Appears in the 1st IEEE Conference on Data Mining (2001)

Frequent Subgraph Discovery*

Michihiro Kuramochi and George Karypis

Department of Computer Science/Army HPC Research Center
University of Minnesota
4-192 EE/CS Building, 200 Union St SE
Minneapolis, MN 55455

Technical Report 01-028

{kuram, karypis}@cs.umn.edu

Last updated on August 7, 2001

Abstract

Over the years, frequent itemset discovery algorithms have been used to solve various interesting problems. As data mining techniques are being increasingly applied to non-traditional domains, existing approaches for finding frequent itemsets cannot be used as they cannot model the requirement of these domains. An alternate way of modeling the objects in these data sets, is to use a graph to model the database objects. Within that model, the problem of finding frequent patterns becomes that of discovering subgraphs that occur frequently over the entire set of graphs. In this paper we present a computationally efficient algorithm for finding all frequent subgraphs in large graph databases. We evaluated the performance of the algorithm by experiments with synthetic datasets as well as a chemical compound dataset. The empirical results show that our algorithm scales linearly with the number of input transactions and it is able to discover frequent subgraphs from a set of graph transactions reasonably fast, even though we have to deal with computationally hard problems such as canonical labeling of graphs and subgraph isomorphism which are not necessary for traditional frequent itemset discovery.

1 Introduction

Efficient algorithms for finding frequent itemsets—both sequential and non-sequential—in very large transaction databases have been one of the key success stories of data mining research [2, 1, 26, 12, 3, 24]. We can use these itemsets for discovering association rules, for extracting prevalent patterns that exist in the datasets, or for classification. Nevertheless, as data mining techniques have been increasingly applied to non-traditional domains, such as scientific, spatial and relational datasets, situations tend to occur on which we can not apply existing itemset discovery algorithms, because these problems are difficult to be adequately and correctly modeled with the traditional market-basket transaction approaches.

An alternate way of modeling the various objects is to use undirected labeled graphs to model each one of object entities—items in traditional frequent itemset discovery—and the relation between them. In particular, each vertex of a graph will correspond to an entity and each edge will correspond to a relation between two entities. In this model both vertices and edges may have labels associated with them which are not required to be unique. Using such a graph representation, a problem of finding frequent patterns then becomes that of discovering subgraphs which occur frequently enough over the entire set of graphs.

*This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, and by Army High Performance Computing Research Center contract number DAAH04-95-C-0008. Access to computing facilities was provided by the Minnesota Supercomputing Institute.

Modeling objects using graphs allows us to represent arbitrary relations among entities. For example, we can convert a basket of items into a graph, or more specifically a clique, whose vertices correspond to the basket's items, and all the items are connected to each other via an edge. Vertex labels correspond to unique identifiers of items, an edge between two vertices u and v represents the coexistence of u and v, and each edge has a label made of the two vertex labels at its both ends. Subgraphs that occur frequently over a large number of baskets will form patterns which include frequent itemsets in the traditional sense when the subgraphs become cliques. The key advantage of graph modeling is that it allows us to solve problems that we could not solve previously. For instance, consider a problem of mining chemical compounds to find recurrent substructures. We can achieve that using a graph-based pattern discovery algorithm by creating a graph for each one of the compounds whose vertices correspond to different atoms, and whose edges correspond to bonds between them. We can assign to each vertex a label corresponding to the atom involved (and potentially its charge), and assign to each edge a label corresponding to the type of the bond (and potentially information about their relative 3D orientation). Once these graphs have been created, recurrent substructures across different compounds become frequently occurring subgraphs.

1.1 Related Work

Developing algorithms that discover all frequently occurring subgraphs in a large graph database is particularly challenging and computationally intensive, as graph and subgraph isomorphisms play a key role throughout the computations.

The power of using graphs to model complex datasets has been recognized by various researchers in chemical domain [21, 20, 7, 5], computer vision [15, 16], image and object retrieval [6, 9], and machine learning [13, 4, 23]. In particular, Dehaspe et al. [7] applied Inductive Logic Programming (ILP) to obtain frequent patterns in the toxicology evaluation problem [21]. ILP has been actively used for predicting carcinogenesis [20], which is able to find all frequent patterns that satisfy a given criteria. It is not designed to scale to large graph databases, however, and they did not report any statistics regarding the amount of computation time required. Another approach that has been developed is using a greedy scheme [23, 13] to find some of the most prevalent subgraphs. These methods are not complete, as they may not obtain all frequent subgraphs, although they are faster than the ILP-based methods. Furthermore, these methods can also perform approximate matching when discovering frequent patterns, allowing them to recognize patterns that have slight variations.

Recently, Inokuchi et al. [14] presented a computationally efficient algorithm called AGM, that can be used to find all frequent *induced* subgraphs in a graph database that satisfy a certain minimum support constraint. A subgraph $G_s = (V_s, E_s)$ of $G = (V, E)$ is *induced* if E_s contains all the edges of E that connect vertices in V_s. AGM finds all frequent induced subgraphs using an approach similar to that used by Apriori [2], which extends subgraphs by adding one vertex at each step. Experiments reported in [14] show that AGM achieves good performance for synthetic dense datasets, and it required 40 minutes to 8 days to find all frequent induced subgraphs in a dataset containing 300 chemical compounds, as the minimum support threshold varied from 20% to 10%.

1.2 Our Contribution

In this paper we present a new algorithm, named FSG, for finding all connected subgraphs that appear frequently in a large graph database. Our algorithm finds frequent subgraphs using the same level-by-level expansion adopted in Apriori [2]. The key features of FSG are the following: (1) it uses a sparse graph representation which minimizes both storage and computation, (2) it increases the size of frequent subgraphs by adding one edge at a time, allowing to generate the candidates efficiently, (3) it uses simple algorithms of canonical labeling and graph isomorphism which work efficiently for small graphs, and (4) it incorporates various optimizations for candidate generation and counting which allow it to scale to large graph databases.

We experimentally evaluated FSG on a large number of synthetic graphs, that were generated using a framework similar to that used for market-basket transaction generation [2]. For problems in which a moderately large number of different types of entities and relations exist, FSG was able to achieve good performance and to scale linearly with the database size. In fact, FSG found all the frequent connected subgraphs in less than 500 seconds from a synthetic dataset consisting of 80000 graphs with a support threshold of 2%. For problems where the number of edge and vertex labels was small, the performance of FSG was worse, as the exponential complexity of graph isomorphism dominates the overall performance. We also evaluated the performance of FSG on the same chemical compound dataset used by AGM. Our results show that FSG is able to find all the frequent connected subgraphs using a 6.5% minimum support in 600 seconds.

2 Frequent Subgraph Discovery

In our problem setting, we have a dataset of transactions D. Each transaction $t \in D$ is a labeled, or colored, undirected graph[1]. Edges and vertices have their labels, or colors. Given a minimum support σ%, we would like to find all *connected* undirected subgraphs that frequently occur in at least $\sigma|D|$ transactions. Table 1 shows the notation we use.

Table 1: Notation

Notation	Description
D	A dataset of graph transactions
t	A transaction of a graph in D
k-(sub)graph	A (sub)graph with k edges
g^k	A k-subgraph
C^k	A set of candidates with k edges
F^k	A set of frequent k-subgraphs
$\mathrm{cl}(g^k)$	A canonical label of a k-graph g^k

The key restriction in our problem statement is that we are finding only subgraphs that are connected. The motivation is primarily that the resulting frequent subgraphs will be encapsulating relations (or edges) between some of entities (or vertices) of various objects. Within this context, connectivity is a natural property of frequent patterns. An additional benefit of this restriction is that it reduces the complexity of the problem, as we do not need to consider disconnected combinations of frequent connected subgraphs.

In developing our frequent subgraph discovery algorithm, we decided to follow the structure of the algorithm Apriori used for finding frequent itemsets [2], because it achieves the most effective pruning compared with other algorithms such as *GenMax*, dEclat [26] and Tree Projection [1].

The high level structure of our algorithm FSG is shown in Algorithm 1. Edges in the algorithm correspond to items in traditional frequent itemset discovery. Namely, as these algorithms increase the size of frequent itemsets by adding a single item at a time, our algorithm increases the size of frequent subgraphs by adding an edge one by one. FSG initially enumerates all the frequent single and double edge graphs. Then, based on those two sets, it starts the main computational loop. During each iteration it first generates candidate subgraphs whose size is greater than the previous frequent ones by one edge (Line 5 of Algorithm 1). Next, it counts the frequency for each of these candidates, and prunes subgraphs that do no satisfy the support constraint (Lines 7–11). Discovered frequent subgraphs satisfy the downward closure property of the support condition, which allows us to effectively prune the lattice of frequent subgraphs.

In Section 2.1, we briefly review some background issues regarding graphs. Section 2.2 contains details of candidate generation with pruning and Section 2.3 describes frequency counting in FSG.

2.1 Graph Representation, Canonical Labeling and Isomorphism

2.1.1 Sparse Graph Representation

Our algorithm uses sparse graph representation to store input transactions, intermediate candidates and frequent subgraphs. This representation saves memory when input transaction graphs are sparse, and speeds up computation.

[1]The algorithm presented in this paper can be easily extended to directed graphs.

Algorithm 1 fsg(D, σ) (Frequent Subgraph)

```
1: $F^1 \leftarrow$ detect all frequent 1-subgraphs in $D$
2: $F^2 \leftarrow$ detect all frequent 2-subgraphs in $D$
3: $k \leftarrow 3$
4: while $F^{k-1} \neq \emptyset$ do
5:    $C^k \leftarrow$ fsg-gen$(F^{k-1})$
6:    for each candidate $g^k \in C^k$ do
7:       $g^k$.count $\leftarrow 0$
8:       for each transaction $t \in D$ do
9:          if candidate $g^k$ is included in transaction $t$ then
10:            $g^k$.count $\leftarrow g^k$.count $+ 1$
11:   $F^k \leftarrow \{g^k \in C^k \mid g^k\text{.count} \geq \sigma|D|\}$
12:   $k \leftarrow k + 1$
13: return $F^1, F^2, \ldots, F^{k-2}$
```

2.1.2 Canonical Labeling

Because we deal with graphs, not itemsets, there are many differences between our algorithm and the traditional frequent itemset discovery. A difference appears when we try to sort frequent objects. In the traditional frequent itemset discovery, we can sort itemsets by lexicographic ordering. Clearly this is not applicable to graphs. To get total order of graphs we use canonical labeling. A canonical label is a unique *code* of a given graph [18, 10]. A graph can be represented in many different ways, depending on the order of its edges or vertices. Nevertheless, canonical labels should be always the same no matter how graphs are represented, as long as those graphs have the same topological structure and the same labeling of edges and vertices. By comparing canonical labels of graphs, we can sort them in a unique and deterministic way, regardless of the representation of input graphs. We denote a canonical label of a graph g by cl(g). It is easy to see that computing canonical labels is equivalent to determining isomorphism between graphs, because if two graphs are isomorphic with each other, their canonical labels must be identical. Both canonical labeling and determining graph isomorphism are not known to be either in P or in NP-complete [10]. A naive way of determining a canonical label is to use a *flattened* representation of the adjacency matrix of a graph. Namely, by concatenating rows or columns of an adjacency matrix one after another we construct a list of integers. By regarding this list of integers as a string, we can obtain total order of graphs by lexicographic ordering. To compute a canonical label of a graph, we have to try all the permutations of its vertices to see which order of vertices gives the minimum adjacency matrix. To narrow down the search space, we first partition the vertices by their degrees and labels, which is a well-known technique called vertex invariants [18]. Then, we try all the possible permutations of vertices inside each partition.

Let us take an example to see how we can reduce the search space of canonical labeling with vertex invariants. Suppose we have a graph of size 3 as shown in Figure 1. Let a, b, c and d denote vertex identifiers, not labels. Two edges of g^3 are labeled with e_0, and the other has a label e_1. Vertices a, b and d have the same label v_0, and only c is labeled with v_1. Assume a canonical label of an adjacency matrix is a string formed by concatenating columns in the upper triangle of an adjacency matrix from left to right. Suppose the following is the initial adjacency matrix of the graph g^3.

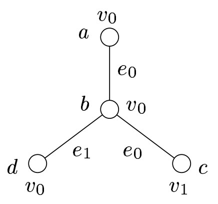


Figure 1: Sample graph g^3

id	a	b	c	d
label	v_0	v_0	v_1	v_0
a	0	e_0	0	0
b	e_0	0	e_0	e_1
c	0	e_0	0	0
d	0	e_1	0	0

By looking at each vertex degree, we can partition them into two groups, one is for degree 1 and the other for degree 2. Vertices a, c and d belong to the first, and b to the second.

id	a	c	d	b
label	v_0	v_1	v_0	v_0
partition		0		1
a	0	0	0	e_0
c	0	0	0	e_0
d	0	0	0	e_1
b	e_0	e_0	e_1	0

Next, by the vertex labels, we can split the first partition into two again, because $v_0 < v_1$ if we compare "v_0" with "v_1" as strings.

id	d	a	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
d	0	0	0	e_1
a	0	0	0	e_0
c	0	0	0	e_0
b	e_1	e_0	e_0	0

There is no further partitioning possible by this simple vertex invariant scheme based on degrees and labels. Thus, we will exhaustively test all the possible permutations of vertices within each partition, and obtain two different permutations of the vertices as shown below. The matrix at the right gives a label of "$000e_0e_1e_0$", while the left one has a label of "$000e_1e_0e_0$".

id	d	a	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
d	0	0	0	e_1
a	0	0	0	e_0
c	0	0	0	e_0
b	e_0	e_1	e_0	0

id	a	d	c	b
label	v_0	v_0	v_1	v_0
partition	0		1	2
a	0	0	0	e_0
d	0	0	0	e_1
c	0	0	0	e_0
b	e_0	e_1	e_0	0

Because $e_0 < e_1$ and "$000e_0e_1e_0$" < "$000e_1e_0e_0$" by string comparison, the label of the right matrix becomes canonical and its adjacency matrix is the canonical representation of g^3, that is, $\mathrm{cl}(g^3) = 000e_0e_1e_0$. By partitioning based on vertex invariants, we only tried 2 permutations in the last step, although the total number of permutations for 4 vertices was $4! = 24$.

Suppose we have a graph with M vertices. By vertex invariants, also suppose we can create N partitions of the vertices, and each partition size is given by p_i for $i = 1, 2, \ldots, N$. Clearly $\sum_{i=1}^{N} p_i = M$. Then, the reduced search space becomes $\prod_{i=1}^{N}(p_i!)$, although the original was $M!$. Of course, vertex invariants do not asymptotically change the computational complexity of canonical labeling [10]. For example, if a given graph is regular, we can not create fine partitions and vertex invariants do not reduce the search space.

2.1.3 Isomorphism

In our algorithm, we need to solve both graph isomorphism and subgraph isomorphism. Graph isomorphism is a problem to determine whether given two graphs g_1 and g_2 are isomorphic, namely, to find a mapping from a set of vertices to another set. Automorphism is a special case of graph isomorphism where $g_1 = g_2$, which means to find a mapping from a graph to itself. Subgraph isomorphism is to find an isomorphism between g_1 and a subgraph of g_2. In other words, it is to determine if a graph is included in the other larger graph. A well-known algorithm for subgraph isomorphism is proposed in [22]. As suggested in [10], graph isomorphism can be directly solved in practice, although it is not known to be either in P or in NP-complete. On the other hand, subgraph isomorphism has been proved to be in NP-complete [11]. Thus, there is no scalable algorithm to solve it. When the size of graphs is small such as 10 vertices or less, however, it is also known that subgraph isomorphism can be feasible even with a simple exhaustive search [10, 22].

A natural way to solve graph isomorphism is, starting from a single vertex in one graph, to try to find a mapping to one of the vertices in the other graph, that is consistent with the labeling. Then, we keep the same process by adding vertices one by one until either we find a complete mapping or we end up with exhausting the search space. When we seek for the next mapping, we have to be careful to keep the consistency of edge and vertex labels. We can reduce the search space more if there are more labels are assigned to edges and vertices, which leads to restriction against mapping. This approach can solve both graph and subgraph isomorphism.

2.2 Candidate Generation

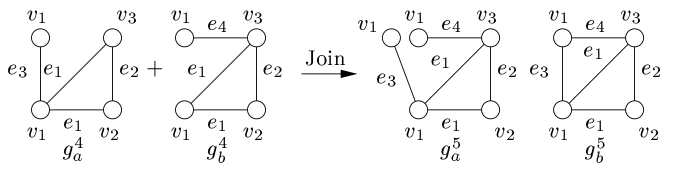
In the candidate generation phase, we create a set of candidates of size $k + 1$, given frequent k-subgraphs. Candidate subgraphs of size $k + 1$ are generated by joining two frequent k-subgraphs. In order for two such frequent k-subgraphs to be eligible for joining they must contain the same $(k - 1)$-subgraph. We will refer to this common $(k - 1)$-subgraph among two k-frequent subgraphs as their *core*.

Unlike the joining of itemsets in which two frequent k-size itemsets lead to a unique $(k + 1)$-size itemset, the joining of two subgraphs of size k can lead to multiple subgraphs of size $k+1$. This is due to three reasons. First, the resulting two $(k + 1)$-subgraphs produced by the joining may differ in a vertex that has the same label in both k-subgraphs. Figure 2(a) is such an example. This pair of graphs g^4_a and g^4_b generates two different candidates g^5_a and g^5_b. The second reason is because a core itself may have multiple automorphisms and each automorphism can lead to a different $(k + 1)$-candidate. An example for this case is shown in Figure 2(b), in which the core—a square of 4 vertices labeled with v_0—has more than one automorphism which result in 3 different candidates of size 6. Finally, two frequent subgraphs may have multiple cores as depicted by Figure 2(c).

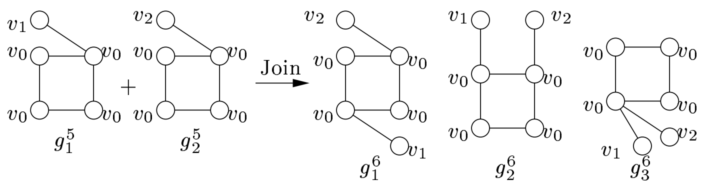
The overall algorithm for candidate generation is shown in Algorithm 2. For each pair of frequent subgraphs that share the same core, the fsg-join is called at Line 6 to generate all possible candidates of size $k + 1$. For each of the candidates, the algorithm first checks if they are already in C^{k+1}. If they are not, then it verifies if all its k-subgraphs are frequent. If they are, fsg-join then inserts it into C^{k+1}, otherwise it discards the candidate (Lines 7–16). The algorithm uses canonical labeling to efficiently check if a particular subgraph is already in C^{k+1} or not.

The key computational steps in candidate generation are (1) core identification, (2) joining, and (3) using the downward closure property of a support condition to eliminate some of generated candidates. A straightforward way of implementing these tasks is to use subgraph isomorphism, graph automorphism and canonical labeling with binary search, respectively. The amount of computation required by the first step, however, can be substantially reduced by keeping some information from the lattice of frequent subgraphs. Particularly, if for each frequent k-subgraph we store the canonical labels of its frequent $(k - 1)$-subgraphs, then the cores between two frequent subgraphs can be determined by simply computing the intersection of these lists. Also to speed up the computation of the automorphism step during joining, we save previous automorphisms associated with each core and look them up instead of performing the same automorphism computation again. The saved list of automorphisms will be discarded once C^{k+1} has been generated.

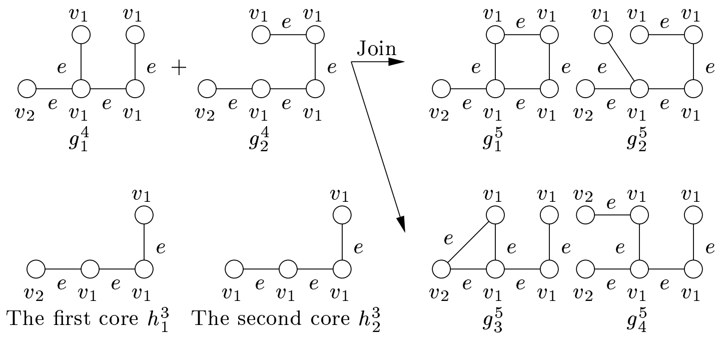
Note we need to perform self join, that is, two graphs g^k_i and g^k_j in Algorithm 2 are identical. It is necessary because, for example, consider transactions without any labels, that is, each transaction in the input is an undirected and unlabeled graph. Then, we will have only one frequent 1-subgraph and one frequent 2-subgraph regardless of a support threshold, because those are the only allowed structures, and edges and vertices do not have labels assigned. From those F^1 and F^2 where $|F^1| = |F^2| = 1$, to generate larger graphs of C^k and F^k for $k \geq 3$, the only way is the self join.


v_1 v_3 e_3 e_1 e_2 v_1 e_1 v_2 g_a^4
+
v_1 e_4 v_3 e_1 e_2 v_1 e_1 v_2 g_b^4
Join
v_1 v_1 e_4 v_3 e_3 e_1 e_2 v_1 e_1 v_2 g_a^5
v_1 e_4 v_3 e_3 e_1 e_2 v_1 e_1 v_2 g_b^5

(a) By vertex labeling


v_1 v_0 v_0 v_0 v_0 g_1^5
+
v_2 v_0 v_0 v_0 v_0 g_2^5
Join
g_1^6
g_2^6
g_3^6

(b) By multiple automorphisms of a single core


v_1 v_1 e e v_2 e v_1 e v_1 g_1^4
+
v_1 e v_1 e v_2 e v_1 e v_1 g_2^4
Join
g_1^5
g_2^5
g_3^5
g_4^5
The first core h_1^3
The second core h_2^3

(c) By multiple cores

Figure 2: Three different cases of candidate joining

2.3 Frequency Counting

Once candidate subgraphs have been generated, FSG computes their frequency. The simplest way of achieving this is for each subgraph to scan each one of the transaction graphs and determine if it is contained or not using subgraph isomorphism. Nonetheless, having to compute these isomorphisms is particularly expensive and this approach is not feasible for large datasets. In the context of frequent itemset discovery by Apriori, the frequency counting is performed substantially faster by building a hash-tree of candidate itemsets and scanning each transaction to determine which of the itemsets in the hash-tree it supports. Developing such an algorithm for frequent subgraphs, however, is challenging as there is no natural way to build the hash-tree for graphs. For this reason, FSG instead uses Transaction ID (TID) lists, proposed by [8, 19, 27, 25, 26]. In this approach for each frequent subgraph we keep a list of transaction identifiers that support it. Now when we need to compute the frequency of g^{k+1}, we first compute the intersection of the TID lists of its frequent k-subgraphs. If the size of the intersection is below the support, g^{k+1} is pruned, otherwise we compute the frequency of g^{k+1} using subgraph isomorphism by limiting our search only to the set of transactions in the intersection of the TID lists.

Algorithm 2 fsg-gen(F^k) (Candidate Generation)

```
1:  $C^{k+1} \leftarrow \emptyset$;
2:  for each pair of $g_i^k, g_j^k \in F^k, i \leq j$ such that $\text{cl}(g_i^k) \leq \text{cl}(g_j^k)$ do
3:    for each edge $e \in g_i^k$ do {create a $(k-1)$-subgraph of $g_i^k$ by removing an edge $e$}
4:      $g_i^{k-1} \leftarrow g_i^k - e$
5:      if $g_i^{k-1}$ is included in $g_j^k$ then {$g_i^k$ and $g_j^k$ share the same core}
6:        $T^{k+1} \leftarrow$ fsg-join($g_i^k, g_j^k$)
7:        for each $g_j^{k+1} \in T^{k+1}$ do
8:          {test if the downward closure property holds for $g_j^{k+1}$}
9:          flag $\leftarrow$ true
10:         for each edge $f_l \in g_j^{k+1}$ do
11:           $h_l^k \leftarrow g_j^{k+1} - f_l$
12:           if $h_l^k$ is connected and $h_l^k \notin F^k$ then
13:             flag $\leftarrow$ false
14:             break
15:         if flag = true then
16:           $C^{k+1} \leftarrow C^{k+1} \cup \{g^{k+1}\}$
17: return $C^{k+1}$
```

Algorithm 3 fsg-join(g_1^k, g_2^k, h^{k-1}) (Join)

```
1:  $M \leftarrow$ detect all automorphisms of $h^{k-1}$
2:  {determine an edge $e_1 \in g_1^k$ that does not appear in $h^{k-1}$}
3:  $e_1 \leftarrow$ NULL
4:  for each edge $e_i \in g_1^k$ do
5:    if $e_i \notin h^{k-1}$ then
6:      $e_1 \leftarrow e_i$
7:      break
8:  {determine an edge $e_2 \in g_2^k$ that does not appear in $h^{k-1}$}
9:  $e_2 \leftarrow$ NULL
10: for each edge $e_i \in g_2^k$ do
11:   if $e_i \notin h^{k-1}$ then
12:     $e_2 \leftarrow e_i$
13:     break
14: $G \leftarrow$ generate all possible graphs of size $k+1$ from $g_1^k$ and $g_2^k$, using $M$
```

3 Experiments

We performed a set of experiments to evaluate the performance of FSG. There are two types of datasets we used. The first type was synthetically generated, and allowed us to study the performance of FSG under different conditions. The second type contains the molecular structures of chemical compounds, which is used to evaluate the performance of FSG for large graphs.

All experiments were done on 650MHz Intel Pentium III machines with 2GB main memory, running the Linux operating system.

3.1 Synthetic Datasets

For the performance evaluation, we generate synthetic datasets controlled by a set of parameters shown in Table 2. The basic idea behind our data generator is similar to the one used in [2], but simpler.

First, we generate a set of $|L|$ potentially frequent connected subgraphs whose size is determined by Poisson distribution with mean $|I|$. For each frequent connected subgraph, its topology as well as its edge and vertex labels are chosen randomly. It has a weight assigned, which becomes a probability that the

Table 2: Synthetic dataset parameters

Notation	Parameter
$\|D\|$	The total number of transactions
$\|T\|$	The average size of transactions (in terms of the number of edges)
$\|I\|$	The average size of potentially frequent subgraphs (in terms of the number of edges)
$\|L\|$	The number of potentially frequent subgraphs
N	The number of edge and vertex labels

subgraph is selected to be included in a transaction. The weights obey an exponential distribution with unit mean and the sum of the weights of all the frequent subgraphs is normalized to 1. We call this set of $|L|$ frequent subgraphs a *seed pool*. The number of distinct edge and vertex labels is controlled by the parameter N. In particular, N is both the number of distinct edge labels as well as the number of distinct vertex labels.

Next, we generate $|D|$ transactions. The size of each transaction is a Poisson random variable whose mean is equal to $|T|$. Then we select one of the frequent subgraphs already generated from the seed pool, by rolling an $|L|$-sided die. Each face of this die corresponds to the probability assigned to a potential frequent subgraph in the seed pool. If the size of the selected seed fits in a transaction, we add it. If the current size of a transaction does not reach its selected size, we keep selecting and putting another seed into it. When a selected seed exceeds the transaction size, we add it to the transaction for the half of the cases, and discard it and move onto the next transaction for the rest of the half. The way we put a seed into a transaction is to find a mapping so that the overlap between a seed and a transaction is maximized.

In the following experiments, we use the combinations of the parameters shown in Table 3.

Table 3: Parameter settings

Parameter	Values
$\|D\|$	10000
$\|T\|$	$5, 10, 20, 40$
$\|I\|$	$3, 5, 7, 10$
$\|L\|$	200
N	$3, 5, 10, 20, 40$

Table 4 shows the amount of time required by FSG to find all the frequent subgraphs for various datasets in which we changed N, $|I|$, $|T|$, and σ. In all of these experiments, the number of transactions $|D|$ was fixed to 10000 and the number of potential frequent subgraphs $|L|$ was set to 200. If the average transaction size $|T|$ is smaller than that of potential frequent subgraphs $|I|$, we omitted such combinations because we can not generate transactions. In some cases, we aborted computation because the running time was too long or because the main memory was exhausted, which are denoted by dashes in the table.

By looking at the table, we can observe a number of interesting points regarding the performance of FSG for different types of datasets. First, as the number of edge and vertex labels N increases, the amount of time required by FSG decreases. For example, when $\sigma = 2\%$, $N = 3$, $|I| = 3$ and $|T| = 10$, it takes 143 seconds, while the running time drops to 16 seconds for $N = 20$. This is because as the number of edge and vertex labels increases there are fewer automorphisms and subgraph isomorphisms, which leads to fast candidate generation and frequency counting. Also by having more edge and vertex labels, we can effectively prune the search space of isomorphism because they work as constraints when we seek for a mapping of vertices. Second, as the size of the average transaction $|T|$ increases the overall running time increases as well. The relative increase is higher when N is small than when N is large. For example, going from $|T| = 5$ to $|T| = 40$ under the setting of $N = 5$, $|I| = 3$ and $\sigma = 2\%$, the running time increases by a factor of 20, whereas for the same set of parameters when $N = 40$, the increase is only by a factor of 4. The reason for that is again having many edge and vertex labels effectively decreases the number of isomorphisms and the search space. With small N and large $|T|$, we can not narrow down efficiently the search space of subgraph isomorphism for frequency counting and the running time increases drastically. Third, as $|I|$ increases the overall running time also increases. Again the relative increase is smaller for larger values of N and smaller values of $|T|$ by the same reason described above.

To determine the scalability of FSG against the number of transactions we performed an experiment in which we used $|D| = 10000$, 20000, 40000 and 80000 with $|L| = 200$, $|I| = 5$ and $|T|$ ranging from 5 to 40. These results are shown in Figure 3. As we can see from the figure, FSG scales linearly with the number of transactions.

Table 4: Running times in seconds for synthetic data sets. We omitted parameter combinations where $|I| > |T|$, because transaction size is too small for potential frequent subgraphs. A dash in the table means we had to abort the computation for the set of parameters because of either memory exhaustion or taking too long time.

N	$\|I\|$	$\|T\|$	Running Time [sec]	
			$\sigma = 2\%$	$\sigma = 1\%$
2	3	5	18	24
		10	143	434
		20	—	—
		40	—	—
2	5	5	27	52
		10	251	2246
		20	—	—
		40	—	—
2	7	10	557	6203
		20	—	—
		40	—	—
2	10	10	—	—
		20	—	—
3	3	5	12	22
		10	30	40
		20	112	390
		40	5817	—
3	5	5	18	32
		10	51	102
		20	189	736
		40	6110	—
3	7	10	66	4512
		20	1953	—
		40	—	—
3	10	10	8290	—
		20	—	—
5	3	5	10	12
		10	20	25
		20	53	71
		40	196	279
5	5	5	24	44
		10	55	80
		20	124	174
		40	340	617
5	7	10	208	770
		20	772	1333
		40	2531	3143
5	10	10	10914	—
		20	—	—
10	3	5	9	17
		10	16	25
		20	35	40
		40	87	98
10	5	5	10	18
		10	20	51
		20	47	119
		40	188	246
10	7	10	190	816
		20	866	1506
		40	2456	3199
10	10	10	10785	—
		20	—	—
20	3	5	9	16
		10	16	28
		20	34	38
		40	78	85
20	5	5	10	19
		10	20	51
		20	48	117
		40	182	233
20	7	10	193	804
		20	884	1667
		40	2524	3271
20	10	10	10520	—
		20	—	—
40	3	5	20	22
		10	27	44
		20	44	47
		40	84	89
40	5	5	20	28
		10	29	60
		20	55	131
		40	177	234
40	7	10	197	1236
		20	861	5273
		40	2456	9183
40	10	10	9687	—
		20	—	—

3.2 Chemical Compound Dataset

We obtained a chemical dataset from [17]. This was originally provided for the Predictive Toxicology Evaluation Challenge [21], which contains information on 340 chemical compounds in two separated files. The first file named `atoms.pl` contains definitions of atoms in compounds. For example, "atm($d1, d1_1, c, 22, -0.133$)" means that a chemical compound $d1$ has an atom whose identifier is $d1_1$, of element carbon, of type 22 and with partial charge -0.133. The other file `bonds.pl` provides bonding information between atoms. A line in the file "bond($d1, d1_1, d1_2, 7$)", for instance, states that in the compound $d1$ its atoms $d1_1$ and $d1_2$ are connected by a type 7 bond. There are 4 different types of bonds (1,2,3 and 7) and 24 different atoms (As, Ba, Br, C, Ca, Cl, Cu, F, H, Hg, I, K, Mn, N, Na, O, P, Pb, S, Se, Sn, Te, Ti and Zn). Also there are 66

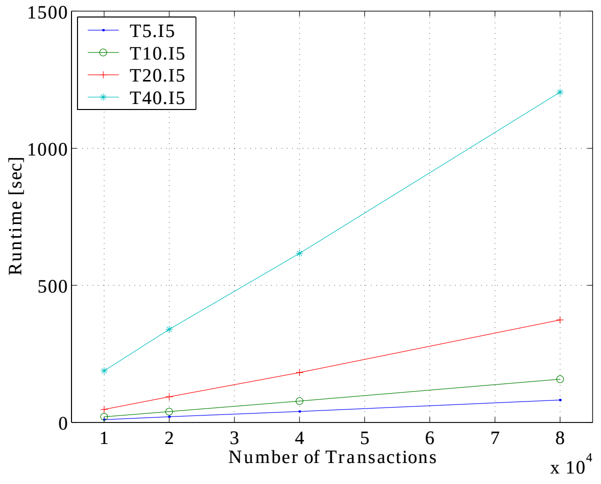


Figure 3: Scalability on the number of transaction

atom types.

We converted the data into graph transactions. Each compound becomes a transaction. Thus, there are 340 transactions in total. Each vertex corresponds to an atom, whose label is made of a pair of the atom element and the atom type. We did not include partial charge to vertex labels because those values were not discretized. Each edge is placed for every bond. Edge label directly corresponds to the bond type. By the conversion, there are 4 edge labels and 66 vertex labels produced in total. The average transaction size was 27.4 in terms of the number of edges, and 27.0 in terms of the number of vertices. Because the number of edges is very close to that of vertices, this dataset is sparse. There are 26 transactions that have more than 50 edges and vertices. The largest transaction contains 214 edges and 214 vertices.

The experimental results by FSG for finding frequent subgraphs are shown in Figure 4. Figure 4(a) shows the running time required for different values of support threshold and Figure 4(b) displays the number of discovered frequent subgraphs on those support levels. With $\sigma = 7\%$, the largest frequent subgraph discovered has 13 vertices.

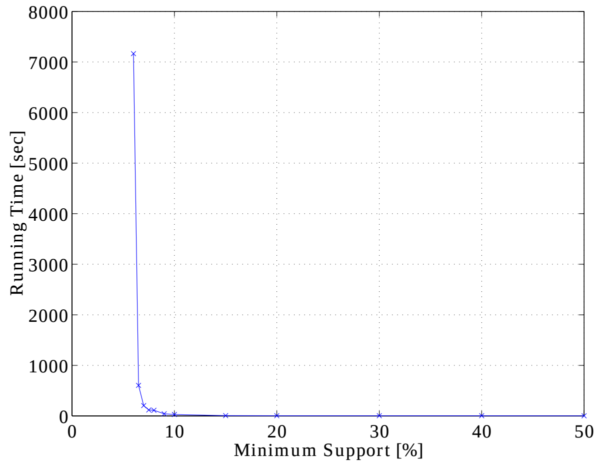
With the support threshold σ below 10%, both the running time and the number of frequent subgraphs increase exponentially. FSG does well even for 7% support as it requires 600 seconds. AGM, a frequent *induced* subgraph discovery algorithm, required about 8 days for 10% and 40 minutes for 20% with almost the same dataset on 400MHz PC [14].

Comparing the performance of FSG on this dataset against those on the synthetic datasets, we can see that it requires more time for this chemical dataset, once we take into account of the difference in the number of transactions. This is because in the chemical dataset, edge and vertex labels have non-uniform distribution. As we decrease the minimum support, larger frequent subgraphs start to appear which generally contain only carbon and hydrogen and a single bonding type. Essentially with $\sigma < 10\%$, this dataset becomes similar to the synthetic datasets where $N = 2$.

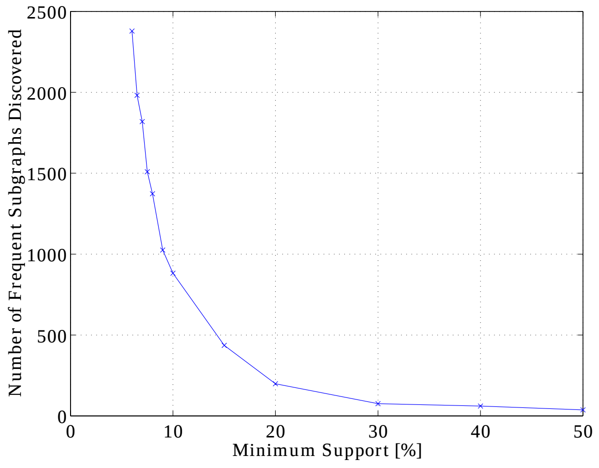
3.3 Summary of Discussions

We summarize the characteristics of FSG performance. First, FSG works better on graph datasets with more edge and vertex labels. During both candidate generation and frequency counting, what FSG essentially does is to solve graph or subgraph isomorphism. Without labels assigned, determining isomorphism of graphs is more difficult to solve, because we can not use labeling information as constraints to narrow down the search space of vertex mapping. We can confirm it by comparing the results in Table 4 with various values of the number of edge and vertex labels, N.

Second, the running time depends heavily on the size of frequent subgraphs to be discovered. If input transactions contain many large frequent patterns such as more than 10 edges, the situation corresponds to



(a) Minimum support σ and running time



(b) Minimum support σ and the number of discovered frequent subgraphs

Figure 4: Performance with the chemical compound dataset

the parameter setting of $|I| = 10$, where FSG will not be likely to finish its computation in a reasonable amount of time. The same thing happened with the chemical dataset with a support threshold less than 10%. If we compare Figure 4(a) and Figure 4(b), we notice the running increases more exponentially than the number of discovered subgraphs does, as we decrease the minimum support. With a lower support criteria, we start getting larger frequent subgraphs and both candidate generation and frequency counting become much heavier. On the other hand, as for the cases of $|I| \leq 5$ in Table 4, FSG runs fast. The result of the chemical dataset is consistent with it. For example, if we use $\sigma = 10\%$ for the chemical dataset, FSG spends 28 seconds to get 882 frequent subgraphs in total. The largest frequent graphs among them have 11 edges, and there are only 10 such frequent 11-subgraphs discovered. Note that AGM discovered about 160000 induced subgraphs under almost the same condition. The reason of this difference is because AGM considers all the possible combinations of frequent induced subgraphs, while FSG only generates frequent connected subgraphs. Once we discovered frequent connected subgraphs, however, we can acquire *disconnected* ones by transforming input graph transactions into basket transactions where items correspond to discovered frequent subgraphs and perform traditional frequent itemset discovery.

Another important factor is the size of a transaction. If the average size of transactions becomes larger, frequency counting by subgraph isomorphism becomes heavier regardless of the size of candidate subgraphs. Traditional frequent itemset finding algorithms are free from this problem. They can perform frequency counting simply by taking the intersection of itemsets and transactions.

As of the number of transactions, FSG requires running time proportional to the size of inputs under the same set of parameters. This is the same as frequent itemset discovery algorithms.

4 Conclusion

In this paper we presented an algorithm, FSG, for finding frequently occurring subgraphs in large graph databases, that can be used to discover recurrent patterns in scientific, spatial, and relational datasets. Our experimental evaluation shows that FSG can scale reasonably well to very large graph databases provided that graphs contain a sufficiently many different labels of edges and vertices.

Acknowledgment

We deeply thank Professor Takashi Washio, Professor Hiroshi Motoda and their research group at the Institute of Scientific and Industrial Research, Osaka University, and Mr. Akihiro Inokuchi at Tokyo Research Laboratory, IBM Japan, Ltd. for providing the source code of AGM and useful comments.

References

[1] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad, and V. Crestana. A tree projection algorithm for generation of large itemsets for association rules. IBM Research Report RC21341, November 1998.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Databases (VLDB)*, pages 487–499. Morgan Kaufmann, September 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th Int. Conf. on Data Engineering (ICDE)*, pages 3–14. IEEE Press, 1995.

[4] C.-W. K. Chen and D. Y. Y. Yun. Unifying graph-matching problem with a practical solution. In *Proceedings of International Conference on Systems, Signals, Control, Computers*, September 1998.

[5] R. N. Chittimoori, L. B. Holder, and D. J. Cook. Applying the SUBDUE substructure discovery system to the chemical toxicity domain. In *Proc. of the 12th International Florida AI Research Society Conference*, pages 90–94, 1999.

[6] V. A. Cicirello. Intelligent retrieval of solid models. Master's thesis, Drexel University, Philadelphia, PA, 1999.

[7] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press, 1998.

[8] B. Dunkel and N. Soparkar. Data organizatinon and access for efficient data mining. In *Proc. of the 15th IEEE International Conference on Data Engineering*, March 1999.

[9] D. Dupplaw and P. H. Lewis. Content-based image retrieval with scale-spaced object trees. In M. M. Yeung, B.-L. Yeo, and C. A. Bouman, editors, *Proc. of SPIE: Storage and Retrieval for Media Databases*, volume 3972, pages 253–261, 2000.

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

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

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

[13] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the SUBDUE system. In *Proc. of the Workshop on Knowledge Discovery in Databases*, pages 169–180, 1994.

[14] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of The 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 13–23, Lyon, France, September 2000.

[15] H. Kälviäinen and E. Oja. Comparisons of attributed graph matching algorithms for computer vision. In *Proc. of STEP-90, Finnish Artificial Intelligence Symposium*, pages 354–368, Oulu, Finland, June 1990.

[16] D. A. L. Piriyakumar and P. Levi. An efficient A* based algorithm for optimal graph matching applied to computer vision. In *GRWSIA-98*, Munich, 1998.

[17] http://oldwww.comlab.ox.ac.uk/oucl/groups/machlearn/PTE/.

[18] R. C. Read and D. G. Corneil. The graph isomorph disease. *Journal of Graph Theory*, 1:339–363, 1977.

[19] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 22–33, May 2000.

[20] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, Berlin, 1997.

[21] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1–6. Morgan-Kaufmann, 1997.

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

[23] K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, 1995.

[24] M. J. Zaki. Fast mining of sequential patterns in very large databases. Technical Report 668, Department of Computer Science, Rensselaer Polytechnic Institute, 1997.

[25] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[26] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute, 2001.

[27] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining,. Technical Report 99-10, Department of Computer Science, Rensselaer Polytechnic Institute, October 1999.