SPIN: Mining Maximal Frequent Subgraphs from Graph Databases

[1]Jun Huan, [1]Wei Wang, [1]Jan Prins, [2]Jiong Yang
[1]Department of Computer Science, University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
[2]Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana-Champaign, IL 61801, USA
[1]{huan, weiwang, prins}@cs.unc.edu, [2]jioyang@cs.uiuc.edu

Abstract

One fundamental challenge for mining recurring subgraphs from semi-structured data is the overwhelming abundance of such patterns. In large graph databases, the total number of frequent subgraphs can become too large to allow a full enumeration using reasonable computational resources. In this paper, we propose a new algorithm that mines only maximal frequent subgraphs, i.e. subgraphs that are not a part of any other frequent subgraphs. This may exponentially decrease the size of the output set in the best case; in our experiments on practical data sets, mining maximal frequent subgraphs reduces the total number of mined patterns by two to three orders of magnitude.

Our method first mines all frequent trees from a general graph database and then reconstructs all maximal subgraphs from the mined trees. Using two chemical structure benchmarks and a set of synthetic graph data sets, we demonstrate that in addition to decreasing the output size our algorithm can achieve a significant speed up over the current state-of-the-art subgraph mining algorithms.

1 Introduction

We focus on finding recurring subgraphs from graph databases that are not part of other frequent subgraphs. Graphs are generic ways to represent complicated data. Mining recurring subgraphs has many applications such as pattern discovery in chemical informatics [2] and bioinformatics [11], efficient storage of semi-structured databases [5], efficient indexing [6], and web information management [22, 16]. One important issue in efficient graph databases mining is handling the huge number of recurring patterns. The phenomenon is well understood in mining "long" frequent itemsets. Given a frequent itemset I, any subset of I is also frequent hence the total number of frequent itemsets grows exponentially in the size of the itemset. Mining graphs have the same problem: any subgraph of a frequent graph is frequent and the total number can grow exponentially. Practically we do observe a sharp increase of total number of subgraphs when we mine even moderate sized graph databases [10].

In this paper, we propose a new graph mining algorithm that mines only maximal frequent subgraphs, which offers several advantages in processing large databases. (1) Maximal subgraph mining significantly reduces the total number of mined subgraphs. In experiments on chemical data sets, the total number of frequent subgraphs is up to one thousand times greater than the number of maximal subgraphs. We can save both space and subsequent analysis effort if the number of mined subgraphs is significantly reduced. (2) Several "pruning" techniques, which are detailed in this paper, can be efficiently integrated into the mining process and dramatically improve the performance of the mining algorithm. (3) We do not loose information since the non-maximal frequent subgraphs can be reconstructed from the maximal subgraphs reported. To get the actual frequency (support) of non-maximal subgraphs requires examination of the original database, but it is certain to be at least as high as the frequency of the maximal subgraph. In addition,

the techniques used in [15] can be easily adapted to approximate the support of all frequent subgraphs within some error bound. (4) In some applications such as discovering structure motifs in a group of homology proteins [8, 11], maximal frequent subgraphs are the subgraphs of most interest since they encode the maximal structure commonalities within the group.

In this paper we show that maximal subgraph mining can be efficiently performed. Our mining method is based on a novel graph mining framework in which we first mine all frequent tree patterns from a graph database and then construct maximal frequent subgraphs from trees. This approach offers asymptotic advantages compared to using subgraphs as building blocks, since tree normalization is a simpler problem than graph normalization. The proposed method enables us to integrate well-developed techniques from mining maximal itemsets and knowledge gained in graph mining into a new algorithm. According to our experimental study, such a combination can offer significant performance speedup in both synthetic and real data sets. The framework of our method is versatile. Depending on the particular tree mining algorithm, the search can be either breadth-first or depth-first (preferred due to its better memory utilization). It can also be designed to mine all frequent subgraphs without major modifications.

In summary, we make three contributions: (1) we propose a novel algorithm SPIN (**SP**anning tree based maximal graph m**IN**ing) to mine maximal frequent subgraphs of large graph databases, (2) we integrate several optimization techniques to speed up the mining process, (3) we perform an extensive analysis of the proposed algorithm on graph databases with different characteristics.

The remainder of the paper is organized as follows. Section 2 presents a formal description of the maximal frequent subgraph mining problem. Section 3 presents the data structure and the proposed search algorithm. Section 4 presents the results of our experimental study using synthetic graph databases and two benchmark chemical data sets. We conclude the paper with a discussion, related works, and conclusion.

2 Background

We start this section defining labeled graphs. We define subgraph isomorphism relation to quantify the support value of a graph in a graph database. Maximal frequent subgraph is defined subsequently. Notations related to trees are introduced at the end of the section.

2.1 Labeled Graph

A *labeled graph* G is a graph where each node and edge has an associated label. We use Σ_V and Σ_E to denote the set of node labels and edge labels respectively. Without loss of generality, we assume a total order $\geq$ on Σ_V and Σ_E. The labeling function δ defines the mappings $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$ [1].

When we study recurring subgraphs in graph databases, it is critical to know whether a graph occurs in another graph, as defined below:

Definition 2.1 *A labeled graph G is **subgraph isomorphic** to another graph G', if there exists an injection $f : V[G] \rightarrow V[G']$ such that*

- $\forall\, u \in V[G], (\delta(u) = \delta'(f(u)))$,
- $\forall\, u, v \in V, ((u, v) \in E[G] \Rightarrow (f(u), f(v)) \in E[G'])$, *and*
- $\forall\, (u, v) \in E[G], (\delta(u, v) = \delta'(f(u), f(v)))$.

where $V[G]$ and $E[G]$ denote the node set and edge set of a graph G.

The injection f is a *subgraph isomorphism* from G to G'. By a slightly abused notation, we refer G as a "subgraph" of G', denoted by $G \subseteq G'$ by omitting the word "isomorphic"; similarly G' is referred to as a *supergraph* of G. A labeled graph G is defined to be *isomorphic* to another graph G' if G and G' are mutually subgraphs. Non-isomorphic subgraph is referred to as a *proper subgraph*, denoted by $G \subset G'$ and similarly G' is referred to as a *proper supergraph* of G. Given a set $\mathcal{G}$ of labeled graphs, the *support* of a graph G is the fraction of graphs in $\mathcal{G}$ in which G occurs.

[1] all graphs (including trees) discussed in this paper are *undirected* and all subgraphs are *connected* subgraphs.

Definition 2.2 *Given a set $\mathcal{G}$ of labeled graphs and a threshold $0 < \sigma \leq 1$, a subgraph G is a frequent maximal subgraph if and only if:*

- $sup_G \geq \sigma$, *and*
- $\nexists G'$, *($G \subset G'$ and $sup_{G'} \geq \sigma$).*

Example 2.1 *In Figure 1 we show three labeled graphs P, Q and S. Both Q and S are subgraphs of P and the mapping f: $q_1 \rightarrow p_1, q_2 \rightarrow p_2, q_3 \rightarrow p_3$ represents an subgraph isomorphism from graph Q to P. The graph Q has support value 2/3 in the set $\{P, Q, S\}$ because Q is a subgraph of P and Q, but not a subgraph of S. Given a threshold $\sigma = 2/3$, both graphs Q and S are maximal frequent subgraphs in the data set $\{P, Q, S\}$ since no proper supergraph of either of them is frequent.*

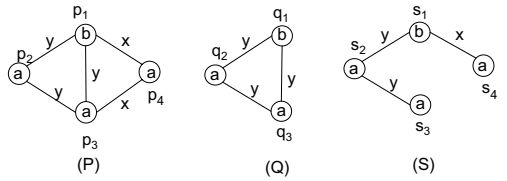

Figure 1. Example of a graph database $\mathcal{G}$ of three labeled graphs with subgraph relation. We assume that the node and edge labels are ordered alphabetically.

The **maximal frequent subgraph mining problem** is given a set $\mathcal{G}$ of labeled graphs and a threshold $0 < \sigma \leq 1$, finding all maximal frequent subgraphs G such that $sup_G \geq \sigma$.

2.2 Labeled Trees

A *(free) tree* is an acyclic connected graph. A *subtree* of a free tree T is a subgraph of T, which implies that every subtree itself is a free tree. A *rooted tree* is a free tree with a node designated as its *root*. A sequence of distinctive nodes in a tree T is a *path* if consecutive nodes are connected by en edge in T. In a rooted tree, a node u on the path from the root to a node v is an *ancestor* of v. If u is v's connected ancestor, u is v's parent and consequently v is u's child. A group of nodes sharing the same parent are *siblings*. A node is an *internal node* if it has at least one child; otherwise, it is a *leaf*. An *ordered rooted tree* is a rooted tree where the children of each internal node are ordered i.e. we can refer the children of a node as its first child, second child, and so on so forth. Tree may have labels, i.e. each node and edge has an associated label. In a labeled tree, a labeling function δ is provided as a mapping from a tree's node/edge to its associated label. We use the same symbol (δ) for both trees and graphs and the difference is usually understood from the context. Given an ordered rooted tree T, we define the *right-most leaf* as the leaf x which all of its ancestors (including x itself) are ordered the least among their siblings; the ordered rooted tree T_l obtained by remove the right-most leaf from T is defined as the *left-most* subtree of T.

Example 2.2 *We present three labeled, ordered, and rooted trees in Figure 2. Throughout the paper, we assume that root of a ordered rooted tree is shown on top and the children of each internal node are ordered left to right. Node t_6 of tree T is its right-most leaf.*

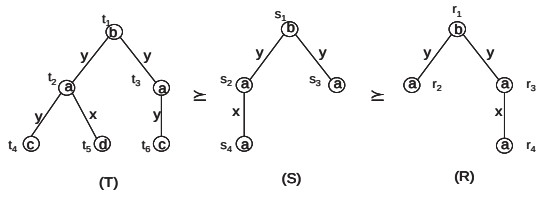

Figure 2. Example of three labeled ordered rooted trees and the order $\succeq$ among such trees.

In the following discussion, we define a string representation of a tree. We introduce two special symbols "$", "#" which we assume no tree may have them as labels. We further assume all labels of trees are ordered alphabetically and are greater than "$", which is greater than "#". For every labeled, ordered, and rooted tree T, there is a unique string presentation as suggested by Chi *et al.* [4]. The way to derive the string representation can be broken into two steps. In the first step, for each edge (u, v) in T (and assume v is u's child), we create an artificial label for v as a two-element sequence $e_l v_l$, where e_l and v_l are the labels of the edge (u, v) and the node v, respectively[2].

[2]Whether an edge label precedes a node label or not is not important. We just use one order throughout the paper.

In the second step, we perform a breadth-first enumeration of the nodes in the tree, starting from the root of the tree and following the order among children when visiting sibling nodes. The string obtained by concatenating the artificial labels of the nodes according to the traversal order, using "$" to group siblings and "#" to label the end of the string, is the string representation of the tree. For example, the tree T shown in Figure 2 has the string representation $b\$yaya\$ycxd\$yc\$\#$. In this example, we start with the label of the root, b. By breadth-first enumeration, we concatenate $yaya$ to b since it is the artificial label of the first and the second children of the root of T. We add $\$$ to the partial-completed string to group the two siblings. We continue the same enumeration for the rest of the nodes in T; add a $\#$ to label the end of the representation; and finally obtain the string $b\$yaya\$ycxd\$yc\$\#$ as T's string representation.

We use string's lexicographical order to define a total order $\succeq$ of all labeled, ordered, and rooted tree i.e. $T \succeq S$ if and only if the T's string representation is lexicographical greater or equal to S's string representation. For example, for the trees shown in Figure 2, we can easily verify that $T \succeq S \succeq R$.

A free tree T can be transformed to an ordered rooted tree by randomly designating a root and arbitrarily imposing order of children for every internal node. Each such tree is referred to as the *rooted ordered version* of T. Given a free tree T, T's *canonical representation*, denoted by $\mathcal{T}(T)$, is the maximal ordered rooted tree among all T's ordered rooted versions; the process to find such canonical representation is referred to as the *tree normalization* procedure and details of such computation can be found in [4, 21]

Before we leave the discuss about trees, we present the following simple theorem, which we are going to use in the next section.

Theorem 2.1 *Given two rooted ordered trees X and Y and the left-most subtree Y_1 of Y, let X's string representation be $x = x_1, x_2, \ldots, \#$ and that of Y_1 is $y = y_1, y_2, \ldots, y_n, \#$, if x is grater than the prefix $y' = y_1, y_2, \ldots, y_n$ of y, then $X \succ Y$.*

Proof 1 *This theorem is the direct consequence of the definition of right-most leaf and string representation of trees.*

3 Maximal Subgraph Mining

In the following discussion, we present a novel framework for mining maximal frequent subgraphs. We show that we can unify tree mining and subgraph mining into one process where we first find all frequent trees from a graph database and then construct frequent cyclic graphs from the mined trees. We developed two procedures to support this new framework. The first one is a graph partitioning method where we group all frequent subgraphs into equivalence classes based on the spanning trees they contain. The second one is a set of pruning techniques which we developed to prune graph partitions entirely or partially for efficient maximal graph mining.

There are advantages in the framework we developed. First, tree related operations, such as isomorphism, normalization, and testing whether a tree is a subtree of another tree, are asymptotically simpler than the related operations for graphs, which are NP-complete. Second, in certain applications such as chemical structure mining, most of the frequent subgraphs are really trees. Last but not least, this framework adapts well to *maximal* frequent subgraph mining, which is the focus of this paper. Using a chemical structure benchmark, we show 99% of cyclic graph patterns and 60% of tree patterns can be eliminated by our optimization technique in searching for maximal subgraphs; further details about the efficiency of the optimization techniques can be found in Section 4.

3.1 Canonical Spanning Tree of a Graph

We define a *subtree* of an undirected graph G as an acyclic connected subgraph of G. A subtree T is a *spanning tree* of G if T contains all nodes in G. There are many spanning trees for a given graph G. We define the maximal one according to a total order defined on trees as the *canonical spanning tree* of G, denoted by $\mathcal{T}(G)$.

Example 3.1 *In Figure 3, we show an example of a labeled graph P with all four-node subtrees of P. Each subtree is represented by its canonical representation and sorted according to the total order $\succeq$. Each such tree is a spanning tree of the graph P and the first one (T_1) is the canonical spanning tree of P.*

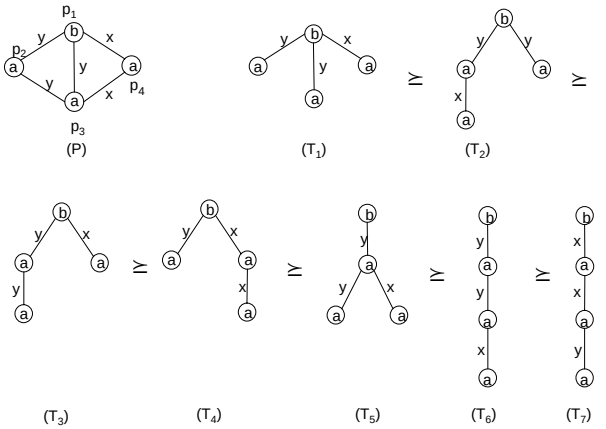

Figure 3. Example of a labeled graph P (upper-left), P's subtrees, spanning trees, and its canonical spanning tree (T_1).

In the following discussion, we study the properties of canonical spanning trees. We show that the canonical spanning tree of a graph G is the maximal one among those of subgraphs of G. Based on this property, we present a greedy algorithm to compute the canonical spanning tree of a graph. Before proving the theorem, we prove a weaker version of it on trees.

Theorem 3.1 *Given two free trees $R \subseteq S$, we have $\mathcal{T}(S) \succeq \mathcal{T}(R)$.*

Proof 2 *We prove the theorem by constructing a rooted, ordered version S_1 of S such that $S_1 \succeq \mathcal{T}(R)$. Upon succeeding in constructing S_1, by the definition of the canonical spanning tree, we have $\mathcal{T}(S) \succeq S_1 \succeq \mathcal{T}(R)$ and hence the proof of the theorem. To show that such S_1 always exists, let $R_1 = \mathcal{T}(R)$ be the canonical representation of R and f be a subtree isomorphism from R_1 to S. Let u be the root of R_1 and $M \subseteq V[S]$ be the set of images of nodes in $V[R_1]$ under f. We construct S_1, the rooted, ordered version of S as follows: the root of the S_1 is $f(u)$; the relative order among siblings $V \subseteq M$ follows the order of the corresponding nodes in R_1; for nodes no in M, their relative order is arbitrary but are all less than their siblings that are in M. An example of such construction is shown in Example 3.2.*

To complete the proof, we show that $S_1 \succeq R_1$. To that end, let $x = x_1, x_2, \ldots, x_h$ be the string representation of R_1 and $y = y_1, y_2, \ldots, y_l$ be the string representation of S_1 and we have the following observation. Let i denote the first position where $x_i \neq y_i$, we claim that we have either one of the following two conditions: (1) $x_i = \$$ and $y_i \in \Sigma_{E[S]}$ and (2) $x_i = \#$ and $y_i \in \Sigma_{E[S]} \cup \{\$\}$. In other words, at the first position i where string x and y differs, the symbol x_i can only be either a # or \$. This is because every node in R_1 has exactly one image in S_1 and hence the first mismatch could not happen between a node/edge label of S_1 and R_1. Given the observation and recall that every label in $\Sigma_{E[S]}$ is sorted greater than \$ and that is greater than #, we have the theorem proved.

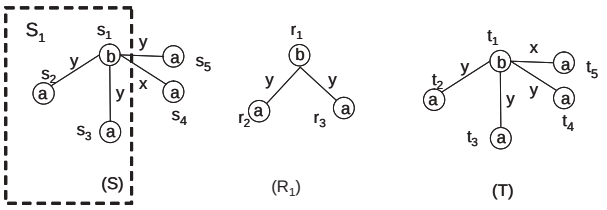

Figure 4. Example showing the canonical tree of tree S is the maximal among that of all subtrees of S.

Example 3.2 *In Figure 4, we show a free tree S and an canonical representation R_1 of a free tree $R \subseteq S$. We map R_1 to a subtree S_1 of S (labeled by a dashed rectangle) using a mapping: $r_1 \rightarrow s_1$, $r_2 \rightarrow s_2$, and $r_3 \rightarrow s_3$ (there are other valid subgraph isomorphisms from R_1 to S and it is left to interested users to valid the proof if we use other subgraph isomorphisms). We construct a rooted, ordered version X of S by using s_1 as the root and s_2, s_3 as the first and the second child of s_1. The relative order of s_4 and s_5 is not important. We just choose to flip them: s_5 as the third child of s_1 and s_4 as the last child. Notice in the figure that t_4 has the edge label x to node t_1 and therefore it corresponds to the node s_5, as stated before. The string representation of R_1 is $byaya\$\#$; that of X is $byayaxaya\$\#$ hence that we have $X \succeq R_1$. By definition we have $\mathcal{T}(S) \succeq X$ and therefore we have $\mathcal{T}(S) \succeq R_1 = \mathcal{T}(R)$.*

We prove a general form of Theorem 3.1 for graphs as follows.

Theorem 3.2 *Given two connected graphs $P \subseteq Q$, we have $\mathcal{T}(Q) \succeq \mathcal{T}(P)$*

Proof 3 *Let $R = \mathcal{T}(P)$ be the canonical spanning tree of P. We claim that we can always find a spanning tree S of Q such that $R \subset S$. The proof is straightforward. First, let $R' = R$. Second, we can find a node u in Q but not in R' and connected u to R' by an edge in Q. Such u always exists as long as R' is not the spanning tree of Q. Continue the process and finally we get a supertree of R which is a spanning tree of Q. By Theorem 3.1, we have $\mathcal{T}(S) \succeq R$. Therefore we have $\mathcal{T}(Q) \succeq \mathcal{T}(S) \succeq \mathcal{T}(P)$ and the proof of the theorem.*

It turns out we can prove an even stronger version of Theorem 3.2, which asserts that the left-most subtree S of a graph Q's spanning tree is maximal to some extents among its "peers". The theorem is stated below:

Theorem 3.3 *Given an n-node graph Q and a tree R that is the left-most subtree of $\mathcal{T}(Q)$, for every $(n-1)$-node subgraph $P \subset Q$, we have $R \succeq \mathcal{T}(P)$.*

Proof 4 *We prove the theorem by contradiction. Let's assume there is a $(n-1)$-node subgraph $P \subset Q$ such that $S = \mathcal{T}(P) \succ R$. We have $\mathcal{T}(Q) \succ S$ (by Theorem 3.2). Then we have $\mathcal{T}(Q) \succ R$ (transitivity). Finally we have $\mathcal{T}(Q) \succ \mathcal{T}(Q)$ (following Theorem 2.1). That leads to contradiction.*

Based on the theorem, we present a greedy search scheme in Table 1 to calculate the canonical spanning tree of a graph G. The algorithm works by first picking up maximal labeled nodes in G as a group of single-node trees. It iteratively grows those trees by attaching an additional node to each of them in all possible ways. The resulting group of trees are inspected one by one. Only those which are maximal among their peers are selected for the next iteration. Because of Theorem 3.2 this scheme is guaranteed to converge to the canonical spanning tree of a graph. Since every tree is a graph, the procedure can be applied to obtain canonical representations of trees also [3].

3.2 Tree-based Equivalence Class

In this section based on the canonical spanning tree we introduced, we introduce a graph partitioning method.

Algorithm Canonical Spanning Tree(G)
begin
1. $S \leftarrow \{u|\ u$ is a single-node tree with the maximal node label in $G\}$
2. **do**
3. $Q \leftarrow \{y|\ y$ is a subtree of G by including one additional node to a tree $x \in S\}$
4. $S \leftarrow \{y|\ y \in Q, \mathcal{T}(y)$ is maximal for all tress in $S\}$
5. **until** S contains spanning tree(s) of G
end

Table 1. An algorithm for finding canonical spanning trees

Definition 3.1 ***Tree-based Equivalence Class:*** *Given two graphs P and Q, we defined a binary relation $\cong$ such that $P \cong Q$ if and only if their canonical spanning trees are isomorphic to each other i.e. $\mathcal{T}(P) = \mathcal{T}(Q)$. The relation $\cong$ is reflexive, symmetric, transitive, and hence an equivalence relation.*

Example 3.3 *In Figure 5, we show subgraphs of the graph P in Figure 3 which are not necessarily trees. Subgraphs are grouped together if they share the same canonical spanning tree. The five non-singleton groups are shown here and the remaining twelve groups are all singletons* [4]

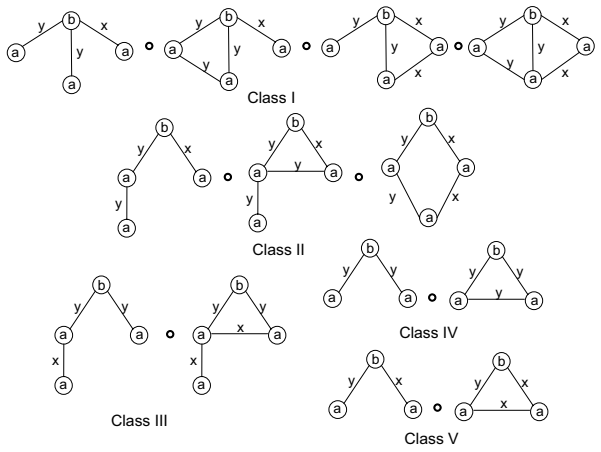

Figure 5. Example of tree based equivalence classes for subgraphs in graph P, presented in Figure 3.

[3] There is a notable difference between our procedure and the method present in [4]. We do not define the center of the tree but rather find the maximal string a free tree can produce. We choose the particular way for an uniform treatment of tree and graph canonical form as stated in Theorem 3.3

[4] Throughout the paper, we are only interested in subgraphs with at least an edge (i.e. excluding frequent nodes as trivial cases).

We outline a new frequent subgraph search algorithm based on the graph partitioning method discussed hereinbefore with two steps: (1) mine all the frequent trees from a graph database and (2) for each such frequent tree T, find all frequent subgraphs whose canonical spanning trees are isomorphic to T. Maximal frequent subgraphs can be found subsequently among frequent ones. We would like to discuss the first step later for two reasons. First, as pointed out in [20], the current subgraph mining algorithms can be easily tailored to find only trees from a graph database by limiting the topology of the patterns. This is true for Closegraph [20] as well as for FFSM [10], which is our previously developed depth-first subgraph mining algorithm. Second, most of the techniques developed for mining subtrees from a forest can also be easily adapted for the same purpose. Therefore, in the following discussion, we focus on step 2, which is how to enumerate the equivalence class of a tree T. The two-step division of the mining algorithm is artificially introduced to make it easy to explain the key ideas of the algorithm without exceeding details.

3.3 Enumerating Graphs from Trees

We define a *joining* operation $\oplus$ between a graph (tree) G and a hypothetical edge connecting any two nodes i, j in G with label e_l such that $G \oplus (i, j, e_l) = G'$ where G' is a supergraph of G with one additional edge between nodes i and j with label e_l. If the graph G already contains an edge between nodes i and j, the joining operation fails and produces nothing. If G' is frequent, we denote the hypothetical edge (i, j, e_l) as a *candidate edge* for G. The above definition can serve as the basis for a recursive definition of the joining operation between a graph G and a candidate edge set $E = \{e_1, e_2, \ldots, e_n\}$ such that $G \oplus E = (G \oplus e_1) \oplus \{e_2, \ldots, e_n\}$.

Let's assume we already calculated the set of candidate edges $C = \{c_1, c_2, \ldots, c_n\}$ from the set of all possible frequent hypothetical edges. We define the *search space* of G, denoted by $G : C$, as the set of graphs which might be produced by joining the graph G and a candidate edge set in the powerset set of C (denoted by 2^C). That is:

$$G : C = \{G \oplus y | y \in 2^C\} \qquad (1)$$

In the following discussion, the group of candidate edges are sometimes referred to as the "tail" of the graph G in its search space. We present a recursive algorithm in Table 3 to enumerate the search space for a graph G and the overall frequent subgraph search algorithm is presented in Table 2.

Example 3.4 *in Figure 6, we single out the largest equivalence class (Class One) from Figure 5. We show a tree K with its tail $C = \{(k_2, k_3, y), (k_3, k_4, x)\}$. K's search space is composed of four graphs $\{K, K_{S1}, K_{S2}, K_{S3}\}$ (K is always included in its search space) and is organized into a "search tree" in analogy to frequent item set mining. This tree structure follows the recursive procedure we present in Table 3 which can be used to explore the search space for a given graph.*

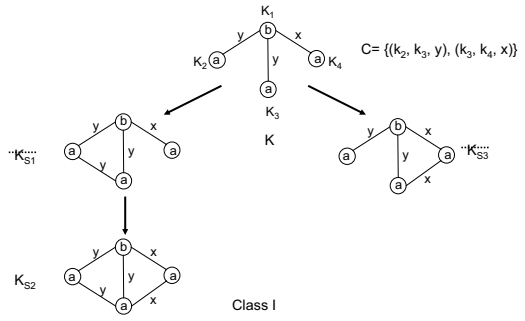

Figure 6. Example of enumerating graph's search space. We use dashed lines on the subgraph K_{S1} and K_{S3} to denote the fact that they will be pruned by an optimization technique which is discussed in Section 3.5.1

Algorithm Maximal Subgraph Mining($\mathcal{G}, \sigma$)
begin
1. $\mathcal{R} \leftarrow \{T | T$ is a frequent tree in $\mathcal{G}\}$
2. $S \leftarrow \{G | G \in$ Expansion(T), $T \in \mathcal{R}\}$
3. return $\{G | G \in S, G$ is maximal $\}$
end

Table 2. An outline of the maximal subgraph mining algorithm

Algorithm Expansion(T)
begin
1. $C \leftarrow \{c|\ c$ is a candidate edge for $T\}$
2. $S \leftarrow$ Search Graphs (T, C)
3. return $\{G|\ G \in S, G$ is frequent, G has the same canonical spanning tree as T has$\}$
end

Algorithm Search Graphs($G, C = \{c_1, c_2, \ldots, c_n\}$)
begin
1. $Q \leftarrow \emptyset$
2. **for** each $c_i \in C$
3. $\quad Q \leftarrow Q \cup$ Search Graphs($G \oplus c_i$, $\{c_{i+1}, c_{i+2} \ldots, c_n\}$)
4. **endfor**
5. return Q
end

Table 3. An algorithm for exploring the equivalence class of a tree T

Algorithm Maximal Subgraph Mining($\mathcal{G}$)
begin
1. $C \leftarrow \{c|\ c$ is a frequent edge in $\mathcal{G}\ \}$
2. ($\mathcal{F}$, $\mathcal{S}$)= Generic-Tree-Explorer(C, $\emptyset$)
3. return $\{G|G \in \mathcal{F}$ and G is maximal $\}$
end

Algorithm Generic-Tree-Explorer(C, R)
begin
4. $Q \leftarrow \emptyset$
5. **for** each $X \in C$
6. $\quad S \leftarrow \{Y|Y$ is a frequent tree of one additional node of $X \in C\}$
7. $\quad S \leftarrow S - R$ # avoiding duplicated search
8. $\quad (U, V) \leftarrow$ Generic-Tree-Explore(S, R)
9. $\quad Q \leftarrow Q \cup U \cup$ Expansion(X) #including cyclic graphs
10. $\quad R \leftarrow R \cup \{X\} \cup V$
11. **endfor**
12. return (Q, R)
end

Table 4. An algorithm for enumerating frequent trees/graphs based on FFSM

3.4 Enumerating Frequent Trees from a Graph Database

We outline a generic depth-first tree enumerating method in Table 4. The algorithm begins by finding all frequent edges from a graph database. The recursive function Generic-Tree-Explorer is invoked to perform a depth-first search where set C holds all trees that are to be enumerated (candidates) and set R holds all trees that have been enumerated. For a given n-node candidate X, all its frequent $n+1$-node supertrees Y are included as the candidates at the next level. Duplicated candidates are detected and removed at line 7. By adding the method to enumerate the search space of a tree (line 9) the algorithm is in fact enumerating all frequent graphs. The updated sets are returned where set Q holds the group of frequent graphs and R holds the set of frequent trees. Finally, maximal frequent subgraphs are computed from set of frequent subgraphs.

Our method to enumerate cyclic graphs is independent of the tree mining method and hence that people can freely choose any tree mining method in the implementation. The one we used, which is considerably more complicated and more efficient, is based on the FFSM algorithm [10] and is detailed in Appendix.

In summary, we developed a different method to enumerate frequent subgraphs. The method may be slightly more efficient than previous ones [5] but the technique for maximal subgraphs involves inefficient postprocessing. In the next section, we introduce optimization techniques to improve maximal frequent subgraphs search.

[5]tree related operations are treated separately from graph related operations and therefore save processing time

3.5 Optimizations: Global and Local Maximal Subgraphs

In this section by developing optimization techniques, we demonstrate that maximal subgraphs can be mined efficiently. Our developed techniques dynamically remove a set of frequent subgraphs that can not be maximal from a search space. To that end, we define a frequent subgraph G to be *locally maximal* if it is maximal in its equivalence class i.e. G has no frequent supergraph(s) that share the same canonical spanning tree as that of G; we refer to a subgraph as *globally maximal* if it is maximal frequent in a graph database. Clearly, every globally maximal subgraph must be locally maximal but not vice versa. Our pruning techniques aim to avoid enumerating subgraphs which are not locally maximal.

3.5.1 Bottom-Up Pruning

The search space of a graph G is exponential in the cardinality of the candidate edges set C. One heuristic to avoid such an exponential search space is to check whether the largest possible candidate $G' = G \oplus C$ is frequent or not. If G' is frequent, each graph in the search space is a subgraph of G' and hence not max-

imal. This heuristics is referred to as the *Bottom-Up Pruning* and can be applied to every step in the recursive search procedure presented in Table 3. By applying the bottom-up pruning to the equivalence class I presented in Figure 5, graph K_{S1} and K_{S3} are pruned.

Dynamic Reordering: An important technique related to the efficiency of the bottom-up pruning is the *dynamic reordering* technique, which works in two ways. First, it trims infrequent candidate edges from the tail of a graph to reduce the size of the search space (an edge candidate can become infrequent after several iterations since other edges are incorporated into the patterns). Second, it rearranges the order of the elements in the tail according to their support value. For example, given a graph's tail C, by dynamic reordering, we sort the elements in C by their support values, from lowest to highest. After this sorting, the infrequent "head" is trimmed. At the end of the remaining tail is a family of elements individually having high support and hence the pattern obtained by grouping them together is likely to still have high support value. This heuristics is widely used in mining maximal itemsets to gain performance. However, without the spanning tree framework, applying dynamic ordering is very difficult in any of the current subgraph mining algorithms, which intrinsically have a fixed order of adding edges to an existing pattern for various performance considerations.

3.5.2 Tail Shrink

Given a graph G and a supergraph G' of G, an *embedding* of G in G' is a subgraph isomorphism f from G to G'. We prefer the term embedding to subgraph isomorphism, though interchangeable, for more intuitive descriptions. In Figure 7, we show a subgraph L and its supergraph P. There are two embeddings of L in P: $(l_1 \rightarrow p_1, l_2 \rightarrow p_2, l_3 \rightarrow p_3, l_4 \rightarrow p_4)$ and $(l_1 \rightarrow p_1, l_2 \rightarrow p_3, l_3 \rightarrow p_2, l_4 \rightarrow p_4)$. We define a candidate edge (i, j, e_l) to be *associative* to a graph G if it appears in every embedding of G in a graph database. In other words, a candidate edge (i, j, e_l) of G is associative if and only if for every embedding f of G in a graph G', G' has the edge $(f(i), f(j))$ with label e_l. One example of an associative edge is the edge (l_1, l_3, y) to the tree L shown in Figure 7.

If a tree T contains a set of associative edges $\{e_1, e_2, \ldots, e_n\}$, any maximal frequent graph G which is a supergraph of T must contain all such edges. Hence we can remove these edges from the tail of T and augment them to T with no risk of missing any maximal ones. This technique is referred to as the *tail shrink* technique. Tail shrink has two advantages: (1) it reduces the search space and (2) it can be used to prune the entire equivalence class in certain cases. To elaborate the latter point, we define a set of associative edges C of a tree T to be *lethal* if the resulting graph $G' = T \oplus C$ has a canonical spanning tree other than that of T i.e. $\mathcal{T}(G') \neq \mathcal{T}(T)$.

We formally prove the correctness of the tail-shrinking optimization by the following theorem. First we prove that if a tree T contains a group A of associative candidates, any maximal supergraph of T must contains all such candidates. If the group of associative candidates is lethal, we prove that any maximal supergraph G of T can not be in T's equivalence class.

Theorem 3.4 *Given a tree T, its tail C, a group of associative edge $A \subseteq C$ of T, and a maximal graph $G \supset T$, we have $G \supseteq T \oplus A$ (I). Further more, if A is lethal, we have $\mathcal{T}(G) \succ \mathcal{T}(T)$ (II).*

Proof 5 *We prove (I) by showing contradiction. Let's assume there is a G such that $G \supset T$ and $G \not\supseteq T \oplus A$, then there is at least one $c = (i, j, e_l) \in A$ that is not part of G. Since c occurs in every instance of T, it occurs in every instance of G. Then we can construct a $G' = G \oplus c \supset G$ which has the same support value of G and hence frequent. This contradicts to the fact that G is maximal. For (II), we have $\mathcal{T}(G) \succeq \mathcal{T}(T \oplus A)$ (by Theorem 3.2) and $\mathcal{T}(T \oplus A) \succ \mathcal{T}(T)$ (by the definition of lethal). Therefore we have $\mathcal{T}(G) \succ \mathcal{T}(T)$.*

Example 3.5 *In Figure 7, we show an example of an associative edge $e = (1, 3, y)$ of L which is lethal to L also. In the same example, the lethal edge e can be augmented to each member of the class II to produce a supergraph with the same support and with a canonical spanning tree sorted greater than L. Therefore the whole class can be pruned away once we detect a lethal edge(s) to the tree L. Interested reader might verify that the equivalence class III shown in Figure 5 also has a lethal edge and the whole class can be pruned away by tail shrink.*

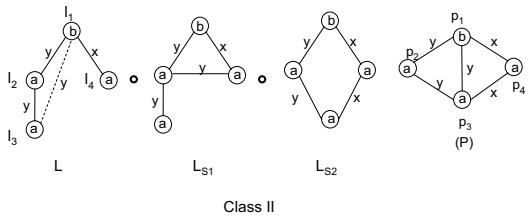

Figure 7. An example showing how tail shrink might be used to prune the whole equivalence class. Edge $e = (l_1, l_3, y)$, denoted by a dashed line to be distinguished from other edges, is associative to tree L and lethal to L as well. The graph obtained by joining L and e should belong to equivalence class I shown in Figure 5.

3.5.3 External-Edge Pruning

In this section, we introduce a technique to remove one equivalence class without any knowledge about its candidate edges. We refer to this technique as the *external-edge pruning*. We define an edge to be an *external edge* for a graph G if it connects a node in G and a node which is not in G. We represent an external edge as a three-element tuple (i, e_l, v_l) to stand for the fact that we introduce an edge with label e_l incident on the node i in a graph G and a node which does not belong to G with node label v_l. An external edge (i, e_l, v_l) is *associative* to a graph G if and only if:

- for every embedding f of G in a graph G', G' has a node v with the label v_l and is not an image of a node in $V[G]$ under f, and
- v connects to the node $f(i)$ with an edge label e_l in G'

.

Example 3.6 *We show two examples of associative external edges in Figure 8. One is (m_1, x, a) for the tree M and another one is $(n_f 1, y, a)$ for the tree N. If a tree T has at least one associative external edge, the entire equivalence class of T can be pruned since the same edge can be augmented to every member of the class. In this example, both equivalence classes IV and V can be eliminated due to the external-edge pruning.*

Once we find that a tree T has an associative external edge, the equivalence class associated with tree T can be eliminated since the same edge can be augmented to each members in T's equivalence class and therefore none of them are maximal.

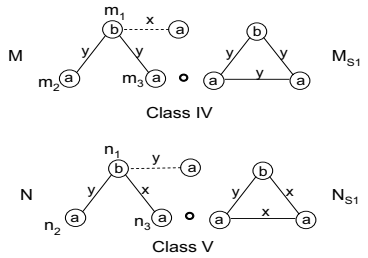

Figure 8. Examples showing external edges and associative external edges.

In summary, we present three pruning techniques to speed up maximal subgraph mining. For the graph P shown in Figure 3, there are a total of twenty five subgraphs of P, including itself and excluding the null graph. These subgraphs are partitioned into five non-singleton classes, shown in Figure 5, and twelve singleton classes (not shown). There is only one maximal subgraph, namely, graph P itself. We have successfully pruned every one of the five non-singleton equivalence classes (P of the equivalence class I is left untouched since it is maximal). What we do not show further is that we can apply the same techniques to the remaining twelve singleton equivalence classes to eliminate all of them. Interested readers might verify that themselves.

Table 5 and Table 6 integrate these optimizations into the basic enumerate technique we presented in Table 2 and Table 3.

3.6 Computational Details

In this section, we discuss computational details that are important for an efficient implementation of the SPIN algorithm. First, we present two algorithms to support the optimizations we designed for SPIN, namely, computing the edge candidates for a tree and determining whether there is an external associated edge for a frequent tree pattern. We then present two procedures to calculate a pattern's support value and

Algorithm MaxSubgraph-Expansion(T)
begin
1. $C \leftarrow \{c|\ c$ is a candidate edge for $G\}$
2. $A \leftarrow \{c|\ c \in C$ and c is associative $\}$
3. **if** A is lethal return $\emptyset$ #tail shrinking
4. $S \leftarrow$ Search Graphs ($T \oplus A, C - A$)
5. return $\{G|\ G \in S, G$ is frequent and maximal, $\mathcal{T}(G) = \mathcal{T}(T)\}$
end

Algorithm Search Graphs($G, C = \{c_1, c_2, \ldots, c_n\}$)
begin
1. **if** $G \oplus C$ is frequent, return $\{G \oplus C\}$ #bottom-up pruning
2. $Q \leftarrow \emptyset$
3. **for**each $c_i \in C$
4. $\quad Q \leftarrow Q \cup$ Search Graphs($G \oplus c_i, \{c_{i+1}, c_{i+2} \ldots, c_n\}$)
5. **endfor**
6. return Q
end

Table 5. An algorithm for exploring the equivalence class of a tree T for maximal subgraph mining

Algorithm Maximal Subgraph Mining($\mathcal{G}$)
begin
1. $C \leftarrow \{c|\ c$ is a frequent edge in $\mathcal{G}\}$
2. ($\mathcal{M}, S$)= Generic-Tree-Explorer(C, $\emptyset$)
3. return $\mathcal{M}$
end

Algorithm Generic-Tree-Explorer(C, R)
begin
4. $Q \leftarrow \emptyset$
5. **for**each $X \in C$
6. $\quad S \leftarrow \{Y|Y$ is a frequent tree of one additional node of $X \in C\}$
7. $\quad S \leftarrow S - R$ # avoiding duplicated search
8. $\quad (U, V) \leftarrow$ Generic-Tree-Explore(S, R)
9. $\quad Q \leftarrow Q \cup U, R \leftarrow R \cup \{X\} \cup V$
10. $\quad$ **if**(X has no external associative edge) # external-edge prunning
11. $\quad\quad Q \leftarrow Q \cup$ Max-subgraph-Expansion(X) #mining cyclic graphs
12. $\quad$ **endif**
13. **endfor**
14. return (Q, R)
end

Table 6. An algorithm for enumerating $\mathcal{M}$: maximal frequent subgraphs.

make sure it is maximal. By those procedures, we guarantee that every pattern reported by SPIN is frequent and maximal.

3.6.1 How to Calculate the Candidate Edge Set?

To calculate the candidate edge set for a tree T, we need to scan all embeddings of T and count how many graphs having an edge candidate e for every possible such edge. If T has n nodes, there are up to $n * (n - 1)/2 * |\Sigma_E|$ edges we need to monitor where $|\Sigma_E|$ is the total number of edge labels in a graph database. This computation is used frequently in the overall search and we want to optimize it as much as possible.

The optimization relies on the depth-first tree search method we introduced. Assuming an edge $e = (i, j, e_l)$ that connects two nodes in a tree T' that is the parent node of the tree T in a depth-first tree enumeration algorithm, one necessary condition for e to be frequent in T is that it must be frequent in T'. In other words, an edge e must be a candidate edge in T' in order to be considered as a candidate edge in T. Thus, to speed up the candidate edge calculation, T can "inherit" all candidate edges from its parent, verify their frequencies, and select those that are frequent. The additional edges T needs to be considered are those that are not part of T', namely, edges that connect the newly introduced node of T and nodes in T'.

3.6.2 How to Find an External Associative Edge?

Determining whether there is an associative external edge for a tree T can be very expensive. This involves scanning every embedding of T and searching for the associative edges. With depth-first search we can use an efficient heuristic. Given a tree pattern T, assuming tree T' is T's parent node in a depth-first enumeration, a *hint* of T is an associative external edge e of T'. When we check the tree T, we first check its hints (assuming there is at least one from its parent node). The reason is that if an edge is associative to a tree T', it is very likely to be associative to its child node also. If at least a hint e is valid, we prune the whole equivalence class of T. During such search process, we might come to situations where we "run out of hints", i.e. there is no valid associative external edge from its parent node for a tree T. In such cases, we employ an expensive search procedure to search every embedding of T to determine the set of associative external edges.

3.6.3 How to Calculate the Support Value for a Cyclic Graph?

Given a correct tree enumeration algorithm, the only thing we need to make sure about the SPIN algorithm is that every cyclic graphs is indeed frequent. A brute force way to guarantee this is performing a linear scan of the graph database and using subgraph isomorphism test to determine the support of a cyclic graph. It turns out we have a much efficient way to perform such

calculation than the brute force one. Given a graph database $\mathcal{G}$, a tree T, and an edge candidate e in T's tail C, the *embedding set* e_s of e is the group of embeddings of graph $T \oplus e$ in $\mathcal{G}$. Given a cyclic graph $G = T \oplus E = \{e_1, e_2, \ldots, e_n\}$, the embedding set of G can be computed efficiently by intersecting the embedding sets of cyclic graphs $T \oplus e$ for all $e \in E$. The support of G is the fraction of graphs those embeddings map to. This is further explained in the following example. .

Example 3.7 *Graph K has two embeddings in the graph database $\mathcal{G} = \{P, Q, R\}$ showing in Figure 1. Those are $\{(k_1 \rightarrow p_1, k_2 \rightarrow p_2, k_3 \rightarrow p_3, k_4 \rightarrow p_4), (k_1 \rightarrow p_1, k_2 \rightarrow p_3, k_3 \rightarrow p_2, k_4 \rightarrow p_4)\}$. Given $e_1 = (k_2, k_3, y)$ and $e_2 = (k_3, k_4, x)$ the cyclic graph $K \oplus e$ has the same two embeddings while the cyclic graph $K \oplus e_2$ has only the first embedding of K. Therefore graph $K \oplus \{e_1, e_2\})$ has only one embedding:$(k_1 \rightarrow p_1, k_2 \rightarrow p_2, k_3 \rightarrow p_3, k_4 \rightarrow p_4)$.*

Where does the efficiency come from? First, we notice that we do not need to perform any subgraph isomorphism checking. Second, we do not even need to scan the database since if we keep the embedding set of each candidate edge, the only operation we need to do is set intersection. Given candidate edge set $E = \{e_1, e_2, \ldots, e_n\}$, computing the support value of a cyclic graph $G = T \oplus E$ is bounded by $O(|E| \times n)$ where n is the total number of embeddings T has. In our implementation, we index embeddings of a tree T by integers 1, 2, ..., n. The embedding set of any cyclic graph $T \oplus e$ can be efficiently encoded as a bit-vector $V = v_1, v_2, \ldots, v_n$ where v_i is the 1 if and only if $T \oplus e$ has the ith embedding of T. The set operation can be efficiently performed by the bitwise and operation.

3.6.4 How to Find Maximal Subgraphs?

We notice that with the optimizations we presented, all reported graphs G are locally maximal, i.e. there is no supergraph of G sharing the same canonical spanning tree. To select globally maximal subgraphs, we need to check whether there is an extern edge e such that if attached, the resulting graph (or tree) is frequent. If there exists such external edge, the graph is not maximal. Otherwise, it is. This is similar to what we did in searching for external associated edge and we could use similar strategy to save computation.

4 Experimental Study

We performed our empirical study using a single processor of a 2.8GHz Pentium Xeon with 512KB L2 cache and 2GB main memory, running RedHat Linux 7.3. The SPIN algorithm is implemented using the C++ programming language and compiled using g++ with O3 optimization. We compared SPIN with two alternative subgraph mining algorithms: FFSM ([10]) and gSpan [19]. Every maximal subgraph reported by SPIN in synthetical and real data sets are cross validated using results from FFSM and gSpan to make sure it is (a) frequent, (b) maximal, and (c) unique.

4.1 Synthetic Dataset

To evaluate the performance of the SPIN algorithm, we first generate a set of synthetic graph databases using a synthetic data generator [13]. There are six parameters to control the behavior of the synthetic graph generator and we summarize them in Table 7. The synthetic data generator works by first creating a set of candidate graphs (the total number are controlled by L) with user specified size (I). The candidates are subsequently selected, replicated and augmented with random selected node and edges to create a graph database. Further details of the synthetic graph generator might be found in [13].

In Figure 10, we represent the performance comparison of SPIN, FFSM, and gSpan algorithms for a synthetic data set with different support values. When the support is set to a pretty high value e.g. 5%, the performance of all three algorithms are pretty close. SPIN scales much better than the other two algorithms as we decrease the support values. At support value 1%, SPIN provides a six and ten fold speed-up over FFSM and gSpan, respectively. We do not show data with support value great than 5% since there is little difference among the three methods.

We then launched a set of four experiments to test the scalability of SPIN for important parameters listed in Table 7. In Figure 11, we compare the performance using different value of I, which specifies the average size of the set of subgraphs to be embedded frequently

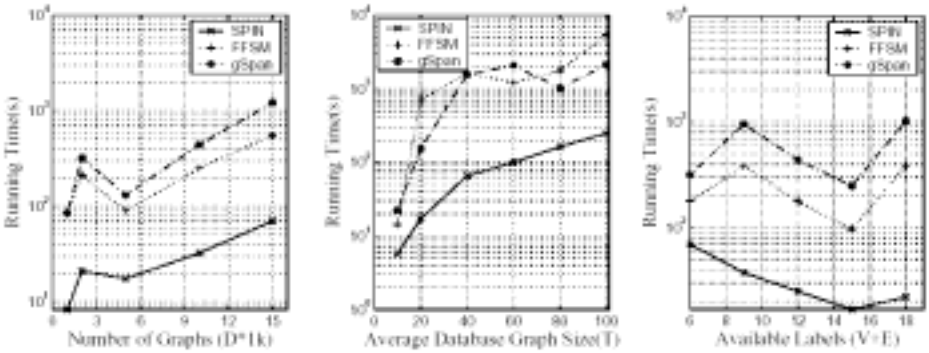

Figure 9. Left: performance comparison with different database sizes (D) using SPIN, FFSM and gSpan. Middle: the same comparison using different average database graph sizes (T) and Right: using different number of available labels. We fix the support value to be 1% in all cases. When we change the total number of labels, we fix the ratio between node labels and edge labels at 1:2

Parameter	meaning
D	total number of graphs in a generated database
T	average graph size (edges)
L	the total number of potentially frequent subgraphs
I	the size of the potentially frequent subgraphs (edges)
V	the total number of node labels
E	the total number of edge labels

Table 7. We list the set of parameters controlling the behavior of the synthetic graph generator. We use a single string, e.g. "$D10kT30L200I11V4E4$" to represent a graph database generated by a particular combination of parameters. For example, "$D10kT30L200I11V4E4$" stands for a synthetic graph database which contains a total of D = 10k (10000) graphs. On average, each graph contains T = 30 edges with up to V =4 vertex labels and $E = 4$ edge labels. There are total $L = 200$ potential frequent patterns in the database with average edge number $I = 11$. We use a fixed value $L = 200$ throughout the paper following previous published papers.

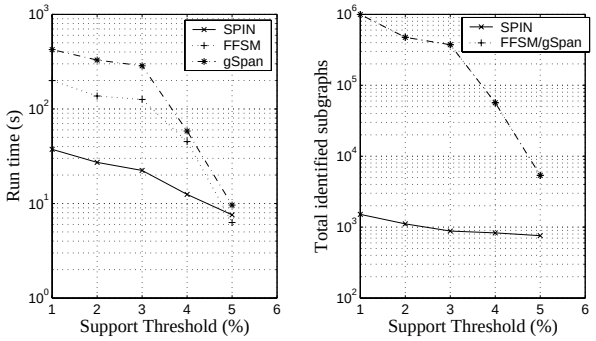

Figure 10. Left: performance comparison under different support values for data set $D10kT30L200I11V4E4$ using SPIN, FFSM and gSpan. Here we follow the common convention of encoding the parameters of a synthetic graph database as a string. Right: Total frequent patterns identified by the algorithms.

in a synthetic graph database. These subgraphs are referred to as *seeds* in [13] and they correlate to the set of maximal frequent subgraphs we need to find. The results demonstrate that SPIN scales well with different sizes of seeds. On the other hand, FFSM and gSpan have to enumerate all subgraphs of these seeds. They clearly experience an exponential growth of the total number of such subgraphs (shown in the right part of Figure 11) as the sizes of the created seeds increase. We notice more than an order of magnitude speed-up in the study and the performance gap can become even larger as we further increase the value of I.

In Figure 9, we further compared the scalability of the SPIN using different database sizes, different average graph transaction sizes, and different number of node/edge labels. Throughout the experimental studies, SPIN always offers a huge performance gain comparing to FFSM and gSpan.

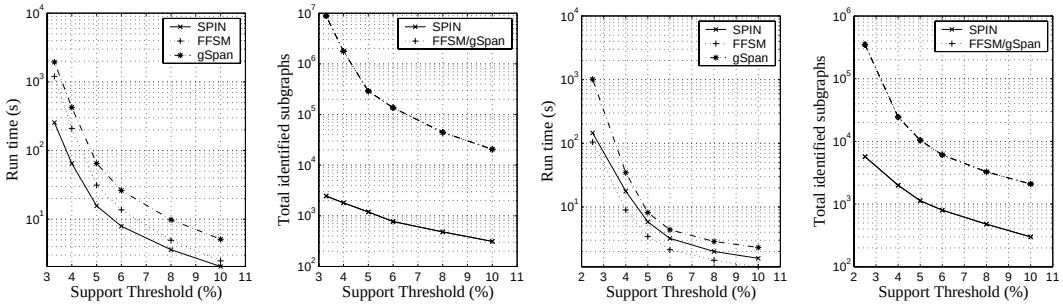

Figure 12. Left: performance comparison under different support values for DTP CA data set using SPIN, FFSM and gSpan. Right: Total frequent patterns identified by the algorithms.

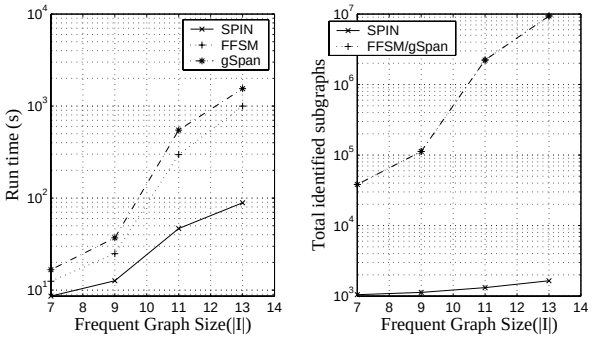

Figure 11. Left: performance comparison with different seed sizes(I) using SPIN, FFSM and gSpan. Right: Total frequent patterns identified by the algorithms.

4.2 Chemical Data Set

We also applied SPIN to two widely used chemical data sets to test its performance. The data sets are obtained from the DTP AIDS Antiviral Screen test, conducted by U.S. National Cancer Institute. In the DTP data set, chemicals are classified into three sets: confirmed active (**CA**), confirmed moderately active (**CM**) and confirmed inactive (**CI**) according to experimentally determined activities against the HIV virus. There are in total of 423, 1083, and 42115 chemicals in the three sets, respectively. For our own purposes, we used all compounds from CA and from CM to form two data sets, which are subsequently referred to as DTP CA and DTP CM, respectively. The DTP data can be downloaded from http://dtp.nci.nih.gov/docs/aids/aids_data.html. The characteristics of these two data sets are summarized in Table 8.

In Figure 12, we show the performance comparison of SPIN, FFSM, and gSpan using the DTP CA data set. We report that SPIN is able to expedite the program up to five(eight) fold, comparing with FFSM(gSpan) at support value 3.3%. Mining only maximal subgraphs can reduce the total number of mined patterns by a factor up to three orders of magnitude in this data set. We also applied the same algorithms to the data set DTP CM. In this case, SPIN has a performance very close to FFSM and both are around eight fold speed-up over gSpan. However, if we impose an additional constraint to let FFSM output the maximal patterns it finds among the set of frequent patterns, SPIN offers a three fold speed-up from FFSM.

5 Related Work

Knowledge discovery from semi-structured data sets is an active topic in the data mining/machine learning community. Many different pattern definitions were proposed from different perspectives such as finding patterns from a single large network [14], finding approximately matched patterns [17], mining patterns using domain knowledge from bioinformatics [9], and finding frequent subgraphs. The later one is the focus of our paper.

Recent subgraph mining algorithms can be roughly classified into two categories. Algorithms in the first category use a level-wise search scheme based on the Apriori property to enumerate the recurrent subgraphs [12, 13]. Rather than growing a graph by one single node/edge at a time, Vanetik *et al.* recently pro-

Data Set	N	$\overline{V}$	$\overline{L_V}$	$maxV$	$minV$	$\overline{E}$	$\overline{L_E}$	$maxE$	$minE$
DTP CA	423	39.56	3.99	190	10	42.2	2.058	196	10
DTP CM	1083	31.8	3.665	220	6	34.25	2.07	234	4

Table 8. The characteristics of the two data sets. N is the total number of graphs in a data set; $\overline{V}$ and $\overline{E}$ specify the average numbers of vertices and edges; $\overline{L_V}$ and $\overline{L_E}$ specify the average numbers of vertex labels and edge labels; and $maxV/minV$ and $maxE/minE$ are the maximal/minimal numbers of vertices/edges of all graphs in a data set, respectively.

posed an Apriori-based algorithm using paths as building blocks with a novel support definition [18].

Algorithms in the second category use a depth-first search to enumerate candidate frequent subgraphs [19, 20, 2, 10]. As demonstrated in these papers, depth first algorithms provide advantages over level-wise search for (1) better memory utilization and (2) efficient subgraph testing, e.g. it usually permits the subgraph test to be performed incrementally at successive levels during the search [10].

Our current work benefits extensively from existing algorithms for maximal itemset mining such as [3, 7] and frequent subtree mining algorithms [1, 21].

6 Conclusion and Future Work

In this paper we present SPIN, an algorithm to mine maximal frequent subgraphs from a graph database. A new framework, which partitions frequent subgraphs into equivalence classes is proposed together with a group of optimization techniques. Compared to current state-of-the-art subgraph mining algorithms such as FFSM and gSpan, SPIN offers very good scalability to large graph databases and at least an order of magnitude performance improvement in synthetic graph data sets. The efficiency of the algorithm is also confirmed by a benchmark chemical data set. The algorithm of compressing large number of frequent subgraphs to a much smaller set of maximal subgraphs will help us to investigate demanding applications such as finding structure patterns from proteins in the future.

Acknowledgement We thank Dr. Jack Snoeyink in the University of North Carolina for helpful discussions about the paper.

References

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, and H. Sakamoto. Efficiently substructure discovery from large semi-structured data. *SDM*, 2002.

[2] C. Borgelt and M. R. Berhold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. International Conference on Data Mining'02*, 2002.

[3] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. *ICDE*, 2001.

[4] Y. Chi, Y. Yang, and R. Muntz. Indexing and mining free trees. *ICDM*, 2003.

[5] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. *in SIGMOD*, pages 431–442, 1999.

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. *In VLDB'97*, 1997.

[7] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. *ICDM*, 2001.

[8] J. Hu, X. Shen, Y. Shao, C. Bystroff, and M. J. Zaki. Mining protein contact maps. *2nd BIOKDD Workshop on Data Mining in Bioinformatics*, 2002.

[9] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining protein family specific residue packing patterns from protein structure graphs. In *Eighth Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 308–315, 2004.

[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. *ICDM*, 2003.

[11] J. Huan, W. Wang, A. Washington, J. Prins, R. Shah, and A. Tropsha. Accurate classification of protein structural families based on coherent subgraph analysis. In *Proc. Pacific Symposium on Biocomputing*, 2004.

[12] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *In PKDD'00*, 2000.

[13] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. International Conference on Data Mining'01*, 2001.

[14] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *SDM*, 2004.
[15] J. Pei, G. Dong, W. Zou, and J. Han. On computing condensed frequent pattern bases. *ICDM*, 2002.
[16] S. Raghavan and H. Garcia-Molina. Representing web graphs. *In Proceedings of the IEEE Intl. Conference on Data Engineering*, 2003.
[17] N. Vanetik and E. Gudes. Mining frequent labeled and partially labeled graph patterns. *ICDE*, 2004.
[18] N. Vanetik, E. Gudes, and E. Shimony. Computing frequent graph patterns from semi-structured data. *Proc. International Conference on Data Mining'02*, 2002.
[19] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. International Conference on Data Mining'02*, 2002.
[20] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. *KDD'03*, 2003.
[21] M. Zaki. Efficiently mining freqeunt trees in a forest. *SIGKDD*, 2002.
[22] M. J. Zaki and C. J. Hsiao. Charm: An efficient algorithm for closed itemset mining. *In SDM'02*, 2002.

A APPENDIX

A.1 Enumerating Frequent Trees from a Graph Database using Modified FFSM

In this section in implementing SPIN we present the method for mining trees from a graph database. The method is based on a depth-first search algorithm called FFSM (Fast Frequent Subgraph Mining), which is designed for enumerating all frequent graph from a graph database [10]. Since every frequent tree is a frequent graph, FFSM can be quite easily tailored to mine tree only.

We highlight several key components of the FFSM algorithm; further details can be found in [10]. Given an n by n adjacency matrix M of a graph (tree), we define its *code* as the sequence of lower triangular entries in the order of $m_{1,1}, m_{2,1}, m_{2,2}, \ldots, m_{n,1}, \ldots, m_{n,n}$ where $m_{i,j}$ is the entry at the ith row and jth column of the matrix M (the upper triangular part is the mirror of the lower one and is omitted). We designate one matrix as the *canonical adjacency matrix* (CAM) of a graph G if it has the largest lexicographic code among all possible adjacency matrices of G. Based on CAM representation of graphs, we designed a compact data structure called CAM tree to assign every frequent subgraph of a graph database a unique node in the CAM tree. We developed two operations: a joining operation, which is subdivided into four cases: case 1, case 2, case 3a and case 3b, and an extension operation to dynamically construct and enumerate the CAM tree.

Based on the FFSM algorithm, the algorithm to enumerate frequent trees without cyclic graphs can be outlined as follows: first we scan a graph database to obtain all frequent edges. Second, we use the joining operation case 3b and the extension operation to perform depth first enumeration of the CAM tree to find all frequent tree. The algorithm is presented in Table 9. We claim that this algorithm is *correct and non-redundant* in that we only report frequent trees and report all of them exactly once. The formal proof of algorithm requires significant materials from the FFSM algorithm and is not directly related to current paper. We omit the proof.

Algorithm Frequent-Graphs($\mathcal{G}$)
begin
1. $C \leftarrow \{c |$ c is a frequent edge in $\mathcal{G}\}$
2. return Modified-FFSM-Explore(C)
end

Algorithm Modified-FFSM-Explore((C)
begin
1. $Q \leftarrow \emptyset$
2. **for** each $X \in C$
3. $S \leftarrow \{$ FFSM-join-case3b(X, Y)$|Y \in C\}$
4. $S \leftarrow S \cup$ FFSM-extension(X)
5. $S \leftarrow \{Y | Y \in S, Y$ is frequent, and it is in its canonical form $\}$
6. $Q \leftarrow Q \cup$ Modified-FFSM-Explore(S)
7. **endfor**
8. return Q
end

Table 9. An algorithm for enumerating frequent trees based on FFSM