Diagonally Subgraphs Pattern Mining

Moti Cohen Ehud Gudes
Department of Computer Science, Ben-Gurion University
Beer-Sheva 84105, Israel
{motic, ehud} @cs.bgu.ac.il

ABSTRACT
In this paper we present an efficient algorithm, called DSPM, for mining all frequent subgraphs in large set of graphs. The algorithm explores the search space in a DFS fashion, while generating candidates in advance to each mining phase just like the Apriori algorithm does. It combines the candidate generation and anti monotone pruning into one efficient operation thanks to the unique mode of exploration. DSPM efficiently enumerates all frequent patterns by using diagonal search, which is a general scheme for designing effective algorithms for hard enumeration problems. Our experiments show that DSPM has better performance, from several aspects, than the current state of the art - gSpan algorithm.

Keywords
graph mining, frequent subgraphs, pattern discovery.

1. INTRODUCTION
Whereas in the past, data mining was mainly applied to structured data and flat files, there is growing interest for mining semi-structured data such as web links [8], chemical compounds [4], or efficient database indexing [9]. The problem of frequent substructure pattern mining is to find frequent subgraphs over a collection of graphs. Frequent subgraph mining serves meaningful structured information such as widespread web access patterns, common protein structures, and shared patterns in object recognition. Another application is to cluster XML documents based on their common structures. Furthermore, a graph is a general data structure which covers all previous well-researched frequent patterns, thus it can fuse the mining process into one framework.

Problem Statement. Given a dataset of transactions *D*, each transaction $t \in D$ is a labeled undirected subgraph. Edges and vertices have their labels. Given a minimum support, *minSup*, DSPM finds all connected structured patterns that occur in at least *minSup* transactions.

Related Work. There are two general approaches in efficient frequent structured patterns mining. The Apriori approach [1,2,3] adopts the breadth-first search which was first developed in the context of association rules by Agrawal and Srikant [10]. The Apriori-based algorithm works as follows: given all connected frequent patterns from size *k*, construct out of this group a set of candidates such that each candidate pattern is from size *k*+1. A candidate generation of size *k*+1 can be accepted, for example, by joining two frequent patterns from size *k* that share a common kernel from size *k*-1. The next step, usually, will be to count support, i.e., how many transactions in DB each one of the candidates occurs. For each candidate with a count above *minSup* will be considered as a frequent pattern. This way we discover at each phase larger and larger frequent patterns, one group after another. The detailed algorithms, in general, distinguish themselves in using different building blocks: vertices in [1], edges in [2], and edge-disjoint paths in [3]. The second approach is DFS exploration, represented by [4,5,6], which adopts a pattern-growth by growing patterns from a single graph directly, that is depth-first search. The algorithms map each pattern to a unique canonical label. By using these labels, a complete order relation is imposed over all possible patterns. This lexicographic order (over all patterns) is also used to impose tree-hierarchy search order over all patterns. One of the characteristics of the (search space) tree is that all nodes at level *k* of the tree represent all connected patterns with *k* edges and only them. An in-order search over the tree enables to discover all frequent patterns as opposed to the BFS approach that discovers all frequent *k*-patterns (patterns with *k* edges) before discovering frequent (*k*+1)- patterns. The main idea of our method is to construct a hybrid which combines the two approaches. The algorithms which most closely related to our current attempt are FSG [2] and gSpan [5]. Our algorithm, DSPM, explores the search space in a depth search. It can use several mining techniques which are especially applicable for DFS algorithms, such as, using transactions id lists like gSpan [5] or maintaining an embedding set for each frequent subgraph like FFSM from [6]. We adopt from gSpan its canonical graph representation and corresponding tree search space. On the other hand, we adopt from FSG its notable frequency anti-monotone pruning (checking for each generated candidate, with *k*+1 edges, if all its subgraphs with *k* edges were found to be frequent in previous step. If not, then the candidate isn't frequent and therefore can be dropped). Our algorithm enjoys from this anti-monotone pruning technique since it keeps all previous mined subgraphs alike Apriori does. Once DSPM explores a pattern from size *k*, only some of the frequent patterns from size *k*-1 were discovered till that moment. Still, we will show that frequency anti-monotone pruning technique can be applied without any special constraint thanks to the unique reverse depth exploration. The reverse depth exploration contributes for easy generation of candidates, fast anti-

DMKD'04, June 13, 2004, Paris, France.

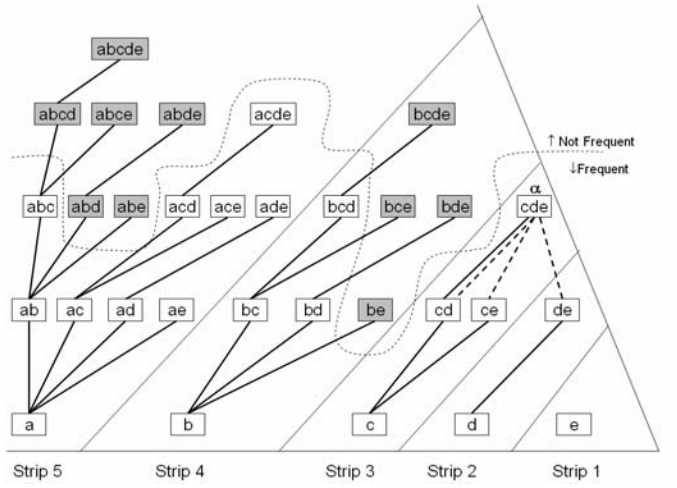

Figure 1. Lattice (Itemset Search Space)

monotone pruning and the ability to explore efficiently more space in a single step of the algorithm unlike previous DFS algorithms. In addition we represent several novel ideas that can be integrated by previous mentioned algorithms.

2. PRELIMINARY CONCEPTS

In section 2.1 we describe the main algorithm idea by developing it for the well-known task of association rules. We will get familiarized with the search space of frequent itemsets and the way DSPM algorithm explores it. As a framework for subgraph mining, section 2.2 defines the search space for the frequent subgraphs problem. It uses the graph representation of gSpan algorithm [5] though it is not bound to this specific representation.

2.1 Algorithm Outline

Itemsets mining is simpler than subgraphs mining in many aspects. Thus, for clarity we choose to start describing DSPM with respect to itemsets. An example of an itemsets tree is illustrated in Figure 1 (forest). Consider the curved-dashed line in Figure 1. The problem of mining frequent itemsets can be viewed as finding a cut through the lattice such that all elements above the cut are infrequent itemsets, and all elements below are frequent itemsets and their counting support is known.

Definition 1 (Prefix Based Lattice). *Let $\tau \in$ {itemsets, sequences, trees, graphs} be a frequent pattern problem. Let τ-order be a complete order over the patterns (i.e., over the canonical representation of patterns), and let τ-space be the corresponding search space of the problem which has a tree shape. Given a pattern p^k, $k>1$, subpatterns(p^k) = { p^{k-1} | p^{k-1} is a subpattern of p^k }. Then, the τ-space is Prefix Based Lattice if* (i) *The parent of each pattern p^k, $k > 1$, is the minimum τ-order pattern from the set subpatterns(p^k).* (ii) *An in-order search over τ-space follows ascending τ-order.* (iii) *The search space is complete.*

The itemset search space as depicted in Figure 1 is a prefix based lattice. From now on we will assume implicitly that for each given τ problem, the corresponding τ-space is defined to be prefix based lattice.

Definition 2 (Reverse Depth Search). *Regular depth search over τ-space which explores the sons of each visited node (pattern) in a descending τ-order.*

As depicted in Figure 1, the lattice is divided into diagonal strips, counted from right to left. The strips are explored one after another, from right to left by DSPM algorithm. Each strip is explored in a reverse depth search by the algorithm. We consider the τ-space as a forest rather than one tree. Each tree in τ-space will correspond to one strip such that the root will be a pattern from size 1.

Property 1 (FAM: Frequency Anti-Monotone). *If a pattern p^k is frequent, then any subpattern of p^k must be frequent. It is equal to say, if p^k isn't frequent then any pattern that contains p^k isn't frequent also.*

FSG algorithm uses property 1, by checking for each candidate c^{k+1} if all its subpatterns from set subpatterns(c^{k+1}) found to be frequent in the previous phase. DSPM applies the same pruning for candidates, that is, although DSPM explores the search space by using a reverse depth search, it can still apply FAM pruning. Consider Figure 1. When candidate {*c, d, e*} is generated, the only discovered itemsets at that point are the ones which appear in strips 1, 2 and 3 only. In this case we can apply FAM checking on {*c, d, e*} because all its 2-subsets (In Figure 1, connected to {*c, d, e*} with dashed lines, that is {*c, d*}, {*c, e*} and {*d, e*}) their frequency were determined already. Is there a possibility that DSPM might visit a candidate that it can't apply on it FAM pruning? i.e., is it possible DSPM will discover {$x_1, x_2, \ldots, x_k$} before determining the frequency of {$x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots x_k$} for some $1 \le i \le k$? The answer is no and the following theorem claims that this technique of exploration can be used not only for itemsets but also for any problem τ as long as the search space is a prefix based lattice.

Theorem 1. *Given a pattern problem τ and the corresponding τ-order and τ-space, then by exploring τ-space in reverse depth search it enables checking FAM for each explored pattern, if all previous mined patterns are kept.* (Proof omitted).

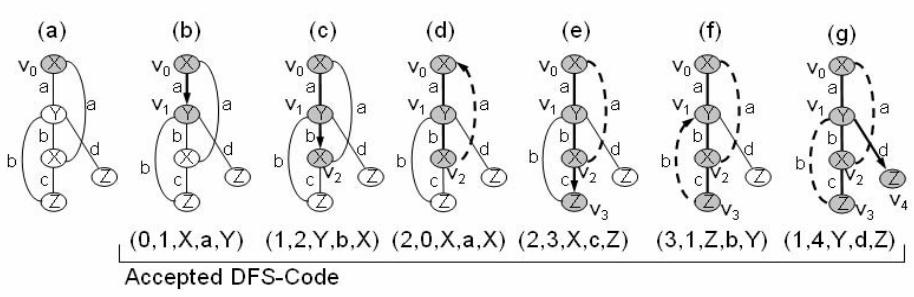

Figure 2: Depth search and its DFS Code

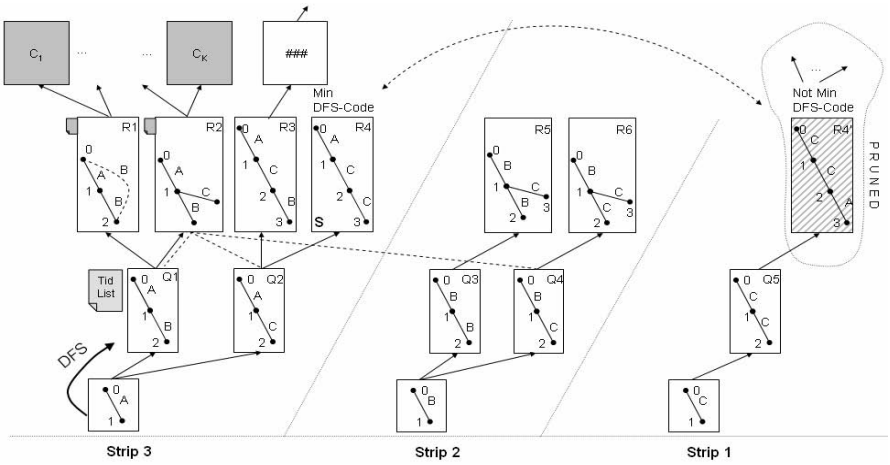

Figure 3: DFS Code Search Space

Intuitively, DSPM generates candidates in advance, the same manner Apriori algorithms does. Unlike Apriori, when the reverse depth search visits p^k, only a part of the frequent patterns of size k are known. And the only joining of p^k can be done with k-patterns which are τ-order bigger than p^k. Nevertheless, there is no need to know the frequent k-patterns which are τ-order smaller than p^k in an attempt to generate sons-candidates to p^k. Because each son of p^k , named c^{k+1}, can be accepted by joining only pairs of k-patterns from the set subpatterns(c^{k+1}) and because p^k is the smallest one in subpatterns(c^{k+1}), then pattern p^k is the last one to be explored by a reverse depth search from all the patterns in the set subpatterns(c^{k+1}). That guarantees for DSPM the ability to extend from each visited pattern p^k all its sons by using join operation with some of the k-patterns discovered so far.

2.2 Lexicographic Ordering in Graphs

This section discusses a canonical graph representation and a corresponding search space, based on [5] work. It includes mapping each graph to a DFS Code (a unique label), building a lexicographic ordering among these codes and constructing a complete search space, with a tree shape, of all the DFS Codes. Since the defined search space is an instance of prefix based lattice, we can use it with DSPM algorithm.

When performing a depth search in a subgraph, if the search visits an edge which leads to a new node that wasn't explored by the search previously, then the edge is a forward edge. Otherwise, it is a backward edge with respect to the given depth search [7]. A corresponding DFS Tree can be constructed from the set of Forward edges. For example, Figure 2(g), without the dashed lines, is a DFS Tree which was accepted from the depth search 2(a)-(g). The depth search defined by [5] explores all backward edges before it finds the next forward edge. The depth search of the vertices forms a linear order. The magnitude of subscripts is used to illustrate this order according to their discovery time [7]. $i < j$ means v_i is discovered before v_j (Figure 2). We denote G subscripted with a DFS tree T by G_T. T is named also a DFS subscripting of G. The *right most path* of G_T is the path from v_0 to v_n, and the right most vertex is v_n.

Consider Figure 2. At each state of the given depth search, another edge is discovered. Under each state we can see 5-tuple that represents the discovered edge. The DFS Code that represents the DFS Tree in Figure 2(g) (which represents the depth search in Figure 2) is the sequence of 5-tuples that is accepted from the depth search. Of course, one subgraph may have many DFS Codes because we can apply different depth search over a single subgraph. For that reason, [5] constructed a lexicographic order

among all valid DFS Codes so we can choose a canonical representation for each subgraph, G, by picking the minimum DFS Code that can be accepted among all DFS Codes.

Since we are aimed to construct a search space for graphs, we need to consider also how we can extend a graph. Given graph G and T_0 (The canonical subscripting of G). Edge e can extend G from the right-most vertex connecting to any other vertices on the right-most path (*backward extension*), or e can extend G from vertices on the right-most path and introduce a new vertex (*forward extension*). We consider these two kinds of restricted extension as *legal* extensions, denoted by $G \bullet e$. This way of extensions fits well for extending DFS Code representation since it follows a continuing depth search.

In a DFS Code search space, each node represents a DFS code, the relation between parent and child node complies with the extension described above. The relation among siblings is consistent with the DFS Lexicographic Order, so the pre-order exploration of DFS Code search space follows the DFS lexicographic order. Figure 3 shows a DFS Code search space. Through depth search of the code tree, all the minimum DFS codes of frequent subgraphs can be discovered in this way.

Theorem 2 (DFS Code Pruning). *(i) Given a DFS Code Tree, by exploring only nodes with Minimum DFS Codes, and pruning all other nodes, it is guaranteed to keep the search space completeness, i.e, Minimum DFS Code can only grow from Minimum DFS Code. (ii) All descendents DFS Codes of infrequent DFS Code in Tree are also infrequent.* (see [5] for proof).

Theorem 3. *DFS Code Tree is Prefix based lattice.*

3. ALGORITHM DETAILS

Section 3.1 illustrates the main procedure of the algorithm using a recursive procedure, named ExploreLattice. The recursive procedure is aimed to explore the DFS Code search space and discover all the nodes which represent frequent subgraphs. Section 3.2 shows how candidates can be generated in each phase. Section 3.3 further develops the generation of candidates and integrates in it a novel technique for fast Frequency Anti-Monotone Pruning (FAM Pruning). Finally, section 3.4 explains about the support counting method.

3.1 Main Procedure

As explained above, DSPM explores the search space in a Reverse depth search. Since it keeps all previous mined frequent patterns and since the defined search space is a prefix based lattice, it can generates from each frequent pattern a group of candidates and also apply FAM pruning by using previous mined patterns. The high level structure of the algorithm is shown in Figure 4.

The procedure gets a transaction set of graphs and minimum support, *minSup*. It returns all frequent subgraphs stored in the container F. Line 1 removes infrequent vertices and edges labels

from the graph set D. Line 2 sorts all representative frequent edges (with two vertices) and store result in E. Now, in a reverse lexicographic order, for each edge $e \in E$ (Line 4) the algorithm constructs a one edge subgraph g^1 from edge e (Line 6), and makes a call at line 10 to the recursive procedure ExploreLattice with g^1. The recursive procedure explores in a reverse DFS search all the induced subtree under subgraph g^1 in the defined search space, i.e., finding all frequent subgraphs whose min DFS Codes appear as a descendant subgraphs of g^1 in the DFS Code search space.

Suppose we have a set of transactions D. Each transaction is a labeled graph. For simplicity, there are different labels on edges but all vertices has the same label v. Suppose also that the frequent edges are A, B and C. Infrequent edges were removed. Therefore E will holds after the sorting at line 3 the set {(v, A, v), (v, B, v), (v, C, v)}.

In the first for loop (with reverse lexicographic order over the edges in E) a subgraph with single edge is constructed from edge (v, C, v) which results with DFS Code (0, 1, v, C, v). For abbreviation we shall write the 1-subgraph as g^1_C. The recursive method ExploreLattice (Line 10) explores the induced subtree of subgraph g^1_C, that is all the frequent subgraphs which contain edges with label C only. Figure 3 (strip 1) shows the explored tree and its frequent subgraphs in the subtree space that are discovered. In the second round, ExploreLattice explores induced subtree of subgraph g^1_B, which is to explore all frequent subgraphs that contain edges with label B and maybe also label C (Figure 3 - strip 2). In the third and last round, ExploreLattice explores induced subtree of g^1_A, i.e., finds out all frequent subgraphs which contains edges with label A, and possibly also labels B and C (Figure 3 - strip 3).

We can improve the algorithm in the following way. In the first round, instead of exploring the tree in Figure 3(strip 1) by mining transaction set D, we can project from D to D^* only occurrences of edges with label C, and mine transaction set D^* instead. This is applicable because the graphs in Figure 3(strip 1) have edges with label C only. In the second round the algorithm needs to explore the tree in Figure 3(strip 2), which contains graphs that their edges have labels B and C only. And so on. This enhancement is reflected in Figure 4 at lines 5, 9 and also line 10 which calls procedure ExploreLattice with D^* instead of D. The outcome is faster searches over smaller projected graphs than the ones in D. Similar approach can be found in [5].

Line 7 builds the transaction ID list (TID list) for subgraph g^1, that is, we keep a list of transaction identifiers that support it. We will do so for any frequent subgraph. Once we need to compute the frequency of a son-candidate of subgraph g^k, the algorithm can

```
DSPM(D, minSup)
1.  Remove infrequent vertices and edges
    considering minSup
2.  E:= all frequent 1-edge graphs in D, sorted
    in ascending DFS lexicographic order
3.  D*:= {}, F:= {}
4.  For each e ∈ E, in reverse lexicographic
    order, do
5.     D*:= D* ∪ { all occurrences of e in D}
6.     g¹:= {e}
7.     TID(g¹):= {t.id | t ∈ D, g¹ is subgraph of
       t}
8.     F¹:=F¹ ∪ {g¹}  /* Fᵏ:={frequent k-
       subgraphs}. Fᵏ⊂F */
9.     sons(g¹):= {mine all frequent 1-edge
       extensions for g¹ from D*}        /*
       sons(g¹) are attached to g¹. */
10.    ExploreLattice(D*,F,g¹, minSup)
11. Return F
```

Figure 4: The Main procedure for mining all frequent subgraphs from transaction set D.

```
ExploreLattice(D*, F, g^k, minSup)
1.  if (sons(g^k) is empty set)
2.     return;
3.  for each g^{k+1} ∈ sons(g^k), in reverse order,
    do
4.     F^{k+1} := F^{k+1} ∪ {g^{k+1}}
5.     GeneratesCandidates(g^{k+1}, F)
6.  if no candidates were produced then
7.     return;
8.  SupportCounting(D*, g^k, minSup)
9.  For each g^{k+1} ∈ sons(g^k), in reverse order,
    do
10.    ExploreLattice(D* , F , g^{k+1} , minSup)
```

Figure 5: The recursive procedure which explores the search space.

limit the support counting only to the set of transactions in the TID of g^k. As soon as the recursive function ends a visit at g^k, it can delete TID list of g^k.

Line 8 inserts each frequent 1-subgraph, g^1, into Frequent-subgraphs data structure *F*. Line 9 finds all frequent 1-edge extensions to g^1. Line 9 mines also 2-subgraph at this stage of the algorithm in an attempt to satisfy the precondition of the recursive function that comes at line 10 (this will become clear in the following paragraph).

Figure 5 presents the recursive procedure ExploreLattice for exploration of DFS Code search space in a reverse depth search. The recursion receives parameter g^k which is a DFS Code to visit, so as to explore all induced subtree of g^k. Following are pre and post condition of ExploreLattice procedure. *Precondition***:** (i) Graph g^k is frequent *k*-subgraph that has its TID list. (ii) None of g^k's descendants were explored yet, except to its sons. *Post condition*: (i) induced subtree of g^k in DFS Code Tree was explored, i.e., all frequent-descendants subgraphs (in DFS Code search space) of g^k were discovered and stored in *F*.

Consider Figure 3 which depicts the subgraphs lattice. All white nodes represents discovered frequent subgraphs up till now, gray nodes represents candidates. The recursion visits node *Q1*. The precondition of the recursion in order to visit *Q1* is that *Q1* has its TID list and its frequent sons are known (white nodes). Line 5 grows from each son of g^k a group of candidates (gray nodes) by calling GenerateCandiates procedure. The support counting method at line 8 counts support for all candidates that were generated from all sons of g^k and in the same time also builds TID lists for all sons of g^k. As a result from line 8, each son of g^k will be linked to two sets. One set will be its TID list and the other one is its frequent sons. Lines 9-10, for each son of g^k, in reverse lexicographic order, a recursive call applied to discover the induced subtree of each son. As can be seen, DSPM explores the search space two steps a head instead of one in order to earn a larger set of candidates that can be enumerated in a single support counting. Since it generates a group of candidates by using FAM pruning, it still needs to handle in each recursion frame only a limited set of candidates.

3.2 Candidates Generation

Algorithm FSG generates candidates of size *k*+1 by joining two frequent *k*-subgraphs. In order for two such frequent *k*-subgraphs to be eligible for joining they must contain the same (*k*-1)-subgraph as their core. This joining procedure is called fsg-join. For each of the generated candidates, FSG algorithm checks if it is already in the set of candidates. If not then it verifies that if all its subgraphs of size *k* are frequent, i.e., FAM pruning. On the other hand, DSPM isn't worried if a candidate was generated before, but only whether the generated candidate can be a son of the frequent subgraph from which it grew, more precisely, whether the DFS Code which was received by extending min DFS Code of a frequent subgraph (by adding one edge) is also a min DFS Code (See Theorem 2(i) – DFS Code Pruning). Thus DSPM generates and validates candidates in three steps as follows: (i) Candidate Generation & FAM Pruning (ii) Validating min DFS-Code. (iii) Support counting.

Consider again Figure 3. It is shown how DFS Code tree is explored by DSPM. The algorithm visits node *Q1* whose frequent sons are known, *R1* and *R2*. The algorithm needs to generate from nodes *R1* and *R2* a set of candidates.

Let's concentrate on *R2*. The algorithm finds from all the 3-subgraphs which produced till that moment (i.e., from all the 3-subgraphs that are τ-order not smaller than *R2*) which ones share a common (*k*-1)-subgraph with *R2* and can be joined with it. Trying to join a given *k*-subgraph with all previous mined frequent *k*-subgraphs is simply unacceptable. As an alternative, we can access directly to all previous mined *k*-subgraphs which share a common core with the subgraph we want to generate its sons candidates by keeping for each frequent (*k*-1)-subgraph, g^{k-1}, a list of extensions to all frequent *k*-subgraphs that can be accepted from g^{k-1} by adding one edge.

The extensions list of g^{k-1} will be constructed from a set of pairs. The first item in a pair will be a reference to a *k*-subgraph which can be accepted from g^{k-1} by adding the edge that is kept as the second item in a pair. Consider Figure 3. Regarding subgraph *Q4*, we can conclude the following: extensions(*Q4*) = {<*R2*,(1, 3, *v*, *A*, *v*)>, <*R3*,(2, 3, *v*, A, *v*)>, <*R5*,(1, 3, *v*, *B*, *v*)>, <*R6*,(1, 3, *v*, C, *v*)>}. The set sons(p^k) is subset of extensions(p^k). For a given subgraph, its set of sons is determined only once along the algorithm execution that is when the recursion visits the subgraph, but the set of extensions is always updated. Consider Figure 5, line 4. Each time we find another frequent k-subgraph, besides from adding it into the frequent set *F*, we now also need to update the extensions set of all its (k-1)-subgraphs.

What we can do now to facilitate generation of candidates from *R2*, is searching only after its subpatterns(*R2*) . Then for each *Qj*∈subpatterns(*R2*), for each extension <*Ri*, *e*> ∈ extensions(*Qj*), we can join *R2* and *Ri* with core *Qj*. This way we can directly access all previous mined 3-subgraphs that share a common core with *R2* and as well we know in advance their common core.

```
GeneratesCandidates(g^k)
1.  T:= {}
2.  for each <g^{k-1},e> ∈ subpatterns(g^k)
3.    T:= T ∪ dfs-join(g^k, <g^{k-1}, e>)
4.  {check AMP for each candidate in T }
5.  for each g^{k+1} ∈ T
6.    if min(g^{k+1}) ≠ g^{k+1}
7.       remove(g^{k+1}) from T
8.  sons-candidates(g^k):= T
```

Figure 6: The Generation of candidates.

In Figure 3 all subpatterns(*R2*) are connected with dashed lines to *R2*. Those are *Q1*, *Q2* and *Q4*. Consider subgraph *Q4*. Subgraph *R2* will be joined with each of 3-subgraphs in extensions(*Q4*), i.e., *R2* (self join), *R3*, *R5* and *R6*, and the common core will be *Q4*. Same technique will be applied also with cores *Q1* and *Q2* in order to generate candidates from *R2*. This idea reflected at lines 2-3 in Figure 6. Procedure dfs-join returns a set of candidates which is received from joining g^k with each of the extensions of core g^{k-1}.

Our join operation can be more efficient relative to fsg-join. Each constructed candidate is designated to be son of *R2* and so several constraints can be added to the join operation. This is explained in the following subsection. Line 4 checks FAM for the generated candidates. If a candidate doesn't obey to FAM it is discarded. In lines 5-7, the algorithm checks, for each candidate, if it has a min DFS Code. Those who don't have are removed from the set *T* (see Theorem 3.2 – DFS Code Pruning). One novel idea of our algorithm is to combine the dfs-join procedure with FAM pruning. This is explained in section 3.3.

The dfs-join procedure. Consider Figure 7. Procedure dfs-join gets one *k*-subgraph g^k, one (*k*-1)-subgraph g^{k-1} and an edge $e \in V(g^k)$, so $g^k \setminus e = g^{k-1}$. Subgraph g^{k-1} holds already all the set of *k*-subgraphs to join with g^k, the ones who kept as frequent extensions to g^{k-1}, while the common core for the join operation is g^{k-1}.

Each edge extension e' to g^{k-1} represents a frequent *k*-subgraph $g^{k'}$ which is received by adding edge e' to g^{k-1}. We can join g^k and $g^{k'}$ with common core g^{k-1} by mapping edge e' to g^k through an isomorphism from g^{k-1} to $g^k \setminus e$. Since we are using min DFS Code to represent subgraphs then it is equivalent to see subgraph g^k as a subscripted graph (see section 2.2 for details). For each edge extension e' to g^{k-1}, we confine all combinations of mapping e' (line 3) to subscripted graph g^k such that e' grows from right most path of g^k as a legal forward edge or backward edge. This restriction causes dropping of many candidates and helps finding more bounded group over the desired frequent subgraphs.

dfs-join finds at line 2 all isomorphisms from g^{k-1} to $(g^k \setminus e)$ and for each given isomorphism σ, it maps edge e' from g^{k-1} to g^k and stores the joining result at line 5 only if $\sigma(e')$ represents a valid forward edge or a backward edge growing in g^k.

As for line 2 in Figure 7, we don't really need to generate all this isomorphisms. The parent procedure GenerateCandiates(g^k , F) already did this job when it found all (*k*-1)-subgraphs of g^k. In fact, it is not the only one to do so. Procedure ExploreLattice, before calling to GenerateCandiates, inserted graph g^k in data structure *F*, and as mentioned, this means that it needs to find all (*k*-1)-subgraphs of g^k and updates their extensions list and this is the first (and last) time to generate all isomorphisms from g^k to all its (*k*-1)-subgraphs. These isomorphisms are passed as a parameter to GenerateCandiates and to dfs-join procedures.

```
dfs-join(g^k, <g^(k-1), e>)
1.  C:= {}
2.  M:= generate all isomorphisms from g^(k-1) to
    (g^k \ e)
3.  for each extension <g^k', e'> ∈ extensions(g^(k-
    1)) do
4.    for each isomorphism σ ∈ M do
5.      C:= C ∪ {generate candidates of size
        k+1 from the set g^k, e, e', g^(k-1) and σ. }
6.  return C
```

Figure 7. The joining procedure.

3.3 Freq. Anti-Monotone Pruning (FAM)

One way to do FAM is to find for each generated candidate, of size *k*+1, all its *k*-subgraphs and checking if each one of them was found to be frequent, like FSG does. This technique demands a massive computation, i.e., applying *k* isomorphisms in the worst case for each *(k+1)*-candidate not to mention the searching cost. We can do much better than that with some compromising over FAM pruning. Instead of applying FAM we would apply only Partial-FAM which on the average is expected (see experiments section – table 1) to give results almost as good as FAM.

Method GenerateCandidates can be changed a little such that the FAM pruning becomes a trivial operation (Figure 6, line 4) with no computation effort. Several of the candidates are generated more than once. We will prove that the number of tries in which the same candidate is generated has a strong relation to FAM pruning.

Consider an edge extension e' to subgraph g^k so that (*k*+1)-subgraph $g^k \bullet e'$ is a nominee to be a frequent child of g^k. Suppose it is. Then each *k*-subgraph of $g^k \bullet e'$ must be frequent also. Now let us look at some $g^{k-1} \in$ subpatterns(g^k) which is received by dropping some edge e from g^k. We can add to g^{k-1} edge e' at the same place we added to g^k (with only one exception), under isomorphism consideration, and get a frequent *k*-subgraph $g^{k-1} \bullet e'$. This is so because $g^{k-1} \bullet e'$ is subgraph of $g^k \bullet e'$. Thus edge e' must appear in the extensions set of g^{k-1} and therefore we would expect that the procedure call dfs-join($<g^k, e>, g^{k-1}$) would return candidate $g^k \bullet e'$ among several others. Since we didn't limit the discussion to specific $g^{k-1} \in$ subpatterns(g^k) thus each calling to dfs-join with each subgraph $g^{k-1} \in$ subpatterns(g^k) would return candidate $g^k \bullet e'$ among several others. We can conclude that a candidate cannot be frequent if exist $g^{k-1} \in$ subpatterns(g^k) such that the result of dfs-join($<g^k, e>, g^{k-1}$) doesn't include the candidate. This simple condition is applied in Figure 6, line 4 by dropping candidates which were generated less than |subpatterns(g^k)| times.

Yet, there is a specific type of edge extension that might be missed. If the edge extension, e', relative to g^k, is a forward edge which grows from a node $v \in V(g^k)$ such that degree(v)=1 then at most one of the (*k*-1)-subgraphs of g^k doesn't have node v, under isomorphism consideration, and therefore cannot be extended with edge e'. Therefore the algorithm need to identify this special extension and exclude it only if it was generated less than |subpatterns(g^k)|-1 times.

3.4 Support Counting

Once a set of candidates have been generated, DSPM algorithm computes their frequency. The simplest way to achieve this frequency is, for each candidate subgraph, to scan each one of the transaction graphs and determine if there is a subgraph isomorphism or not. Even so, having to compute this isomorphism is particularly expensive and this approach is not feasible for large datasets. DSPM algorithm uses another technique in order to make this heavy task to be more efficient.

Following are pre and post condition for SupportCouting procedure. *Precondition:* (i) Graph g^k is frequent *k*-subgraph that has its TID list. (ii) None of g^k's descendants where explored yet, except to his sons. (iii) Each son of g^k, signed g^{k+1}, has a set of

Table 1. Using Chemical Compound dataset. (1) Running time comparsion of DSPM and gSpan. (2) Number of candidates comparsion of DSPM and FSG.

minSup	(1) running time <sec>		(2) # of candidates		Frequent Patterns
	DSPM	gSpan	DSPM	FSG	
0.0911	0.85	1.81	1276	1168	1049
0.0588	2.0	4.6	2705	2694	2326
0.0294	23.5	49.7	24119	24064	22758
0.0205	128.4	240.9	139790	139666	136949

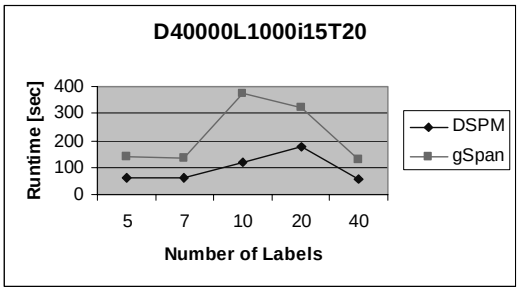

Figure 8. Runtime with respect to Number of different labels in graph-set.

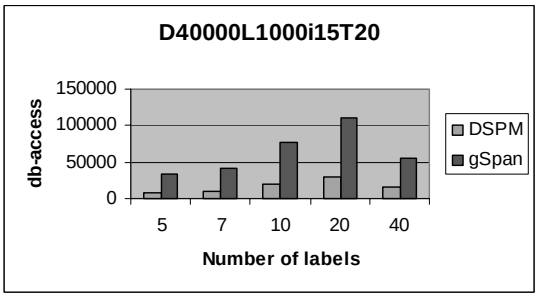

Figure 9. Number of db-access.

candidates that might be its frequent sons. *Postcondition:* (i) All frequent sons of g^k have their own TID lists. (ii) All frequent grandchildren of g^k are known.

The method applies subgraph isomorphism with graph g^k over all the projected transactions that appear in TID of g^k. For each occurrence of g^k in some transaction the algorithm tries to extend the occurrence to each of g^k 's children. For a successful extension to a child g^{k+1}, the algorithm adds to TID list of g^{k+1} the current transaction ID and tries to further extend to each one of the candidate-sons of g^{k+1}. If the algorithm succeeds to further extend it to a candidate that grows from g^{k+1} then it updates the support counting of this candidate.

After we have finished counting support, the method checks which candidates have support above *minSup* and update those frequent ones to be frequent-sons of their parents. If none of g^{k+1} extensions is frequent the sons(g^{k+1}) becomes empty which is the exit condition of the recursive procedure ExploreLattice.

4. PERFORMANCE STUDY

The experiments were carried out on Intel 2.0GHz machine with 256MB main memory, running Windows XP operating system, and compiled by Visual C++ 6.0. Besides implementing DSPM we have also implemented gSpan algorithm.

Both algorithms use underneath two nontrivial components of subgraph isomorphism and canonical labeling (isomorphism problem) for graphs. It is crucial to use the same subgraph-isomorphism, canonical-labeling and graph-representation components for both algorithms, in order to have an accurate comparison. As we stated, we choose to use TID list for DSPM just like gSpan and not list of embeddings like FFSM [6] or some other kind of list so as to prove that DSPM's efficiency is not an outcome of keeping more informative lists than TID lists but a result of efficient candidate generation and effective exploring and pruning of search space.

Synthetic Graph-Sets. The first part of the experiments included testing synthetic data sets. The datasets generator, provided by Kuramochi [2,12], is controlled by a set of parameters and outputs a set of synthetic graphs. Figure 8 shows the overall running time of DSPM and gSpan with *minSup*=0.01 over different graph-sets with varied number of labels. All graph-sets are with D = 40000 transactions; The average size of transactions, in terms of the number of edges is T=20; The average size of potentially frequent kernels is I=15; and the number of potentially frequent kernels is L= 1000.

Both algorithms have support method that scans the dataset each iteration to determine the support of a set of candidates (which share a common kernel). Let's assume that an access to dataset to count support for a set of candidates considered to be one *db-access*. Each db-access results with many subgraphs isomorphism for measuring support value (which might also cost with an access to physical memory for big dataset). Then the number of times gSpan access database is equal to the number of discovered frequent patterns exactly. Each time it finds a frequent pattern, gSpan tries to extend it and therefore another db-access to count. DSPM on the other hand explores recursively the search space two

steps a head and that is why the number of db-access it needs is no more than the number of frequent-patterns-nodes in tree-space without the leaves. In fact it is even less because it can prune a branch in the search space without any db-access thanks to FAM pruning it applies in advance. Figure 9 shows the number of db-accesses we count for each one of the tests from Figure 8.

Chemical Compound Datasets. Table 1(1) shows the running time of DSPM and gSpan over chemical compound dataset for predictive Toxicology Evaluation[1] which was also tested by [3,6]. Table 1(2) shows the number of candidates that was generated by DSPM compared to the number of candidates that was reported by FSG [6] for the same dataset with the same support-thresholds. The main inefficiently of FSG results from the special care it gives to each candidate in the generation phase and from executing support-counting per candidate. FSG uses in addition intersection of *tid*-lists to prune even further the number of candidates, but as comes out from table 1(2), the pruning technique of DSPM is almost as strong as the one of FSG whereas DSPM uses trivial operation for that purpose, with least cost.

5. CONCLUSIONS

We formulated our frequent graph mining framework in terms of reverse depth search and a prefix based lattice which is also applicable to other known types of patterns. We suggested the DSPM algorithm among many others possible algorithms that can be developed for making full use of prefix based lattice properties. DSPM algorithm uses several new techniques for graph mining which can be adapted rather easily by the predecessor algorithms such as effective candidate generation, fast anti-monotone pruning and mass support counting for a large set of candidates in a single pass. As is shown in our experiments DSPM adopted successfully ideas from two inherently different approaches for pattern mining (DFS and BFS) with no compromising over the running time, and with better results over the best known algorithm gSpan.

Acknowledgement. We thank Mr. Michihiro Kuramochi and Dr. George Karypis from University of Minnesota for providing the synthetic data generator.

6. REFERENCES

[1] A. Inokuchi, T. Washio, and H. M otoda. An Apriori-based algorithm for mining frequent substructures from graph data. In PKDD'00.

[2] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. Technical report, 2002. http://www.cs.umn.edu/~kuram/papers/fsg-long.pdf.

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

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

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

[6] Jun Huan, Wei Wang, Jan Prins. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In. ICDM'03.

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

[8] M. J. Zaki. Efficiently mining frequent trees in a forest. In SIGKDD'02.

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

[10] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In VLDB'94, pages 487-499, Sept. 1994.

[11] N. Vanetik, E. Gudes. Mining Frequent Labeled and Partially Labeled Graph Patterns. In ICDE'04.

[12] M. Kuramochi and G. Karypis, Finding Frequent Patterns in a Large Sparse Graph, SIAM International Conference on Data Mining, 2004.

[13] Jiawei Han and Micheline Kamber, "Data mining Concepts and Techniques", Morgan Kaufman Publications, 2001

[14] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In ICDE, pages 443--452, 2001.

[1] http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE