CloseGraph: Mining Closed Frequent Graph Patterns[*]

Xifeng Yan
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
xyan@uiuc.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
hanj@uiuc.edu

ABSTRACT

Recent research on pattern discovery has progressed from mining frequent *itemsets* and *sequences* to mining structured patterns including *trees, lattices*, and *graphs.* As a general data structure, *graph* can model complicated relations among data with wide applications in bioinformatics, Web exploration, and etc. However, mining large graph patterns is challenging due to the presence of an exponential number of frequent subgraphs. Instead of mining all the subgraphs, we propose to mine *closed frequent graph patterns.* A graph g is *closed* in a database if there exists no proper supergraph of g that has the same support as g. A closed graph pattern mining algorithm, CloseGraph, is developed by exploring several interesting pruning methods. Our performance study shows that CloseGraph not only dramatically reduces unnecessary subgraphs to be generated but also substantially increases the efficiency of mining, especially in the presence of large graph patterns.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining

General Terms

Algorithms, Performance, Experimentation

Keywords

frequent graph, graph representation, closed pattern, canonical label

1. INTRODUCTION

Frequent graph pattern mining raised great interest in data mining community recently. Many scientific and commercial applications require the discovery of structural patterns in large data sets, which go beyond *sets*, *sequences*, and *trees* towards *lattices, graphs*, and *other complicated structures.* Certainly, as a general data structure, graph can meet the demands of modelling complicated relations among data. We have witnessed several algorithms developed for this challenging problem [10, 11, 16, 15, 3].

Frequent graph pattern mining shares a common problem which also exists in frequent itemset and sequence mining. Based on the Apriori principle [1], a frequent n-edge labeled graph may contain 2^n frequent subgraphs. This raises a serious problem for mining complete subpatterns: A graph pattern with 64 edges, which is not unusual in molecule structures and chemical compounds, may generate 2^{64} frequent subgraphs, although most subgraphs actually deliver nothing interesting but redundant information if all of them share the same support. For example, among 422 chemical compounds that are confirmed to be active in an AIDS antiviral screen dataset provided by Developmental Theroapeutics Program in NCI/NIH, there are nearly 1,000,000 frequent graph patterns if the minimum support is 5%. This makes the further analysis on frequent graphs nearly impossible.

In the context of itemset and sequence mining, an interesting solution, called mining *closed frequent itemsets and sequences* [13], has been proposed to overcome this difficulty [14, 4, 18, 17]. A frequent pattern I is *closed* if there exists no proper super-pattern of I with the same support in the dataset. The counterpart alternative in graph mining, *mining closed frequent graphs*, may lead to a promising direction as well. For example, for the dataset mentioned above, among these 1 million frequent graphs, only about 2,000 are closed frequent graphs. If further analysis (such as classification or clustering) is performed on the set of closed graphs instead of on the whole set of frequent graphs, it will achieve the same accuracy with less redundancy and better efficiency. However, until recently, this crucial alternative has not been touched. This is, based on our analysis, partly because of the complexity in graphs.

The goal of this paper is to develop an efficient method for mining closed graph patterns. Previous studies in closed itemset mining [14, 4, 18] and subsequence mining [17] have demonstrated that such mining may lead to more efficient methods than mining all patterns by developing some sophisticated techniques to prune the pattern search space. However, some elegant properties of itemsets and sequences do not hold in graphs. For example, our recently discovered property about equivalence of projected database in

[*]The work was supported in part by U.S. National Science Foundation NSF IIS-02-09199, Univ. of Illinois, and an IBM Faculty Award.

sequential pattern mining [17] can hardly be applied in the graph mining if we do not use a huge amount of extra storage and computation. Thus, it requires exploring different approaches to accommodate the specific properties in graph patterns.

There are two general approaches in efficient graph pattern mining. The first, represented by [10, 11, 15], extends the Apriori-based candidate generation-and-test approach [1] to graph pattern mining. The detailed algorithms distinguish themselves in using different building blocks: vertices in [10], edges in [11], and edge-disjoint paths in [15]. The second approach, represented by [16, 3], adopt a pattern-growth philosophy [8] by growing patterns from a single graph directly. It adopts space-efficient depth-first search. Such a method avoids two costly operations: (1) joining two k-edge frequent graphs to generate $(k+1)$-edge graph candidates, and (2) checking the frequency of these candidates separately. These two operations usually are the performance bottlenecks of the Apriori-like algorithms.

In [16], we demonstrated that gSpan, a new graph mining algorithm based on the pattern-growth approach, outperforms FSG[11], the reported best algorithm in the category of the Apriori-based, candidate generation-and-test approach. However, it is still challenging to develop an efficient closed graph mining algorithm based on gSpan since gSpan builds a very strict order in graph patterns. In this paper, we present CloseGraph (Closed Graph pattern mining), the first algorithm for mining closed frequent graph patterns, and show that CloseGraph is highly efficient. It outperforms gSpan by a factor of 4 to 10 when the frequent graphs are large (for example, graphs with more than 32 edges). The success of the method is based on the development of the novel concepts of *equivalent occurrence* and *early termination* that help CloseGraph prune the search space substantially with small additional cost. We also illustrate some cases where early termination may fail and miss some patterns. By detecting and eliminating these cases, we guarantee the completeness and soundness of the closed graph patterns discovered by CloseGraph.

The remaining of the paper is organized as follows. Section 2 introduces the concept of closed frequent graph pattern mining and the notations to be used throughout the paper. A search framework is illustrated in Section 3. In Section 4, the methods for testing equivalent occurrence, early termination, as well as failure detection are presented. Section 5 formulates the algorithm of CloseGraph. We report our performance result in Section 6, discuss the extension of our method and related work in Section 7, and conclude our study in Section 8.

2. PRELIMINARY CONCEPTS

As a general data structure, labeled graph can be used to model complicated patterns among data such as relationship patterns. A *labeled graph* has labels associated with its edges and vertices. We denote the vertex set of a graph g by $V(g)$, the edge set by $E(g)$. A label function, l, can map a vertex or an edge to a label. A graph g is a subgraph of another graph g' if there exists a subgraph isomorphism from g to g'.

DEFINITION 1 (SUBGRAPH ISOMORPHISM). *A subgraph isomorphism is an injective function $f: V(g) \to V(g')$, such that (1)$\forall u \in V(g)$, $l(u) = l'(f(u))$, and (2)$\forall (u,v) \in E(g)$, $(f(u), f(v)) \in E(g')$ and $l(u,v) = l'(f(u), f(v))$, where l and l' are the label function of g and g' respectively.*

If g is a *subgraph* of g', then g' is a *supergraph* of g, denoted by $g \subseteq g'$ (*proper supergraph*, if $g \subset g'$). Given a labeled graph dataset, $D = \{G_1, G_2, \ldots, G_n\}$, $support(g)$ (or $frequency(g)$) denotes the percentage (or number) of graphs (in D) in which g is a subgraph. The set of **frequent graph patterns**, FS, includes all the graphs whose support is no less than a minimum support threshold, *min_sup*. The set of **closed frequent graph patterns**, CS, is defined as follows, $CS = \{g | g \in FS \text{ and } \nexists g' \in FS \text{ such that } g \subset g' \text{ and } support(g) = support(g')\}$. Since CS includes no graph that has a proper supergraph with the same support, we have $CS \subseteq FS$. The problem of **closed frequent graph mining** is to find the complete set of CS in the graph dataset D with a given *min_sup*.

Since most of interesting graph patterns are connected graphs, we first study mining labeled connected undirected graphs without multiple edges. Extensions of our method for mining other kinds of graph structures such as unlabeled graphs, graphs with self-loops and multiple edges, directed graphs, disconnected graphs, and so on, will be examined in the discussion section.

EXAMPLE 1. *Figure 1 is a sample labeled graph dataset, D, where three labeled graphs are presented. Among these graphs, each vertex and edge are assigned a label. Let min_sup be 2. The alphabetic order is taken as the default lexicographical order.*

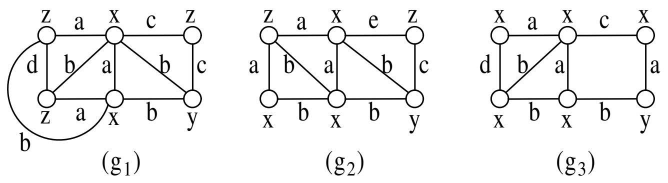


Figure 1: A Sample Graph Dataset D

A graph g can be extended by adding a new edge e. Let the new graph denoted by $g \diamond_x e$. Edge e may or may not introduce a new vertex to g. If e introduces a new vertex, we denote the new graph by $g \diamond_{xf} e$, otherwise, $g \diamond_{xb} e$. Algorithm 1 illustrates a naive frequent graph mining algorithm. It finds all the frequent graphs, closed or non-closed. For each discovered graph g, it performs the extension recursively until all the frequent graphs with g embedded are discovered. Line 4 shows the termination condition: When the support of a graph is less than *min_sup*, it is unnecessary to extend it any more.

NaiveGraph is simple, but not efficient. The key issue is the inefficiency of extending g to $g \diamond_x e$: The same graph can be extended in different ways. For an n-edge graph, it may have n different ways to be formed from $(n-1)$-edge graphs if we do not consider isomorphism. We call a graph that is discovered again a *duplicate graph.* Line 1 in Algorithm 1 gets rid of duplicate graphs. The number of duplicate graphs may be huge. It raises some severe problems. First, the generation and support computation of duplicate graphs waste time. Second, it is nontrivial to tell whether a graph is a duplicate. Generally, we have to

Algorithm 1 NaiveGraph(g, D, min_sup, S)

```
Input: A graph $g$, a graph dataset $D$, and $min\_sup$.
Output: The frequent graph set $S$.

1: if $g$ exists in $S$ then return;
2: else insert $g$ to $S$;
3: scan $D$ once, find every edge $e$ such that
     $g$ can be extended to $g \diamond_x e$ and it is frequent;
4: for each frequent $g \diamond_x e$ do
5:     Call NaiveGraph($g \diamond_x e$, $D$, $min\_sup$, $S$);
6: return;
```

compute its canonical label and check whether it was discovered before. Third, should we extend g if we find g is a duplicate? If there exists at least one graph that can grow only from this duplicate graph, we still need to extend it. As we can see, these three interleaved problems affect the efficiency of the algorithm. They suggest that g should be extended as conservatively as possible in order to reduce the generation of duplicate graphs. To satisfy this requirement, we developed gSpan [16] where an efficient canonical labeling system and a lexicographic ordering in graphs are built. gSpan has the following salient properties: (1) it reduces the generation of duplicate graphs; (2) it does not need to search previous discovered frequent graphs in order to detect duplicates; and (3) it never extends any duplicate graph but still guarantees the completeness. CloseGraph is built on gSpan and formulates the early termination conditions to make gSpan "return" as early as possible in closed graph mining. In the following two sections, we focus on two major techniques used in CloseGraph: (1) right-most extension and DFS (depth-first search) lexicographic order for frequent graph generation; and (2) equivalent occurrence and early termination for non-closed graph pruning.

3. LEXICOGRAPHIC ORDERING

This section introduces several techniques developed to represent and extend graphs efficiently. It includes mapping a graph to a DFS code (a sequence), building a lexicographic ordering among these codes, and mining DFS codes based on this lexicographic order.

3.1 DFS Subscripting

When performing a depth-first search [5] in a graph, a corresponding DFS tree can be constructed. Figure 2(a) is a frequent subgraph for the sample dataset D in Figure 1. Figure 2(a)-(d) are the same graph. The darkened edges in Figure 2(b)-(d) construct three different DFS trees for this graph. When building a DFS tree, the depth-first discovery of the vertices forms a linear order. We use the magnitude of subscripts to illustrate this order according to their discovery time [5]. $i < j$ means v_i is discovered before v_j. We denote G subscripted with a DFS tree T by G_T. T is named a *DFS subscripting* of G.

Given G_T, we call the first node traversed in G_T, v_0, the *root*, and the last node traversed, v_n, the *right-most vertex.* The straight path from v_0 to v_n is named the *right-most path.* In Figure 2(b)-(d), three different subscriptings are generated. The right-most path is (v_0, v_1, v_3) in Figure 2 (b) and (c), and (v_0, v_1, v_2, v_3) in Figure 2(d).

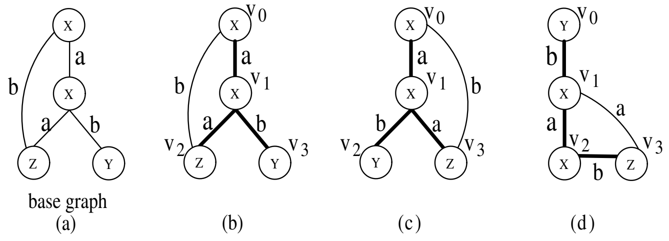


Figure 2: DFS subscripting

Given G_T, the *forward edge* (*tree edge* [5]) set contains all the edges in the DFS tree, denoted by E_T^f, and the *backward edge* (*back edge* [5]) set contains all the edges which are not in the DFS tree, denoted by E_T^b. For example, the darkened edges in Figure 2(b)-(d) are forward edges while the undarkened ones are backward edges. From now on, (v_i, v_j) (simply written as (i, j)) is viewed as an ordered pair to represent an edge. If $(v_i, v_j) \in E(G)$ and $i < j$, it is a forward edge; otherwise, a backward edge. The *forward edge of* v_i means there exists a forward edge (i, j) and $i < j$. The *backward edge of* v_i means there exists a backward edge (i, j) and $i > j$. In Figure 2(b), $(1, 3)$ is the forward edge of v_1, but not of v_3, and $(2, 0)$ is the backward edge of v_2.

3.2 Right-Most Extension

In Algorithm 1, NaiveGraph requires extending g in any possible position, which will result in a huge number of duplicate graphs. We would like to show that there is a more clever way to extend graphs. gSpan restricts the extension as follows: Given g and a DFS tree T in g, e can be extended from the right-most vertex connecting to any other vertices on the right-most path (*backward extension*); or e can be extended from vertices on the right-most path and introduce a new vertex (*forward extension*). We call these two kinds of restricted extension as *right-most extension*, denoted by $g \diamond_r e$ (for simplicity, we omit T here). This restricted extension is different from $g \diamond_x e$ described in NaiveGraph.

EXAMPLE 2. *If we want to extend the graph in Figure 2(b), the backward extension candidates can be* (v_3, v_0)*. The forward extension candidates can be edges extending from* v_3*,* v_1*, or* v_0 *with a new vertex introduced.*

Since we may have different DFS subscriptings for the same graph, we want to select one from them as *base subscripting* and conduct right-most extension on the base subscripting. Otherwise, right-most extension cannot reduce the generation of duplicate graphs because there are many different subscriptings to extend from the same graph.

3.3 DFS Code

For each subscripted graph, we can map it into an edge sequence. We build an order among these sequences and select the subscripting that generates the minimum sequence as its base subscripting. There are two kinds of orders in this process: (1) edge order, which maps edges in a subscripted graph into a sequence; and (2) sequence order, which builds the order among sequences. We introduce edge order in this subsection and sequence order in the next subsection.

Intuitively, the DFS tree has defined the discovery order of forward edges. For the graph shown in Figure 2(b), the forward edges are discovered in the order $(0,1),(1,2),(1,3)$. Now we can insert backward edges into the order. Given a vertex v, all of its backward edges should appear just before its forward edges. If v does not have any forward edge, we put its backward edges just after the forward edge where v is the second vertex. For vertex v_2 in Figure 2(b), its backward edge $(2,0)$ should appear just after (1,2) since v_2 does not have any forward edge. Among the backward edges from the same vertex, we can enforce an order: Given v_i and its two backward edges, $(i,j_1),(i,j_2)$, if $j_1 < j_2$, then edge (i,j_1) will appear before edge (i,j_2). So far, we complete the ordering of the edges in a graph. Based on this order, we can translate a graph into a sequence. A complete sequence for Figure 2(b) is $(0,1),(1,2),(2,0),(1,3)$.

Formally we can define a linear order, $\prec_T$, in R^2 if we only consider the subscripts of edges. $e_1 \prec_T e_2$ holds if one of the following statements is true (assume $e_1 = (i_1,j_1), e_2 = (i_2,j_2)$):

(i) $e_1, e_2 \in E_T^f$, and $j_1 < j_2$ or $i_1 > i_2 \wedge j_1 = j_2$.
(ii)$e_1, e_2 \in E_T^b$, and $i_1 < i_2$ or $i_1 = i_2 \wedge j_1 < j_2$.
(iii) $e_1 \in E_T^b$, $e_2 \in E_T^f$, and $i_1 < j_2$.
(iv) $e_1, \in E_T^f$, $e_2 \in E_T^b$, and $j_1 \leqslant i_2$.

EXAMPLE 3. *For simplicity, we represent an edge by a 5-tuple, $(i,j,l_i,l_{(i,j)},l_j)$, where l_i and l_j are the labels of v_i and v_j respectively and $l_{(i,j)}$ is the label of the edge between them. For example, (v_0, v_1) in Figure 2(b) is represented by $(0,1,X,a,X)$. Table 1 shows the edge order for the DFS subscriptings in Figure 2(b)-(d).*

edge	γ_0	γ_1	γ_2
e_0	$(0,1,X,a,X)$	$(0,1,X,a,X)$	$(0,1,Y,b,X)$
e_1	$(1,2,X,a,Z)$	$(1,2,X,b,Y)$	$(1,2,X,a,X)$
e_2	$(2,0,Z,b,X)$	$(1,3,X,a,Z)$	$(2,3,X,b,Z)$
e_3	$(1,3,X,b,Y)$	$(3,0,Z,b,X)$	$(3,1,Z,a,X)$

Table 1: DFS code for Figure 2(b)-(d)

DEFINITION 2 (DFS CODE). *Given G_T, an edge sequence (e_i) can be constructed based on $\prec_T$, such that $e_i \prec_T e_{i+1}$, where $i = 0, \ldots, |E|-1$. (e_i) is a DFS code, denoted by $code(G,T)$.*

Table 1 shows three different DFS codes generated by DFS subscriptings in Figure 2(b)-(d). As one can see, actually we build a one-to-one mapping between a subscripted graph and a DFS code. When the context is clear, we treat a subscripted graph and its DFS code as the same. All the notations on subscripted graphs can also be applied to DFS codes. The graph represented by a DFS code α is written as g_α.

3.4 DFS Lexicographic Order

We want to build an order among the DFS codes generated for a graph so that we can define a minimum DFS code for this graph. Since we are dealing with labeled graphs, the label information should be considered as one of the ordering factors, which can be used to break a tie when two edges have the same subscript, but different labels. We let $\prec_T$ take the first priority, the vertex label l_i take the second priority, the edge label $l_{(i,j)}$ take the third, and the vertex label l_j take the fourth to determine the order of two edges. For example, the first edges for the three DFS codes shown in Table 1 are $(0,1,X,a,X)$, $(0,1,X,a,X)$, and $(0,1,Y,b,X)$ respectively. All of them share the same $(0,1)$ subscript. So $\prec_T$ cannot tell the difference among them. But using label information, following the order of first vertex label, edge label, and second vertex label, we have $(0,1,X,a,X) < (0,1,Y,b,X)$. Based on this order, given DFS codes $\alpha = (a_0, a_1, ..., a_m)$ and $\beta = (b_0, b_1, ..., b_n)$, if $a_0 = b_0, ..., a_{t-1} = b_{t-1}$ and $a_t < b_t$ $(t \leqslant min(m,n))$, then we say $\alpha < \beta$. According to this order definition, we have $\gamma_0 < \gamma_1 < \gamma_2$ for the DFS codes listed in Table 1.

The above discussion builds an order on the DFS codes of the same graph. We can extend this order definition in the DFS codes of different graphs. This ordering is one of our key contributions in gSpan. The formal definition of DFS code order is given as follows.

DEFINITION 3 (DFS LEXICOGRAPHIC ORDER). *Suppose $Z = \{code(G,T) \mid T$ is a DFS subscripting of $G\}$, i.e., Z is a set containing all DFS codes of all connected labeled graphs. Suppose there is a linear order ($\prec_L$) in the label set (L), then the lexicographic combination of $\prec_T$ and $\prec_L$ is a linear order ($<$) in $R^2 \times L \times L \times L$ (the space of $(i,j,l_i,l_{(i,j)},l_j)$). **DFS Lexicographic Order** is a linear order defined as follows. If $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n), \alpha, \beta \in Z$, then $\alpha \leqslant \beta$ iff either of the following is true.*

(i) $\exists t, 0 \leqslant t \leqslant min(m,n), a_k = b_k$ *for* $k < t, a_t < b_t$
(ii) $a_k = b_k$ *for* $0 \leqslant k \leqslant m$, *and* $m \leqslant n$.

EXAMPLE 4. *Assume we have a 2-edge graph which has a DFS code $((0,1,X,a,X)(1,2,X,b,X))$. This graph is different from the graph in Figure 2(a). Using the DFS lexicographic order, we can compare $((0,1,X,a,X)(1,2,X,b,X))$ with any code in Table 1. It is greater than γ_0, but smaller than γ_1.*

DEFINITION 4 (MINIMUM DFS CODE). *Given G, $Z(G) = \{code(G,T) \mid T$ is a DFS subscripting of G $\}$. Based on DFS lexicographic order, $min(Z(G))$ is called **Minimum DFS Code** of G. If $code(G,T_0) = min(Z(G))$, we call T_0 the base subscripting of G.*

Code γ_0 in Table 1 is the minimum DFS code of the graph in Figure 2(a). We use $min(\alpha)$ to denote the minimum DFS code of the graph represented by code α. Minimum DFS code can be considered as canonical label.

So far, we defined DFS code, minimum DFS code, and base subscripting. For every graph, we only conduct the right-most extension on its base subscripting and ignore other possible subscriptings. From now on, *the right-most extension of G specifically means the right-most extension on the base subscripting of G.* Can this extension method guarantee the completeness of the mining result? The answer is "*yes*". We first have the following result.

THEOREM 1 (COMPLETENESS). *Performing right-most extension in* NaiveGraph *guarantees the completeness of mining result.*

When performing the right-most extension in NaiveGraph, it is possible that α is the minimum (i.e., representing a base subscripting), but $\alpha \diamond_r e$ is not. In this case, should we conduct the right-most extension on this non-minimum DFS code (i.e., it is not a base subscripting)? The answer is "*no*".

LEMMA 1. *Performing only the right-most extension on the minimum DFS codes in* NaiveGraph *guarantees the completeness of the mining result.*

We achieved these two major results in gSpan. The detailed proof and implementation are available in [16]. Algorithm 2 outlines the framework. The difference between gSpan and NaiveGraph is the right-most extension and terminating extension on non-minimum DFS codes (Algorithm 2 lines 1-2). We replace the existence judgement in Algorithm 1 Line 1-2 with the inequation $s \neq min(s)$. Actually, $s \neq min(s)$ is more efficient to calculate. Line 5 requires exhaustive enumeration of s in D. This exhaustive enumeration will be utilized by CloseGraph.

Figure 3 shows the search space of gSpan, where each link represents a possible right-most extension. The right-most extension takes place when we extend $(k-1)$-edge graphs to the k-edge ones. If we find two DFS codes s and s' represent the same graph and $s < s'$, by Lemma 1, we can completely stop searching any descendant of s'. gSpan can generate graphs strictly in DFS lexicographic order: graphs with smaller minimum DFS codes will be discovered first.

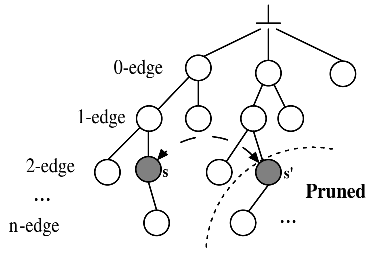


Figure 3: Search Space

Algorithm 2 gSpan(s, D, min_sup, S)

```
Input: A DFS code $s$, a graph dataset $D$, and $min\_sup$.
Output: The frequent graph set $S$.

1: if $s \neq min(s)$, then
2:     return;
3: insert $s$ into $S$;
4: set $C$ to $\varnothing$;
5: scan $D$ once, find every edge $e$ such that
     $s$ can be right-most extended to frequent $s \diamond_r e$;
     insert $s \diamond_r e$ into $C$;
6: sort $C$ in DFS lexicographic order;
7: for each $s \diamond_r e$ in $C$ do
8:     Call gSpan($s \diamond_r e$, $D$, $min\_sup$, $S$);
9: return;
```

Although gSpan can achieve competitive performance compared with other algorithms, it cannot help much when the graph size is large. The exponential growth on the number of frequent subgraphs makes the mining impossible. Certainly, it is inefficient to find the complete frequent graph set first, and then filter it to obtain the closed ones. We need to find an efficient method to mine the closed frequent graphs directly.

4. EQUIVALENT OCCURRENCE

In CloseGraph, we want to find certain condition that if it is satisfied, then for some s in Figure 3, all of its descendant supergraphs will not be closed, and need not to be checked. For such case, we can completely prune the search branch of s. That is to say, in Algorithm 2 line 1, besides the original condition of $s \neq min(s)$, we probably can add another condition that lets the recursive function return as "early" as possible. In this section, we discuss a possible condition that we can utilize in CloseGraph. Later we will show that the efficient implementation of this condition largely relies on gSpan's pattern-growth model and its exhaustive enumeration method.

Let $\varphi(g, g')$ represent the number of possible subgraph isomorphisms of g in g', i.e., the number of different injective functions existing in Definition 1.

DEFINITION 5 (OCCURRENCE). *Given g and $D = \{G_1, G_2, \ldots, G_n\}$, the occurrence of g in D is the sum of the number of subgraph isomorphisms of g in every graph of D, i.e. $\sum_{i=1}^{n} \varphi(g, G_i)$, denoted by $\mathcal{I}(g, D)$.*

EXAMPLE 5. *For the graph g in Figure 2(a) and g_1 in Figure 1(1), $\varphi(g, g_1) = 2$. As to g_2 in Figure 1(2), $\varphi(g, g_2) = 1$. For the sample dataset in Figure 1, $\mathcal{I}(g, D) = 3$.*

Suppose $g' = g \diamond_x e$, f is a subgraph isomorphism of g in G, and f' is a subgraph isomorphism of g' in G. If $\exists \rho$, ρ is a subgraph isomorphism of g in g', $\forall v, f(v) = f'(\rho(v))$, then we call f *extendable* and f' an *extended subgraph isomorphism* from f. Intuitively, if we have f, we can extend f to f' since we extend g to g' by adding one edge. We denote the number of such extendable f by $\phi(g, g', G)$.

DEFINITION 6 (EXTENDED OCCURRENCE). *Given $g' = g \diamond_x e$ and $D = \{G_1, G_2, \ldots, G_n\}$, the extended occurrence of g' in D w.r.t g is the sum of the number of extendable subgraph isomorphisms of g (w.r.t g') in every graph among D, i.e. $\sum_{i=1}^{n} \phi(g, g', G_i)$, denoted by $\mathcal{L}(g, g', D)$.*

Equivalent Occurrence. *Given $g' = g \diamond_x e$ and D, if $\mathcal{I}(g, D) = \mathcal{L}(g, g', D)$, we say that g and g' have the* ***Equivalent Occurrence****, which means wherever g occurs in D, g' occurs. Can we conclude that g'' is not closed if $g \subset g''$ and $g' \not\subseteq g''$?*

If it is true, actually we need only extend g', instead of g. We call this **Early Termination** since we terminate searching the supergraphs of g except g'. Since $\varphi(g, G_i) \geqslant \phi(g, g', G_i)$, we have $\mathcal{I}(g, D) \geqslant \mathcal{L}(g, g', D)$. When they are equal, it can be concluded that wherever g occurs in G_i, g' must also occur exactly in the same place. Thus g must be accompanied with e in any place where g occurs. Therefore, some supergraph g'' of g without the edge e will not be closed since we have $support(g'') = support(g'' \diamond_x e)$. Here is an example.

EXAMPLE 6. *For the graph g_1 in Figure 4(1), the occurrence of g_1 in the sample dataset is $\mathcal{I}(g_1, D) = 2 + 1 + 0 = 3$. For the graph g_2 in Figure 4(2), its extended occurrence w.r.t. g_1 is $\mathcal{L}(g_1, g_2, D) = 2 + 1 + 0 = 3$, which is equal to*

$\mathcal{I}(g_1, D)$. That means $x \overset{a}{—} x$ always accompanies g_1 between the vertices labeled with "x". Thus, even without computing the support of the graph in Figure 4(3), we can conclude that its support must be equal to the support of Figure 4(4). Thus it is not closed. A further conclusion is that we need not grow from Figure 4(1), but from Figure 4(2).

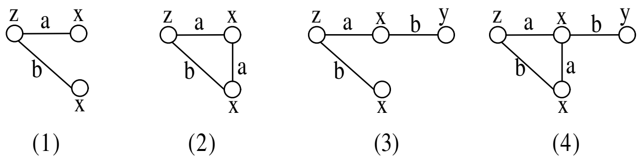


Figure 4: Equivalent Occurrence

4.1 Failure of Early Termination

Unfortunately, Early Termination does not hold for every supergraph of g. There are several cases where Early Termination cannot be applied. One of them is illustrated below. We develop techniques to check these cases in order to make Early Termination work correctly.

CASE 1. *Suppose a dataset consists of two graphs shown in Figure 5(1) and 5(2). We want to find closed graphs whose minimum support is 2. Let g be $x \overset{a}{—} y$ and g' be $x \overset{a}{—} y \overset{b}{—} x$. As we can see, $y \overset{b}{—} x$ always occurs with $x \overset{a}{—} y$. Can we terminate extending g since $\mathcal{I}(g, D) = \mathcal{L}(g, g', D)$ in this case? We cannot. Otherwise, we will miss the pattern shown in Figure 5(3). This graph cannot be extended from $x \overset{a}{—} y \overset{b}{—} x$. If we add edge $y \overset{b}{—} x$ into Figure 5(3), the new graph will not be frequent any more. That means we cannot only extend g' in this case. We say there is a failure of Early Termination in g', $y \overset{b}{—} x$ is the failure point.*

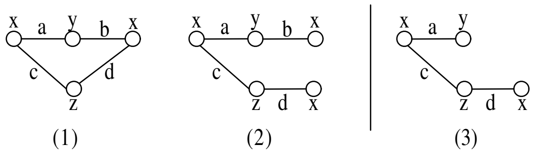


Figure 5: Failure of Early Termination

Case 1 shows a situation where Early Termination does not work. If we can eliminate this case, Early Termination still works in the remaining cases. The following theorem sets the condition where Early Termination can be applied.

THEOREM 2 (EARLY TERMINATION). *Given $g' = g \diamond_x e$ and D, $\mathcal{I}(g, D) = \mathcal{L}(g, g', D)$, if $\forall h$, $g \subset h$, $g' \not\subset h$, $\mathcal{I}(h, D) = \mathcal{L}(h, h \diamond_{xf} e^1, D)$ or $\mathcal{I}(h, D) = \mathcal{L}(h, h \diamond_{xb} e, D)$ is true, then it is unnecessary to extend g except g'.*

Proof.(draft) If $\mathcal{I}(h, D) = \mathcal{L}(h, h \diamond_{xf} e, D)$ or $\mathcal{I}(h, D) = \mathcal{L}(h, h \diamond_{xb} e, D)$, then we can conclude that one of the following must be true: (1) $\forall G \in D$, if $h \subset G$, then $h \diamond_{xf} e \subset G$; or (2) $\forall G \in D$, if $h \subset G$, then $h \diamond_{xb} e \subset G$. Therefore, we have either $support(h) = support(h \diamond_{xf} e)$ or $support(h) = support(h \diamond_{xb} e)$. In either way, h is not closed. Thus it is unnecessary to extend g except g'. ■

[1]Since h contains g, but not g', e is the edge which can be added into g in h to construct g' in h. Therefore, $h \diamond_{xf} e$ and $h \diamond_{xb} e$ will contain g'.

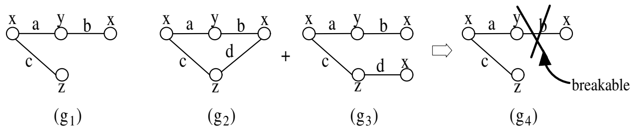


Figure 6: Detect Failure of Early Termination

4.2 Detecting Failure of Early Termination

Now the key problem is how to detect whether there exists h which fails the condition of Theorem 2. Figure 6 shows our approach to detect the occurrence of Case 1. It works in a passive way, which means we do not actively look for h. Instead, at the beginning we assume there is no violation case of Early Termination and just commit to it: Any frequent graph with $x \overset{a}{—} y$ embedded must contain $x \overset{a}{—} y \overset{b}{—} x$ for the dataset in Figure 5(1) and 5(2). Then we extend $x \overset{a}{—} y \overset{b}{—} x$ to g_1 shown in Figure 6. At this stage everything is fine. We continue extending g_1 to g_2 and g_1 to g_3. We find g_2 and g_3 are not frequent (their support is less than 2). However, we find the extensions from g_1 to g_2 and g_1 to g_3 add a common edge $z \overset{d}{—} x$, though g_3 introduces a new vertex and g_2 does not. It gives us a hint: If we decouple edge $x \overset{a}{—} y$ and $y \overset{b}{—} x$, we probably can grow a graph like Figure 5(3). Actually we can break the edge $y \overset{b}{—} x$ (we say $y \overset{b}{—} x$ *breakable*) and then introduce a forward edge $z \overset{d}{—} x$, which invalidates our commitment previously made that $x \overset{a}{—} y$ and $y \overset{b}{—} x$ must be together. We need to grow $x \overset{a}{—} y$ separately. It helps us find a graph which is ignored if we just apply Early Termination. There are other similar failure cases which can be handled with care. The detection of these failure cases of Early Termination can guarantee the completeness of mining result.

Furthermore, since we only have interests in frequent graphs, we can apply an additional requirement for the case shown in Figure 6: The sum of *support*(g_2) and *support*(g_3) must be no less than *min_sup*. Otherwise, it is impossible that the support of the graph in Figure 5(3) will be greater than or equal to *min_sup*.

5. CLOSEGRAPH: DESIGN AND IMPLEMENTATION

In previous sections, we discussed two techniques: Rightmost extension for frequent graph mining and Early Termination for closed graph mining. We put them together in CloseGraph. In this section, we illustrate the major problems we meet in formulating CloseGraph.

When we extend g by adding one more edge in gSpan, we enumerate each occurrence of g in the graph dataset. Thus, we only need to maintain a counter to calculate $\mathcal{I}(g, D)$. When we enumerate g, we search for the occurrence of $g' = g \diamond_x e$ and update the support of g'. We add another counter to calculate the extended occurrence of g', $\mathcal{L}(g, g', D)$. Thus, both $\mathcal{I}(g, D)$ and $\mathcal{L}(g, g', D)$ can be obtained nearly for free, which is largely contributed by the pattern-growth model and exhaustive enumeration method adopted by gSpan. In the first glance, it seems trivial to apply Early Termination

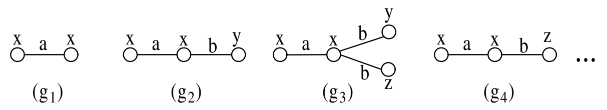


Figure 7: Pattern Generation Order

condition in gSpan. Actually, it is not.

The advantage of gSpan is its right-most extension approach in graph pattern growth. It reduces the generation of duplicate graphs. As we know, gSpan performs depth-first search and generates the DFS codes in DFS lexicographic order. gSpan avoids extending g in all the possible ways. It only conducts the right-most extension on the vertices in the right-most path. That is, g is extended to $g \diamond_r e$ instead of $g \diamond_x e$. If there exist some supergraphs g' that cannot be right-most extended from g in gSpan, we will lose the trace of $\mathcal{L}(g, g', D)$ for these supergraphs, which, to some extent, will miss some opportunities to find the situations where Early Termination can be applied.

CloseGraph proposes a trade-off between NaiveGraph and gSpan. For each graph g, we check all the possible frequent g', where $g' = g \diamond_x e$, to obtain $\mathcal{L}(g, g', D)$. However, we only conduct the right-most extension of g. Certainly, we must pay special attention to detect the failure cases of Early Termination in CloseGraph.

Even if we implement CloseGraph in this way, we cannot fully take advantage of Early Termination since we generate the graphs in DFS lexicographic order. Figure 7 shows an example. Suppose g_1, g_2, g_3, and g_4 are generated in that order and $\mathcal{I}(g_1, D) = \mathcal{L}(g_1, g_4, D)$. If we stop right-most extending g_1 and only extend g_4, we may lose g_3 (g_2 is not closed) and some of its supergraphs forever. For example, g_3 cannot be generated from g_4 [2]. Thus, we need to extend g_1 to g_2, and so on. Therefore, Early Termination cannot be applied in this case if we strictly follow DFS lexicographic order. However, in cases where $g' = g \diamond_x e$, $\mathcal{I}(g, D) = \mathcal{L}(g, g', D)$, and $g > g'$, we can use Early Termination.

Besides using Early Termination to remove some non-closed frequent graphs, we also need to get rid of the remaining non-closed frequent graphs. One approach is whenever we discover a frequent graph, we compare it with the previously discovered graphs to see whether it is closed. In CloseGraph, the following lemma can help doing this in a clever way.

LEMMA 2. *Given graphs g and g', if $g \subset g'$ and $support(g) = support(g')$, then $\exists h$, $h = g \diamond_x e$ and $h \subseteq g'$, s.t. $support(g) = support(h)$.*

Lemma 2 shows that if we want to check whether a graph is closed, we only need to check the support of its supergraphs that have one more edge. We mentioned before that CloseGraph checks all the possible frequent g', where $g' = g \diamond_x e$, to obtain $\mathcal{L}(g, g', D)$. Therefore, $support(g')$ can be obtained as a side-product when we calculate $\mathcal{L}(g, g', D)$. If there exists a graph g' such that $support(g) = support(g')$, g is not closed. Otherwise, it is closed. Thus, CloseGraph need not record any previously discovered graphs to prune newly discovered non-closed graphs. A closed frequent graph is output whenever it is found. This method is different from CloSpan that we developed in [17], where a closed sequential pattern candidate set need to be stored in order to do post-pruning.

[2] g_3 can be right-most extended from g_4, but it will be ignored since the DFS code of g_3 generated in this way is not the minimum DFS code of g_3.

CloseGraph works in three major steps recursively: (1) it generates a frequent graph; (2) it applies Lemma 2 to check whether this graph is closed; and (3) it checks the condition of Early Termination and its failure cases to determine whether the graph should be extended.

Algorithm 3 CloseMining(D, min_sup, S)

```
Input: A graph dataset $D$ and $min\_sup$.
Output: The closed frequent graph set $S$.

1: remove infrequent vertices and edges;
2: $S^0 \leftarrow$ code of frequent graphs with single vertex;
3: $S \leftarrow S^0$;
4: for each code $s \in S^0$ do
5:     CloseGraph($s$, $NULL$, $D$, $min\_sup$, $S$);
```

Algorithm 3 illustrates the framework which includes the necessary preprocessing step. It removes infrequent vertices and edges. Then it calls CloseGraph recursively by doing depth-first search and right-most extension. The search order of closed graphs is consistent with the DFS lexicographic order.

Algorithm 4 CloseGraph(s, p, D, min_sup, S)

```
Input: A DFS code $s$, its parent $p$,
       a graph dataset $D$, and $min\_sup$.
Output: The closed frequent graph set $S$.

1: if $s \neq min(s)$, then
2:     return;
3: if $\exists e', g' = g_p \diamond_x e'$ and $g' < g_s$ and $\mathcal{I}(g_p, D) = \mathcal{L}(g_p, g', D)$
     and $g_p$ is not a failure case of early termination then
4:     return;
5: set $C$ to $\varnothing$;
6: scan $D$ once, find every edge $e$ such that
     $s$ can be extended to frequent $s \diamond_x e$;
    insert $s \diamond_x e$ into $C$;
7: detect any possible failure of early termination in $s$;
8: if $\nexists s \diamond_x e \in C$, $support(s) = support(s \diamond_x e)$ then
9:     insert $s$ into $S$;
10: remove $s \diamond_x e$ from $C$ which cannot be
     right-most extended from $s$;
11: sort $C$ in DFS lexicographic order;
12: for each $s \diamond_r e$ in $C$ do
13:     Call CloseGraph($s \diamond_r e$, $s$, $D$, $min\_sup$, $S$);
14: return;
```

Algorithm 4 outlines the pseudo code of CloseGraph. The framework of CloseGraph is similar to gSpan. However, it performs a major improvement using the search space pruning techniques developed above. That is, before conducting the right-most extension on a discovered graph, CloseGraph first checks whether there exists Early Termination in Algorithm 4 line 3-4, which is a variance of the condition we previously mentioned. If the condition is satisfied and g' does not cause the failure of Early Termination, it is unnecessary to continue extension of g_s since all its possible descendants must not be closed.

6. EXPERIMENTS AND PERFORMANCE STUDY

A comprehensive performance study was conducted in our experiments on both real and synthetic datasets. The real data set we tested is an AIDS antiviral screen chemical compound dataset [3]. For the latter, we use a synthetic data generator provided by Kuramochi and Karypis [11].

All the experiments are done on a 1.7GHZ Intel Pentium-4 PC with 1GB main memory, running RedHat 7.3. Both gSpan and CloseGraph are implemented in C++ with STL library support and compiled by g++ with -O3 optimization. CloseGraph shares the exact same routines and source codes of the right-most extension, subgraph isomorphism and minimum DFS code calculation with gSpan. Thus, the performance curve mainly reflects the effectiveness of Early Termination in CloseGraph. The mining result of CloseGraph is cross-checked with gSpan. In each experiment, we also show the performance of an Apriori-like algorithm, FSG [11].

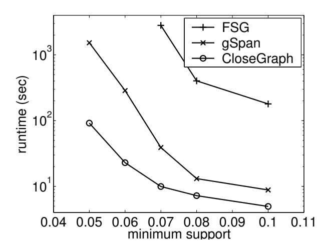
We are interested in discovery of frequent structures. Additional mining operations can be constructed on these frequent structures: For example, finding contrast structures which are frequent in one class but infrequent in another, or using the frequent structures as features to label unclassified structures.

The AIDS antiviral screen compound dataset from Developmental Theroapeutics Program in NCI/NIH is available publicly. We select the most up-to-date release, March 2002 Release. The dataset contains 43,905 chemical compounds. The results of the screening tests can be categorized into three classes: **CA**: confirmed active; **CM**: confirmed moderately active; and **CI**: confirmed inactive. Among these 43,905 compounds, 422 of them belong to **CA**, 1081 are of **CM**, and the remaining are in class **CI**.

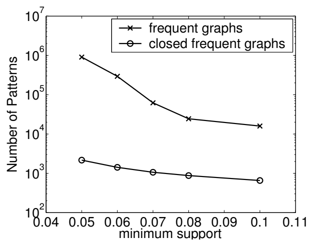
We want to discover the frequent structures in the class **CA** and **CM** compounds. We remove all the hydrogens in these compounds. The most popular atoms in these two datasets are C, O, N, S, etc. There are 21 kinds of atoms in class **CA** compounds whereas 25 in class **CM**. Three kinds of bonds are popular in these compounds: single-bond, double-bond, and aromatic-bond. On average, each class **CA** compound has 40 vertices and 42 edges. The maximum one has 188 vertices and 196 edges. Each class **CM** compound has 32 vertices and 34 edges on average. The maximum one has 221 vertices and 234 edges.

Figure 8(a) shows the runtime with *min_sup* varying from 10% to 5% on the class CA compound dataset. The number of frequent graph patterns and closed frequent graph patterns are shown in Figure 8(b). As we can see, CloseGraph outperforms gSpan by a factor of 10 when *min_sup* is close to 5%. Both of them are better than FSG. Meanwhile, CloseGraph generates fewer patterns than gSpan. The ratio between frequent graphs and closed ones is close to 100 : 1. It demonstrates that closed pattern mining can deliver more compact mining results. We also tested the performance of WARMR [6], an ILP program. WARMR can not complete the task in 2 hours for 10% minimum support. It takes

[3]http://dtp.nci.nih.gov/docs/aids/aids_data.html.



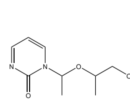
(a) performance

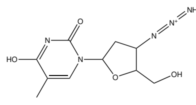


(b) number of patterns

Figure 8: Mining Patterns in class CA Compounds

208 seconds and 6400 seconds for 30% and 20% minimum support. In both cases, it takes *gSpan* less than 2 seconds.


(a) 20%


(b) 10%

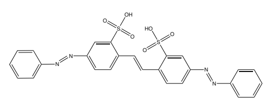
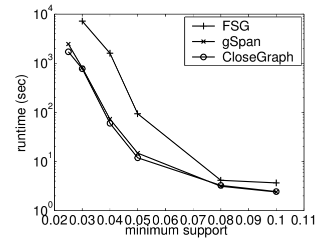

(c) 5%

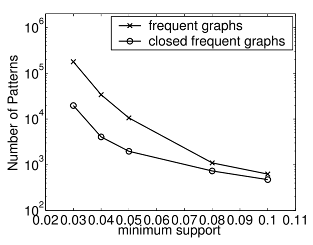
Figure 9: Discovered Patterns in class CA

Figure 9 shows the largest graph patterns we discovered in three different minimum support thresholds: 20% in Figure 9(a) (14 edges), 10% in Figure 9(b) (21 edges), and 5% in Figure 9(c) (42 edges).

Next, we conduct experiments on class **CM** compounds. The performance and the number of discovered patterns are shown in Figure 10. The ratio between frequent graphs and closed ones is around 10 : 1. The largest pattern with 5% support has 23 edges. That means the compounds in class **CA** share larger and more common chemical fragments. The compounds in class **CM** are more diverse and do not converge on some specific structures. It explains why CloseGraph does not achieve a similar speedup in this dataset. However, gSpan needs extra cost to compute closed frequent graphs, which is not shown in the figure. For example, with 2.5% support, gSpan takes 5675 seconds to compute the closed graphs while CloseGraph only takes 1713 seconds.



(a) performance



(b) number of patterns

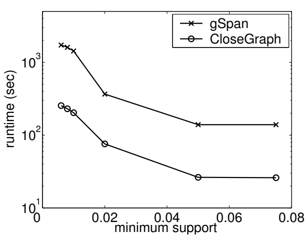
Figure 10: Mining Patterns in class CM Compounds

We then tested CloseGraph on a series of synthetic graph datasets. The synthetic graph datasets are using a procedure similar to that described in [1]. Table 2 shows major parameters in this generator and their meanings, as denoted in [11]. The synthetic data generator works as follows: First, it generates a set of L potential frequent graphs as seeds. They have I edges on average. Then, it randomly picks several seeds and merges them (overlaps these seeds as much as possible) to construct a new graph in the dataset. More details can be referred to [11]. A user can set parameters to decide the number of graphs (D) wanted and their average size (T). For a dataset which has 10000 graphs, each graph has 20 edges on average, the potential frequent graph has 10 edges on average, 200 potential frequent graphs, and 40 available labels, we represent this dataset as $D10kN40I10T20L200$. In our experiments, L is set to 200 and $L200$ is omitted in our legend specification.

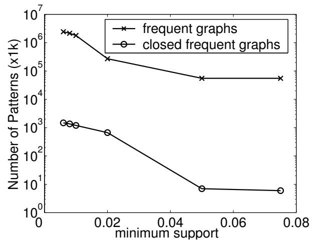
abbr.	meaning
D	the total number of graphs in the dataset
N	the number of possible labels
T	the average size of graphs in terms of edges
I	the average size of potentially frequent graphs
L	the number of potentially frequent graphs

Table 2: Parameters for Synthetic Graph Generator

Figure 11 shows the runtime and mining result for the dataset D10kN40I12T20. The synthetic graphs have 27 edges and 16 vertices on average. Compared with the real dataset, CloseGraph has a similar performance gain in this synthetic dataset. We do not show the runtime of FSG in this dataset and the following ones since the runtime of FSG is out of the bound in each figure.



(a) performance



(b) number of patterns

Figure 11: Varying Support for D10kN40I12T20

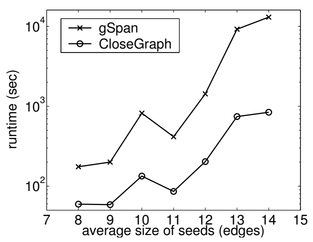
We then test the performance of CloseGraph by changing some major parameters in the synthetic data. The impact of different parameters is presented on the runtime of each algorithm. Two parameters in Table 2 are selected as varied ones: (1) the average size of potentially frequent graphs (seeds), and (2) the average size of graphs in the dataset. For each experiment, only one parameter varies with the others fixed. The experimental results are shown in Figure 12.

The performance study clearly shows that CloseGraph performs better than gSpan and FSG. Considering that it takes extra computation to obtain closed frequent graphs from mined frequent graphs in gSpan and FSG, the speedup achieved by CloseGraph should be more significant.

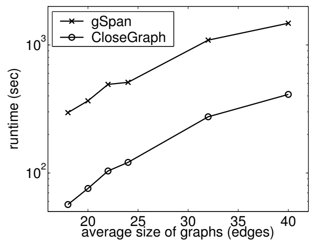
7. DISCUSSION

In this section, we discuss the extensions of CloseGraph for mining other kinds of graphs and the related work.

7.1 Mining Other Kinds of Graphs



(a) D10kN40I?T20: 1%



(b) D10kN40I12T?: 2%

Figure 12: Performance vs. Varying Parameters

So far, we have proposed and investigated a general framework, CloseGraph, for mining *closed*, *labeled*, *connected*, *undirected*, frequent subgraph patterns. Here we show that the framework can be extended to mine other kinds of graphs.

1. Mining *unlabeled or partially labeled* graphs. We build a label set which contains the original label set (*empty* if the graphs are all unlabeled) and a new empty label, ϕ. Label ϕ is assigned to all the vertices and edges that do not have labels. With this transformation, CloseGraph can directly mine unlabeled or partial labeled graphs.
2. Mining *non-simple* graphs. Non-simple graphs may have self-loop (i.e., an edge joins a vertex to itself) and multiple edges (i.e., several edges connect two same vertices). In CloseGraph, we always first grow backward edges and then forward edges. In order to accommodate self-loops, the growing order can be changed to *backward edges, self-loops, and forward edges*. Multiple edges can appear in these three kinds of edges. If we allow two neighbored edges in a DFS code to share the same vertices, actually the definition of DFS lexicographic order can accommodate multiple edges. Thus CloseGraph can mine non-simple graphs efficiently too.
3. Mining *directed* graphs. We use a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$, to represent an undirected edge. For directed edges, a new state can be introduced to form a 6-tuple, $(i, j, d, l_i, l_{(i,j)}, l_j)$, where d represents the direction of an edge. Let $d = +1$ represent the direction from i (v_i) to j (v_j), whereas $d = -1$ be that from j (v_j) to i (v_i). Notice that the sign of d is not related with the forwardness or backwardness of an edge. When growing a graph with one more edge, we allow this edge to have two choices of d, which only introduces a new state in the growing procedure and need not change the framework of CloseGraph.
4. Mining *disconnected* graphs. There are two cases: (1) the graphs in the dataset may be disconnected; (2) the graph patterns may be disconnected. For the first case, we can transform the original dataset by adding a virtual vertex to connect the disconnected graphs in each graph. Then we apply CloseGraph on the new graph dataset. For the second case, we redefine the DFS code. A disconnected graph pattern can be viewed as a set of connected graphs, $r = \{g_0, g_1, \ldots, g_m\}$, where g_i is a connected graph, and $0 \leqslant i \leqslant m$. Since each graph can be mapped to a minimum DFS code, a disconnected graph r can be translated into a code, $\gamma = (s_0, s_1, \ldots, s_m)$, where s_i is the minimum DFS code of g_i. The order of g_i in r is irrelevant. We can enforce an order among s_i such that $s_0 \leqslant s_1 \leqslant \ldots \leqslant s_m$. γ can be extended by either adding

one-edge s_{m+1} ($s_m \leqslant s_{m+1}$) to the code or by extending s_m, ..., and s_0. This procedure can be repeatedly conducted in a depth-first search way. When checking the frequency of γ in the graph dataset, make sure that $g_0, g_1, \ldots$, and g_m are disconnected with each other.

5. Mining *trees*. By removing backward edges, CloseGraph is ready to mine tree structures. The efficiency of this dwarfed version of frequent graph mining was demonstrated in [2]. It is expected that CloseGraph can achieve high performance in tree-structure mining.

7.2 Related Work

There were many previous studies on discovering common structure patterns among chemical compounds and proteins [7]. They targeted different measures of structure similarity and pair-wise algorithms to find max similarity between two structures. In this paper, we investigated the issue of mining frequent subgraphs among a large graph dataset. Dehaspe et al. [6] applied inductive logic programming approach to predict chemical carcinogenicity by mining frequent substructures. Holder et al. [9] proposed SUBDUE to do approximate substructure pattern discovery based on the minimum description length principle and optional background knowledge. These systems either do not demonstrate mining efficiency or cannot discover the complete set of frequent graphs. Inokuchi et al. [10] and Kuramochi and Karypis [11] proposed Apriori-based algorithms to discover all frequent substructures, and they also show that their methods mine the complete set of structured patterns, and have better performance than the previous ones. Recently, there are studies by Vanetik et al. [15] and Borgelt et al. [3]. The former applies an Apriori-like algorithm, but uses edge-disjoint paths as building blocks. [15] also introduces a new support definition in a graph. The framework proposed by [3] is similar to gSpan. However, gSpan systematically solved several difficult problems, including efficient detection and elimination of duplicate graphs.

Other related work in frequent substructure mining includes frequent tree structure mining [2, 19], discovering frequent graphs with geometric constraints [12], and etc.

8. CONCLUSIONS

In this paper, we investigated the problem of *mining closed frequent graph patterns in large graph data sets*, a critical problem in graph pattern mining because mining *all* patterns are inherently inefficient and redundant. Several new concepts are introduced in this study including *right-most extension, equivalent occurrence, early termination* and *its failure detection.* A CloseGraph method is implemented and our performance study demonstrates its high efficiency over gSpan and FSG. To the best of our knowledge, this is the first piece of work on closed graph pattern mining.

There are many interesting research problems related to CloseGraph that should be pursued further. One possible improvement over CloseGraph is the further improvement of failure detection. In our experiments, we have found that in order to detect one failure situation, we have to sacrifice half of the performance. A better failure detection algorithm may further improve the performance. Moreover, the incorporation of user-specified constraints in closed graph pattern mining and the extension CloseGraph to mining other complicated structured patterns are interesting topics for future research.

Acknowledgements. The synthetic data generator is kindly provided by George Karypis and Michihiro Kuramochi at the University of Minnesota. We are also thankful to Siegfried Nijssen for his script program in WARMR, Yifan Li, Yanli Tong, and anonymous referees for their helpful suggestions.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94.*

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM'02.*

[3] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finging relevant substructures of molecules. In *ICDM'02.*

[4] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *ICDE'01.*

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 2nd ed.* The MIT Press, Cambridge, MA, 2001.

[6] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *KDD'98.*

[7] I. Eidhammer, I. Jonassen, and W. R. Taylor Structure comparison and structure patterns. In *J. of Comp. Bio. 7, 2000.*

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD'00.*

[9] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the subdue system. In *KDD'94.*

[10] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD'00.*

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM'01.*

[12] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. In *ICDM'02.*

[13] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99.*

[14] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *DMKD'00.*

[15] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *ICDM'02.*

[16] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. *UIUC-CS Tech. Report: R-2002-2296*, (a 4-page short version in *ICDM'02*).

[17] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *SDM'03.*

[18] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SDM'02.*

[19] M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD'02.*