical to each other, that is, there is a mapping from V_1 to V_2 such that each edge in E_1 is mapped to a single edge in E_2 and vice versa. In the case of labeled graphs, this mapping must also preserve the labels on the vertices and edges. An **automorphism** is an isomorphism mapping where $G_1 = G_2$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of **subgraph isomorphism** is to find an isomorphism between G_2 and a subgraph of G_1, i.e., determine whether or not G_2 is included in G_1.

Given a subgraph G_s and a graph $\mathcal{G}$, two embeddings of G_s in $\mathcal{G}$ are called **identical** if they use the same set of edges of $\mathcal{G}$, and they are called **edge-disjoint** if they do not have any edges of $\mathcal{G}$ in common. Given a set of all embeddings of a particular subgraph G_s in a graph $\mathcal{G}$, the **overlap graph** of G_s is a graph obtained by creating a vertex for each non-identical embedding and creating an edge for each pair of non-edge-disjoint embeddings. An example of a subgraph and its overlap graph are shown in Figure 2.

The notation that we will be using throughout the paper is shown in Table 1.

2.1 Canonical Labeling

One of the key operations required by any frequent subgraph discovery algorithm is a mechanism by which to check whether two subgraphs are identical or not. One way of performing this check is to perform a graph isomorphism operation. However, in cases in which many such checks are required among the same set of subgraphs, a better way of performing this task is to assign to each graph a unique *code* (i.e., a sequence of bits, a string, or a sequence of numbers) that is invariant on the ordering of the vertices and edges in the graph. Such a code is referred to as the **canonical label** of a graph $G = (V, E)$ [61, 18], and we will denote it by $\text{cl}(G)$. By using canonical labels, we can check whether or not two graphs are identical by checking to see whether they have identical canonical labels. Moreover, by comparing the canonical labels we can obtain a complete ordering of a set of graphs in a unique and deterministic way, regardless of the

Table 1: Notation used throughout the paper

Notation	Description
k-subgraph	A connected subgraph with k edges (also written as a size-k subgraph)
G^k, H^k	Graphs of size k
$E(G)$	Edges of a graph G
$V(G)$	Vertices of a graph G
cl(G)	Canonical label of a graph G
dia(G)	Diameter of a graph G
a, b, c, e, f	Edges
u, v	Vertices
$d(v)$	Degree of a vertex v
$l(v)$	Label of a vertex v
$l(e)$	Label of an edge e
$H = G - e$	H is a graph obtained by the deletion of edge $e \in E(G)$
$\mathcal{G}$	Input graph
$\mathcal{G}_i$	$\mathcal{G}$'s connected component
$\mathcal{S}(G^{k+1})$	Set of all connected size-k subgraphs of G^{k+1}
$\mathcal{M}(G) = \{m_i\}$	All embeddings of a subgraph G in $\mathcal{G}$
$\mathcal{A}(G) = \{e_i\}$	All anchor edges of a subgraph G in $\mathcal{G}$
C	Candidate subgraph
$\mathcal{C}^k$	Set of candidates with k edges
$\mathcal{C}$	Set of all candidates
F	Frequent subgraph
$\mathcal{F}^k$	Set of frequent k-subgraphs
$\mathcal{F}$	Set of all frequent subgraphs
k^*	Size of the largest frequent subgraph in $\mathcal{G}$
L_E	Set of all edge labels in $\mathcal{G}$
L_V	Set of all vertex labels in $\mathcal{G}$

original vertex and edge ordering.

A simple way of defining the canonical label of a graph is as the string obtained by concatenating the upper triangular entries of the graph's adjacency matrix when this matrix has been symmetrically permuted so that this string becomes the lexicographically largest (or smallest) over the strings that can be obtained from all such permutations. This is illustrated in Figure 1 that shows a graph G^3 and the permutation of its adjacency matrix[1] that leads to its canonical label "*aaazyx*". In this code, "*aaa*" was obtained by concatenating the vertex-labels in the order that they appear in the adjacency matrix and "*zyx*" was obtained by concatenating the columns of the upper triangular portion of the matrix. Note that any other permutation of G^3's adjacency matrix will lead to a code that is lexicographically smaller (or equal) to "*aaazyx*". If a graph has $|V|$ vertices, the complexity of determining its canonical label using this scheme is in $O(|V|!)$ making it impractical even for moderate size graphs. Note that the problem of determining the canonical label of a graph is equivalent to determining isomorphism between graphs, because if two graphs are isomorphic with each other, their canonical labels must be identical. Both canonical labeling and determining graph isomorphism are not known to be either in P or in NP-complete [18].

In practice, the complexity of finding a canonical labeling of a graph can be reduced by using various heuristics to narrow down the search space or by using alternate canonical label definitions that take advantage of special properties that may exist in a particular set of graphs [53, 52, 18]. As part of our earlier research we have developed such canonical labeling algorithm that fully makes use of edge- and vertex-

[1]The symbol v_i in the figure is a vertex ID, not a vertex label, and blank elements in the adjacency matrix means there is no edge between the corresponding pair of vertices.

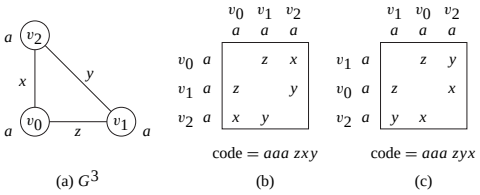


Figure 1: Simple examples of codes and canonical adjacency matrices

labels for fast processing and various vertex invariants to reduce the complexity of determining the canonical label of a graph [45, 46]. Our algorithm can compute the canonical label of graphs containing up to 50 vertices extremely fast and will be the algorithm used to compute the canonical labels of the different subgraphs in this paper.

2.2 Maximum Independent Set

As discussed later in Section 4, our frequent subgraph discovery algorithm focuses on finding subgraphs whose embeddings are edge-disjoint. A critical step in obtaining this set of edge-disjoint embeddings for a particular subgraph is to find the maximum independent set of its overlap graph. Given a graph $G = (V, E)$, a subset of vertices $I \subset V$ is called **independent** if no two vertices in I are connected by an edge in E. An independent set I is called **maximal independent set** for every vertex v in I if there is an edge in E that connects v to a vertex in $V \setminus I$. A maximal independent set I is called **maximum independent set** (MIS) if I contains as many vertices of V as possible.

The problem of finding the MIS of a graph was among the first problems proved to be in NP-complete [19], and remains so even for bounded degree graphs. Moreover, it has been shown that the size of MIS cannot be approximated even within a factor of $n^{1-o(1)}$ in polynomial time [17]. However, the importance of the problem and its applicability to a wide-range of domains has attracted a considerable amount of research. This research has been focused on developing both faster exact algorithms as well as approximate algorithms. The faster exact algorithm to date is the algorithm by Robson [62] that solves the MIS problem in time $O(1.211^n)$, making it possible to solve in reasonable amount of time problem instances containing up to around 100 vertices. In this study, we used a fast implementation of the exact **maximum clique** (MC) problem solver *wclique* [57] instead of those fast exact MIS algorithms. Because the MIS problem on a graph G is equivalent to the MC problem on a G's complement graph $\bar{G}$, we can use *wclique* as a fast exact MIS algorithm (EMIS). Heuristic algorithms focus on finding maximal independent sets whose size is bounded in terms of the size of the optimal solution, and a number of such methods have been developed [27, 6, 38, 25].

One of the most widely used heuristic is the **greedy algorithm** (GMIS) which selects a vertex of the minimum degree, deletes that vertex and all of its neighbors from the graph, and repeats this process until the graph becomes empty. A recent detailed analysis of the GMIS algorithm has

shown that it produces reasonably good approximations of the MIS for bounded- and low-degree graphs [25]. In particular, for a graph G with a maximum degree Δ and an average degree $\overline{d}$, the size $|I|$ of the MIS satisfies the following:

$$|I| \leq \min\left(\frac{\Delta+2}{3}|\mathsf{GMIS}(G)|, \frac{\overline{d}+2}{2}|\mathsf{GMIS}(G)|\right) \quad (2.1)$$

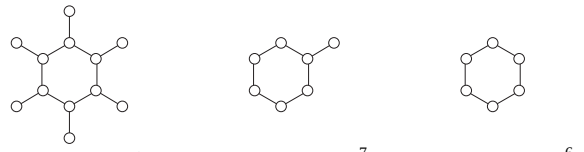
where $|\mathsf{GMIS}(G)|$ is the size of the approximate MIS found by the GMIS algorithm.

3 Related Work

The previous research on finding frequent subgraphs in graph datasets falls under two categories. The first category contains algorithms for finding subgraphs that occur multiple times in a single input graph [71, 28, 21, 65] and are directly related to the algorithms presented in this paper, whereas the second category contains algorithms that find subgraphs that occur frequently across a database of small graphs [14, 31, 43, 45, 33, 8, 68, 32, 29, 30, 44]. Between these two classes of algorithms, those developed for the latter problem are in general more mature as they have moderate computational requirements and scale to large datasets.

In the rest of this section, we will describe on the related research for the single-graph setting as it is directly related to the topic of the paper.

The most well-known algorithm for finding recurring subgraphs in a single large graph is the SUBDUE system, originally developed in 1994, and improved over the years [28, 10, 12, 11]. SUBDUE is an approximate algorithm and finds patterns that can compress the original input graph by substituting those patterns with a single vertex. In evaluating the extent to which a particular pattern can compress the original graph it uses the minimum description length (MDL) principle, and employs a heuristic beam search to narrow the search-space. These approximations improve its computational efficiency but at the same time it prevents it from finding subgraphs that are indeed frequent. GBI [71] is another greedy heuristics based algorithm similar to SUBDUE. Ghazizadeh and Chawathe [21] developed an algorithm called SEuS that uses a data structure called *summary* to construct a lossy compressed representation of the input graph. This summary is obtained by collapsing together all the vertices of the input graph that have the same label and is used to quickly prune infrequent candidates. As the authors indicate, this summary data-structure is useful only when the input graph contains a relatively small number of frequent subgraphs with high frequency, and is not effective if there are a large number of frequent subgraphs with low frequency. Finally, Vanetik, Gudes and Shimony [65] presented an algorithm for finding all frequently occurring subgraphs from a single labeled undirected graph using the maximum number of edge-disjoint embeddings of a graph as a measure of its frequency. Each subgraph is represented by its minimum number of edge-disjoint paths (*path number*), and use a level-by-level approach to grow the patterns based on their path-number. Their emphasis is on efficient candidate generation and no special attention is paid for frequency counting.



(a) Size-12 graph $\mathcal{G}$ (b) Size-7 subgraph G^7 (c) Size-6 subgraph G^6

Figure 3: Patterns with the non-monotonic frequency

4 Discovering Frequent Patterns in a Single Graph: Problem Definition

A fundamental issue that needs to be considered by any frequent subgraph discovery problem formulation similar to the single-graph setting is the counting method of the occurrence frequency. In general, there are two possible methods of the frequency counting. According to the first method, two embeddings of a subgraph are considered different, as long as they differ by at least one edge (i.e., non-identical). As a result, arbitrary overlaps of embeddings of the same subgraph are allowed. On the other hand, by the second method, two embeddings are considered different, only if they do not share edges (i.e., they are edge-disjoint). These two methods are illustrated in Figure 2. In this example, there are three possible embeddings of the subgraph shown in Figure 2(1) in the input graph of Figure 2(2). Two of these embeddings (Figures 2(3) and (5)) do not share any edges, whereas the third embedding (Figure 2(4)) shares edges with the other two. Thus, if we allow overlaps, the frequency of the subgraph is 3, and if we do not it is 2.

These two ways of counting the frequency of a subgraph lead to problems with dramatically different characteristics. If we allow arbitrary overlaps between non-identical embeddings, then the resulting frequency is not any longer downward closed (i.e., the frequency of a subgraph does not monotonically decrease as a function of its length). This is illustrated in Figure 3. Both G^7 and G^6 are subgraphs of $\mathcal{G}$. Although the smaller subgraph G^6 has only one non-identical embedding, the larger G^7 has six non-identical embeddings. On the other hand, if we determine the frequency of each subgraph by counting the maximum number of its edge-disjoint embeddings, then the resulting frequency is downward closed [65].

Being able to take advantage of a frequency counting method that is downward closed is essential for the computational tractability of most frequent pattern discovery algorithms. For this reason, our problem formulations uses edge-disjoint embeddings. Given this, one way of formulating the frequent subgraph discovery problem for the single-graph setting as follows [65]:

Definition 1 (Exact Discovery Problem) *Given an input graph $\mathcal{G}$ which is undirected and labeled, and a parameter f, find all connected undirected labeled subgraphs that have at least f edge-disjoint embeddings in $\mathcal{G}$.*

Unfortunately quite often this problem can be intractable. By

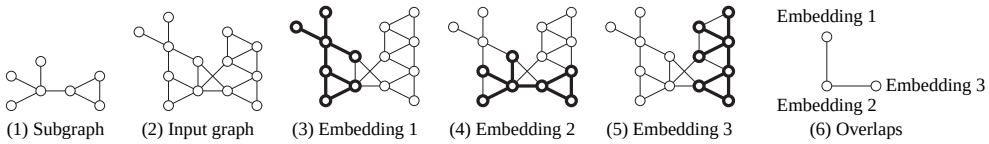


Figure 2: Overlapped embeddings

this definition, in order to determine if a subgraph is frequent or not, we need to find whether the overlap graph of its non-identical embeddings contain an independent set whose size is at least f. When a subgraph is relatively frequent compared to the frequency threshold f, by using approximate MIS algorithms we can quickly tell that such a subgraph is actually frequent. However, in the cases in which the approximate MIS algorithm does not find a sufficiently large independent set, the exact MIS needs to be computed before a pattern will be kept or discarded. Depending on the resulting size of the maximum independent set, the subgraph will be identified as frequent or infrequent. Also, if we need not only to find frequent subgraphs, but also to find their exact frequency, then the exact MIS needs to be computed on the overlap graph of every pattern. In both cases, because solving the exact MIS problem is in NP-complete (see Section 2.2), the above definition of the frequent subgraph discovery problem cannot be tractable, even for a relatively simple input graph.

To make the problem more practical, we propose two alternative formulations that can find frequent subgraphs without solving the exact MIS problem.

Definition 2 (Approximate Discovery Problem) *Given an input graph $\mathcal{G}$ which is undirected and labeled, and a parameter f, find connected undirected labeled subgraphs that have at least f edge-disjoint embeddings in $\mathcal{G}$ as much as possible.*

Definition 3 (Upper Bound Discovery Problem) *Given an input graph $\mathcal{G}$ which is undirected and labeled, and a parameter f, find all connected undirected labeled subgraphs such that an upper bound on the number of its edge-disjoint embeddings is above the threshold f.*

Essentially the solutions for those two problems become a subset and a superset of the solution for Definition 1, respectively. The first formulation, Definition 2, which asks for a subset of the solution of Definition 1, requires that the embeddings of each subgraph form an overlap graph that has an *approximate MIS* whose size is greater than or equal to f. The second formulation, Definition 3, which asks for a superset of the solution of Definition 1, requires that an upper bound on the size of the exact MIS of this overlap graph is greater than or equal to f. Note that as discussed in Section 2.2, such upper bounds can be easily obtained for both the GMIS algorithm as well as for other approximate algorithms.

5 Algorithms for Finding Frequent Subgraphs in a Large Graph

We developed two algorithms, called HSIGRAM [2] and VSIGRAM, which find all frequent subgraphs according to Definitions 1–3 described in Section 4. In both algorithms, the frequent patterns are conceptually organized in a form of a lattice that is referred to as the *lattice of frequent subgraphs*. The kth level of this lattice contains all frequent subgraphs with k edges (i.e., size-k subgraphs), and a node at level k representing a subgraph G^k is connected to at most k nodes at level $k-1$, each corresponding to a distinct (i.e., non-isomorphic) connected size-$(k-1)$ subgraph of G^k. The goal of both HSIGRAM and VSIGRAM is to identify the various nodes of this lattice and the frequency of the associated subgraphs.

The difference between the two algorithms is the method they use to discover (i.e., generate) the nodes of the lattice. HSIGRAM follows a horizontal approach and discovers the nodes in a breadth-first fashion, whereas VSIGRAM follows a vertical approach and discovers the nodes in a depth-first fashion. Both horizontal and vertical approaches have been previously used to find frequent subgraphs in the graph-transaction setting [33, 44, 68, 8] and have their origins on algorithms developed for finding frequent itemsets and sequences [2, 3, 26, 72].

A detailed description of HSIGRAM and VSIGRAM is provided in the rest of this section.

5.1 Horizontal Algorithm: HSIGRAM

The general structure of HSIGRAM is shown in Algorithm 1 (the notation used in the pseudo-code is shown in Table 1). HSIGRAM takes as input the graph $\mathcal{G}$, the minimum frequency threshold f, and the parameter *MIS_type* that specifies the particular problem definition (as discussed in Section 4). It starts by enumerating all frequent single- and double-edge subgraphs in $\mathcal{G}$, and then enters its main computational loop (Lines 7–10). During each iteration, HSIGRAM first generates all candidate subgraphs of size $k+1$ by joining pairs of size-k frequent subgraphs (Line 8) and then computes their frequency (HSIGRAM-COUNT in Line 11). The candidate subgraphs whose frequency is lower than the minimum threshold f are discarded and the remaining are kept for the next level of the algorithm. The computation terminates when no frequent subgraphs are generated during a particular iteration.

The two key components of the HSIGRAM algorithm that significantly affect its overall computational complexity are

[2] SiGraM stands for Single Graph Miner.

Algorithm 1 HSIGRAM($\mathcal{G}$, *MIS_type*, f)

```
1: ▷ f is the minimum frequency threshold.
2: ▷ MIS_type is either approximate, exact or upper bound.
3: F ← ∅
4: F^1 ← all frequent size-1 subgraphs in G
5: F^2 ← all frequent size-2 subgraphs in G
6: k ← 2
7: while F^k ≠ ∅ do
8:    C^{k+1} ← HSIGRAM-GEN(F^{k-1}, F^k, f)
9:    F^{k+1} ← ∅
10:   for each candidate C in C^{k+1} do
11:      C.freq ← HSIGRAM-COUNT(C, MIS_type)
12:      if C.freq ≥ f then
13:         add C to F^{k+1}
14:   F ← F ∪ F^{k+1}
15:   k ← k + 1
16: return F
```

Algorithm 2 HSIGRAM-GEN($\mathcal{F}^{k-1}$, $\mathcal{F}^k$, f)

```
1: C^{k+1} ← ∅
2: for each F in F^{k-1} do
3:    for each pair F_i, F_j in F.children do
4:       C ← join F_i and F_j based on F
5:       ▷ test if the downward closure property holds.
6:       S(C) ← all connected size-k subgraphs of C
7:       P(C) ← two primary subgraphs of size k
8:       skip ← false
9:       for each S in S(C) do
10:         if S.freq < f then
11:            skip ← true
12:            break
13:      if skip ≠ true then
14:         add C to C^{k+1}
15:         ▷ P(C) = {H_1, H_2}
16:         add C to H_1.children and to H_2.children
17: return C^{k+1}
```

Algorithm 3 HSIGRAM-COUNT(C^{k+1}, *MIS_type*)

```
1: (M(C^{k+1}), A(C^{k+1})) ← HSIGRAM-EMBED(C, G)
2: G ← build an overlap graph from M(C^{k+1})
3: {G_1, G_2, ..., G_m} ← decompose G
4: f_MIS ← 0
5: for each G_i in {G_1, G_2, ..., G_m} do
6:    if G_i is easy to handle then
7:       f_MIS ← f_MIS + |EMIS(G_i)|
8:    else if MIS_type = approximate then
9:       f_MIS ← f_MIS + |GMIS(G_i)|
10:   else if MIS_type = exact then
11:      f_MIS ← f_MIS + |EMIS(G_i)|
12:   else if MIS_type = upper bound then
13:      f_MIS ← f_MIS + |GMIS(G_i)| min((Δ + 2)/3, (d̄ + 2)/2)
14: ▷ S(C^{k+1}) is a set of all connected size-k subgraphs in C^{k+1}
15: f_p ← the lowest frequency among S(C^{k+1})
16: return min(f_MIS, f_p)
```

the methods used to perform candidate generation and to compute the frequency of the candidate subgraphs. In the rest of this section we provide additional details on how these operations are performed and describe various optimizations that are designed to reduce their runtime.

5.1.1 Candidate Generation

HSIGRAM generates candidate subgraphs of size $k+1$ by joining two frequent size-k subgraphs. In order for two such frequent size-k subgraphs to be eligible for joining each of the two must contain the same size-$(k-1)$ connected subgraph. The simplest way to generate the complete set of candidate subgraphs is to join all pairs of size-k frequent subgraphs that have a common size-$(k-1)$ subgraph. Unfortunately, the problem with this approach is that a particular size-k subgraph may have up to k different size-$(k-1)$ subgraphs and as a result, if we consider all such possible subgraphs and perform the resulting join operations, we will end up generating the same candidate pattern multiple times, and generating a large number of candidate patterns that are not downward closed. Such an algorithm would spend a significant amount of time identifying unique candidates and eliminating non-downward closed candidates (both of which operations are non-trivial as they require to determine the canonical label of the generated subgraphs).

HSIGRAM addresses both of these problems by only joining two frequent subgraphs if and only if they share a certain, properly selected, size-$(k-1)$ subgraph. Algorithm 2 shows the pseudo-code for the candidate generation, where the properly selected size-$(k-1)$ subgraph is denoted by F. For each frequent size-k subgraph F_i, let $\mathcal{P}(F_i) = \{H_{i,1}, H_{i,2}\}$ be the two size-$(k-1)$ connected subgraphs of F_i such that $H_{i,1}$ has the smallest canonical label and $H_{i,2}$ has the second smallest canonical label among the various connected size-$(k-1)$ subgraphs of F_i. We will refer to these subgraphs as the *primary subgraphs* of F_i. Note that if every size-$(k-1)$ subgraph of F_i is isomorphic to each other, $H_{i,1} = H_{i,2}$ and $|\mathcal{P}(F_i)| = 1$. HSIGRAM will only join two frequent subgraphs F_i and F_j, if and only if $\mathcal{P}(F_i) \cap \mathcal{P}(F_j) \neq \emptyset$, and the join operation will be done with respect to the common size-$(k-1)$ subgraph(s). The proof that this approach will correctly generate all valid candidate subgraphs is presented in [44]. This candidate generation approach dramatically reduces the number of redundant and non-downward closed patterns that are generated and leads to significant performance improvements over the naive approach [45].

5.1.2 Frequency Counting

HSIGRAM-COUNT in Algorithm 3 computes the frequency of a candidate subgraph C by first identifying all of its embeddings, constructing the overlap graph of these embeddings, and then, based on the *MIS_type* parameter, finding an approximate or exact MIS of this overlap graph. The outline of this process is shown in Algorithms 4 and 3. In the rest of this section we first describe how the various embeddings are identified followed by a description of the method used to efficiently compute the desired maximal independent sets.

Embedding Identification In order to identify all the embeddings of a candidate C, HSIGRAM-EMBED shown in Algorithm 4 needs to solve the subgraph isomorphism problem. Performing the subgraph isomorphism for every candidate from scratch may be expensive, especially when an input graph is large. HSIGRAM-EMBED reduces this computational requirement by using *anchor edges*. An anchor edge is a partial embedding of a candidate C and works as

Algorithm 4 HSIGRAM-EMBED($C, \mathcal{G}$)

1: $\triangleright$ $\mathcal{A}$: a set of all anchor edges of C
2: $\mathcal{A} \leftarrow$ intersection of anchor edges across $\mathcal{S}(C)$
3: $\triangleright$ collect all unique embeddings of C into $\mathcal{M}$
4: $\mathcal{M} \leftarrow \emptyset$
5: **for each** anchor edge e **in** $\mathcal{A}$ **do**
6: $\quad \mathcal{M}_e \leftarrow$ all embeddings of C that includes the edge e
7: $\quad \mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_e$
8: $\triangleright$ collect all unique anchor edges of C into $\mathcal{A}$
9: $\mathcal{A} \leftarrow \emptyset$
10: **for each** embedding m **in** $\mathcal{M}$ **do**
11: $\quad e \leftarrow$ choose one edge from m arbitrarily
12: $\quad$ add e to $\mathcal{A}$
13: **return** $(\mathcal{M}, \mathcal{A})$

a constraint of the subgraph isomorphism problem in which narrows down the search space only around the anchor edge.

More specifically, HSIGRAM-EMBED creates and uses anchor edges as follows. First, the list of anchor edges are created right after frequency counting for size-$(k-1)$ frequent subgraph, by converting the list of its non-identical embeddings. These edges will be used later for counting a candidate of size k. Let F_i denote a frequent subgraph of size $k-1$ and suppose F_i has N non-identical embeddings in total. After the frequency counting, F_i has a list of all its embeddings $\mathcal{M}(F_i) = \{m_1, \ldots, m_N\}$. An anchor edge e of an embedding m_i of F is an edge in $E(\mathcal{G})$ that is also a part of m_i. For every m_i, HSIGRAM-EMBED arbitrarily chooses an edge and adds it to $\mathcal{A}(F_i)$ (Line 11 in Algorithm 4). Because of overlapped embeddings, some embeddings may lead to the same anchor edge.

Now, in the next iteration, suppose a k-candidate C contains a frequent $(k-1)$-subgraph F_i. Because there are k edges in $E(C)$, C may have up to k distinct such frequent subgraphs of size $k-1$, and each F_i holds the anchor edge list. Before starting the frequency counting of C, first HSIGRAM-EMBED selects one of F_i whose frequency is the lowest among $\{F_i\}$. For each $e_n \in \mathcal{A}(F_i)$, HSIGRAM-EMBED checks if there is an edge $e_m \in \mathcal{A}(F_j)$ for all $j \neq i$ such that the shortest path length between e_n and e_m, denoted by d, is within the diameter of C, denoted by $\mathrm{dia}(C)$. If there is such an edge e_m from every $\mathcal{A}(F_j)$ for $j \neq i$, e_n may be a part of an embedding of C, because if C is a frequent subgraph of size k, there must be a set of frequent subgraphs of size $k-1$ inside the same embedding of C. To compute the exact path length between edges e_n and e_m in $\mathcal{G}_i$ requires all pairs shortest paths, which may be computationally expensive when $|E(\mathcal{G}_i)|$ is large. HSIGRAM-EMBED bounds this length d by the difference between two lengths, $|d_n - d_m|$, where d_n and d_m are the shortest path lengths from an arbitrarily chosen vertex $v \in V(\mathcal{G}_i)$ to e_n and e_m respectively. If e_n and e_m are in the same embedding of C_i, always $d \leq \mathrm{dia}(C)$ holds and $d_n \leq d_m + d$. Thus, if $|d_n - d_m| \leq \mathrm{dia}(C)$ is true, then e_n and e_m may belong to the same embedding of C , otherwise e_n and e_m cannot be in the same embedding (see Figure 4). If e_n cannot find such e_m from every $\mathcal{A}(F_j)$ for $j \neq i$, e_m is removed from $\mathcal{A}(F_i)$ (Line 2). Because the subgraph isomorphism will be performed for each e_n, this pruning procedure can effectively reduce the run-time.

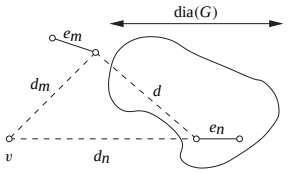


Figure 4: Distance estimation between two edges

Finally, after removing unnecessary anchor edges, for each of the remaining anchor edges, all the subgraph isomorphisms of C are repeatedly identified and the set of embeddings $\mathcal{M}$ is built (Line 6).

Computing the Frequency The frequency of each subgraph C^{k+1} is computed by the HSIGRAM-COUNT function shown in Algorithm 3. In particular, HSIGRAM-COUNT computes two different frequencies. The first, denoted by f_{MIS}, is computed based on the size of the MIS of the overlap graph created from the embeddings of C^{k+1}. The second, denoted by f_p, is the least frequency of all the connected size-k subgraphs of C^{k+1} (Line 15), which represents an upper bound on C^{k+1}'s frequency derived entirely from the lattice of frequent subgraphs. In the case in which f_{MIS} is computed using Definition 3, the frequency bound provided by f_p may actually be tighter, and thus may lead to more effective pruning. For this reason, the overall frequency of C^{k+1} is obtained by taking the minimum of f_{MIS} and f_p.

The frequency f_{MIS} is computed as follows (Lines 2–13). Given a pattern and all of its non-identical embeddings, HSIGRAM-COUNT generates its overlap graph G. Then, HSIGRAM-COUNT decomposes G into its connected components $G_1, G_2, \ldots, G_m$ ($m \geq 1$). Next, for each connected component G_i, it checks the maximum degree of its vertices and if it is less that or equal to two (a cycle or a path), it computes its maximum independent set directly by the EMIS algorithm because it is trivial to compute the exact MIS for this class of graphs (Line 7). If the maximum degree is greater than two, HSIGRAM-COUNT uses either the result of the GMIS algorithm (Line 9), the result of the EMIS algorithm (Line 11), or the upper bound on the size of the exact MIS (Equation 2.1). The summation of those MIS sizes for the components is the final value of f_{MIS}. Note that the decomposition of the overlap graph into its connected components allow us to take advantage of the properties of the special graphs and also obtain tighter bounds for each component as the maximum degree for some of them will be lower than the maximum degree of the entire overlap graph.

In addition, every edge is marked if it is included in any embedding of a frequent subgraph. Unmarked edges are removed before proceeding to the next iteration.

5.2 Vertical Algorithm: VSIGRAM

The most computationally expensive step in the HSIGRAM algorithm is frequency counting as it needs to repeatedly perform subgraph isomorphism computations. The overall time can be greatly reduced if instead of storing only the anchor-edges we store the complete set of embeddings across

successive levels of the algorithm. Because of HSIGRAM's level-by-level structure, these complete embeddings need to be stored for the entire set of frequent and candidate patterns of each successive pair of levels. This substantially increases the memory requirements of this approach, making it impractical for the most of interesting datasets. On the other hand, within the context of a vertical algorithm, storing the complete set of embeddings is feasible since we need to do that only for the subgraphs along the path from the current node to the root. Thus, a vertical algorithm has potentially a computational advantage over a horizontal algorithm, which motivated the development of VSIGRAM.

However, before developing efficient algorithms that generate the lattice of frequent subgraphs in a depth-first fashion two critical steps need to be addressed. The first step is the method that is used to ensure that the same node of the lattice and the depth-first subtree rooted at that node should not be discovered and explored multiple times. This is important because each node at level k will be connected to up to k different nodes at level $(k-1)$. As a result, if there are no mechanisms by which to prevent the repeated generation of the same node, a depth-first algorithm will end-up performing redundant computations (i.e., generating the same nodes multiple times), adversely impacting the overall performance of the algorithm. VSIGRAM eliminates these redundant computations by assigning each node at level k (corresponding to a subgraph F^k) to a unique parent node at level $k-1$ (corresponding to a subgraph F^{k-1}, such that only F^{k-1} is allowed to create F^k. The subgraph F^{k-1} is called *the generating parent* of F^k. Details on how this is achieved is provided in Section 5.2.1.

The second step is the method that is used to create successor nodes in the course of the traversal. In the case of HSIGRAM, this corresponds to the candidate generation phase, and is performed by joining the frequent subgraphs of the previous level. However, since the lattice is explored in a depth-first fashion, such joining-based approach will not work, as the algorithm may not have yet discovered the required frequent subgraphs. To address this problem, VSIGRAM creates the successor nodes (i.e., extended subgraphs) by analyzing all the embeddings of the current subgraph F^k, and identifying the distinct one-edge extensions to these embeddings that are sufficiently frequent. The frequent extensions for which F^k is the generating parent are then used as the successor nodes during the depth-first traversal.

The general structure of VSIGRAM is shown in Algorithm 5. VSIGRAM starts by determining all frequent size-1 patterns and then uses each one of them as the starting point of a recursive depth-first extension (VSIGRAM-EXTEND function). VSIGRAM-EXTEND takes as input a size-k frequent subgraph F^k and all of its embeddings $\mathcal{M}(F^k)$ in $\mathcal{G}$ and proceeds as follows. For each size-k embedding $m \in \mathcal{M}(F^k)$, it identifies and stores every possible size-$(k+1)$ subgraph in $\mathcal{G}$ that contains m. From this set of subgraphs, it extracts all size-$(k+1)$ subgraphs which are not isomorphic to each other and stores them in $\mathcal{C}^{k+1}$. Then, VSIGRAM-EXTEND eliminates from $\mathcal{C}^{k+1}$ all the subgraphs that do not have F^k as their

Algorithm 5 VSIGRAM

```
VSIGRAM(G, MIS_type, f)
 1: F ← ∅
 2: F^1 ← all frequent size-1 subgraphs in G
 3: for each F^1 in F^1 do
 4:    M(F^1) ← all embeddings of F^1
 5: for each F^1 in F^1 do
 6:    F ← F ∪ VSIGRAM-EXTEND(F^1, G, f)
 7: return F

VSIGRAM-EXTEND(F^k, G, MIS_type, f)
 1: F ← ∅
 2: for each embedding m in M(F^k) do
 3:    C^{k+1} ← C^{k+1} ∪ {all (k+1)-subgraphs of G containing m}
 4: for each C^{k+1} in C^{k+1} do
 5:    if F^k is not the generating parent of C^{k+1} then
 6:       continue
 7:    compute C^{k+1}.freq from M(C^{k+1})
 8:    if C^{k+1}.freq < f then
 9:       continue
10:    add C^{k+1} to F
11: return F
```

generating parent (Lines 5–6) or are infrequent (Lines 7–8). The subgraphs remaining in $\mathcal{C}^{k+1}$ are the frequent subgraphs of size-$(k+1)$ obtained by an one-edge-extension of F^k and are used as input for the next recursive call. The recursion terminates when $\mathcal{C}^{k+1} = \emptyset$, and the depth-first search backtracks.

In the rest of this section we provide additional details on how the various operations are performed and describe various optimizations that are designed to reduce VSIGRAM's run-time.

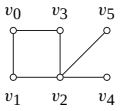
5.2.1 Generating Parent Identification

The scheme that VSIGRAM uses to determine the generating parent of a particular subgraph is as follows. Suppose a size-$(k+1)$ frequent subgraph F^{k+1} is just created by extension from a size-k frequent subgraph F^k. By the canonical labeling, the order of edges and vertices in F^{k+1} is uniquely determined. VSIGRAM removes the last edge that does not disconnect F^{k+1} and obtains another size-k subgraph F.

If F is isomorphic to F^k then F^k becomes the generating parent of F^{k+1}, and VSIGRAM keeps the further exploration from F^{k+1}. Similar type of approaches have been used earlier in the context of vertical algorithms for the graph-transaction setting [65, 68]. All of these share the same idea, which avoids redundant frequent pattern generation and traverses the lattice of patterns as if it was a tree.

5.2.2 Efficient Subgraph Extension

Starting from a frequent size-k subgraph, VSIGRAM obtains the extended subgraphs of size $k+1$ by adding an additional edge (while preserving connectivity) to all of its possible embeddings. Specifically, for each embedding m of a frequent k-subgraph F, VSIGRAM enumerates all the edges that can be added to m to form a size-$(k+1)$ extended subgraph. Each of those edges is represented by a tuple of 5 elements $s = (x, y, u, v, e)$, called a *stem*, where x and y are the vertex



(a) Graph G

ϕ_0	v_0	v_1	v_2	v_3	v_4
ϕ_1	v_0	v_1	v_2	v_3	v_5
ϕ_2	v_0	v_3	v_2	v_1	v_4
ϕ_3	v_0	v_3	v_2	v_1	v_5
cvid	v_0	v_1	v_2	v_1	v_4
ϕ^*	ϕ_0	ϕ_0	ϕ_0	ϕ_2	ϕ_1

(b) Canonical vertex ID and automorphsm

Figure 5: Size-6 graph G, canonical vertex IDs, and canonical automorphism

IDs of the edge in $\mathcal{G}$, u and v, $u < v$, are the corresponding vertex IDs in F, and e is the label of the edge. For u and v, if there is no corresponding vertex in F, -1 is used to show that it is outside the subgraph F.

However, because of the automorphism of the subgraph F, we cannot use this stem representation directly. For a particular embedding m of a frequent subgraph F in $\mathcal{G}$, there may be more than one vertex mapping of the subgraph onto the embedding. If we simply used a pair of vertex IDs of the subgraph to represent a stem, depending on the mapping, the same edge addition might be considered a different stem, which would result in the wrong frequency of the subgraph. To avoid this problem, every time a stem is generated, its representation is normalized as follows . vSiGraM enumerates all possible automorphisms of F, denoted by $\{\phi_i\}$. By an appropriate ϕ_i we obtain the canonical vertex ID for every vertex $v \in V(F)$. The *canonical ID* of a vertex v, denoted by $\text{cvid}(v)$, is defined as

$$\text{cvid}(v) = \min_i \phi_i(v).$$

The automorphism with the least subscript that gives the canonical ID for v is called the *canonical automorphism*, denoted by ϕ^*_v.

$$\phi^*_v = \arg\min_{\phi_i} \phi_i(v), \quad i < j \text{ if } \phi_i(v) = \phi_j(v)$$

For example, given the size-6 graph G shown in Figure 5(a), $\text{cvid}(v_3) = v_1$ and $\phi^*_{v_3} = \phi_2$. Figure 5(b) shows cvid and ϕ^* for every vertex in G. Note that although $\phi_3(v_3)$ is also v_1, because ϕ_2 has the smaller subscript, 2, $\phi^*_{v_3}$ is ϕ_2. Now for each stem $s = (x, y, u, v, e)$, $\phi^*(u, v) = (u', v')$ are defined as follows.

$$\begin{aligned} u' &\equiv \text{cvid}(u),\ v' \equiv \phi^*_u(v) && \text{if } \text{cvid}(u) \le \text{cvid}(v) \\ u' &\equiv \phi^*_v(u),\quad v' \equiv \text{cvid}(v) && \text{otherwise} \end{aligned}$$

Then, stem s is rewritten as (x, y, u', v', e), which is automorphism invariant representation of s and is used by vSiGraM to properly determine the frequency of size-$(k+1)$ extended subgraphs.

Additional Optimization: Keeping Track of Edge Creation Status Each frequent subgraph maintains a three-dimensional table, called a *connection table*. Each element in the table is denoted by $\text{ct}(u', v', e)$ which shows if it is possible to form an edge between the vertices u' and v' whose edge label is e. Every time a stem (x, y, u', v', e) is discarded, the corresponding element in the connection table is updated to show that it is now impossible to create an edge with a label e between u' and v'. If $\text{ct}(u', v', e)$ is deactivated for a frequent subgraph of size k, then for any $l > k$, there should not be any frequent subgraph that has an edge between u' and v' with the edge label e. We can reduce the number of stems to be generated by looking up the connection table during the stem enumeration phase.

5.2.3 Frequency Counting

In the vertical algorithm, when a size-$(k+1)$ extension is processed, there is only one size-k frequent subgraph visible, the generating parent. vSiGraM's frequency counting is similar to hSiGraM-Count, except for the computation of f_p (see Line 15 in Algorithm 3). hSiGraM enforces the downward closure property on the frequency of a size-$(k+1)$ candidate, by using the least frequency of all size-k subgraphs of the candidate. vSiGraM cannot take the same step because vSiGraM does not hold all size-k frequent subgraphs at the time a size-$(k+1)$ extended subgraph is created. Instead vSiGraM simply uses the frequency of the size-k generating parent from which the current size-$(k+1)$ extension is obtained. As a result, vSiGraM's pruning is looser than that of hSiGraM.

6 Experimental Evaluation

In this section, we study the performance of the proposed algorithms with various parameters and real datasets. All experiments were done on dual AMD Athlon MP 1800+ (1.53 GHz) machines with 2 GBytes main memory, running the Linux operating system. All the run-times reported are in seconds.

6.1 Datasets

We used six different datasets, each obtained from a different domain, to evaluate and compare the performance of hSiGraM and vSiGraM. The basic characteristics of these datasets are shown in Table 2. Note that even though some of these graphs consist of multiple connected components, the hSiGraM and vSiGraM algorithm treat them as one large graph and discover the frequent patterns according to Definitions 1–3 described in Section 4.

The *Aviation* and *Credit* datasets are obtained from [64]. The Aviation dataset is originally from the Aviation Safety Reporting System Database and the Credit dataset is from the UCI machine learning repository [7]. The directed edges in the original graph data were converted into undirected ones. For the Aviation dataset, we removed undirected edges to show "near_to" relation between two vertices because those edges form cliques which makes this graph difficult to mine.

The *Citation* dataset was created from the citation graph used in KDD Cup 2003 [37]. Each vertex in this graph corresponds to a document and each edge corresponds to a citation. Because our algorithms are for undirected graphs, the direction of these citations was ignored. Since the original dataset does not have any meaningful label for vertices, we

generated vertex labels as follows. We first used a clustering algorithm to form clusters of the document abstracts into 50 thematically coherent topics, and then assigned the cluster ID as the label to the corresponding vertices. For the edges, we used as labels the difference in the publication year of the two papers. For example, if two papers were published in 1997 and 2002, an edge is created between those two document vertices with the label "5". Finally, because some of the vertices in the resulting graph had a very high degree (i.e., authorities and hubs), we kept only the vertices whose degree was less or equal to 15.

The *Contact Map* dataset is made of 170 proteins from the Protein Data Bank [5] with pairwise sequence identity lower than 25%. The vertices in these graphs correspond to the different amino acids and the edges connect two amino acids if they are either at consecutive sequence positions or they are in contact in their 3D structure. To amino acids are considered to be in contact if the distance between their C_α atoms is less than 8 Å. Furthermore, while creating the graphs we only considered non-local contacts that are defined as the contacts between amino acids whose sequence separation is at least six amino acids.

The *DTP* dataset is a collection of 2,319 chemical compounds randomly selected from the dataset of 223,644 chemical compounds provided by the Developmental Therapeutics Program (DTP) at National Cancer Institute[3]. Note that each chemical compound forms a connected component and there are 2,319 such components in this dataset. Each vertex corresponds to an atom and its label represents the atom type. An edge is formed between two vertices if the corresponding two atoms are connected by a bond. The type of a bond is used as an edge label, and there are three distinct edge labels.

Finally, the *VLSI* dataset was obtained from the International Symposium on Physical Design '98 (ISPD98) benchmark suite[4] and corresponds to the netlist of a real circuit. The netlist was converted into a graph by first removing any nets that are longer than four and then using a star-based approach to replace each net (i.e., hyperedge) by a set of edges. Note that for this dataset we limited the size of the largest discovered pattern to five edges. This is because for the values of the frequency threshold used in our experiments, the only frequent patterns that contained more than five edges were paths, and because of the highly connected nature of the underlying graph, there were a very large number of such paths, making it hard to find these longer path patterns in reasonable amount of time.

6.2 Results

Table 3 shows the results obtained by the HSIGRAM and VSIGRAM algorithms for the different datasets, for a wide range of the minimum frequency threshold values f, and the three different MIS-based problem definitions. For each experiment, Table 3 shows the amount of time (in seconds) required by the particular algorithm, the total number of patterns that were discovered, and size of the largest pattern. Entries in the table marked with "—" represents experiments that were aborted because of high computational requirements.

Table 2: Datasets used in the experiments

Dataset	Connected Components	Vertices	Edges	Labels	
				Vertex	Edge
Aviation	2703	101185	196964	6173	51
Credit	700	14700	28000	59	20
Citation	16999	29014	42064	50	12
Contact Map	170	33443	224488	21	2
DTP	2319	41190	86140	58	3
VLSI	2633	12752	23084	23	1

From these results we can see that as expected, for all datasets and algorithms, as the value of f decreases, the runtime for finding the frequent patterns increases as well. The rate of increase in runtime follows the corresponding rate of increase in the number of patterns that are being discovered. Besides that, the results in this table help illustrate the relation between the two key variables in these experiments, which are the type of the particular algorithm (HSIGRAM vs VSIGRAM) and the type of frequency calculation (approximate MIS, exact MIS, or upper bound MIS).

In general, the amount of time required by VSIGRAM is smaller than that required by HSIGRAM. In fact, as the value of the frequency threshold decreases, VSIGRAM is up to five times faster than HSIGRAM. This is true across all datasets for the approximate and exact MIS problem formulation, and for those datasets for which the upper bound MIS formulation leads to the same number of frequent patterns for both algorithms. As discussed in Section 5.2, the reason for that performance advantage is the fact that by keeping track the embeddings of the frequent subgraphs along the depth-first path, VSIGRAM spends significantly less time in subgraph isomorphism related computations than HSIGRAM does.

However, for certain datasets, when the upper bound MIS formulation is used, VSIGRAM ends up generating significantly more patterns than those generated by HSIGRAM. For example, in the case of the DTP dataset and $f = 20$, VSIGRAM generates almost 16 times more patterns than HSIGRAM. In such cases, the amount of time required by VSIGRAM is substantially greater than that required by HSIGRAM (32.4 times greater in the DTP example). The reason for that is the fact that because of its depth-first nature, VSIGRAM cannot take advantage of the frequent subgraph lattice to get a tight upper bound on the frequency of a subgraph based on the frequency of all of its subgraphs, and it bases its upper bound only on the frequency of the generating parent. On the other hand, because of its level-by-level nature, HSIGRAM can use the information from all its subpatterns, and obtains better upper bounds (see discussion in Section 5.1.2).

Comparing the different MIS-based problem formulations, we can see that the one based on the approximate MIS usually leads to the fastest execution time for both algorithms. Moreover, for datasets for which the various overlap graphs

[3] DTP 2D and 3D Structural Information. http://dtp.nci.nih.gov/docs/3d_database/structural_information/structural_data.html

[4] http://vlsicad.cs.ucla.edu/~cheese/ispd98.html

Table 3: Run-time in seconds and the number of found frequent patterns for the different datasets. this is that and that is this and it is what and what is it.

		Run-time[sec]						Number of Found Patterns						Largest Pattern Size					
	f	Apprx.		Exact		UB		Apprx.		Exact		UB		Apprx.		Exact		UB	
		H	V	H	V	H	V	H	V	H	V	H	V	H	V	H	V	H	V
Aviation	2000	308	130	306	130	320	130	833	833	833	833	833	833	8	8	8	8	8	8
	1750	779	342	787	342	789	341	2249	2249	2249	2249	2249	2249	9	9	9	9	9	9
	1500	1603	743	1674	745	1584	739	5207	5207	5207	5207	5207	5207	10	10	10	10	10	10
	1250	2726	1461	2720	1496	2781	1486	11087	11087	11087	11087	11087	11087	12	12	12	12	12	12
	1000	5256	3667	5158	3683	5596	3818	30331	30331	30331	30331	30331	30331	13	13	13	13	13	13
Citation	100	0.1	0.0	0.1	0.0	0.1	0.0	6	6	6	6	7	11	1	1	1	1	2	5
	50	0.1	0.1	0.1	0.1	0.6	—	39	39	39	39	113	—	2	2	2	2	7	—
	20	0.6	0.3	0.9	0.5	139	—	266	266	266	266	12203	—	3	3	3	3	16	—
	10	4.0	1.5	4.2	1.9	—	—	986	986	988	988	—	—	5	5	5	5	—	—
Contact Map	400	3	2	3	2	10	—	100	100	100	100	246	—	2	2	2	2	8	—
	300	10	3	10	3	183	—	186	186	186	186	2358	—	2	2	2	2	10	—
	200	44	9	45	9	—	—	505	505	505	505	—	—	3	3	3	3	—	—
	100	362	63	356	71	—	—	3183	3183	3186	3186	—	—	5	5	5	5	—	—
	50	3505	607	3532	632	—	—	29237	29237	29298	29298	—	—	6	6	6	6	—	—
Credit	500	0	0	0	0	0	0	24	24	24	24	24	24	3	3	3	3	3	3
	200	10	4	10	4	9	4	1325	1325	1325	1325	1325	1325	7	7	7	7	7	7
	100	49	20	45	21	45	20	11696	11696	11696	11696	11696	11696	9	9	9	9	9	9
	50	169	78	172	80	169	78	73992	73992	73992	73992	73992	73992	11	11	11	11	11	11
	20	2019	461	1855	468	1880	462	613884	613884	613884	613884	613884	613884	13	13	13	13	13	13
DTP	500	92	20	86	21	96	30	109	109	109	109	153	226	7	7	7	7	12	13
	200	101	23	100	24	115	38	414	414	415	415	641	916	9	9	9	9	15	15
	100	113	27	114	27	169	64	1244	1244	1244	1244	2484	3788	12	12	12	12	16	18
	50	145	34	134	35	247	103	4028	4028	4028	4028	8295	13622	14	14	14	14	18	21
	20	243	86	249	83	616	19998	21477	21477	21478	21478	52180	824702	16	16	16	16	20	81
	10	813	311	882	294	2018	—	112535	112535	112539	112539	232810	—	21	21	21	21	21	—
VLSI	200	11	3	—	—	37	8	137	137	—	—	347	415	5	5	—	—	5	5
	150	13	4	—	—	46	9	156	156	—	—	437	503	5	5	—	—	5	5
	100	42	7	—	—	54	10	379	379	—	—	519	609	5	5	—	—	5	5
	75	49	8	—	—	56	10	409	409	—	—	571	679	5	5	—	—	5	5
	50	236	15	—	—	282	17	683	683	—	—	946	1051	5	5	—	—	5	5
	25	428	18	—	—	469	20	1452	1452	—	—	1907	2131	5	5	—	—	5	5

Note. Dashes indicate the computation was aborted because of the too long run-time or memory exhaustion.
f: the minimum frequency threshold, H: HSIGRAM, V: VSIGRAM, Approx.: with approximate MIS, Exact: with exact MIS, UB: with upper bound MIS

are reasonably small (this is true for all our datasets except VLSI), the exact MIS-based formulation leads to small execution time as well. Also, the upper bound MIS formulation tends to be slower than the other two primarily because it generates more patterns. However, the advantage of the upper bound formulation over the one based on the exact MIS can be seen for the VLSI graph for which the resulting overlap graph was large, and exact MIS computations could not finish in reasonable amount of time. Finally, comparing the number of patterns found by the approximate and the exact MIS-based formulations, we can see that, in general, the approximate algorithm fails to discover a very small number of patterns.

Table 4: SUBDUE Results

Dataset	Run-time [sec]	Number of Patterns	Pattern Size			Frequency of Found Patterns		
Aviation	—	—	—	—	—	—	—	—
Citation	8812	3	27	26	27	1	1	1
Contact Map	5043	3	224	223	223	1	1	1
Credit	517	3	6	5	5	341	395	387
DTP	1525	3	2	2	6	4957	4807	1950
VLSI	16	3	1	1	1	773	773	244

6.3 Performance Comparison with Existing Algorithms

Comparison with SUBDUE We ran SUBDUE [28] version 5.0.6[5] on the same datasets described in Section 6.1 and

[5] Although this version is not the latest one, it runs significantly faster than the current latest version, 5.0.8.

measured the run-time, the number of discovered patterns, their size, and their frequency. These results are shown in Table 4. These results were obtained by using SUBDUE's default settings for all but the VLSI dataset. For the VLSI dataset, we run SUBDUE so that to find subgraphs that contain at most five edges, as was done in the case of HSIGRAM and VSIGRAM. Note that SUBDUE's default settings returns at most three subgraphs that were determined to be the most important.

Because of the inherent differences between SUBDUE and our algorithms, it is impossible to perform a direct comparison of the results that they generate. For this reason our comparisons will focus mostly on highlighting some key points. First, the amount of time required by SUBDUE is in general, considerably higher than that required by our algorithms. For example, SUBDUE did not finish the computation for the Aviation dataset after spending four entire days. Also for the Citation and Contact Map datasets, SUBDUE could not find any meaningful patterns at all, as the patterns that it found had a frequency of one. For the Credit dataset with the minimum frequency threshold of 50, both HSIGRAM and VSIGRAM with upper bound MIS spent 169 and 78 seconds respectively to discover the same number of subgraphs, 73992. The largest pattern has 11 edges and had a frequency of 58. In contrast, the largest pattern found by SUBDUE had six edges with a frequency of 341. This indicates that if there are small subgraphs that have relatively high frequency, SUBDUE will focus on them and will not discover the larger patterns. We can see the similar result for the DTP dataset. The size of the patterns SUBDUE found are very small, 2–6 edges, but their frequency is very high. On the other hand, the results in Table 3 show that with the minimum frequency threshold 20, both HSIGRAM and VSIGRAM under exact MIS spend 249 and 83 seconds respectively to find 21,478 frequent subgraphs, and the largest size is 16.

Comparison with SEuS The SEuS [21] algorithm is designed to find all frequent subgraphs in a single-graph setting. However, when determining the frequency of a subgraph they consider all embeddings irrespective of whether they are disjoint or not. As a result, a subgraph may have high frequency even though it has small number of edge-disjoint embeddings because of overlapped embeddings. In [21], the run-time of SEuS on the PTE chemical dataset[6] is reported. SEuS (SEuS-S1) spent more than 20 seconds to find 34 frequent subgraphs, that is 1.4 frequent subgraphs per second. On the same dataset given the minimum frequency threshold of 500, VSIGRAM with upper bound MIS requires 20 seconds to find 168 frequent subgraphs, which translates to 8.4 frequent subgraphs per second. Similarly, with the Credit dataset (which is called "Credit-4" in [20]), SEuS-S1 spent 50 seconds to produce 48 frequent subgraphs (one frequent subgraphs per second), while VSIGRAM with upper bound MIS finds 1,325 frequent subgraphs in four seconds for the minimum frequency threshold 200 (331 frequent subgraphs per second).

[6] ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Datasets/carcinogenesis/progol/carcinogenesis.tar.Z

7 Conclusions

In this paper we addressed the problem of finding all the subgraphs that have many edge-disjoint embeddings in a large sparse graph, a step critical to discovering patterns in graph datasets. We studied three distinct formulations of the problem that were motivated by the complexity of identifying the maximum set of edge-disjoint embeddings of a subgraph, and developed two frequent subgraph mining algorithms for solving them. These algorithms are based on the horizontal and vertical paradigms, respectively. Our experimental evaluation on many real datasets showed that for most datasets and problem formulations both algorithms achieve good performance, with the vertical algorithm being two-to-five times faster.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. of 1998 ACM-SIGMOD Int. Conf. on Management of Data*, 1998.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, September 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th Int. Conf. on Data Engineering (ICDE)*, pages 3–14. IEEE Press, 1995.

[4] B. Berendt, A. Hotho, and G. Stumme. Towards semantic web mining. In *International Semantic Web Conference (ISWC)*, pages 264–278, 2002.

[5] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000.

[6] P. Berman and T. Fujito. On the approximation properties of independent set problem in degree 3 graphs. In *Proc. of Workshop on Algorithms and Data Structures*, pages 449–460, 1995.

[7] C. L. Blake and C. J. Merz. UCI repository of machine learning databases, 1998.

[8] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

[9] L. P. Chew, D. Huttenlocher, K. Kedem, and J. Kleinberg. Fast detection of common geometric substructure in proteins. In *Proc. of the 3rd ACM RECOMB International Conference on Computational Molecular Biology*, 1999.

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

[11] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[12] D. J. Cook, L. B. Holder, and S. Djoko. Knowledge discovery from structural data. *Journal of Intelligent Information Systems*, 5(3):229–245, 1995.

[13] L. De Raedt and S. Kramer. The level-wise version space algorithm and its application to molecular fragment finding. In *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

[14] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press, 1998.

[15] M. Deshpande, M. Kuramochi, and G. Karypis. Automated approaches for classifying structures. In *Proc. of the 2nd Workshop on Data Mining in Bioinformatics (BIOKDD '02)*, 2002.

[16] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure based approaches for classifying chemical compounds. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM)*, 2003. to appear.

[17] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proc. of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 2–12, 1991.

[18] S. Fortin. The graph isomorphism problem. Technical Report TR96-20, Department of Computing Science, University of Alberta, 1996.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[20] S. Ghazizadeh and S. Chawathe. Discovering freuqent structures using summaries. Technical Report CS-TR-4364, Department of Computer Science, University of Maryland, 2002.

[21] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In *Proc. of the 5th International Conference on Discovery Science*, 2002.

[22] J. Gonzalez, L. B. Holder, and D. J. Cook. Application of graph-based concept learning to the predictive toxicology domain. In *Proc. of the Predictive Toxicology Challenge Workshop*, 2001.

[23] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of Molecular Biology*, 229:707–721, 1993.

[24] V. Guralnik and G. Karypis. A scalabale algorithm for clustering sequence datasets. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, 2001.

[25] M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, 1997.

[26] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Dallas, TX, May 2000.

[27] D. S. Hochbaum. Efficient bounds for the stable set, vertex cover, and set packing problems. *Discrete Applied Mathematics*, 6:243–254, 1983.

[28] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the SUBDUE system. In *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, pages 169–180, 1994.

[29] M. Hong, H. Zhou, W. Wang, and B. Shi. An efficient algorithm of frequent connected subgraph extraction. In *Proc. of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-03)*, volume 2637 of *Lecture Notes in Computer Science*, pages 40–51. Springer-Verlag, 2003.

[30] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomophism. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM'03)*, 2003. to appear.

[31] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, pages 13–23, Lyon, France, September 2000.

[32] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, March 2003.

[33] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.

[34] D. Jensen and H. Goldberg, editors. *Artificial Intelligence and Link Analysis Papers from the 1998 Fall Symposium*. AAAI Press, 1998.

[35] I. Jonyer, D. J. Cook, and L. B. Holder. Discovery and evaluation of graph-based hierarchical conceptual clusters. *Journal of Machine Learning Research*, 2:19–43, 2001.

[36] I. Jonyer, L. B. Holder, and D. J. Cook. Hierarchical conceptual structural clustering. *International Journal on Artificial Intelligence Tools*, 10(1–2):107–136, 2001.

[37] KDD Cup 2003. http://www.cs.cornell.edu/projects/kddcup/datasets.html.

[38] S. Khanna, R. Motwani, M. Sudan, and U. V. Vazirani. On syntactic versus computational views of approximability. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 819–830, 1994.

[39] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[40] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The Web as a graph: Measurements, models and methods. *Lecture Notes in Computer Science*, 1627, 1999.

[41] C. Ko. Logic induction of valid behavior specifications for intrusion detection. In *IEEE Symposium on Security and Privacy (S&P)*, pages 142–155, 2000.

[42] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopoloties in a set of protein structures. *Journal of computational biology*, 3(2):289–306, 1996.

[43] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in HIV data. In *Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-01)*, pages 136–143, 2001.

[44] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*. in press.

[45] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, November 2001.

[46] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. Technical Report 02-026, University of Minnesota, Department of Computer Science, 2002.

[47] W. Lee and S. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4), 2000.

[48] N. Leibowitz, Z. Y. Fligelman, R. Nussinov, and H. J. Wolfson. Multiple structural alignment and core detection by geometric hashing. In *Proc. of the 7th International Conference on Intelligent Systems in Molecular Biology*, pages 169–177, Heidelberg, Germany, August 1999.

[49] N. Leibowitz, R. Nussinov, and H. J. Wolfson. MUSTA—

a general, efficient, automated method for multiple structure alignment and detection of common motifs: application to proteins. *Journal of computational biology*, 8(2):93–121, 2001.

[50] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, 2001.

[51] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *4th Internation Conference on Knowledge Discovery and Data Mining*, 1998.

[52] B. D. McKay. Nauty users guide. http://cs.anu.edu.au/∼bdm/nauty/.

[53] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[54] E. M. Mitchell, P. J. Artymiuk, D. W. Rice, and P. Willett. Use of techniques derived from graph theory to compare secondary structure motifs in proteins. *Journal of Molecular Biology*, 212:151–166, 1989.

[55] R. J. Mooney, P. Melville, L. R. Tang, J. Shavlik, I. de Castro Dutra, D. Page, and V. S. Costa. Relational data mining with inductive logic programming for link discovery. In *National Science Foundation Workshop on Next Generation Data Mining*, November 2002.

[56] S. H. Muggleton. Scientific knowledge discovery using Inductive Logic Programming. *Communications of the ACM*, 42(11):42–46, 1999.

[57] P. R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:195–205, 2002.

[58] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *Proc. of the 8th ACM SIGKDD Internal Conference on Knowlege Discovery and Data Mining (KDD'2002)*, Edmonton, AB, Canada, July 2002.

[59] X. Pennec and N. Ayache. A geometric algorithm to find small but highly simialar 3D substructures in proteins. *Bioinformatics*, 14(6):516–522, 1998.

[60] J. W. Raymond. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *J. Chem. Inf. Comput. Sci.*, 42:305–316, 2002.

[61] R. C. Read and D. G. Corneil. The graph isomorph disease. *Journal of Graph Theory*, 1:339–363, 1977.

[62] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.

[63] A. Srinivasan, R. D. King, S. H. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, 1997.

[64] SUBDUE databases. http://cygnus.uta.edu/subdue/databases/index.html.

[65] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, pages 458–465, 2002.

[66] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, I. Rigoutsos, and K. Zhang. Finding patterns in three dimensional graphs: Algorithms and applications to scientific data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):731–749, July/August 2002.

[67] S. Wasserman, K. Faust, and D. Iacobucci. *Social Network Analysis : Methods and Applications*. Cambridge University Press, 1994.

[68] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

[69] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, 2003.

[70] K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, 1995.

[71] K. Yoshida, H. Motoda, and N. Indurkhya. Graph-based induction as a unified learning framework. *Journal of Applied Intelligence*, 4:297–328, 1994.

[72] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, 2003.