FOGGER: An Algorithm for Graph Generator Discovery[*]

Zhiping Zeng[1], Jianyong Wang[2], Jun Zhang[3], Lizhu Zhou[4]
Department of Computer Science and Technology
Tsinghua University, Beijing, 100084, P.R.China
{clipse.zeng[1], zhangjun03[3]}@gmail.com, {jianyong[2], dcszlz[4]}@tsinghua.edu.cn

ABSTRACT

To our best knowledge, all existing graph pattern mining algorithms can only mine either closed, maximal or the complete set of frequent subgraphs instead of graph generators which are preferable to the closed subgraphs according to the ***Minimum Description Length*** principle in some applications. In this paper, we study a new problem of frequent subgraph mining, called frequent connected graph generator mining, which poses significant challenges due to the underlying complexity associated with frequent subgraph mining as well as the absence of Apriori property for graph generators. Whereas, we still present an efficient solution FOGGER[1] for this new problem. By exploring some properties of graph generators, two effective pruning techniques, ***backward edge pruning*** and ***forward edge pruning***, are proposed to prune the branches of the well-known DFS code enumeration tree that do not contain graph generators. To further improve the efficiency, an effective index structure, ADI++, is also devised to facilitate the subgraph isomorphism checking. We experimentally evaluate various aspects of FOGGER using both real and synthetic datasets. Our results demonstrate that the two pruning techniques are effective in pruning the unpromising parts of search space, and FOGGER is efficient and scalable in terms of the base size of input databases. Meanwhile, the performance study for graph generator-based classification model shows that generator-based model is much simpler and can achieve almost the same accuracy for classifying chemical compounds in comparison with closed subgraph-based model.

*This work was supported in part by National Natural Science Foundation of China under grant No. 60873171, 60573061 and 60833003, National Basic Research Program of China under Grant No. 2006CB303103, Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology(TNList), Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, State Education Ministry of China.

[1]FOGGER stands for **F**requent c**O**nnected **G**raph **G**en**ER**ator

Keywords

Graph mining, graph generator, graph classification

1. INTRODUCTION

As graph structures provide a general way to model a variety of relationships among different objects and arise naturally in a wide range of disciplines and applications, graph pattern mining has raised great interest and become a very active topic in data mining research. One of the essential problem formulations in graph pattern mining is frequent subgraph mining based on a given input graph database and a user-specified minimum support threshold, which has shown various applications in bioinformatics[8], subgraph-index based data management[28, 33, 3, 25] and so on. Many efficient algorithms[23, 32, 21, 18, 20] have been developed and widely used in real applications, typical examples including AGM[11], TreeMiner[30], FSG[12].

However, due to the intractable ***combinatorial explosion problem***, frequent subgraph mining is a very time-consuming process and usually generates a huge number of frequent subgraphs. For example, as shown in [26], nearly 1,000,000 frequent subgraphs with a minimum support threshold of 5% can be mined even from the small and sparse CA dataset which contains 422 active chemical compounds in AIDS antiviral screen dataset provided by NCI/NIH. The huge number of frequent subgraphs also makes some further data analysis like feature selection and chemical compound classification a very challenging task.

Fortunately, not all the frequent subgraphs are of interest from the application point of view. We can divide the complete set of frequent subgraphs into a set of equivalence classes. Informally speaking, a set of frequent subgraphs forms an equivalence class if and only if they are supported by the same set of input graphs. The maximal subgraphs in each equivalence class are called closed subgraphs, while the minimal ones are called graph generators. As we can see, because both the set of frequent graph generators and the set of frequent closed subgraphs are subsets of all frequent subgraphs, they tend to be more concise. In addition, to mine graph generators(or closed subgraphs) only, we can devise more effective optimization techniques to prune some parts of the search space which contain no graph generator(or closed subgraph). Thus, both graph generator mining and closed subgraph mining can be potentially more efficient than all frequent subgraph mining too. To our best knowledge, although there is some work on mining frequent closed subgraphs[26] or fully connected closed subgraphs only[24], no attention has been paid to graph generator mining.

Compared with closed subgraph mining, graph generator mining has its own advantage and deserves some research efforts. From the graph data classification point of view, graph generators are usually preferable to their corresponding closed subgraphs according to the MDL principle [16, 15, 6], which has a sound statistical foundation rooted in the well-known Bayesian inference and Kolmogorov complexity. We will explain this in detail in Section 2.2 after the introductionn of some necessary preliminaries. Moreover, we will compare the average pattern size and the number of patterns in the result set of graph generators with those in the result set of closed subgraphs, and show the effectiveness of the generator-based graph classification model in the performance study section.

To our best knowledge, this work is the first attempt to design an algorithm for mining frequent connected graph generators. We summarize our contributions as follows:

- We formally define the problem of mining frequent connected graph generators, explore their properties, and present the first graph generator mining algorithm, FOGGER.
- Two novel pruning techniques whose effectiveness are verified in the performance study section are proposed to prune the unpromising parts of search space.
- An effective index structure, ADI++, is devised to assist the subgraph isomorphism checking, which can be widely used as an underlying index structure for a large number of existing graph pattern mining algorithms.
- Extensive study has been conducted to validate the algorithm's efficiency and scalability, and demonstrate its utility in graph data classification such as classifying chemical compounds.

The rest of this paper is organized as follows. Section 2 gives the problem formulation, explains the motivation of graph generator mining and explores some properties of graph generators. Section 3 presents our solution for frequent connected graph generator mining. It first gives a brief introduction about the DFS(depth-first search) code tree enumeration framework and introduces two novel pruning techniques. And then, generator checking scheme is described as well as ADI++ structure which can facilitate the subgraph isomorphism checking. At the end of this section, the integrated algorithm for mining frequent connected graph generators, FOGGER, is outlined. Section 4 investigates the comprehensive performance study followed by related work discussion in Section 5. At last, this study concludes with Section 6.

2. PRELIMINARIES

In this section, we introduce some preliminary concepts and notations about frequent subgraph mining, formulate and motivate the problem of frequent connected graph generator mining. Some properties of graph generators are also explored.

2.1 Graph Generators and Closed Graphs

In this paper, only simple graph structures are considered, i.e., undirected graphs without multi-edges and self-loops. An ***undirected labeled graph*** G can be represented by a 6-tuple, $G = (V, E, L_v, L_e, F_v, F_e)$, where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, L_v and L_e are the sets of vertex labels and edge labels respectively, and $F_v : V \rightarrow L_v$ and $F_e : E \rightarrow L_e$ are labeling functions assigning labels to vertices and edges respectively. G is said to be ***connected*** if at least one path exists between any pair of vertices in V, otherwise it is called a ***disconnected*** graph. In addition, singleton graphs are considered as connected graphs. G_1 is ***graph isomorphic*** to another graph G_2 iff there exists a bijection $f : V_1 \rightarrow V_2$ such that for any vertex $v \in V_1$, $f(v) \in V_2 \wedge F_v(v) = F_v(f(v))$, and for any edge $(u, v) \in E_1$, $(f(u), f(v)) \in E_2 \wedge F_e(u, v) = F_e(f(u), f(v))$. G_1 is a ***subgraph*** of G_2 iff $V_1 \subseteq V_2$ and $E_1 \subseteq E_2 \cap (V_1 \times V_1)$, denoted by $G_1 \sqsubseteq G_2$ or $G_1 \sqsubset G_2$ (i.e., $G_1 \sqsubseteq G_2$ but $G_1 \neq G_2$). Equivalently, G_2 is a ***supergraph*** of G_1 and is said to contain G_1. Moreover, if G_1 is isomorphic to a subgraph g of G_2, G_1 is said to be ***subgraph isomorphic*** to G_2, and g is called an ***instance*** of G_1 in G_2.

A ***graph database*** $\mathcal{D}$ is defined as a set of input graphs whose cardinality is denoted by $|\mathcal{D}|$. Figure 1 shows a graph database example containing four input graphs which will be used as our running example in the following discussion. Assume $\mathcal{P}$ is the set of subgraph patterns each of which is contained by at least one input graph in $\mathcal{D}$. The ***Galois connection***[4] between $2^{\mathcal{D}}$ and $2^{\mathcal{P}}$ is a couple of functions (f, g) where $f(p) = \{d \in \mathcal{D} \,|\, p \sqsubseteq d\}$ and $g(D) = \{p \,|\, \forall d \in D, p \sqsubseteq d\}$. Intuitively, $f(p)$ is the set of all input graphs in $\mathcal{D}$ containing p, and dually, $g(D)$ is the set of all subgraph patterns in $\mathcal{P}$ shared by all input graphs in D. The ***Galois closure operators*** are the following functions: $h = g \circ f$ and $h' = f \circ g$, where $\circ$ denotes the composition of functions. Given an subgraph pattern p, $h(p) = g(f(p))$ is called the ***closure*** of p.

The closure induces an ***equivalence relation*** $\sim_{\mathcal{D}}$ by $p_1 \sim_{\mathcal{D}} p_2$ iff $h(p_1) = h(p_2)$. Thus, the ***equivalence class*** of a subgraph pattern p, denoted by$[p]$, is defined as the set $\{p' | h(p') = h(p)\}$. For an equivalence class $[p]$, if $p_0 \in [p]$ and $\nexists\, p' \in [p]$ such that $p_0 \sqsubset p'$, p_0 is a maximal pattern in $[p]$ and is said to be ***closed***; otherwise, if $\nexists\, p' \in [p]$ such that $p' \sqsubset p_0$, p_0 is a minimal pattern in $[p]$ and is called a ***generator***. For the patterns in the form of itemsets, there is one and only one closed pattern in each equivalence class[14]. But for connected graph patterns, there could be more than one closed patterns in an equivalence class. The ***support*** of a subgraph pattern p, denoted by $sup(p)$, is defined as the number of input graphs containing p, i.e., $sup(p) = |f(p)|$. A subgraph p is said to be ***frequent*** if its support is no less than a user-specified threshold min_sup.

Problem Statement: *Given an input graph database $\mathcal{D}$ and a minimum support threshold min_sup, we study the problem of mining the complete set of graph generators which are frequent and also connected.*

For simplicity, if the context is clear we will omit the graph database $\mathcal{D}$. Moreover, since we only consider connected graphs, if not explicitly stated, the notation "graph" means a connected graphs by default in the rest of this paper.

2.2 MDL Favors Generators

In Section 1, we stated that graph generators are preferable to closed subgraphs according to MDL principle. Here we give a theoretical description of this principle and explain the reason. A crude two-part version of MDL principle[7] can be described as follows: let $\mathcal{H} = \{H_1, H_2, \cdots, H_n\}$ be

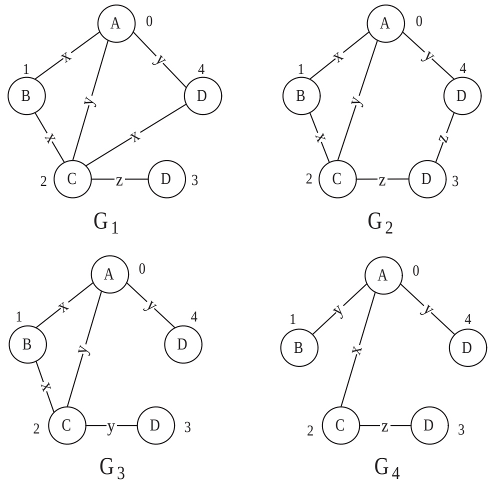


Figure 1: An Example of Graph Database

a set of candidate models, each of which contains a set of point hypotheses, the best point hypothesis $H \in H_1 \cup H_2 \cup \cdots \cup H_n$ to explain D is the one which minimizes the sum $L(H)+L(D|H)$, where $L(H)$ is the length of the description of this hypothesis and $L(D|H)$ is the length of the description of D when encoded with the help of H. The best model to explain D is the smallest model containing the selected H.

The same as [5] and [14], we apply this principle in the context of graph generators and closed subgraphs. Assume Q is an equivalence class of some data D, g and c are a generator and a closed subgraph in Q respectively. Let $D_Q = f(g) = f(c)$, then g and c are two hypotheses describing D_Q. Because g and c occur in the same data D_Q, $L(D_Q|c) = L(D_Q|g)$ holds. Furthermore, g is no "greater" than c, then $L(g) \leq L(c)$. Therefore, we can get that $L(c) + L(D_Q|c) \geq L(g) + L(D_Q|g)$. According to MDL, g is preferable to c for describing the data D_Q. The advantages of graph generators over closed subgraphs are also confirmed by the result data in classifying chemical compounds conducted in Section 4.

2.3 Properties of Graph Generators

Based on the definition of generators, one way to get all generators in an equivalence class $[p]$ is to elicit them from all patterns belonging to $[p]$. In order to determine whether p' belongs to $[p]$ or not, we can test whether $h(p) = h(p')$ holds. However, the computation of $h(p) = g(f(p))$ is an expensive operation. $f(p)$ is needed first, and then $g(f(p))$ can be calculated based on $f(p)$. Given a set D of input graphs, the computation of $g(D)$ is equivalent to discover the complete set of frequent subgraphs in D with the support threshold of 100% which itself is a tough problem. Therefore, we need to find an efficient method to determine whether $h(p) = h(p')$. The lemma introduced next provides us such an efficient method.

LEMMA 2.1. $h(p_1) = h(p_2)$ *iff* $f(p_1) = f(p_2)$.

PROOF. **Sufficiency**. According to the definition of $h(p)$, if $f(p_1) = f(p_2)$, $g(f(p_1)) = g(f(p_2))$ must hold. Therefore, $h(p_1) = h(p_2)$ holds.
Necessity. Since $h(p_1) = h(p_2)$, $g(f(p_1)) = g(f(p_2))$ must hold. Because $p_1 \in g(f(p_1))$, then $p_1 \in g(f(p_2))$. From the definition of the function g, $\forall\, d \in f(p_2)$, $p_1 \sqsubseteq d$. Moreover, according to the definition of f, we know that $d \in f(p_1)$. Therefore, $\forall\, d \in f(p_2)$, $d \in f(p_1)$, i.e. $f(p_2) \subseteq f(p_1)$. In the same way, we can prove that $f(p_1) \subseteq f(p_2)$. Consequently, $f(p_1) = f(p_2)$. $\Box$

Based on Lemma 2.1, in order to determine whether $h(p_1) = h(p_2)$, we only need to examine whether $f(p_1) = f(p_2)$ or not. Thus, the computation of $g(f(p_1))$ and $g(f(p_2))$ are ignored, and the determination is much easier. This lemma also indicates that all subgraph patterns in an equivalence class are contained in the same set of input graphs. Moreover, given two subgraph patterns p_1 and p_2, if $p_1 \sqsubset p_2$ and $sup(p_1) = sup(p_2)$, $f(p_1) \supseteq f(p_2)$ and $|f(p_1)| = |f(p_2)|$ hold. Therefore, $f(p_1) = f(p_2)$ holds, i.e., $h(p_1) = h(p_2)$.

In [14], the authors introduced and proved the Apriori property for itemset generators: every proper-subset of an itemset generator is also an itemset generator. Depending on this nice property, several effective pruning techniques were proposed, and an efficient algorithm for mining the frequent itemset generators was introduced. This nice property, however, no longer holds for graph generators. For example, assume there is a graph database D_0 consisting of four graphs C_1, C_2, C_3 and C_4 shown in Figure 2. By setting min_sup=1, we know that C_1 is a graph generator in D_0, but C_2 is not a graph generator even it is a proper-subgraph of C_1.

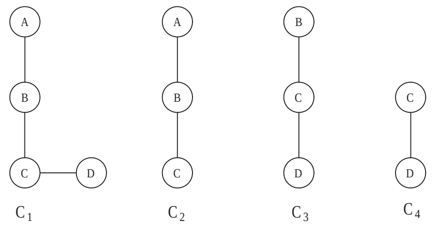


Figure 2: Apriori Property Absence for Graph Generators

As illustrated in the above example, the Apriori property does not hold for graph generators. Thus, the Apriori property-based pruning techniques used in itemset generator mining[14] cannot be applied to graph generator mining. In addition, graphs in general have undesirable theoretical properties with regard to algorithmic complexity. In terms of complexity theory, subgraph isomorphism is NP-Complete. Furthermore, no efficient algorithm is known to perform systematic enumeration of the subgraphs of a given graph. Consequently, all of the above factors make it quite difficult and challenging to discover frequent graph generators.

3. EFFICIENT MINING OF GENERATORS

In this section, we illustrate how to efficiently mine the complete set of frequent graph generators from input graph databases. First, we give a brief introduction about DFS code tree enumeration framework which is adopted to enumerate frequent subgraphs in this paper. And then, two novel pruning techniques are introduced to prune the branches of the numeration tree that do not contain graph generators. A generator checking scheme is devised to discover graph generators. To improve the efficiency, an effective underlying index structure, ADI++, is devised to assist subgraph isomorphism checking. Integrated the above techniques, an efficient algorithm FOGGER is presented at the end of this section.

3.1 DFS Code Tree Enumeration Framework

All frequent subgraph mining algorithms need an enumeration framework to guarantee that they can discover the complete set of frequent subgraphs. ***DFS code tree*** enumeration framework[27] is an effective and widely used method to enumerate frequent connected subgraphs. In this paper we also adopt this well-known enumeration framework. DFS code enumeration framework is based on the ***minimum DFS code***, which is a good canonical representation of graphs and has a nice property: two graphs g and g' are graph isomorphic to each other iff $min(g) = min(g')$(here $min(g)$ denotes the minimum DFS code of graph g). Moreover, with the help of minimum DFS code, the problem of mining frequent subgraph patterns is reduced to mining frequent minimum DFS codes, which are sequences with constraints preserving the connectivity of graphs. Please refer to [27] for more details about the minimum DFS code and the DFS code tree enumeration framework.

Figure 3 illustrates the DFS code tree for enumerating the frequent subgraphs with $min\_sup = 2$ from our running example graph database shown in Figure 1. There is a pair of numbers in the form of "a:b" near each node in the tree, which represents the order of this node being enumerated and the support of this node in the graph database respectively. In the rest of this paper, we use the order to represent a node in the DFS code tree. For instance, node 14 indicates the subgraph $\langle 0, 1, A, y, C\rangle$. In Figure 3, the nodes with gray background are frequent generators, and there are 26 frequent subgraphs with $min\_sup = 2$, only seven of which are generators. Many unpromising parts of search space should not been enumerated. In the next subsection, optimization techniques will be described to prune some of these unpromising branches by exploring some properties of graph generators.

At first, we introduce two important terms, ***forward edge*** and ***backward edge***, which will be used in the following discussion. For an edge in the DFS code representation of a graph, if its start vertex is discovered before its end vertex in DFS search, it is a forward edge; otherwise, it is a backward edge. For example, in node 2 of Figure 3, $\langle 1, 2, B, x, C\rangle$ is a forward edge while $\langle 2, 0, C, y, A\rangle$ is a backward edge.

LEMMA 3.1. *For a connected graph, if any number of backward edges are removed, the derived subgraph is still connected; furthermore, if no edge starts from the end vertex of a forward edge, the subgraph derived by removing this forward edge is still connected.*

The proof of the above lemma is simple, thus it is omitted here. Based on Lemma 3.1, we know that the removal of backward edges or some special forward edges which satisfy the specific constraint from a connected graph will not cause the disconnectivity. It will be applied in the proof of two pruning techniques introduced next.

3.2 Pruning Techniques

Besides the underlying complexity associated with frequent subgraph mining, the Apriori property no longer holds for graph generators. Thus, devising effective pruning techniques for graph generator mining is quite difficult and challenging. In this subsection, we introduce two novel pruning techniques whose effectiveness is confirmed by the experimental results shown in Section 4.

Given a node p in the DFS code tree(i.e., a subgraph pattern), let $E_x(p)$ denote the set of valid extensible DFS edges which can be used to extend p to get "bigger" subgraph patterns. For a node $p = (a_0, a_1, ..., a_n)$ and a DFS edge $e \in E_x(p)$, the number of input graphs which contain the subgraph pattern $(a_0, a_1, ..., a_n, e)$ in the input database is called the ***conditional support*** of e w.r.t. p, denoted by $sup(e|p)$. For simplicity, in the following we use $p \diamond e$ to denote the subgraph pattern $(a_0, a_1, ..., a_n, e)$. Accordingly, we know that $sup(p \diamond e) = sup(e|p)$ holds. Based on the definition of graph generators, we can easily get the following lemma.

LEMMA 3.2. *For a subgraph pattern p and a DFS edge $e \in E_x(p)$, if $sup(e|p) = sup(p)$, $p \diamond e$ cannot be a generator.*

The above lemma is evident and the proof is omitted here. Thereby, any subgraph pattern $p \diamond e$ with $sup(e|p) = sup(p)$ is definitely not a generator. Moreover, $p \diamond e$ is said to be a ***generator candidate*** if $sup(e|p) < sup(p)$. Apparently, only generator candidates might be graph generators, and graph generators can only be discovered from the set of generator candidates.

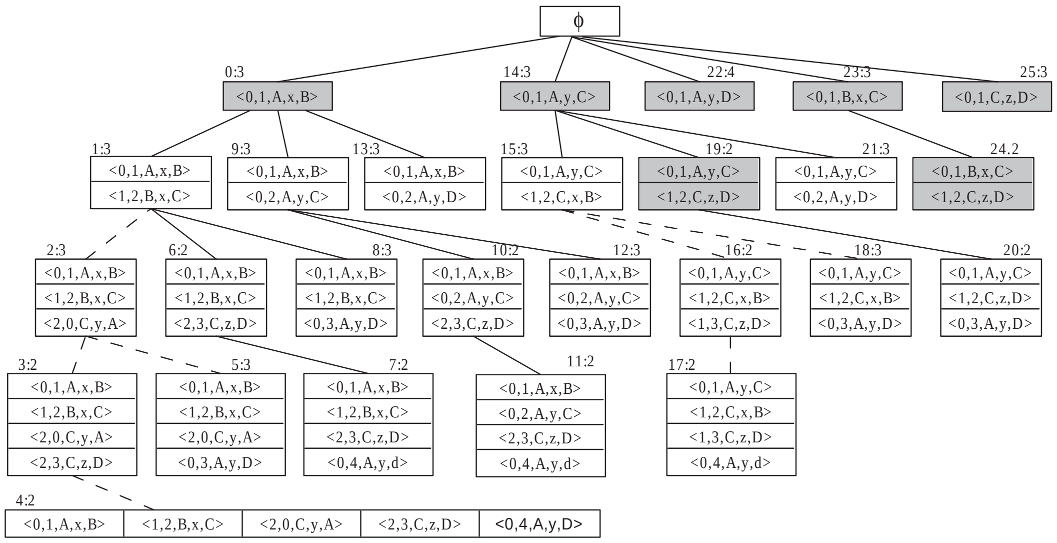
To date, using DFS code tree enumeration framework, a straightforward solution can be devised to discover the complete set of graph generators from the set of generator candidates. However, this rudimentary approach is rather brute-force and will take an unacceptable time. In the following, we elaborate on pruning techniques which can improve the efficiency. For a node p in the DFS code tree, the branch rooted by p can be pruned only on the condition that all descendants of p are not graph generators. More specifically, for any descendant p' of p, if there exists a connected subgraph which is derived by removing one edge from p' and possesses the support of $sup(p')$, the branch rooted by p can be safely pruned. Based on this idea, two novel pruning techniques are introduced in the following.

3.2.1 Backward Edge Pruning

LEMMA 3.3. *Given a subgraph pattern $p = (a_0, a_1, ..., a_n)$ ($n \geq 0$) in the DFS code tree, let $p' = p \diamond (e, b_0, b_1, \ldots, b_m)$ ($m \geq 0$) be a descendant of p in the DFS code tree, if $e \in E_x(p)$ is a backward extensible edge for each instance of p, then $sup(p') = sup(p'')$ must hold where $p'' = p \diamond (b_0, b_1, \ldots, b_m)$.*

PROOF. Since e is a backward edge, according to Lemma 3.1, p'' is connected which is derived by removing e from p'. Moreover, because $p'' \sqsubset p'$, $f(p'') \supseteq f(p')$ holds. Assume $f(p'') \neq f(p')$, then at least one input graph d_0 exists s.t. $d_0 \in f(p'') - f(p')$, i.e., $p'' \sqsubseteq d_0$ and $p' \not\sqsubseteq d_0$. Accordingly, d_0 must contain some instances of p. Because e is an extensible edge for all instances of p, for any instance t_p of p in d_0, t_p has an extensible DFS edge e. Furthermore, since inserting backward edges to a subgraph pattern does not introduce new vertices to the current subgraph pattern[2], any super-

[2]This is a very important property. As shown in the discussion of forward edge pruning, without this property we cannot state that each t' can be extended with e.



ϕ
0:3 <0,1,A,x,B>
14:3 <0,1,A,y,C>
22:4 <0,1,A,y,D>
23:3 <0,1,B,x,C>
25:3 <0,1,C,z,D>
1:3 <0,1,A,x,B> <1,2,B,x,C>
9:3 <0,1,A,x,B> <0,2,A,y,C>
13:3 <0,1,A,x,B> <0,2,A,y,D>
15:3 <0,1,A,y,C> <1,2,C,x,B>
19:2 <0,1,A,y,C> <1,2,C,z,D>
21:3 <0,1,A,y,C> <0,2,A,y,D>
24.2 <0,1,B,x,C> <1,2,C,z,D>
2:3 <0,1,A,x,B> <1,2,B,x,C> <2,0,C,y,A>
6:2 <0,1,A,x,B> <1,2,B,x,C> <2,3,C,z,D>
8:3 <0,1,A,x,B> <1,2,B,x,C> <0,3,A,y,D>
10:2 <0,1,A,x,B> <0,2,A,y,C> <2,3,C,z,D>
12:3 <0,1,A,x,B> <0,2,A,y,C> <0,3,A,y,D>
16:2 <0,1,A,y,C> <1,2,C,x,B> <1,3,C,z,D>
18:3 <0,1,A,y,C> <1,2,C,x,B> <0,3,A,y,D>
20:2 <0,1,A,y,C> <1,2,C,z,D> <0,3,A,y,D>
3:2 <0,1,A,x,B> <1,2,B,x,C> <2,0,C,y,A> <2,3,C,z,D>
5:3 <0,1,A,x,B> <1,2,B,x,C> <2,0,C,y,A> <0,3,A,y,D>
7:2 <0,1,A,x,B> <1,2,B,x,C> <2,3,C,z,D> <0,4,A,y,d>
11:2 <0,1,A,x,B> <0,2,A,y,C> <2,3,C,z,D> <0,4,A,y,d>
17:2 <0,1,A,y,C> <1,2,C,x,B> <1,3,C,z,D> <0,4,A,y,d>
4:2 <0,1,A,x,B> <1,2,B,x,C> <2,0,C,y,A> <2,3,C,z,D> <0,4,A,y,D>

Figure 3: DFS code Tree of Frequent Subgraphs in Figure 1 with $min\_sup = 2$

graph t' of t_p without e can be extended with e to generate a "bigger" connected graph. Because $p'' \sqsubseteq d_0$, at least one instance of p'' resides in d_0, denoted by $t_{p''}$. Obviously, $t_{p''}$ is a supergraph of t_p without e, thus, $t_{p''}$ can be extended with e to generate a bigger connected graph which is graph isomorphic to p'. Consequently, $p' \sqsubseteq d_0$ which contradicts the assumption that $p' \not\sqsubseteq d_0$. Therefore, $f(p') = f(p'')$, namely, $sup(p') = sup(p'')$. $\square$

Based on Lemma 3.3, if e is a backward extensible edge for each instance of p, all descendants of $p \diamond e$ cannot be generators. Moreover, $p \diamond e$ itself is also not a generator because $sup(p) = sup(e|p)$. Therefore, the branch rooted by $p \diamond e$ can be pruned safely. This pruning technique is called ***backward edge pruning***. For instance, the edge $\langle 2, 0, C, y, A\rangle$ is a backward extensible edge for all three instances of node 1 in Figure 3. According to the backward edge pruning, the branch rooted by node 2 will make no contribution to the discovery of graph generators and can be safely pruned. Therefore, many unpromising nodes will not be enumerated and the enumeration process will become more efficient.

Furthermore, if e is an extensible edge for each instance of p, $sup(p) = sup(e|p)$ must hold. Hence, if $sup(p) \neq sup(e|p)$, it is not necessary to apply the backward edge pruning technique. Over relational graph databases in which vertex labels are distinct, this pruning technique will be very efficient because at most one instance exists in each input graph for a subgraph pattern. Therefore, in this case once a backward extensible edge e s.t. $sup(p) = sup(e|p)$ encountered, we can stop growing p with e. Note that the prunable backward edges here correspond to associative edges in *Tail Shrink* proposed in [10]. The difference is that Huan et al. aim at maximal graphs, therefore these associated edges are removed from the tail of the tree and are augmented to this tree without missing any maximal ones.

3.2.2 *Forward Edge Pruning*

In previous discussion, a backward edge-based pruning technique is proposed. Intuitively, we intend to devise forward edge-based pruning techniques. Based on Lemma 3.1, if no edge starts from the end vertex of a forward edge, the removal of this forward edge can preserve the connectivity. Accordingly, for a subgraph pattern $p_n = (a_0, a_1, ..., a_n)$ where a_n is a forward edge, if b_1 in $p' = p_n \diamond (b_1, \cdots, b_m)$ is a forward edge which does not start from the end vertex of a_n, according to the DFS code tree enumeration framework, no valid DFS edge among $b_2, \ldots, b_m$ will connect to the end vertex of a_n. Thus, the removal of a_n from p' will preserve the connectivity of the derived subgraph $p'' = p_{n-1} \diamond (b_1, \cdots, b_m)$. Based on the idea of Lemma 3.3, perhaps we would make an assumption that if a_n is a forward extensible edge for each instance of p_n, $sup(p') = sup(p'')$ holds. Unfortunately, this is not true.

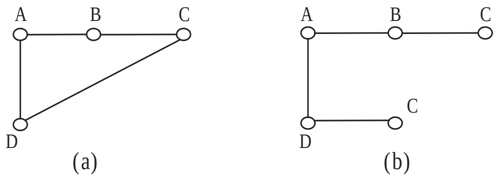


Figure 4: A Failure Case

For example, let $p_n = \langle 0, 1, A, B\rangle\langle 1, 2, B, C\rangle$, $p' = p_n \diamond \langle 0, 3, A, D\rangle\ \langle 3, 4, D, C\rangle$ and $p'' = \langle 0, 1, A, B\rangle\ \langle 0, 2, A, D\rangle\ \langle 2, 3, D, C\rangle$ (edge labels are ignored). Considering a graph database containing two input graphs shown in Figure 4, $\langle 1, 2, B, C\rangle$ is a forward extensible edge for each instance of $\langle 0, 1, A, B\rangle$, and $\langle 0, 3, A, D\rangle$ is a forward extensible edge of p_n which does not start from the end vertex of $\langle 1, 2, B, C\rangle$. In this case, $sup(p') = 1$ and $sup(p'') = 2$, i.e., $sup(p') \neq sup(p'')$. Thus, the assumption is not true. The reason for this failure is that even though $\langle 1, 2, B, C\rangle$ is a forward extensible edge for each instance of $\langle 0, 1, A, B\rangle$, the vertex C in $\langle 1, 2, B, C\rangle$ is contained in the instance of p'' in Figure 4 a). Consequently, although Figure 4 a) contains p'', it does not contain p'. Based on this analysis, a subgraph pattern p_n is called a ***pruning candidate*** if both of the following two constraints are satisfied:

1. p_{n-1} is ϕ or a_n is a forward extensible edge for each instance of p_{n-1};
2. the end vertex of a_n in each instance of p_n will not be contained in any instance of p''.

LEMMA 3.4. *If p_n is a pruning candidate and $e \in E_x(p_n)$ is a forward edge that does not start from the rightmost vertex of p_n, p' and p'' have the same support, i.e. $sup(p') = sup(p'')$.*

PROOF. First, according to Lemma 3.1, p'' is a connected graph. Because $p'' \sqsubset p'$, $f(p'') \supseteq f(p')$ holds. Assume $f(p'') \supset f(p')$ and $d_0 \in f(p'') - f(p')$, then $p'' \sqsubseteq d_0$ and $p' \not\sqsubseteq d_0$. Because p_n is a pruning candidate, a_n is a forward extensible edge for each instance of p_{n-1} and the end vertex of a_n in each instance of p_n will not reside in any instance of p''. Accordingly, for each instance of p'' in d_0, a_n can be attached to this instance to get a "bigger" subgraph pattern which is graph isomorphic to p'. Thus, $p' \sqsubseteq d_0$ which contradicts with the assumption that $p' \not\sqsubseteq d_0$. Consequently, the assumption does not hold and we can know that $f(p') = f(p'')$, namely, $sup(p'') = sup(p')$. □

Based on Lemma 3.4, if constraints are satisfied, $p_n \diamond e$'s descendants including $p_n \diamond e$ itself cannot be graph generators. Thus, we can safely prune the branch rooted by $p_n \diamond e$. We call this pruning technique the ***forward edge pruning***. However, to determine whether the end vertex of a_n will be contained in the instances of p'' or not is still an open problem. Actually, it is quite difficult to predict this. But if the following condition is satisfied, we can state that the end vertex of a_n will never be contained in any instance of p'': ***in each input graph containing p_n, the end vertex of a_n in each instance only connects to vertices that are already contained in this instance***. For example, $\langle 1, 2, C, x, B\rangle$ is a forward extensible edge for each instance of node 14 in Figure 3, the vertex B which is the end vertex of this forward edge only connects to vertices that are already contained in each instance of node 15. Thus, node 15 in Figure 3 is a pruning candidate. Moreover, because $\langle 1, 3, C, z, D\rangle$ is forward extensible edge of node 15 and does not start from the right most vertex of node 15, node 16 can be safely pruned according to the forward edge pruning technique.

The solid lines among different nodes in Figure 3 are valid enumeration paths after applying both backward edge pruning or forward edge pruning techniques, while the dashed lines are invalid. The effectiveness of these two pruning techniques in both pruning the unpromising branches and improving the efficiency of enumeration process will be confirmed in the experimental study.

3.3 Efficient Generator Discovery

In previous discussion, an efficient numeration framework integrated with two effective pruning techniques is introduced to enumerate frequent subgraphs. However, not all remaining frequent subgraphs are graph generators. In this subsection, we address the problem of how to elicit graph generators efficiently.

3.3.1 Generator Checking Scheme

As stated in Section 3.2, graph generators can only be discovered from generator candidates. To elicit graph generators, we maintain a set of selected generator candidates during the enumeration process, denoted by R, in which elements are grouped according to their supports. While a generator candidate p is encountered, we are trying to insert it into R. According to the DFS code tree enumeration framework, all proper-subgraphs of p that are enumerated before p must reside in the path from the root of the search tree to p. Thus, if one proper-subgraph of p in this path has the support $sup(p)$, p cannot be a generator candidate; if p is a generator candidate, all proper-subgraphs of p in this path have supports greater than $sup(p)$. Therefore, while inserting a generator candidate p into R, no proper-subgraph of p with support $sup(p)$ will exist in R. Whereas, some proper-supergraphs of p with support $sup(p)$ might exist in R, and these patterns should be removed from R since they are not graph generators. For example, node 6 in Figure 3 is a generator candidate, but it is not a graph generator because node 24 is a proper-subgraph of it with the same support.

Consequently, only one operation should be performed while inserting a generator candidate into R, i.e., removing elements in R s.t. they are proper-supergraphs of p and possess the support of $sup(p)$. Because elements in R are grouped by their supports, proper-supergraph checking only needs to performed in the group with the support of $sup(p)$. In general, however, the problem of subgraph isomorphism is NP-Complete. Thus, the proper-supergraph checking is a very time-consuming operation. Accordingly, we devise an effective underlying index structure ADI++, which is an enhancement of ADI structure introduced in [22], to facilitate the subgraph isomorphism checking problem.

3.3.2 The ADI++ Structure

Chen Wang et al.[22] analyzed previous algorithms on frequent subgraph mining, such as FSG[12], gSpan[27] and CloseGraph[26], and pointed out that random access to elements in graph databases and checking the graph isomorphism are frequent and expensive operations in these algorithms. Accordingly, ADI structure, which supports these frequent and expensive operations very well, was devised to facilitate the scalable mining of frequent subgraphs. In this paper, we introduce a novel index structure, ADI++, which is an enhancement of ADI. Besides preserving all advantages of ADI, ADI++ can be exploited to assist in solving the (sub)graph isomorphism checking problem efficiently.

Similar to ADI structure, ADI++ structure is a three-level index for edges, graph-ids and adjacency information. An example is shown in Figure 5, where two graphs G_1 and G_2 in Figure 1 are indexed. Different with ADI structure, all edges which are indexed by their labels in the edge table are ordered and numbered. The index of each edge in a graph is unique and can be used to identify an edge uniquely. Except for this little difference, the edge table and graph-ids list in ADI++ structure are identical to that in ADI structure. However, we exploit a different structure to store the adjacency information of edges. Each input graph corresponds to an edge block, and edges in a graph are stored as a vector of edges in an edge block, but not a linked list.

First, vertices and edges in a single graph are ordered separately. The order used here can be any one user preferred. In this paper, we order vertices and edges according to the order in which they are scanned from this graph. Note that, vertex index starts with 0, while edge index starts with 1. The reason for this difference is because a special edge

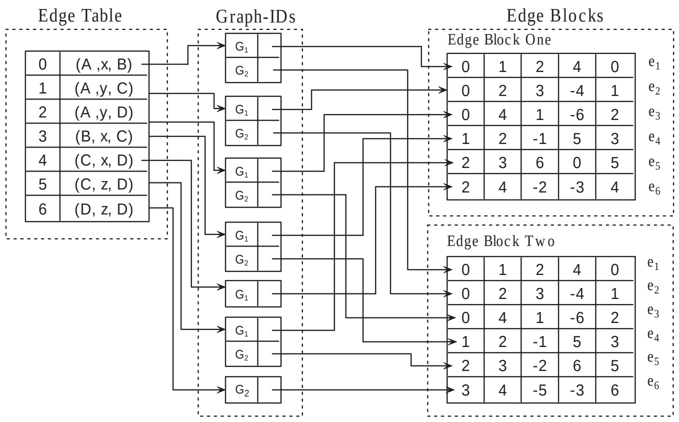


Figure 5: The ADI++ Structure for G_1 and G_2 in Figure 1

with index 0 is introduced to indicate an empty edge. For instance, the numbers near each vertex of G_1 and G_2 in Figure 1 is the order in which the vertices are scanned. Second, each edge in edge blocks is represented by a 5-tuple $\langle s_n, e_n, s_l, e_l, i\rangle$, where s_n is the start vertex index, e_n is the end vertex index and $s_n < e_n$, i is the index of this edge in the edge table, s_l is the index of next edge which contains the vertex s_n. If no such a next edge exists, s_l will be the index of the first edge in this edge block which contains the vertex s_n; if no other edge contains this vertex s_n, s_l will be 0. Moreover, if the start vertex of the edge which contains vertex s_n is s_n, s_l will be positive; otherwise s_l will be negative. The same method can be used to calculate the value of e_l. The only difference is that e_l focuses on the vertex e_n. For example, taking into account the edge $\langle A, y, C\rangle$ connecting vertices 0 and 2 in G_1, then $s_n = 0$ and $s_e = 2$. The index of ⟨A,y,C⟩ in the edge table is 1, then $i = 1$. Because the third edge $e_3 = \langle 0,4,1,-6,2\rangle$ in the first block of Figure 5 is the first next edge containing the vertex 0 as its start vertex, thus $s_l = 3$. Meanwhile, the first next edge containing vertex 2 is the forth edge $e_4 = \langle 1,2,-1,5,3\rangle$ of which vertex 2 is its end vertex, thus $e_l{=}-4$. Therefore, the tuple for edge e_2 is ⟨0,2,3,-4,1⟩. The construction of ADI++ structure is trivial and users can implement it easily. Hereby, the construction of ADI++ structure will not be described here due to the page limitation.

Comparing with ADI, in terms of storage, only one more cell[3] is needed in ADI++ than ADI for each edge in edge blocks. In terms of usage, besides preserving all advantages of ADI, ADI++ can provide more functions. First, in ADI++, once given an edge index in an edge block, the vertex labels and edge label of this edge can be easily identified due to the introduction of i. But this operation cannot be accomplished in ADI. Second, edges sharing the common vertex are "linked" as a cycle in ADI++. Thus, we can enumerated all the edges sharing the common vertex through any edge in this cycle. But in ADI, this can be accomplished only by providing the first edge in the linked list. Third, edges are represented by 5-tuples consisting of 5 integers, it is easier and more convenient either to store on the disk or to be held in main memory than using linked list. At last, ADI++ can be used to assist in solving (sub)graph isomorphism checking problem which will be illustrated in the following.

[3]Note here a cell may correspond to one space unit for storing an integer data type or a long integer type depending on the concrete implementation

3.3.3 Solving Subgraph Isomorphism using ADI++

Although minimum DFS code can uniquely represent a graph and can be used to determine graph isomorphism efficiently, it cannot be applied to determine subgraph isomorphism. For example, node 24 in Figure 3 is a proper-subgraph of node 4, but their minimum DFS codes are quite different and we cannot determine the subgraph isomorphism relationship between them by their minimum DFS codes. In general, subgraph isomorphism problem is NP-Complete. It will take an unacceptable time to determine subgraph isomorphism relationship between two graph. Fortunately, ADI++ structure can facilitate this problem.

For any subgraph pattern, one of its instance in an input graph corresponds to a set of edges in its corresponding edge block. Since all edges in an edge block are numbered with unique numbers, each instance can be represented by a set of edge indexes, i.e., a set of integers. And then, we can get the following two lemmas.

LEMMA 3.5. *Given a graph pattern G, let $\mathcal{T}(G)$ be the set of instances of G in the input graph database, $\mathcal{E}(t)$ be the set of edge indices of the instance t, and $\mathcal{I}(t)$ be the graph index in which t resides. Another graph pattern G' is subgraph isomorphic to G iff $\forall t \in \mathcal{T}(G)$ and $\exists t' \in \mathcal{T}(G')$ s.t. $\mathcal{E}(t') \subseteq \mathcal{E}(t)$ and $\mathcal{I}(t) = \mathcal{I}(t')$.*

LEMMA 3.6. *For two graph patterns G and G', if $\exists t \in \mathcal{T}(G)$ and $\exists t' \in \mathcal{T}(G')$ s.t. $\mathcal{E}(t') \subseteq \mathcal{E}(t)$ and $\mathcal{I}(t) = \mathcal{I}(t')$, G' is subgraph isomorphic to G.*

The above two lemmas are straightforward and their proofs are simple, so we omit them here. Nevertheless, these two simple lemmas provide a significant help to subgraph isomorphism checking. Assume $s \in R$, t_s is an instance of s and p is a new generator candidate which will be inserted into R. On the one hand, if $p \sqsubset s$, according to Lemma 3.5, there must exist $t \in \mathcal{T}(p)$ s.t. $\mathcal{E}(t) \subset \mathcal{E}(t_s)$ and $\mathcal{I}(t) = \mathcal{I}(t_s)$; otherwise, p is not a proper-subgraph of s. On the other hand, if there exists $t \in \mathcal{T}(p)$ s.t. $\mathcal{E}(t) \subset \mathcal{E}(t_s)$ and $\mathcal{I}(t) = \mathcal{I}(t_s)$, p must be a proper-subgraph of s according to Lemma 3.6. From the above analysis we can know that for each element in R, one instance's information(edge indices and graph index in which it resides) is adequate to determine whether this element is a proper-supergraph of a new generator candidate or not. Therefore, maintaining one instance's information for each element in R makes subgraph isomorphism checking easier and more efficient. Moreover, this maintenance does not consume too much space because the number of elements in R would be not too large and for each element only $m + 1$ integers need to be maintained where m is the edge number of the corresponding element.

Further more, *non-minimum pruning*, i.e. $p \neq min(p)$, is a very significant pruning technique in gSpan[27] which avoids many redundant parts of search space. However, computing minimum DFS code for a graph is quite expensive. With the help of ADI++, this pruning technique would become more efficient. According to DFS code tree enumeration framework, if $p \neq min(p)$, the graph $min(p)$ must be enumerated before p. Essentially, the function of $p \neq min(p)$

is to determine whether p is graph isomorphic to a graph enumerated before. Because graph isomorphism is a special case of subgraph isomorphism, ADI++ structure also can be used to improve the efficiency of non-minimum pruning technique in gSpan. Due to the page limitation, it will not be discussed further.

3.4 The FOGGER Algorithm

By integrating the DFS code tree enumeration framework, two pruning techniques and the generator checking scheme introduced in previous subsections, we present the first solution, FOGGER, for mining the complete set of frequent graph generators from graph databases.

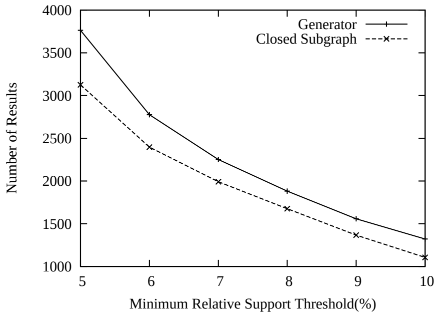
Before running FOGGER shown in ALGORITHM 1, we first compute the set of frequent edges from the input graph database, and remove the infrequent ones. This procedure can reduce the size of input graphs significantly when min_sup is high. After this preprocess, we use FOGGER to mine the complete set of frequent graph generators. For the current subgraph pattern p, we first compute the extensible edge set $E_x(p)$ and their conditional supports(line 1). According to the Apriori property of frequent subgraph patterns, the edges in $E_x(g)$ whose conditional supports are less than min_sup can be removed (line 2). After then, the backward edge pruning technique can be applied to delete edges in $E_x(p)$ that are extesible edges for each instance of p (lines 3-4). Furthermore, if p is a pruning candidate, the forward edge pruning technique can be applied to remove the forward edge e in $E_x(p)$ which does not start from the rightmost vertex of p(lines 6-7). For each edge e remained in $E_x(p)$, if $p \diamond e$ is not a minimum DFS code, $p \diamond e$ can be ignored according to the nonminimum-pruning technique(lines 10-11). If e's conditional support is less than $sup(p)$, we insert the pattern $p \diamond e$ to the generator candidate set rs and remove the proper-supergraph patterns of $p \diamond e$ with support $sup(p \diamond e)$ in rs(lines 13-14). After this, we invoke the FOGGER(D, $p \diamond e$, min_sup, rs) recursively to continue the enumeration(line 16). After the enumeration finished, the elements remained in rs is the complete set of frequent graph generators in the input graph database.

Algorithm 1 FOGGER(D, p, min_sup, rs)

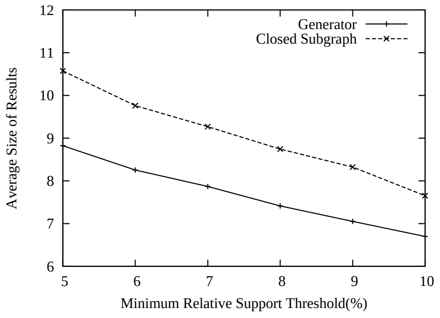
```
Input: $D$-the input graph database; $p$-the subgraph pattern being inspected, $min\_sup$-the minimum support threshold; $rs$-the set of generator candidate
 1. Compute $E_x(p)$ and their conditional supports;
 2. Remove edges in $E_x(p)$ s.t. $sup(e|p) < min\_sup$;
 3. if $e \in E_x(p)$ is a backward edge for each instance of $p$ then
 4.     remove $e$ from $E_x(p)$;
 5. end if
 6. if $p$ is a pruning candidate and $e \in E_x(p)$ is a forward edge
    which does not start from the rightmost vertex of $p$ then
 7.     remove $e$ from $E_x(p)$;
 8. end if
 9. for each edge $e$ remained in $E_x(p)$ do
10.     if $p \diamond e \neq min(p \diamond e)$ then
11.         continue;
12.     end if
13.     if $sup(e|p) < sup(p)$ then
14.         insert $p \diamond e$ into $rs$ according to the generator checking
            scheme;
15.     end if
16.     invoke FOGGER($D, p \diamond e, min\_sup, rs$);
17. end for
```

4. EXPERIMENTAL RESULTS

We conducted a comprehensive performance study to evaluate various aspects of algorithm FOGGER. All algorithms were implemented in C++ using STL, and all experiments were performed on a PC running Fedora 8 Linux and with an AMD Sempron 1.8GHz CPU and 1G MB of main memory installed.



(a) # patterns



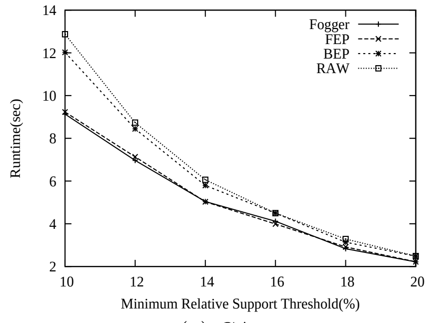
(b) Average pattern size

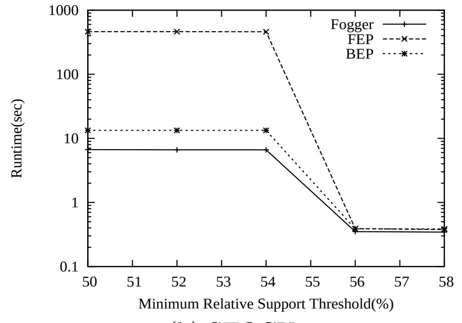
Figure 6: Graph Generators vs. Closed Subgraphs(CA)

In the experiments, we used a variety of real and synthetic datasets to evaluate the FOGGER algorithm. The first two real datasets, CA and CM, were derived from the AIDS antivirus screen compound database from the DTP in NCI/NIH which has been widely used in many previous studies [26, 24, 9]. CA contains 422 confirmed active chemical compounds, while CM contains 1081 confirmed moderately active chemical compounds. The third real dataset is "0.96-STOCK" which was generated from the stocks' daily price according to the correlation coefficient of 0.96. More descriptions about the STOCK dataset can be found in [24]. Meanwhile, the synthetic datasets were generated by a package provided by Kuramochi and Karypis. The parameters in generating synthetic datasets are the same as those in [13], and we set $D = 10k$, $T = 40$, $i = 10$, $L = 200$, $E = 200$, and $V = 200$ as the default values if not explicitly stated. Thus, a dataset generated by the default values contains $10k$ graphs, the average graph size is 40, the average size of frequent subgraphs is 10, and the number of potential frequent subgraphs, the number of edge labels, and the number vertex labels are 200, 200, and 200, respectively.

4.1 Graph Generators vs. Closed Subgraphs

To our best knowledge, FOGGER is the first graph generator mining algorithm, thus, we cannot find a competitive


(a) CA


(b) STOCK

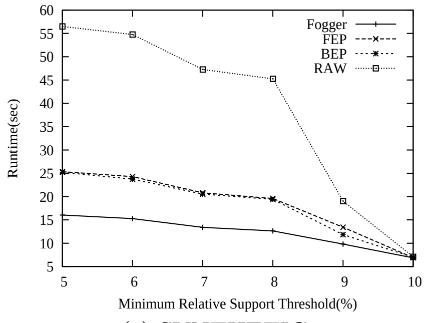
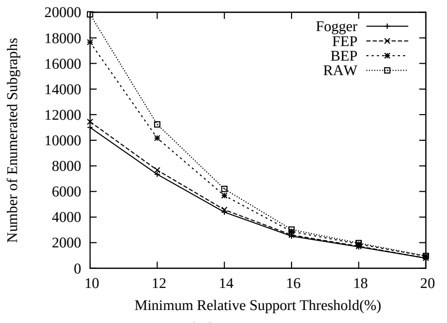
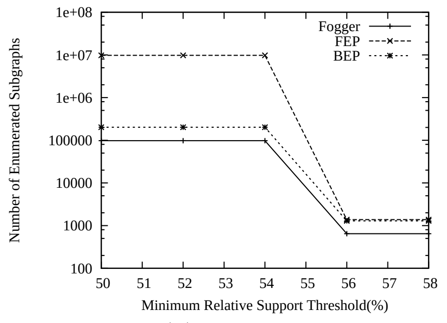

(c) SYNTHETIC

Figure 7: Effectiveness Test of Pruning Techniques (Runtime)


(a) CA


(b) STOCK

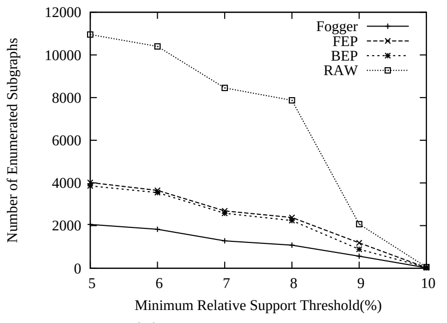

(c) SYNTHETIC

Figure 8: Effectiveness Test of Pruning Tech. (# enumeration)

algorithm which performs the same task in order for comparison with FOGGER. However, as the set of graph generators and closed subgraphs are two representative types of subgraph patterns which can form equivalence classes, it will be interesting if we compare their average size and number of patterns[4]. In the experiments, we used the latest version of CloseGraph package [26] for mining closed subgraphs. Since the current implementation of CloseGraph only accepts input graphs and generates patterns with a maximum size of 254 edges and 254 vertices, we conducted comparison experiments on real dataset CA only. Figure 6 shows their comparison results. From Figure 6(a) we see that the number of graph generators is slightly greater than that of closed subgraphs, while from Figure 6(b), we see the average size of graph generators is smaller than the average size of closed subgraphs.

4.2 Evaluation of Pruning Techniques

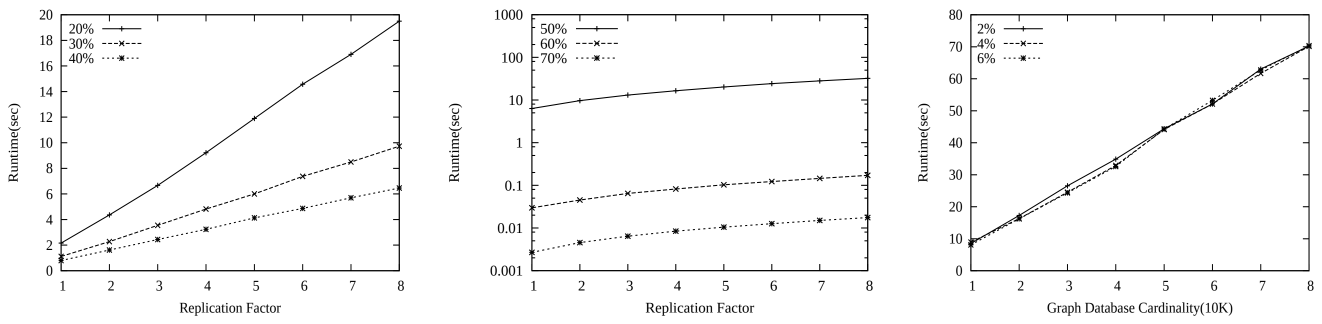
We then evaluated the effectiveness of the pruning techniques and the efficiency of FOGGER on various datasets. Figures 7 and 8 compare the runtime and the number of enumerated subgraphs under conditions with different combinations of pruning techniques on CA, STOCK and the synthetic datasets. For instance, the caption "BEP" denotes a variant of FOGGER which applies *backward edge pruning* only, "FEP" denotes a variant of FOGGER which applies *forward edge pruning* only, and "Raw" denotes a variant of FOGGER without applying any pruning techniques. From Figure 7, we see that these two pruning techniques are very effective in improving the efficiency, and from Figure 8 we can observe that both of these two pruning techniques are also effective in pruning the unpromising parts of search space. Note that because Raw takes a rather long time to finish on dataset STOCK, the experimental results of Raw for dataset STOCK are not shown in the corresponding figures.

Furthermore, for the CA dataset we observe that the forward pruning technique plays a significant role both in pruning the unpromising parts of search space and improving the efficiency. While for the STOCK dataset, the backward pruning performs well. This difference is caused by the characteristics of the input datasets. We examined the results obtained from the different datasets. Most of the generators discovered from CA dataset are tree structures which do not contain backward edges; while for the STOCK dataset, most of the results contain cycles, namely, many backward edges exist in the discovered generators. Moreover, we can observe the fact that the lower the minimum support threshold, the more effective these two pruning techniques, which validates the effectiveness of the newly proposed pruning techniques.

4.3 Scalability of FOGGER

Meanwhile, an extensive scalability study was also conducted in terms of the base size on both real and synthetic datasets. We first replicated the CA and STOCK datasets

[4]As shown in Figure 6, graph generator mining and closed subgraph mining output a quite different result set, thus, their efficiency is not comparable. However, our performance study shows that FOGGER and CloseGraph have comparable performance. Due to limited space, we omit it from the paper.



(a) CA (b) STOCK (c) SYNTHETIC

Figure 9: Scalability Test

from 1 to 8 times and ran FOGGER with three different minimum relative supports. As shown in Figure 9(a) and Figure 9(b), it is evident that FOGGER shows linear scalability in runtime against the number of input graphs for the STOCK and CA datasets. By varing the base size from $10k$ to $80k$, a series of synthetic datasets were produced. We get the runtime of FOGGER with three different minimum relative supports of 2%, 4% and 6% as shown in Figure 9(c). Similarly, FOGGER shows good scalability in runtime against the base size of synthetic datasets. Therefore, FOGGER has a good scalability in terms of the base size.

4.4 Evaluation of Graph Generator-based Classification Model

One typical application of FOGGER algorithm is to select a subset of high quality graph generators discovered by FOGGER and use them as features to build classification models. As the graph generators of the same equivalence class have the same support and confidence, they have a similar power from the classification point of view, we only randomly retained one generator for each equivalence class. For each selected graph generator we first computed its confidence w.r.t. each class label, used the highest one as its confidence and ranked the graph generators in their confidence descending order, and for the graph generators with the same confidence, we further ranked them in support descending order. We then applied the sequential covering paradigm to select a set of high confidence graph generators. This procedure of feature selection is the same as the one used in association classifier CBA. Finally we built two classification models based on the selected graph generators. The first one is the associative classification rule based CBA-like classifier. The second one is the SVM. For SVM, we adopted the LIBSVM implementation (see http://www.csie.ntu.tw/~cjlin/libsvm). In the experiments, we used the RBF kernel function (i.e., $e^{(-\gamma * |u-v|^2)}$), and other parameter values were set as follows: svm_type (type of SVM) is ν-SVC, ν (the parameter ν of ν-SVC) equals 0.1, γ (i.e., γ in kernel function) equals 0.5, ϵ (i.e., the tolerance of termination criterion) equals 0.0001. We applied the same procedure to choose a set of closed subgraphs as features to feed CBA and SVM and build another two classifiers. We merged the chemical compound datasets CA and CM into one, which contains 422 input graphs labeled as CA and 1081 input graphs labeled as CM. We evaluated various classifiers based on 10-fold cross validation.

Table 2 shows the classification accuracy comparison of graph generator based classifiers and closed subgraph-based classifiers. In the experiments, we varied the minimum support from 30% to 10% and ran FOGGER and CloseGraph on the training dataset and used the method described above to find a set of graph generators and a set of closed subgraphs, which can be further used to build CBA and SVM classifiers respectively. The experimental results in Table 2 show that the graph generator based classifiers have almost the same accuracy as the corresponding closed subgraph based classifiers. Since for each equivalence class of patterns, we only randomly retained one pattern, thus, we get the same number of graph generators as that of closed subgraphs which are used for building classification model. Table 1 compares average generator size (i.e., average number of edges) with average closed subgraph size. We see that the average size of graph generators is much smaller than that of closed subgraphs. As graph classification testing involves subgraph isomorphism checking, the smaller the graph patterns, the faster the subgraph isomorphism checking, thus graph generator based models is more efficient than closed subgraph based models in terms of classification testing. This depicts an advantage of graph generator mining over closed subgraph mining from the graph classification point of view.

5. RELATED WORK

Frequent subgraph mining has been widely studied and many efficient algorithms have been proposed [23, 32, 21, 18, 20, 30, 9]. AGM[11], FSG[12] and gSpan[27] are three typical algorithms for mining the complete set of frequent subgraphs. Due to the exponential number of frequent subgraphs, attention has been paid to closed subgraphs[26, 29, 24, 31] and maximal subgraphs[10, 19, 17] to reduce the number of results. Yan et al. [26] first proposed an efficient algorithm, CloseGraph, to mine frequent closed subgraph patterns which not only dramatically reduces the unnecessary subgraphs to be generated but also substantially improves the efficiency of mining process in the presence of large graph patterns. Subsequently, efficient algorithms(e.g., CloseCut and Splat [29], CLAN[24], Cocian[31]), are proposed for mining closed subgraphs which satisfy some specific connectivity constraints such as cliques and quasi-cliques. Meanwhile, efficient algorithms(e.g., SPIN[10] and Margin[19]) are also introduced to mine maximal frequent subgraphs. However, to our best knowledge, no attention has been paid to the discovery of frequent graph generators.

Table 1: Average size comparison: graph generators vs. closed subgraphs (i.e., # edges)

min_sup	# graph generators	Avg. graph generator size	# closed subgraphs	Avg. closed subgraph size
30%	55	2.74	55	5.36
25%	63	3.11	63	5.79
20%	80	3.47	80	6.11
15%	95	4.07	95	7.12
10%	113	4.64	113	9.33
Avg.	81.2	3.606	81.2	6.742

Table 2: Classification accuracy comparison of graph generator based classifiers and closed subgraph-based classifiers.

min_sup	generator based CBA	closed graph based CBA	generator based SVM	closed graph based SVM
30%	81.1	81.2	81.7	82.2
25%	83.2	83.3	83.4	83.8
20%	85.5	85.7	86.4	86.8
15%	86.7	86.6	87.2	87.3
10%	82.8	82.1	83.5	83.7
Avg.	83.86	83.78	84.44	84.76

While in another community of itemset mining, itemset generator has been widely studied[2, 1, 14]. Jinyan Li et al. [14] has proposed an efficient algorithm Gr-growth to mine frequent itemset generators. Unfortunately, the Apriori property no longer holds for graph generators. Therefore, effective pruning techniques devised for frequent itemset generator mining based on this nice property cannot be applied to graph generator mining, which makes graph generator mining a quite challenging task. So far, we are not aware of any previous work on mining graph generators.

6. CONCLUSION AND DISCUSSION

In this paper, we have introduced the problem of mining frequent connected graph generators from graph databases. We presented a novel algorithm, FOGGER, which efficiently mines frequent connected graph generators. Two pruning techniques, *backward edge pruning* and *forward edge pruning*, are used in the algorithm to reduce the search space and to improve the computational efficiency. In addition, an underlying index structure, ADI++, is proposed to accelerate the subgraph isomorphism checking operation. We used a variety of real and synthetic datasets to study the performance of the FOGGER algorithm. Experimental study confirms that FOGGER is efficient and scalable in terms of the base size of input databases. The results in classifying chemical compounds demonstrate the advantages of graph generator-based models over closed subgraph-based models from the graph classification point of view.

Currently, FOGGER can only mine connected graph patterns but not disconnected. In future, we plan to devise disk-based algorithms for mining disconnected graph patterns. In addition, we plan to explore the graph generator-based graph classification problem in wide applications.

Acknowledgement

The authors are grateful to Dr. Michihiro Kuramochi, Prof. George Karypis, Dr. Xifeng Yan and Prof. Jiawei Han for providing us the graph dataset generator, the AIDS databases, and the CloseGraph package.

7. REFERENCES

[1] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 972–986, London, UK, 2000. Springer-Verlag.

[2] J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Approximation of frequency queris by means of free-sets. In *PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 75–85, London, UK, 2000. Springer-Verlag.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872, Beijing, China, 2007. ACM.

[4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. Translator-C. Franzke.

[5] Q. Gao, M. Li, and P. Vitányi. Applying mdl to learn best model granularity. *Artificial Intelligence*, 121(1-2):1–29, 2000.

[6] P. D. Grünwald, I. J. Myung, and M. A. Pitt. *Advances in Minimum Description Length: Theory and Applications (Neural Information Processing)*. The MIT Press, 2005.

[7] P. Grünwald. A tutorial introduction to the minimum description length principle. *The Computing Research Repository*, math.ST/0406077, 2004.

[8] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(1):213–221, 2005.

[9] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM '03: Proceedings of the Third IEEE*

International Conference on Data Mining, page 549, Washington, DC, USA, 2003. IEEE Computer Society.

[10] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586, Seattle, WA, USA, 2004. ACM.

[11] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23, London, UK, 2000. Springer-Verlag.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[13] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph*. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[14] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI '06, Proceedings of the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Application of Artificial Intelligence*. AAAI Press, 2006.

[15] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[16] J. Rissanen. Modelling by the shortest data description. *Automatica*, 14:465–471, 1978.

[17] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu. Mining maximal quasi-bicliques to co-cluster stocks and financial ratios for value investment. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 1059–1063, Washington, DC, USA, 2006. IEEE Computer Society.

[18] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 687–696, San Jose, California, USA, 2007. ACM.

[19] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 1097–1101, Washington, DC, USA, 2006. IEEE Computer Society.

[20] R. M. H. Ting and J. Bailey. Mining minimal contrast subgraph patterns. In *SDM'06: Proceedings of the Sixth SIAM International Conference on Data Mining*, Bethesda, MD, USA, 2006. SIAM.

[21] H. Tong, C. Faloutsos, and Y. Koren. Fast direction-aware proximity for graph mining. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 747–756, San Jose, California, USA, 2007. ACM.

[22] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 581–586, Seattle, WA, USA, 2004. ACM.

[23] J. Wang, W. Hsu, M. L. Lee, and C. Sheng. A partition-based approach to graph mining. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 74, Atlanta, USA, 2006. IEEE Computer Society.

[24] J. Wang, Z. Zeng, and L. Zhou. Clan:an algorithm for mining closed cliques from large dense graph databases. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 73, Atlanta, USA, 2006. IEEE Computer Society.

[25] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE '07, IEEE 23rd International Conference on Data Engineering*, pages 976–985, Istanbul, Turkey, 2007.

[26] X. Yan and J. H. and. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, Washington, D.C., 2003. ACM.

[27] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 721, Washington, DC, USA, 2002. IEEE Computer Society.

[28] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346, Paris, France, 2004. ACM.

[29] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 324–333, Chicago, Illinois, USA, 2005. ACM.

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

[31] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–802, Philadelphia, PA, USA, 2006. ACM.

[32] S. Zhang, J. Yang, and V. Cheedella. Monkey: Approximate graph mining based on spanning trees. In *ICDE '07, IEEE 23rd International Conference on Data Engineering*, pages 1247–1249, Istanbul, Turkey, 2007.

[33] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 938–949, Vienna, Austria, 2007. VLDB Endowment.