gSpan: Graph-Based Substructure Pattern Mining

Xifeng Yan   Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign

September 3, 2002

Abstract

With the successful development of efficient and scalable algorithms for mining frequent itemsets and sequences, it is natural to extend the scope of study to a more general structured pattern mining problem: *mining frequent subgraph patterns*. There are many applications in chemistry, biology, computer networks, and World-Wide Web that require mining such patterns. In this paper we investigate new approaches for frequent graph-based pattern mining in graph datasets and propose a novel algorithm called *gSpan* (graph-based Substructure pattern mining), which discovers frequent substructures without candidate generation. gSpan builds a new lexicographic ordering among graphs and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan enumerates not only all frequent subgraphs but also induced subgraphs. We evaluate the performance of gSpan in both synthetic datasets and a chemical compound dataset. The experimental results show that our method substantially outperforms FSG, the best performing frequent subgraph mining algorithm reported so far, by an order of magnitude. Moreover, the algorithm shows very good parallel and scaleup properties and can incorporate constraints nicely in graph mining.
Keywords substructure mining, frequent pattern, structured pattern, graph isomorphism, lexicographic order

1 Introduction

Frequent pattern mining has been an active research theme in data mining with many efficient and scalable techniques developed for mining associate rules [1], frequent itemsets [6, 13, 27, 7], sequential patterns [2, 17, 19], and trees [26, 3]. However, many scientific and commercial applications may require to find frequent structured patterns in large datasets, which may go beyond *sets*, *sequences*, and *trees* into *lattices, graphs*, and *other complicated structures.*

As a general data structure, graph, especially labeled and/or attributed graph, can be used to model many complicated relations among data. Labels for vertices and edges can represent different attributes of entities and relations among them. For example, in chemical compounds, the labels

for vertices can be different types of atoms, and the labels for edges can be different types of bonds. In electronic transactions, labels for vertices represent account owners while edges between these owners indicate occurrences of payments. Since graph can model more complicated objects, it has been intensively used in chemical informatics [5, 23, 4, 22], computer vision [25], video indexing [20], text retrieval [12], etc.

The problem of frequent subgraph mining is to find frequent subgraphs over a collection of graphs. Frequent subgraph mining delivers meaningful structured information such as hot web access patterns, common protein structures, and shared patterns in object recognition. It can also be used in fraud detection to catch similar fraud transaction patterns from millions of electronic payments. Another application is to cluster XML documents based on their common structures. Furthermore, graph is a general data structure which covers almost all previous well-researched frequent patterns, thus, it can unify the mining process into the same framework. Therefore, frequent subgraph mining has raised great interests.

The kernel of frequent subgraph mining is graph/subgraph isomorphism test. In the past three decades, lots of well-known isomorphism test algorithms were developed, such as J. R. Ullmann's Backtracking [24] and B. D. McKay's Nauty [18], as well as plenty of approximate ones. However, the frequent subgraph mining problem has not been explored well. In chemical informatics, some efficient algorithms were designed to discover common substructures possessed by a set of chemical compounds [5, 23]. L. Dehaspe et al. [10] applied inductive logic programming to the problem of predicting chemical carcinogenicity by using frequent substructures. However these systems are not targeted to large scale datasets. L. B. Holder et al. [14] proposed SUBDUE to do approximate substructure pattern discovery based on the minimum description length principle and optional background knowledge. Since SUBDUE uses a computationally-constrained beam search, it cannot discover the complete set of frequent patterns. Recently, Inokuchi et al. [15] proposed an Apriori-based algorithm, called AGM, to discover all frequent substructures. Kuramochi and Karypis [16] further developed the idea using a more efficient graph representation structure and edge-growth instead of vertex-growth. Their algorithm, called FSG, demonstrates a dramatic performance improvement: for the same chemical compound dataset, FSG is able to find all frequent connected subgraphs in 10 minutes with a 6.5% minimum support, whereas it takes 40 minutes to 8 days to discover them by AGM with a varied support from 20% to 10% [16].

AGM and FSG both take advantage of Apriori-like level-wise approaches, firstly introduced by R. Argrawal and R. Srikant [1]. These algorithms, though reduce search space, still bear three nontrivial but inherent problems [13, 19]: huge candidate set generation, multiple scans of database, and difficulties at mining long patterns. Some previous work has proposed to record transaction IDs (*Tids*) for each pattern to minimize the effort of multiple scans by filtering out unnecessary tests when new patterns grow from the old ones. However, because of a huge candidate set maintained at each level, this optimization seems unaffordable when millions of transactions are taken into consideration. Shenoy et al. [21] introduced a vertical mining technique using *compressed bitvectors*

to compress *Tids* for each discovered patten, while Zaki [27] developed a better one called *diffset*, which significantly reduces the storage space of *Tids*. However, the major computational cost for frequent subgraph mining shifts from large storage of transaction IDs to heavy candidate testings. The cost is nontrivial in determining whether a graph is a frequent subgraph. As [16] showed, even for a 340 chemical compounds dataset, it took 28 to 600 seconds to discover all frequent subgraphs with minimum supports from 10% to 6.5%. Therefore, the exponential growth nature of subgraph isomorphism test plays a major role in the total cost. The Apriori-like algorithms suffer two additional costs:

(1) **Costly subgraph isomorphism test**. Since subgraph isomorphism is an NP-complete problem, no polynomial algorithm can solve it. Thus, testing of false candidates (*false test* or *false search*) degrades the performance a lot.

(2) **Costly candidate generation**. The generation of size $(k+1)$ subgraph candidates from size k frequent subgraphs is more complicated and costly than that of itemsets as observed by Kuramochi and Karypis [16].

Clearly, it is necessary to develop new methods to overcome these difficulties. Recently, there have been reports on successful algorithms, like PrefixSpan [19] and TreeMinerV [26] at mining sequential patterns and trees, respectively. Both explore depth-first search and straightforward pattern growth based on a single subpattern. A similar strategy is also adopted by FREQT [3]. These previous studies give us much confidence to explore new algorithms for mining frequent graphs.

In this paper, we develop *gSpan*, a new graph-based substructure mining algorithm without candidate generation, which aims to avoid the two most significant costs mentioned above.

gSpan adopts depth-first search (DFS) as opposed to breadth-first search (BFS) used inherently in Apriori-like algrothrims. We design a new canonical labeling system (*DFS lexicographic order*) to support DFS. Each graph is assigned a unique *minimum DFS code*. Based on DFS codes, a hierarchical search tree is constructed. By pre-order traversal of the tree, gSpan discovers all frequent subgraphs with required support. Since the design combines the subgraph isomorphism test and frequent subgraph growth into one procedure, gSpan dramatically accelerates the mining process. We introduce methods for partitioning of the frequent graphs according to the DFS lexicographic order and projection of graph datasets to fit the partitions. The partition and projection deliver good parallel and scaleup properties. We demonstrate how to apply pre-pruning, post-pruning and partial count pruning to optimize gSpan. We also extend gSpan to mine frequent induced subgraphs, which indicates the good extensibility of gSpan.

We evaluate the performance of gSpan using a synthetic data generator kindly provided by Kuramochi in University of Minnesota, which is the same generator also used in [16]. Our experimental results indicate that gSpan outperform FSG significantly by an order of magnitude.

The remaining of the paper is organized as follows. In Section 2, we define the basic concepts of frequent subgraph mining problem. Section 3 is an overview of the whole system and our design consideration. The major theoretical foundation of DFS Code, Minimum DFS Code, DFS Code Tree, and their properties are introduced in Section 4. Section 5 formulates the algorithm of gSpan. We present the analysis of the algorithm, its extension and the performance study in Section 6, and Section 7 summarizes our work.

2 Preliminary Concepts

In this paper, we focus on undirected labeled simple graph. However, it is trivial to fit our algorithm to directed graph and unlabeled graph. With some modification, our algorithm can be extended to process non-simple graphs with self-loops and multiple edges. The definition of labeled graph is given as follows.

Definition 1 (Labeled Graph) *A labeled graph can be represented by a 4-tuple,* $G = (V, E, L, l)$*, where*

V *is a set of vertices,*
$E \subseteq V \times V$ *is a set of edges,*
L *is a set of labels,*
$l : V \cup E \rightarrow L$*,* l *is a function assigning labels to the vertices and the edges.*

This definition can be generalized to include partially labeled graphs if the label set L includes an empty label.

Definition 2 (Isomorphism, Automorphism, Subgraph Isomorphism) *An isomorphism is a bijective function* $f : V(G) \rightarrow V(G^{'})$*, such that*

$\forall u \in V(G)$, $l_G(u) = l_{G'}(f(u))$, *and*
$\forall (u, v) \in E(G)$, $(f(u), f(v)) \in E(G^{'})$ *and* $l_G(u, v) = l_{G'}(f(u), f(v))$.

An automorphism of G *is an isomorphism from* G *to* G*. A subgraph isomorphism from* G *to* $G^{'}$ *is an isomorphism from* G *to a subgraph of* $G^{'}$*. If* f *is only injective, then* G *is monomorphic to* $G^{'}$*.*

Induced subgraph is a subgraph contained in another graph that maintains edge counts. *Induced subgraph isomorphism* can be considered as constrained subgraph isomorphism. In Section 5.4, a variant of our algorithm is formulated to solve induced subgraph mining problem under the same framework.

Definition 3 (Frequent Subgraph Mining) *Given a graph dataset,* $GS = \{G_i \mid i = 0...n\}$*, and*

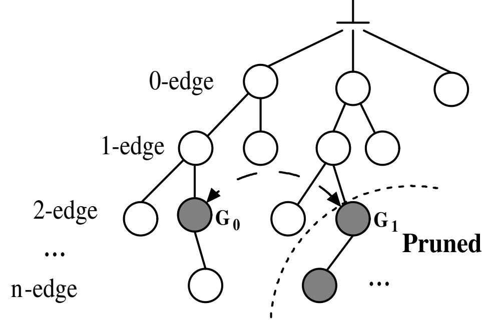


Figure 1: A Search Space: DFS Code Tree

a minimum support, minSup, let

$$\varsigma(g, G) = \begin{cases} 1 & \textit{if } g \textit{ is isomorphic to a subgraph of } G, \\ 0 & \textit{if } g \textit{ is not isomorphic to any subgraph of } G. \end{cases}$$

$$\sigma(g, GS) = \sum_{G_i \in GS} \varsigma(g, G_i)$$

$\sigma(g, GS)$ denotes the occurrence frequency of g in GS, i.e., the support of g in GS. ***Frequent Subgraph Mining*** *is to find every graph, g, such that $\sigma(g, GS)$ is greater than or equal to minSup.*

We assume that each frequent subgraph is connected. This restriction reduces the complexity of the problem and also reflects the connectivity property of hidden frequent structures in most situations.

3 An Overview

Our goal is to design an algorithm which can find frequent subgraphs one by one, from small to large ones. Therefore, a hierarchical search space should be built to facilitate the search. Figure 1 shows a sample abstract search space. The 0-edge nodes represent subgraphs which contain a single vertex. The 1-edge nodes represent subgraphs containing one edge, e.g., $X \stackrel{a}{-} Y$. The n-edge nodes are the children of the corresponding (n-1)-edge nodes. Literally, these nodes represent subgraphs which grow from (n-1)-edge graphs by adding one more edge. We develop a lexicographic order for these nodes to discover frequent subgraphs efficiently. We assign a lexicographic label to each node in this search space. The smaller lexicographic label a node has, the earlier the node will be discovered. In other words, we can enumerate subgraphs in increasing lexicographic order, which is consistent with the depth-first traversal of the search space. The search space also should guarantee easy-growing from high level to lower level and easy-backtracking from low level to higher level.

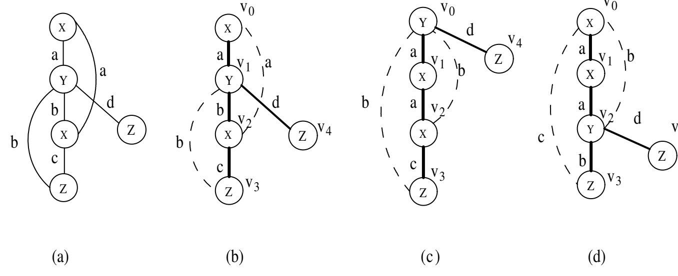


Figure 2: Depth-First Search Tree and its Forward/Backward Edge Set

It may be difficult to have a design that no two nodes in the search space represent the same subgraph. However, if we know that a subgraph has been discovered, then we can discard this duplicate one. Moreover, in our design, we prove that if the subgraph that a node represents has been discovered, all graphs that its descendent nodes represent must have been discovered too. Thus, we can discard them. For example, in Figure 1, if G_1 is the same as G_0, i.e., it has been discovered before the node that represents G_1 is reached, then all of its children must have been discovered too. Thus the subtree rooted from $G1$ can be pruned. This property is critical to keep the algorithm efficient. The following sections illustrate the supporting theorems and the algorithm design in detail.

4 Lexicographic Ordering in Graphs

Depth-first search is popularly used in graph algorithms. We define a *DFS code* for a given DFS tree in a graph. Using DFS code, we develop a new canonical labeling system to map a graph to a unique sequence. We build a *DFS Code Tree* to model the relations among all graphs, in which each node represents one graph and any graph can find its nodes in the DFS Code Tree although the mapping is not one-to-one. One graph can have several DFS codes (nodes) in the DFS Code Tree. We assign its first code (in the pre-order search of the DFS Code Tree), called *Minumum DFS Code*, to that graph as its canonical label. By pruning any node which contains non-minimum DFS code, we reduce the size of the search tree and hence the search space.

Depth-First Search (DFS) Tree. When performing a depth-first search [9] in a graph, we can construct a depth-first search tree. For example, graphs in Figure 2(b)-2(d) are isomorphic to that in Figure 2(a). The thickened edges in Figure 2(b)-2(d) represent three different DFS trees for the graph in Figure 2(a). It is clear that for one graph, there are lots of ways to construct different DFS trees by selecting different starting points and different growing edges.

DFS Subscripting. We denote DFS tree as T. The depth-first discovery of the vertices forms a linear order. We use subscripts to label this linear order according to their discovery time [9].

Given a DFS tree T, $v_i \prec_T v_j$ means v_i is discovered before v_j. Each vertex is assigned a subscript from 0 to $n-1$ if there are n vertices in G, such that the magnitude of subscripts reflects the relation: $\forall i,j,\ v_i \prec_T v_j$ *iff* $i < j$. We call v_0 the *root* and v_{n-1} the *rightmost vertex*. The straight path from the root to the rightmost vertex is named the *rightmost path*. In Figure 2(b)-2(d), three different subscriptings are generated for the graph in Figure 2(a). The right most path is (v_0, v_1, v_4) in Figure 2(b), (v_0, v_4) in Figure 2(c), and (v_0, v_1, v_2, v_4) in Figure 2(d). The rightmost vertex and the vertices on the rightmost path play an important role in growing patterns. We denote such subscripted G as G_T.

Forward Edge Set and Back Edge Set. Given G_T, forward edge (*tree edge* [9]) set contains all edges in the DFS tree, and backward edge (*back edge* [9]) set contains all edges which are not in the DFS tree. For the sake of simplicity, from now on, the (u,v) representation of an edge is written as an ordered pair where the subscripts of the vertices indicate which set the edge belongs to. We denote $E_{f,T} = \{e \mid \forall i,j,\ i<j,\ e=(v_i,v_j) \in E\}$ as the forward edge set in G_T, and $E_{b,T} = \{e \mid \forall i,j,\ i>j,\ e=(v_i,v_j) \in E\}$ as the backward edge set in G_T. In Figure 2(b), (v_0, v_1), (v_1, v_2), (v_2, v_3), and (v_1, v_4) constitute the forward edge set while (v_3, v_1) and (v_2, v_0) constitute the backward edge set.

We define two partial orders, $\prec_{f,T}$ on $E_{f,T}$, and $\prec_{b,T}$ on $E_{b,T}$, assume $e_1 = (v_{i_1}, v_{j_1}), e_2 = (v_{i_2}, v_{j_2})$,

$$e_1 \prec_{f,T} e_2, \forall e_1, e_2 \in E_{f,T}, \tag{1}$$

if and only if $j_1 < j_2$.

and

$$e_1 \prec_{b,T} e_2, \forall e_1, e_2 \in E_{b,T}, \tag{2}$$

if and only if one of the following holds.

$$(i) i_1 < i_2;\ (ii) i_1 = i_2 \ and\ j_1 < j_2.$$

Canonical Labeling and Minimum DFS Code. One way to solve graph isomorphism problem is to calculate the canonical labels of two graphs. If their canonical labels are the same, then these graphs are isomorphic to each other. There are two well-known algorithms to compute the canonical label. One is a variant of J. R. Ullmann's Backtracking algorithm, the other is B. D. McKay's Nauty algorithm. The latter is very efficient and powerful for its successful application of group theory [11]. For small sized graphs and sparse graphs, Cordella et al. [8] developed a better algorithm called VF using State Space Representation (SSR).

The basic idea of Nauty comes from the property that the canonical labeling of two isomorphic graphs should be the same. A naive but general canonical labeling system can be constructed by concatenating rows or columns of the adjacency matrix of a graph. For each permutation of

its vertices, we can get a string representation of this graph. Among these strings, a minimum lexicographic string is taken as the canonical label of the graph. With the construction of canonical label, it is trivial to prove that isomorphic graphs have one and only one canonical label. Yet it is possible that different permutations can work out the same canonical label independently because of automorphism.

We design a new canonical labeling system, called *minimum DFS encoding.* Besides the above two partial relations: $\prec_{f,T}$ and $\prec_{b,T}$, we introduce a third partial order between forward edge and backward edge, $\prec_{bf,T}$ on E, assume $e_1 = (v_{i_1}, v_{j_1}), e_2 = (v_{i_2}, v_{j_2})$,

$$e_1 \prec_{bf,T} e_2, \tag{3}$$

if and only if one of the following holds.

$$\begin{aligned} (i) \quad & e_1 \in E_{b,T}, e_2 \in E_{f,T}, i_1 < j_2; \\ (ii) \quad & e_1 \in E_{f,T}, e_2 \in E_{b,T}, j_1 \leqslant i_2. \end{aligned}$$

Theorem 1 *The relation $\prec_{E,T}$ defined by combining the partial orders, (1), (2), and (3), is a linear order on E. (proof omitted)*

The linear order $\prec_{E,T}$ can be defined in a simplified way: for all the edges in G (assume $e_1 = (i_1, j_1), e_2 = (i_2, j_2)$), (i) if $i_1 = i_2$ and $j_1 < j_2$, $e_1 \prec_{E,T} e_2$; (ii) if $i_1 < j_1$ and $j_1 = i_2$, $e_1 \prec_{E,T} e_2$; and (iii) if $e_1 \prec_{E,T} e_2$ and $e_2 \prec_{E,T} e_3$, $e_1 \prec_{E,T} e_3$.

Definition 4 (DFS Code) *Given a DFS tree T for a graph G, an edge sequence (e_i) can be constructed based on $\prec_{E,T}$, such that $e_i \prec_{E,T} e_{i+1}$, where $i = 0, \ldots, |E| - 1$. (e_i) is called a DFS code, denoted as code(G, T).*

Basically, a DFS code can be constructed in the following way. First, add a new vertex and a forward edge that connects one vertex in the old code with this new vertex. Then, add all backward edges that connect this new vertex to other vertices in the old code. Repeat this procedure to grow the code until all edges are included in the DFS code. For example, for the DFS tree shown in Figure 2(b), the DFS code of the graph is $((v_0, v_1), (v_1, v_2), (v_2, v_0), (v_2, v_3), (v_3, v_1), (v_1, v_4))$. For the DFS tree in Figure 2(c), the DFS code is $((v_0, v_1), (v_1, v_2), (v_2, v_0), (v_2, v_3), (v_3, v_0), (v_0, v_4))$. We can see that, for the same graph, different DFS trees can generate different DFS codes.

We represent an edge as (v_i, v_j), its two vertices' labels $l(v_i)$, $l(v_j)$, and its own label $l(v_i, v_j)$. For the sake of simplicity, we combine all of them into a 5-tuple: $(i, j, l_i, l_{(i,j)}, l_j)$. The values of i and j agree with DFS Subscripting for a given T. For example, (v_0, v_1) in Figure 2(b) is represented by $(0, 1, X, a, Y)$. Table 1 shows the corresponding DFS codes for Figure 2(b), 2(c), and 2(d). Since DFS code is a representation of a graph based on a depth-first search tree, there are some restrictions.

edge no.	(b) α	(c) β	(d) γ
0	$(0,1,X,a,Y)$	$(0,1,Y,a,X)$	$(0,1,X,a,X)$
1	$(1,2,Y,b,X)$	$(1,2,X,a,X)$	$(1,2,X,a,Y)$
2	$(2,0,X,a,X)$	$(2,0,X,b,Y)$	$(2,0,Y,b,X)$
3	$(2,3,X,c,Z)$	$(2,3,X,c,Z)$	$(2,3,Y,b,Z)$
4	$(3,1,Z,b,Y)$	$(3,0,Z,b,Y)$	$(3,0,Z,c,X)$
5	$(1,4,Y,d,Z)$	$(0,4,Y,d,Z)$	$(2,4,Y,d,Z)$

Table 1: DFS code for Figure 2(b), 2(c), and 2(d)

Property 1 (DFS Code's Neighborhood Restriction) *Given G, T, $\alpha = code(G,T)$, assume $\alpha = (a_0, a_1, ..., a_m)$, $m \geqslant 2$, and two neighbors are a_k and a_{k+1} $(0 \leqslant k < m)$. Let $a_k = (i_k, j_k, l_{i_k}, l_{(i_k,j_k)}, l_{j_k})$, and $a_{k+1} = (i_{k+1}, j_{k+1}, l_{i_{k+1}}, l_{(i_{k+1},j_{k+1})}, l_{j_{k+1}})$. Then a_k and a_{k+1} must agree with the following rules:*

rule 1. if a_k is a backward edge, then either of the following holds.

(i) if a_{k+1} is a forward edge, $i_{k+1} \leqslant i_k$ and $j_{k+1} = i_k + 1$;

(ii) if a_{k+1} is a backward edge, $i_{k+1} = i_k$ and $j_k < j_{k+1}$.

rule 2. if a_k is a forward edge, then either of the following holds.

(i) if a_{k+1} is a forward edge, $i_{k+1} \leqslant j_k$ and $j_{k+1} = j_k + 1$;

(ii) if a_{k+1} is a backward edge, $i_{k+1} = j_k$ and $j_{k+1} < i_k$.

The forms of rule 1 and rule 2 share some similarity. The property restricts that, for example, we cannot exchange the positions of edge 2 and edge 3 in code α in Table 1. Among the codes generated by all possible DFS trees for a graph, we have to pick one as its canonical label. The following discussion focuses on how to design a linear order among these codes. Since we are dealing with labeled graphs, we need to consider the label information as one of the ordering factors.

Definition 5 (DFS Lexicographic Order) *Suppose $Z = \{code(G,T) \mid T \text{ is a DFS tree of } G\}$, i.e., Z is a set containing all DFS codes for all the connected labeled graphs. Suppose there is a linear order $(\prec_L)$ in the label set (L), then the lexicographic combination of $\prec_{E,T}$ and $\prec_L$ is a linear order $(\prec_e)$ on the set $E_T \times L \times L \times L$. **DFS Lexicographic Order** is a linear order defined as follows. If $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, ..., a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, ..., b_n), \alpha, \beta \in Z$, then $\alpha \leqslant \beta$ iff either of the following is true.*

$$(i) \quad \exists t, 0 \leqslant t \leqslant min(m,n), a_k = b_k \; for \; k < t, a_t \prec_e b_t$$

$$(ii) \quad a_k = b_k \; for \; 0 \leqslant k \leqslant m, \; and \; n \geqslant m.$$

In another word, **DFS Lexicographic Order in Z** is a linear order defined as follows. If $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, ..., a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, ..., b_n), \alpha, \beta \in Z$, then $\alpha \leqslant \beta$ iff either of the following is true. Assume the forward edge set and backward edge set for T_α and T_β are $E_{\alpha,f}$, $E_{\alpha,b}$, $E_{\beta,f}$, and $E_{\beta,b}$ respectively. Also let $a_t = (i_a, j_a, l_{i_a}, l_{(i_a,j_a)}, l_{j_a})$ and $b_t = (i_b, j_b, l_{i_b}, l_{(i_b,j_b)}, l_{j_b})$,

(i) *for some* $t, 0 \leqslant t \leqslant min\{m, n\}$, *we have* $a_k = b_k$ *for* $k < t$, *and**

$$a_t < b_t = \begin{cases} true\ if\ a_t \in E_{\alpha,b}\ and\ b_t \in E_{\beta,f}. \\ true\ if\ a_t \in E_{\alpha,b},\ b_t \in E_{\beta,b},\ and\ j_a < j_b. \\ true\ if\ a_t \in E_{\alpha,b},\ b_t \in E_{\beta,b},\ j_a = j_b,\ and\ l_{(i_a,j_a)} < l_{(i_b,j_b)}. \\ true\ if\ a_t \in E_{\alpha,f},\ b_t \in E_{\beta,f},\ and\ i_b < i_a. \\ true\ if\ a_t \in E_{\alpha,f},\ b_t \in E_{\beta,f},\ i_a = i_b,\ and\ l_{i_a} < l_{i_b}. \\ true\ if\ a_t \in E_{\alpha,f},\ b_t \in E_{\beta,f},\ i_a = i_b,\ l_{i_a} = l_{i_b},\ and\ l_{(i_a,j_a)} < l_{(i_b,j_b)}. \\ true\ if\ a_t \in E_{\alpha,f},\ b_t \in E_{\beta,f},\ i_a = i_b,\ l_{i_a} = l_{i_b},\ l_{(i_a,j_a)} = l_{(i_b,j_b)},\ and\ l_{j_a} < l_{j_b}. \end{cases}$$

(ii) $a_k = b_k$ *for* $0 \leqslant k \leqslant m$, *and* $n \geqslant m$.

For the graph in Figure 2 (a), there exist tens of different DFS codes. Three of them, which are based on the DFS trees in Figure 2(b)-2(d) are listed in Table 1. According to DFS lexicographic order, code γ is less than code α, which is less than code β.

Definition 6 (Minimum DFS Code) *Given a graph* G, $Z(G) = \{code(G, T) \mid \forall T,\ T$ *is a DFS tree for* $G\}$, *based on DFS lexicographic order, the minimum one,* $min(Z(G))$, *is called* ***Minimum DFS Code*** *of* G. *It is also the canonical label of* G.

We use $min(G)$ to denote $min(Z(G))$, and $min(\alpha)$ to denote the minimum DFS code of the graph represented by the DFS code α. It is clear that among all codes generated by the possible DFS trees for the graph in Figure 2(a), γ in Table 1 is its minimum DFS code. Figure 2(d) shows the corresponding DFS tree for this code. **DFS Lexicographic Order in graphs** is that, given two graphs G and G', $G < G'$, if and only if $min(G) < min(G')$. Without explicit mentioning, the DFS lexicographic ordering discussed below is that on DFS codes instead of graphs. Till now, we have built minimum DFS encoding for graphs.

Theorem 2 *Given two graphs* G *and* G', G *is isomorphic to* G' *if and only if* $min(G) = min(G')$. *(proof omitted)*

Thus the problem of mining frequent connected subgraphs is equivalent to mining their corresponding minimum DFS codes. This problem turns to be a sequential pattern mining problem with slight difference, which conceptually can be solved by existing sequential pattern mining algorithms.

*Implied by DFS Code's Neighborhood Restriction, because $a_k = b_k$ *for* $k < t$, if $a_t \in E_{\alpha,b},\ b_t \in E_{\beta,b}$, then $i_a = i_b$ and $l_{i_a} = l_{i_b}$; if $a_t \in E_{\alpha,f},\ b_t \in E_{\beta,f}$, then $j_a = j_b$ in this case.

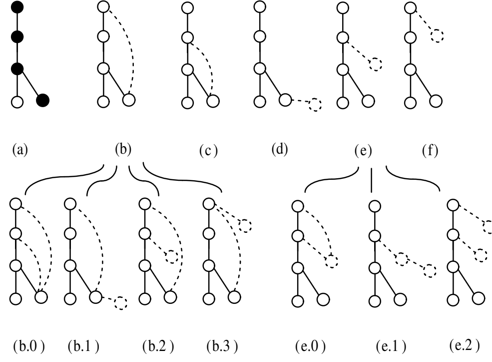


Figure 3: DFS Code/Frequent Pattern Growing

Definition 7 (DFS Code's Parent and Child) *Given a DFS code* $\alpha = (a_0, a_1, ..., a_m)$*, any valid DFS code* $\beta = (a_0, a_1, ..., a_m, b)$ *is called* α*'s* ***child****, and* α *is called* β*'s* ***parent****. We denote* $children(\alpha) = \{\beta \mid \forall\beta,\ \alpha\, is\, \beta' s\, parent\}$.

According to DFS Code's Neighborhood Restriction (Property 1), a valid DFS code which grows from another DFS code cannot have arbitrary edge growth. In the definition of DFS Code's Parent and Child, the code β cannot add a new edge b to an arbitrary position in the graph that its parent code represents. In fact, to construct a valid DFS code, b must be an edge which only grows from the vertices on the rightmost path. In Figure 3, the graph shown in 3(a) has several potential children with one edge growth, which are shown in 3(b)-3(f) (assume the darkened vertices constitute the rightmost path). Among them, 3(b), 3(c), and 3(d) grow from the rightmost vertex while 3(e) and 3(f) grow from other vertices on the rightmost path. Backward edges can only grow from the rightmost vertex while forward edges can grow from vertices on the rightmost path. This restriction is similar to TreeMinerV's equivalence class extension [26] and FREQT's rightmost expansion [3] in frequent tree discovery. The enumeration order of these children is enhanced by the DFS lexicographic order, i.e., it should be in the order of 3(b), 3(c), 3(d), 3(e), and 3(f). 3(b.0)-3(b.3) are children of 3(b), and 3(e.0)-(e.2) are children of 3(e).

Definition 8 (DFS Code Tree) *In a DFS Code Tree, each node represents a DFS code, the relation between parent node and child node complies with the relation described in Definition 7. The relation between siblings is consistent with the DFS lexicographic order. That is, the pre-order search of DFS Code Tree follows the DFS lexicographic order. The Tree is denoted as* $\mathbb{T}$.

Given a label set L, a DFS Code Tree can be constructed by Definition 8, which should contain all possible graphs for this label set. Later, we will illustrate how to enumerate frequent subgraphs by following the pre-order traversal of this tree. Figure 1 shows a DFS Code Tree, the $(n+1)_{th}$ level of the tree has nodes which contain DFS codes of n-edge graphs. *DFS code* and *node in the*

DFS Code Tree are equivalent in the sense that one can be derived from the other. Any valid DFS code has a unique corresponding node in the DFS Code Tree, and any node in the DFS Code Tree contains a valid DFS code. Certainly, some of the nodes contain a minimum DFS code while others do not.

Property 2 (DFS Code Tree Covering) *DFS Code Tree contains minimum DFS codes for all graphs.*

Definition 9 (DFS Code's Ancestors and Descendants) *Given two DFS codes, α and β, in $\mathbb{T}$, if there is a straight path from α to β, then α is called an ancestor of β, and β is called a descendant of α, denoted by $anc(\beta) = \{all\ ancestors\ of\ \beta\}$, and $des(\alpha) = \{all\ descendants\ of\ \alpha\}$.*

Theorem 3 (Frequency Antimonotone) *If a graph G is frequent, then any subgraph of G is frequent. If G is not frequent, then any graph which contains G is not frequent. It is equal to say, if a DFS code α is frequent, $\forall\ \beta \in anc(\alpha)$, β is frequent. If α is not frequent, $\forall\ \beta \in dec(\alpha)$, β is not frequent.*

It is observed that in the DFS Code Tree $\mathbb{T}$ there probably exist different DFS codes for a graph. According to the definition of Minimum DFS Code, the first occurrence of DFS code of a graph in $\mathbb{T}$ (pre-order) is its minimum DFS code. By Theorem 4, we want to prove that the pruning of non-minimum DFS codes for the same graph and their descendants still preserves the DFS Code Tree Covering property.

Algorithm 1 Construction of a DFS code.

step 1. initialize $i = 0$, compare a_0 with e;
    if $a_0 > e$, select e as the first element for the DFS code, goto step 3.
step 2. if $(a_0, ..., a_{i-1}, e)$ is invalid or $(a_0, ..., a_i) \leqslant (a_0, ..., a_{i-1}, e)$
        $i = i + 1$;
        if $(i < n)$ goto step 2;
    otherwise, $(a_0, \ldots, a_{i-1}, e)$ forms the first $i + 1$ elements for the DFS code. goto step 3.
step 3. build the remaining elements from $i + 1$ to $n + 1$.

Theorem 4 (DFS Code Growth) *Given a DFS code β, if $\alpha = min(\beta)$,$\alpha < \beta$. Let $\mathbb{D}_\gamma = \{\eta \,|\, \forall\eta,\, \eta < \gamma\}$, $\forall\delta, \delta \in$ children(β), i.e. any valid DFS code generated by one edge growth from β, $min(\delta) \in \mathbb{D}_\alpha \cup children(\alpha) \subseteq \mathbb{D}_\beta$.*

Proof. Suppose $\beta = (b_0, b_1, \ldots, b_n)$, $\alpha = min(\beta) = (a_0, a_1, \ldots, a_n)$, $\alpha \in \mathbb{D}_\beta$. We grow a new edge e from G_β to $G_{\beta,e}$. Using the rightmost expansion, one code for $G_{\beta,e}$ can be $(b_0, b_1, \ldots, b_n, e)$. However, we can apply Algorithm 1 to build another DFS code for $G_{\beta,e}$ based on α.

The DFS code from step 3 is in two cases: (i) $(a_0, \ldots, a_{i-1}, e, a'_i, \ldots, a'_n)$, and (ii) $(a_0, \ldots, a_n, e)$. Both of them belong to $\mathbb{D}_\alpha \cup children(\alpha)$, which is a subset of $\mathbb{D}_\beta$. $\square$

This theorem shows that if a node in the DFS Code Tree does not contain a minimum DFS code, then its minimum DFS code must appear before this node in the pre-order search of the DFS Code Tree. For the sake of simplicity, a code (or node) is called *duplicate code* (or *duplicate node*) if and only if it is not minimum, i.e., the graph it represents (or its minimum code) must have appeared (or have been discovered) before this code (node) is reached in the pre-order search of the DFS Code Tree. The subgraph that a duplicate code (node) represents is called *duplicate subgraph*.

Theorem 5 (DFS Code Pruning) *Given a graph G, its DFS codes in $\mathbb{T}$, $\alpha_0, \alpha_1, ..., \alpha_n, \forall i, j \leqslant n, \alpha_i \leqslant \alpha_j$ (by DFS lexicographic order), and α_0 is the minimum DFS code for G. The remaining DFS Code Tree after pruning $\alpha_i (1 \leqslant i \leqslant n)$ and all its descendants (its subtree) still preserves Property 2.*

Proof. Given β, $\beta \neq min(\beta)$. Let $\mathbb{D}_\beta = \{\eta \mid \forall \eta, \eta < \beta\}$. Suppose we can prove the following proposition,

$$\forall p \in des(\beta), if\ min(p) \in \mathbb{D}_\beta, then\ \forall q \in children(p), min(q) \in \mathbb{D}_\beta.$$

Because $\beta \neq min(\beta)$, $\forall \delta, \delta \in\ children(\beta)$ we have $min(\delta) \in \mathbb{D}_\beta$, and by induction, $\forall p \in des(\beta) \cup \{\beta\}$, then $min(p) \in \mathbb{D}_\beta$. Now we prove this proposition.

Let $p \in\ des(\beta)$, $min(p) \in \mathbb{D}_\beta$. $\forall q \in children(p)$, by DFS Code Growth theorem, $min(q) \in \mathbb{D}_{min(p)} \cup children(min(p)) \subseteq \mathbb{D}_\beta$. $\square$

Theorem 6 (DFS Duplicate Code Cardinality) *Given a graph G, $|E(G)| \geqslant 2$, $\mathcal{Z}(G) = \{\beta | \forall \beta \in Z(G), \exists \alpha, \alpha = min(\alpha), \beta \in\ children(\alpha)\}$, then $|\mathcal{Z}(G)| \leqslant |E(G)| \times |V(G)|$, that is, the number of G's codes which are children of other minimum codes is bounded by the product of the number of its edges and vertices.*

Proof. Assume G has m edges ($m \geqslant 2$), i.e., $m = |E(G)|$. G has at most m different connected subgraphs, each of which has $m-1$ edges (remove one edge from G). These subgraphs generate at most m different minimum DFS codes. For each minimum DFS code, we add back the m_{th} edge to form possible codes of G. If the removed edge is a forward edge, there are at most $|V(G)| - 1$ vertices from which a new forward edge can grow. Otherwise, if it is a backward edge, it must grow from the rightmost vertex. There are at most $|V(G)| - 2$ vertices (If self-loop and multi-edge are permitted, it is $|V(G)|$), one of which can be the vertex to form this backward edge. Therefore, $|\mathcal{Z}(G)| \leqslant |E(G)| \times |V(G)|$. $\square$

This bound is not tight because the m_{th} edge cannot be added to an arbitrary position mentioned above in order to keep the topology of G. For many graphs, $|\mathcal{Z}(G)|$ is less than $|E(G)|$. Since we

prune all duplicate codes and their descendants, we do not have chance to access the duplicate codes which are children of other duplicate codes. Our algorithm gSpan formulated below only accesses those duplicate codes which are children of other minimum DFS codes. Therefore, for each frequent subgraph, the number of duplicate codes gSpan accesses is bounded by the product of the number of its edges and vertices.

These theorems build a solid foundation for discovering all frequent subgraphs. By pre-order searching of the DFS Code Tree, it is guaranteed that we can enumerate all potential frequent subgraphs. The DFS Code Pruning of duplicate nodes in the tree still makes this searching complete and sound while Frequency Antimonotone helps finding all frequent subgraphs without violating the completeness and soundness.

5 The gSpan Algorithm

In this section, we formulate our gSpan algorithm based on DFS lexicographic order and its properties given and proved above. gSpan uses a sparse adjacency list representation to store graphs. Algorithm 2 lays out the pseudo code of the framework.

Algorithm 2 GraphSet_Projection($\mathbb{GS}$,$\mathbb{S}$).

1: sort labels of the vertices and edges in $\mathbb{GS}$ by their frequency;
2: remove infrequent vertices and edges;
3: relabel the remaining vertices and edges in descending frequency;
4: $\mathbb{S}^1 \leftarrow$ all frequent 1-edge graphs in $\mathbb{GS}$;
5: sort $\mathbb{S}^1$ in DFS lexicographic order;
6: $\mathbb{S} \leftarrow \mathbb{S}^1$;
7: **for each** edge $e \in \mathbb{S}^1$ **do**
8: initialize s with e, set $s.GS = \{g \mid \forall g \in \mathbb{GS}, e \in E(g)\}$; (only graph ID is recorded)
9: Subgraph_Mining($\mathbb{GS}$, $\mathbb{S}$, s);
10: $\mathbb{GS} \leftarrow \mathbb{GS} - e$;
11: **if** $|\mathbb{GS}| < minSup$;
12: **break**;

Step 1 (line 1-6): Remove infrequent vertices and edges from the graph set $\mathbb{GS}$. Relabel the remaining ones in descending frequency. Add all frequent 1-edge graphs into $\mathbb{S}^1$ and sort them in DFS lexicographic order. For example, we have a label set {A,B,C,...} for vertices, {a,b,c,...} for edges. Each 1-edge graph has only one edge such as $(0,1,A,a,A)$, $(0,1,A,a,B)$, ..., and the like. According to DFS lexicographic order, $(0,1,A,a,A) < (0,1,A,a,B) < \ldots$. Since we resort the labels of vertices and edges separately, it is possible that even a vertex labeled "A" has the most frequent occurrences and an edge labeled "a" has the most frequent occurrences, but

the combination, (A, a, A), may not occur most frequently. Thus, if $(0, 1, A, a, A)$ is a frequent subgraph in $\mathbb{GS}$, then $(0, 1, A, a, A)$ has a very high support in $\mathbb{GS}$ if "highest" cannot be claimed. $\mathbb{S}^1$ is important in both Step 2 and Step 3. In Step 2, the 1-edge frequent subgraphs are the seeds that generate lots of children (2-edge frequent subgraphs) followed by more descendants (3 or more edges frequent subgraphs). In Step 3, we use the elements in $\mathbb{S}^1$ to project the whole graph set $\mathbb{GS}$.

Step 2 (line 8-9): For each 1-edge frequent subgraph, Subgraph_Mining grows all nodes in the subtree (Figure 1) rooted at this 1-edge graph. This part will be elaborated later.

Step 3 (line 10): Shrink each graph in the graph set $\mathbb{GS}$ by removing the edge after all descendants of this 1-edge graph have been searched. For example, say (A, a, A), (B, b, C), (C, c, C) are all frequent edges in $\mathbb{GS}$, in the first round (line 7), Subgraph_Mining finds out all frequent subgraphs which contain (A, a, A). In the second round, Subgraph_Mining finds out all frequent subgraphs which contain (B, b, C) but do not contain any edge of (A, a, A). In the third round, Subgraph_Mining finds out all structures which contain only (C, c, C). Therefore, (A, a, A) can be removed from the dataset before the second round runs. The same happens to (B, b, C) before the third round runs. This step projects $\mathbb{GS}$ into a smaller graph set with less vertices, edges and graphs. Thus it makes the successive mining procedure faster and faster.

Step 4 (line 7,11): Terminate when all frequent 1-edge graphs and their descendants are generated. The program should terminate at line 7 instead of line 11 in all cases. Line 11 is included for completeness.

Subprocedure 1 Subgraph_Mining($\mathbb{GS}$, $\mathbb{S}$, s).

1: **if** $s \neq min(s)$
2:     **return**;
3: $\mathbb{S} \leftarrow \mathbb{S} \cup \{s\}$;
4: generate all s' potential children with one edge growth;[†]
5: Enumerate(s);
6: **for each** c, c is s' child **do**
7:     **if** $support(c) \geq minSup$
8:         $s \leftarrow c$;
9:         Subgraph_Mining($\mathbb{GS}$, $\mathbb{S}$, s);

In each recursive run, Subgraph_Mining($\mathbb{GS}$, $\mathbb{S}$, s) grows one edge from s and discover all frequent children of s (s is a DFS code or a node in the DFS Code Tree). The recursion in Subgraph_Mining follows the pre-order traversal of DFS Code Tree in Figure 1. Thus, the discovery order of frequent subgraphs follows DFS lexicographic order, i.e., the minimum DFS code of previously discovered subgraphs should be less than that of later discovered ones. $s \neq min(s)$ prunes duplicate subgraphs and all their descendants. Theorem 5 proves that such pruning does not affect Property 2. There-

[†]This is not efficient. A better one is given in Section 5.2.

fore, it is guaranteed that the pruning does not affect the completeness of the result. The pruning efficiently reduces the search space and redundant computation. (Remember that any test of false frequent subgraphs costs a lot, and even a less costly test to determine $s \neq min(s)$ is computationally expensive.) If s is the minimum DFS code of the graph it represents, Subgraph_Mining adds s to its frequent subgraph set. Then it generates all potential children with one edge growth, and recursively runs Subgraph_Mining on each child. The generation of potential children is not costly. It is trivial to do so by following the rules forced by Property 1. A more efficient approach will be introduced by Section 5.2.

Subprocedure 2 Enumerate(s).

1: **for each** $g \in s.GS$ **do**
2: enumerate the next occurrence of s in g;
3: **for each** c, c is s' child and occurs in g **do**
   $c.GS \leftarrow c.GS \cup \{g\}$;
4: **if** g covers all children of s **break**;

By enumerating all occurrences (more accurately until all possible children are discovered) of s in each graph, Enumerate(s) (Subprocedure 2) counts the occurrences of all the children of s in the graph. For example, Figure 4(a) shows a frequent subgraph discovered through previous routines, the possible children are shown in Figure 3(b)-3(f). Figure 4(b) is a graph in a graph dataset. In Figure 4(c), a thickened line or cycle marks an occurrence of 4(a) in 4(b). Dotted lines and cycles are illustrated as potential candidates of 4(a)'s children (with one more edge). By Property 1, (W, a, Z) is not a valid child. By counting the occurrences of the children over all the graphs where s takes place, we know which child is a frequent subgraph too. Furthermore, we record IDs of all graphs which contain the frequent subgraph. It is used by Enumerate in the next round to find its children. By executing Subgraph_Mining and Enumerate alternately, the complete set of frequent subgraphs is generated in DFS lexicographic order.

There is no requirement for specific subgraph isomorphism testing procedure in line 2 of Subprocedure 2 Enumerate(s). J. R. Ullmann's backtracking [24] can be used, and B. D. Mckay's Nauty algorithm works too. We design our own enumerating engine which basically backtracks in the depth-first search tree. It combines the subgraph matching and DFS code growth into one process, thus it saves lots of computation and makes the algorithm work for induced subgraph isomorphism problem too. The enumerating engine works as follows. Given a minimum DFS code of a subgraph, we want to find all occurrences of this subgraph in another graph from a dataset. We start from the first edge in the DFS code to try to find a matching to one of the edges in that graph. Then we repeatedly do the same work by adding new edges. Sometimes we have to do one edge backtracking if we cannot find a new edge for the current matching. This process repeats until either we find a matching or fail. Since gSpan needs to find all occurrences of a graph, once we find

a matching and do some children counting, we continue trying to find a new matching. Because all occurrences are discovered in one scan, the total cost is not much. In the next part, we discuss several optimizations. At the end of this section, we give out the analysis of the algorithm and its extension.

5.1 The Nontrivial $s \neq min(s)$ Pruning

$s \neq min(s)$ prunes all DFS codes which are not minimum. It significantly reduces unnecessary computation on duplicate subgraphs and their descendants. There are two ways to do that. The first is cutting off any child whose code is not minimum after line 4 of Subgraph_Mining. It is called pre-pruning which happens before the real counting in line 5. The second is cutting off them in line 1. It is called post-pruning which happens after the real counting. The first approach is costly since most of duplicate subgraphs are not even frequent. However, we cannot completely postpone the pruning since the counting of duplicate frequent subgraphs, which have been discovered before, is a waste. Therefore, our algorithm adopts a trade-off between pre-pruning and post-pruning. It first prunes any child that explicitly has been discovered. The pruning happens in four stages: (1) If the first edge of s' minimum DFS code is e_0, then a potential child of s does not contain any edge which is smaller than e_0. (2) For any backward edge growth from s, assume $(v_i, v_j), (i > j)$, this edge should be no smaller than any edge which is connected to v_j in s. For example, in Figure 4(b), the edge (X, a, Z) should not be added to the graph 4(a) to form a new child since (X, a, Z) is less than (X, c, Y). The adding of (X, a, Z) makes the minimum DFS code of the new graph less than that of its parent in Figure 4(a), thus, it must have been discovered before. (3) Since all edges grow from vertices on the rightmost path, to some extent, it prunes those children growing from vertices in other positions. For example, by Property 1, (W, a, Z) is pruned. (4) Post-pruning is applied to the remaining unpruned nodes. A naïve way to calculate $min(s)$ is to generate all the DFS codes of the graph that s represents and then pick up the smallest one. The generation procedure is almost the same as enumerating all automorphisms of s. A heuristic way is to take advantage of Definition 5 and Definition 6, i.e., whenever some part of DFS code is generated, it is compared with s. If it is less than s, it can be concluded that s is not the minimal one and thus it can be discarded. After these four steps, all duplicate codes and their descendants are discarded. Thus gSpan does not generate any duplicate subgraphs.

5.2 Partial Count Pruning

In Subprocedure 1, given a frequent subgraph s, we need to generate all potential children of s with one edge growth. There are three approaches to do that with small cost. In the first approach, all possible children are generated at the beginning. When we search the occurrences of s in the graph set, any valid child of s is counted. In this way, some counts for possible children may be zero. This approach is too naive. The second approach uses an ad hoc way, which need not

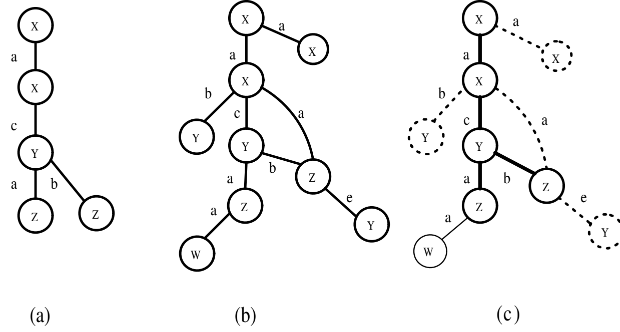


Figure 4: Subgraph Growing

generate any possible children before counting. Instead, whenever we find a new child of s, we add it into the children list of s, and count it. In this way, each count for a child must be at least one. Given a graph $s = (e_0, e_1, \ldots, e_n)$, its possible children set $PS = \{c \mid \forall e_x, c = (e_0, e_1, \ldots, e_n, e_x)\}$, its available children set $VS = \{c \mid c \in PS,\ c\ is\ a\ subgraph\ in\ GS\}$, $VS \subseteq PS$. The first approach generates PS and then counts. The second approach generates VS in an ad hoc way and then counts. The possible optimizations of pruning duplicates in these two approaches have been discussed in Section 5.1. The third approach takes advantage of Frequency Antimonotone. By analyzing the DFS code growing in Figure 3, we find that there are lots of information that can be used to prune unnecessary counting. For example, In Figure 3, all children of 3(a) in 3(c) form can provide enough information to generate the potential children of 3(b) in 3(b.0) form. The same phenomena happens in 3(d) for 3(b.1), 3(e) for 3(b.2), and 3(f) for 3(b.3). If $(e_0, e_1, \ldots, e_n, e_x, e_y)$ is frequent, then $(e_0, e_1, \ldots, e_n, e_x)$ and $(e_0, e_1, \ldots, e_n, e_y)$ must be frequent if both of them are valid. Therefore, $(e_0, e_1, \ldots, e_n, e_x)$ and $(e_0, e_1, \ldots, e_n, e_y)$ provide a good indication for the possible frequent occurrence of $(e_0, e_1, \ldots, e_n, e_x, e_y)$. Sometimes $(e_0, e_1, \ldots, e_n, e_y)$ may be invalid. For example, it seems that for 3(e.1) no child of 3(a) can provide any hint for possible candidates of 3(e.1) without complicated computation done. Therefore, in such case, we still use one of the first two approaches. The bottom line is that such case only represents a small portion of all frequent subgraphs. For the first two approaches, we have to count all s's children no matter whether they are frequent or not. The third approach proposes partial count pruning. That is, for all s' children in VS, we still need to access them, but we don't update counter for those obvious infrequent ones. It only counts some of s' children. This pruning is different from the popular one that Apriori-like algorithms use, where the pruning of any $(k+1)$-subgraph (if one of its k-subgraphs is not frequent) is to reduce unnecessary testing. In gSpan, partial count pruning is to reduce unnecessary counter updating. It does not rely on any complicated hash function to check k-closure in this case. Our experiments show that its cost is nearly negligible, but this optimization makes the program run one time faster. It is one of the most significant speedups we obtain besides what the algorithm itself achieves.

5.3 Analysis

gSpan does a pre-order search of the DFS Code Tree. Property 2 and Theorem 5 guarantee that the DFS Code Tree covers all the graphs. Theorem 3 shows gSpan does not miss any frequent subgraph after pruning. Therefore, its completeness and soundness are proved. The major advantages of gSpan against AGM and FSG are listed as follows:

- **No Candidate Generation and False Test.** In gSpan, just as PrefixSpan, the frequent $(k+1)$-edge subgraphs grow from k-edge frequent subgraphs directly. It does not perform candidate generation and test any false candidate opposed to Apriori-like algorithms. Furthermore, compared with those for mining frequent itemsets, candidate generation and false test for subgraph mining is quite costly. FSG does some optimization on it. gSpan completely avoids the computation of this part.

- **Space Saving from Depth-First Search.** gSpan is a depth-first search algorithm, while Apriori-like ones, such as AGM and FSG, adopt breadth-first search strategy. All of them can take advantage of recording relevant transaction IDs to avoid some redundant false search and test. However, BFS suffers from much higher I/O and memory usage than DFS, especially in their peak space usage situation. It is highly possible that a DFS algorithm can fit in memory while its corresponding BFS one cannot.

- **Quickly Shrunk Graph Dataset.** As depicted in Algorithm 2, at each iteration the mining procedure is performed in such a way that the whole graph dataset is shrunk to the one containing a smaller set of graphs, with each having less edges and vertices. The speedup is larger when less graphs and edges are involved during the mining process.

Subgraph isomorphism problem is an NP-complete problem. Therefore, the runtime of gSpan should be exponential. If measured by the number of subgraph and/or graph isomorphism tests, the runtime can be bounded by $O(kFS + rF)$, where k is the maximum number of subgraph isomorphisms existing between a frequent subgraph and a graph in the dataset, F is the number of frequent subgraphs, S is the dataset size, and r is the maximum number of duplicate codes of a frequent subgraph that grow from other minimum codes. kFS bounds the number of isomorphism tests that should be done in order to find frequent supergraphs from discovered frequent subgraphs. rF bounds the maximum number of $s \neq min(s)$ operations. When considering the value of k, in some extreme situations, for example, two complete graphs without labels, k (the possible subgraph isomorphisms between them) can be P_n^m , where m and n are the number of these two graphs' vertices ($m \leqslant n$). However, k is pretty small for sparse graphs with diverse labels. Another important factor is r. It has been proved in Theorem 6 that for any subgraph G, the number of duplicate nodes of G which grow from other minimum DFS codes is bounded by the product of the number of its edges and vertices. Therefore, r is bounded by the maximum product of the number

of edges and vertices that a frequent subgraph has. Because of pre-pruning, r is much smaller than that number for real datasets.

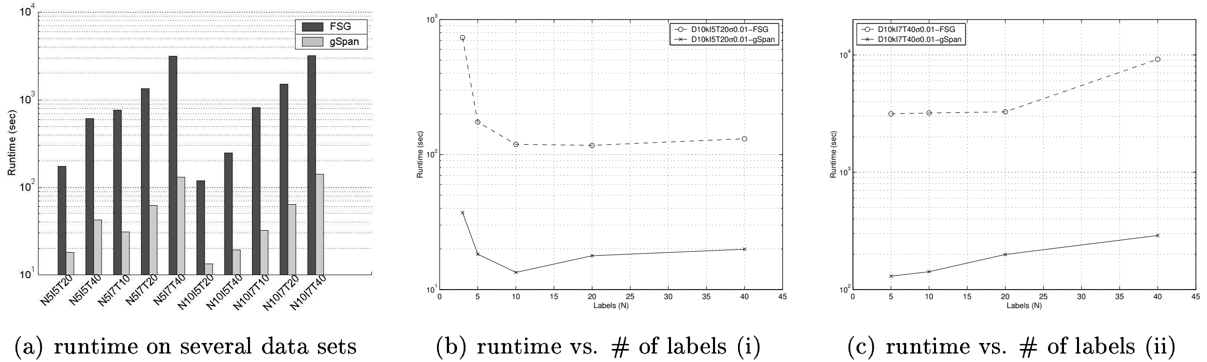
5.4 From Subgraph Isomorphism to Induced Subgraph Isomorphism

In previous sections, we discussed how to find frequent subgraphs in a graph dataset. Here, we formulate a modified version of gSpan, called induced-gSpan, which can discover frequent induced subgraphs. The induced subgraph definition says if H is an induced subgraph of G, then $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, and if two vertices appear in both graphs, the edge between them must also be present in both of them. Therefore, it is a subgraph with constraint(s). We make the following modification on gSpan. First, we cannot shrink the graph set any more since we have to check whether some edges in G are present in H if H is induced subgraph isomorphic to G. But we still can shrink the graph set by vertices. Second, we grow the patterns in the same way of gSpan, however, we need to test whether those patterns are really isomorphic to induced subgraphs with a minimum support. We define an intermediate graph, I. I is subgraph isomorphic to G and $I - \{v\}$ is induced subgraph isomorphic to G, where v is the rightmost vertex of I for a DFS tree in G. Generation of frequent induced subgraphs is conducted in two phases: (1) generate intermediate graphs which are frequent in the graph set above $minSup$; and (2) when growing and counting the children of frequent intermediate graphs, induced subgraph isomorphism is checked for these intermediate graphs, and a new count is calculated and compared with $minSup$. If it is still greater than $minSup$, output it. The easy fitness of gSpan to induced subgraph problem indicates that gSpan can be flexibly extended to solve other constrained frequent subgraph mining problems.

6 Experiments and Performance Study

A comprehensive performance study has been conducted in our experiments on both synthetic and real world datasets. We use a synthetic data generator provided by Kuramochi and Karypis [16]. The real data set we tested is a chemical compound dataset. Our performance tests show that gSpan outperforms FSG by 6 to 45 times on synthetic datasets whereas about 15-100 times faster on the chemical compound dataset. gSpan also demonstrates a better scalability over FSG since it succeeds in completing the mining processes with lower thresholds and larger datasets.

All experiments of gSpan and induced-gSpan are done on a 500MHZ Intel Pentium III PC with 448 MB main memory, running Red Hat Linux 6.2. We also implemented our version of FSG which achieves similar performance as that reported in [16]. Here we compare the performance of gSpan with that of FSG reported in [16] if the result is available, otherwise we show our own implementation result based on the same datasets. They did the test on a 650MHZ Intel Pentium III PC with 2GB main memory, also running the Linux OS. Although there exist some differences in these two testing environments, the great speedup achieved by gSpan makes such inaccuracy



(a) runtime on several data sets (b) runtime vs. # of labels (i) (c) runtime vs. # of labels (ii)

Figure 5: Runtime: gSpan vs. FSG

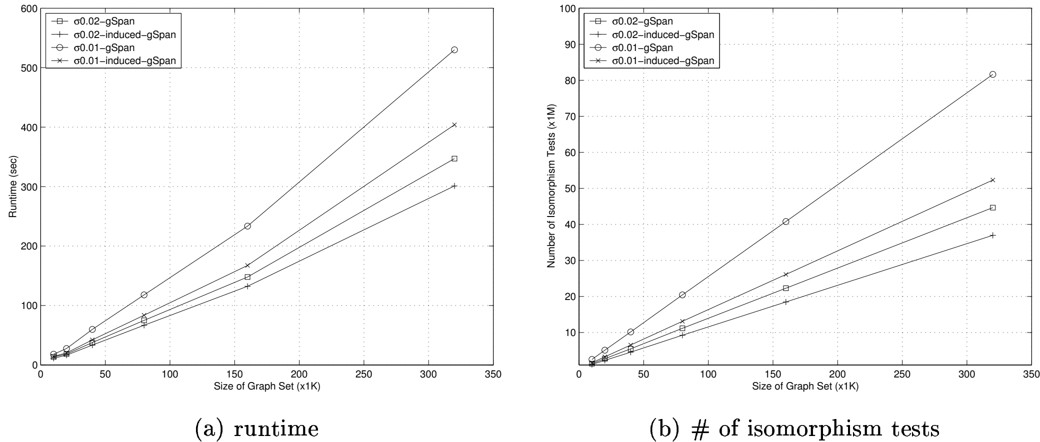
irrelevant. We use the VFLib Graph Matching Library (Version 2.0) provided by Cordella, et al. [8] to verify and debug the experimental results. The frequent subgraphs discovered are double checked by their package and no duplicate or missing pattern is observed.

Synthetic Datasets. The synthetic datasets are generated using a similar procedure described in [1]. Kuramochi et al. applied a simplified procedure in their graph data synthesis. The details about how to generate the datasets were described in [16]. We use the data generator provided by Kuramochi. The synthetic data sets can be described by five parameters: (1) $|D|$, the total number of graphs generated, (2) $|T|$, the average size of graphs in terms of edges, (3) $|I|$, the average size of potential frequent subgraphs (*frequent kernels*), (4) $|L|$, the number of potentially frequent kernels, and (5) $|N|$, the number of possible labels, as denoted in [16]. For a dataset which has 10000 graphs, each graph has 20 edges in average, the potential frequent subgraph has 10 edges in average, 200 potential frequent kernels, and 4 available labels, we represent this dataset as $D10kN4I10T20L200$. Since $|L|$ is always set to 200, we omit $L200$. Another parameter is the minimum support, σ. We use percentage to represent it.

We test the performance of gSpan in various synthetic datasets and compare it with FSG. Then we illustrate the situations where gSpan works better above its average performance. The third experiment shows that gSpan scales linearly in the size of dataset.

Figure 5(a) shows the runtime comparison between gSpan and FSG for datasets with fixed $|D| = 10k$, $|L| = 200$, and $\sigma = 0.01$. The datasets vary by possible labels ($|N|$), the size of frequent kernels ($|I|$), and the average size of graphs ($|T|$). In this series of datasets, gSpan outperforms FSG by a factor from 6 to 30.

Figure 5(b) and 5(c) show the runtime comparisons on two other series of datasets. One has fixed $|D| = 10k$, $|L| = 200$, $|I| = 5$, $|T| = 20$, and $\sigma = 0.01$. The other has fixed $|D| = 10k$, $|L| = 200$, $|I| = 7$, $|T| = 40$, and $\sigma = 0.01$. Both of them have various $|N|$ from 5 to 40. We find that, compared with FSG, the fewer labels the dataset has, the faster gSpan runs; the larger each graph is, the faster gSpan performs. As the experiments show in Figure 5(b), the speedup for



(a) runtime (b) # of isomorphism tests

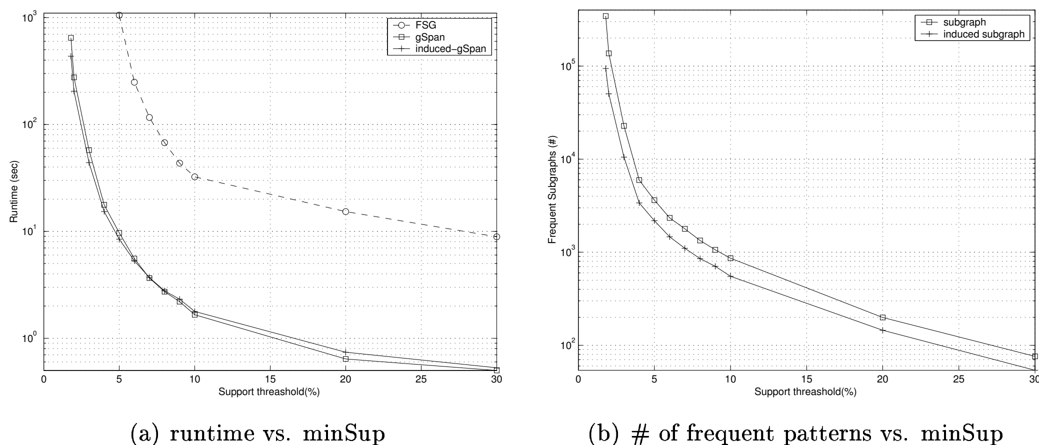
Figure 6: Scalability on the Size of Graph Set (Synthetic Dataset)

dataset $D10kI5T20\sigma0.01$ is down to around 6 for small graphs with lots of labels. In that case, the structure of frequent subgraphs turns to be simple and the total number of frequent subgraphs turns to be small. However, the performance gain is enlarged to 15∼45 when the graphs turn to be bigger and denser, as the runtime result of $D10kI7T40\sigma0.01$ shows in Figure 5(c).

Figure 6 shows the scalability of gSpan and induced-gSpan. The experiments are done for synthetic datasets whose sizes vary from 10k to 320k. The other parameters are fixed, $|N| = 10$, $|I| = 5$, and $|T| = 20$. It tests two different minimum supports, 0.01 and 0.02. With the increasing size of graph datasets, the runtime increases linearly. It is an inherent result since the synthetic data generator controls the number of frequent subgraphs by the parameters, L, I, and T. Therefore, the number of frequent subgraphs is not sensitive to the size of datasets, but to the minimum support σ if other parameters are fixed. This property is not valid for extreme parameters. Our experiment shows under fixed $|L|$, $|I|$, $|T|$, and σ, the number of frequent subgraphs does not change a lot. Figure 6(b) shows the total number of graph isomorphism tests is proportional to the size of the datasets. It is proved that only the magnitude of dataset size S is increased in $O(kFS + rF)$ for this series of datasets. Therefore, the runtime is proportional to the size of the datasets too. Figure 6(a) also shows the performance of induced-gSpan, which is used to mine frequent induced subgraphs. It is observed that generally the runtime of mining frequent induced subgraphs is less than that of mining frequent subgraphs. The reason is although more computation is needed to enumerate induced subgraphs in the graph dataset, the number of frequent induced subgraphs is less than that of frequent subgraphs.

Chemical Compound Dataset. This chemical compound dataset is the same one used in [16]. Readers can refer to this URL ‡ to retrieve it. We apply the same construction method used in [16] to transform the data into a graph set. Figure 7 illustrates the runtime and the number of discovered frequent patterns as the minimum support varies from 2% to 30%. The total memory

‡http://oldwww.comlab.ox.ac.uk/oucl/groups/machlearn/PTE.



(a) runtime vs. minSup (b) # of frequent patterns vs. minSup

Figure 7: Runtime and # of Frequent Patterns (Chemical Compound Dataset)

consumption is less than 100M for any point of gSpan and induced-gSpan shown in the figure. The runtime in Figure 7(a) shows the scalability of gSpan. gSpan achieves better performance by 15-100 times compared with FSG. For example, if we use $\sigma = 0.06$, gSpan spends 5.29 seconds compared with 248 seconds used by FSG. While FSG aborts computation under 5% because of too long running time or exhausted main memory, gSpan can continue the mining process until 1.5% support threshold is reached. Figure 7(b) shows the number of frequent patterns we find for both subgraphs and induced subgraphs.

We try to understand why gSpan has such a higher speedup rate in this chemical compound dataset compared with that in synthetic datasets. After analyzing the structures of the data and discovered frequent subgraphs, we find those chemical compounds have lots of tree-like structures. It is consistent with the inherent DFS tree structure used in gSpan. Therefore, it reduces the search space more efficiently. Another reason is that there are only four kinds of bonds connecting atoms, which make the occurrences of labels in each compound very dense and possibly generates long patterns. Thus, it takes advantage of gSpan. The dense distribution of labels results in bigger performance loss of FSG than that of gSpan. Considering the runtime curve in Figure 5(b), the runtime gap between gSpan and FSG is widened while $|N|$ turns to be smaller for the dataset $D10kI5T20\sigma0.01$.

7 Conclusions

In this paper, we investigated issues for frequent subgraph mining and addressed the possible inefficiencies in the previous work. We introduced a new lexicographic ordering system and formulated gSpan to mine frequent subgraphs more efficiently than the existing ones. gSpan outperforms FSG by an order of magnitude and is capable to mine larger frequent subgraphs in a bigger graph set with lower minimum supports. gSpan is promising not only because it mines various kinds of

subgraphs efficiently but also because it can be applied (with some minor extensions or revisions) to mining nearly all kinds of frequent substructures, including sequences, trees, and lattices. It provides a general framework for scalable mining of more complicated patterns. The practice of extending gSpan to mine frequent induced subgraphs indicates the flexibility of gSpan to solve other constraint problems.

There are many interesting research problems related to gSpan that should be pursued further. For example, the extension of gSpan to mining frequent closed- and max- subgraphs is an important problem for further study because this will lead to the generation of a much less number of frequent subgraphs without loss much useful information.

Acknowledgements. The synthetic data generator is kindly provided by Mr. Michihiro Kuramochi and Professor George Karypis at the Department of Computer Science/Army HPC Research Center, University of Minnesota. Mr. Kuramochi also gave helpful comments on several issues. Dr. Pasquale Forggia, at Dipartimento di Informatica e Sistemistica Università di Napoli " Federico II ", provided helpful suggestions about the usage of VFlib graph matching library. We also thank Guoming Shou and Yanli Tong for their comments. The work was supported in part by U.S. National Science Foundation NSF IIS-02-09199, University of Illinois, and an IBM Faculty Award.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, March 1995.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. 2002 SIAM Int. Conf. Data Mining*, Arlington, VA, April 2002.

[4] R. Attias and J. E. Dubois. Substructure systems: concepts and classifications. *Journal of Chemical Information and Computer Sciences*, 30:2–7, 1990.

[5] D. M. Bayada, R. W. Simpson, and A. P. Johnson. An algorithm for the multiple common subgraph problem. *Journal of Chemical Information and Computer Sciences*, 32:680–685, 1992.

[6] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 85–93, Seattle, WA, June 1998.

[7] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 443–452, Heidelberg, Germany, April 2001.

[8] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Graph matching: A fast algorithm and its evaluation. In *Proceedings of the 14th Int. Conf. on Pattern Recognition (ICPR-16)*, pages 1582–1584, Aug. 1998.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001, Second Edition.

[10] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 30–36, New York, NY, Aug. 1998.

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

[12] B. Liu G. Cong, L. Yi and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. 2002 SIAM Int. Conf. Data Mining*, Arlington, VA, April 2002.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[14] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 169–180, Seattle, WA, July 1994.

[15] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PDKK'00*, pages 13–23, 2000.

[16] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, San Jose, CA, Nov. 2001.

[17] H. Mannila, H Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

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

[19] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[20] K. Shearer, H. Bunke, and S. Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.

[21] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 22–33, Dallas, TX, May 2000.

[22] S. Su, D. J. Cook, and L. B. Holder. Knowledge discovery in molecular biology: Identifying structural regularities in proteins. *Intelligent Data Analysis*, 3:413–436, 1999.

[23] Y. Takahashi, Y. Satoh, and S. Sasaki. Recognition of largest common fragment among a variety of chemical structures. *Analytical Sciences*, 3:23–28, 1987.

[24] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[25] E. K. Wong. Model matching in robot vision by subgraph isomorphism. *Pattern Recognition*, 25(3):287–304, 1992.

[26] M. J. Zaki. Efficiently mining frequent trees in a forest. Technical Report 01-7, Department of Computer Science, Rensselaer Polytechnic Institute, 2001 (to appear KDD 2002).

[27] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. 2002 SIAM Int. Conf. Data Mining*, pages 457–473, Arlington, VA, April 2002.