An Efficient Algorithm for Discovering Frequent Subgraphs*

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 02-026

{kuram, karypis}@cs.umn.edu

Last updated on June 27, 2002

Abstract

Over the years, frequent itemset discovery algorithms have been used to find interesting patterns in various application areas. However, as data mining techniques are being increasingly applied to non-traditional domains, existing frequent pattern discovery approach cannot be used. This is because the transaction framework that is assumed by these algorithms cannot be used to effectively model the datasets in these domains. An alternate way of modeling the objects in these datasets is to represent them using graphs. 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, called FSG, for finding all frequent subgraphs in large graph databases. We experimentally evaluate the performance of FSG using a variety of real and synthetic datasets. Our results show that despite the underlying complexity associated with frequent subgraph discovery, FSG is effective in finding all frequently occurring subgraphs in datasets containing over 100,000 graph transactions and scales linearly with respect to the size of the database.

Index Terms Data mining, scientific datasets, frequent pattern discovery, chemical compound datasets.

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, 29, 15, 3, 27]. These itemsets can be used 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,

*This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274 and ACI-0133464 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.

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 the object's entities—items in traditional frequent itemset discovery—and the relation between them. In particular, each vertex of the 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, the problem of finding frequent patterns then becomes that of discovering subgraphs which occur frequently enough over the entire set of graphs.

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 [25, 24, 7, 5], computer vision [18, 19], image and object retrieval [6, 11], and machine learning [16, 4, 26]. In particular, Dehaspe et al. [7] applied Inductive Logic Programming (ILP) to obtain frequent patterns in the toxicology evaluation problem [25]. ILP has been actively used for predicting carcinogenesis [24], 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 [26, 16] 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. [17] 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 [17] 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, called 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 strategy adopted by Apriori [2]. The key features of FSG are the following: (i) it uses a sparse graph representation that minimizes both storage and computation; (ii) it increases the size of frequent subgraphs by adding one edge at a time, allowing it to generate the candidates efficiently; (iii) it incorporates various optimizations for candidate generation and frequency counting which enables it to scale to large graph databases; and (iv) it uses sophisticated algorithms for canonical labeling to uniquely identify the various generated subgraphs without having to resort to computationally expensive graph- and subgraph-isomorphism computations.

We experimentally evaluated FSG on three types of datasets. The first two datasets correspond to various chemical compounds containing over 100,000 transactions and large patterns, and the third type corresponds to various graph datasets that were synthetically generated using a framework similar to that used for market-basket transaction generation [2]. Our results illustrate that FSG can operate on very large graph databases and find all frequently occurring subgraphs in reasonable amount of time. For example, in a dataset containing over 100,000 chemical compounds, FSG can discover all subgraphs that occur in at least 3% of the transactions in approximately fifteen minutes. Furthermore, our detailed evaluation using the synthetically generated graphs show that for datasets that have a moderately large number of different vertex- and/or edge-labels, FSG is able to achieve good performance as the transaction size increases and scales linearly with the database size.

1.3 Organization of the Paper

The rest of the paper is organized as follows. Section 2 defines the graph model that we use, reviews some graph-related definitions, and introduces the notation that is used in the paper. Section 3 formally defines the problem of frequent subgraph discovery and discusses the modeling strengths of the discovered patterns and the challenges associated with finding them in a computationally efficient manner. Section 4 describes in detail the FSG algorithm that we developed for solving the problem of frequent subgraph discovery. Section 5 describes the various optimizations that we developed for efficiently computing the canonical labeling of the patterns. Section 6 provide a detailed experimental evaluation of the FSG algorithm on a large number of real and synthetic datasets. Finally, Section 7 provides some concluding remarks and discusses future research directions in further improving the performance of FSG.

2 Definitions, Assumptions, and Background Information

A graph $G = (V, E)$ is made of two sets, the set of vertices V and the set of edges E. Each edge itself is a pair of vertices, and throughout this paper we assume that the graph is undirected, *i.e.*, each edge is an unordered pair of vertices. Furthermore, we will assume that the graph is *labeled*. That is, each vertex and edge has a label associated with it that is drawn from a predefined set of vertex labels (L_V) and edge labels (L_E). Each vertex (or edge) of the graph is not required to have a unique label and the same label can be assigned to many vertices (or edges) in the same graph. If all the vertices and edges of the graph have the same vertex and edge label assigned to them, we will call this graph *unlabeled*.

Given a graph $G = (V, E)$, a graph $G_s = (V_s, E_s)$ will be a *subgraph* of G if and only if $V_s \subseteq V$ and $E_s \subseteq E$. Also, G_s will be an *induced subgraph* of G if $V_s \subseteq V$ and E_s contains all the edges of E that connect vertices in V_s. A graph is *connected* if there is a path between every pair of vertices in the graph.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if they are topologically identical to each other, that is, there is a mapping from V_1 to V_2 such that each edge in E_1 is mapped to a single edge in E_2 and vice versa. In the case of labeled graphs, this mapping must also preserve the labels on the vertices and edges. An *automorphism* is an isomorphism mapping where $G_1 = G_2$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of *subgraph isomorphism* is to find an isomorphism between G_2 and a subgraph of G_1, *i.e.*, determine whether or not G_2 is included in G_1.

The *canonical label* of a graph $G = (V, E)$, cl(G), is defined to be a unique *code* (*e.g.*, string) that is invariant on the ordering of the vertices and edges in the graph [21, 12]. As a result, two graphs will have the same canonical label if they are isomorphic. Canonical labels are extremely useful as they allow us to quickly (i) compare two graphs, and (ii) establish a complete ordering of a set of graphs in a unique and and deterministic way, regardless of the original vertex and edge ordering. Canonical labels play a critical role in our frequent subgraph discovery algorithm and Section 5 describes the various approaches that we developed for defining what constitutes a canonical label and how to compute them efficiently. However, the problem of determining the canonical label of a graph 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 [12].

3 Frequent Subgraph Discovery—Problem Definition

The input for the frequent subgraph discovery problem is a set of graphs $\boldsymbol{D}$, each of which is an undirected labeled graph[1], and a parameter σ such that $0 < \sigma \leq 1.0$. The goal of the frequent subgraph discovery is to find all *connected* undirected subgraphs that occur in at least $\sigma|\boldsymbol{D}|\%$ of the input graphs. We will refer to each of the graphs in $\boldsymbol{D}$ as a *graph transaction* or simply *transaction* when the context is clear, to $\boldsymbol{D}$ as the (graph) transaction database, and to σ as the *support* threshold.

There are two key aspects in the problem statement above. The first has to do with the fact that we are only interested in subgraphs that are connected. This is motivated by the fact that the resulting frequent subgraphs will be encapsulating relations (or edges) between some of the 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.

The second key element has to do with the fact that we allow the graphs to be labeled, and as discussed in Section 2, each graph (and discovered pattern) can contain vertices and/or edges with the same label. This greatly increases our modeling ability, as it allow us to find patterns involving multiple occurrences of the same entities and relations, but at the same time it makes the problem of finding such frequently curing subgraphs non-trivial. The reason for that is that if we assume that all the vertices and edges in a particular graph have a unique label, then we can easily model each graph as a set of edges, in which each edge (v, u) will have a unique *identifier* derived from the labels of the two vertices and the label of the edge itself. In this representation, each particular edge identifier will occur once, allowing us to model it as a set of items (each identifier becomes a distinct item), and

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

then use existing frequent itemset discovery algorithms to find the frequently occurring subgraphs. Note that these discovered subgraphs will not be connected, but their connectivity can be easily checked during a post-processing phase. However, this transformation from frequent subgraph discovery to frequent itemset discover cannot be performed if the graph contains edges that get mapped to the same edge identifier (*i.e.*, it has at least two edges with the same edge label and incident vertex labels). For this type of problems, any frequent subgraph discovery algorithm needs to correctly identify how a particular subgraph maps to the vertices and edges of each graph transaction, that can only be done by solving many instances of the subgraph isomorphism problem, which has been shown to be in NP-complete [13].

4 FSG—Frequent Subgraph Discovery Algorithm

In developing our frequent subgraph discovery algorithm, we decided to follow the level-by-level structure of the Apriori algorithm used for finding frequent itemsets in market-basket datasets [2]. The motivation behind this choice is the fact that the level-by-level structure of the Apriori algorithm achieves the highest amount of pruning compared with other algorithms such as *GenMax* [29], dEclat [29] and Tree Projection [1].

The high-level structure of our algorithm, called 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 6–11). Discovered frequent subgraphs satisfy the downward closure property of the support condition, which allows us to effectively prune the lattice of frequent subgraphs. The notation used in this algorithm and in the rest of this paper is explained in Table 1.

Algorithm 1 fsg($\boldsymbol{D}, \sigma$) (Frequent Subgraph)

```
1:  F^1 ← detect all frequent 1-subgraphs in D
2:  F^2 ← detect all frequent 2-subgraphs in D
3:  k ← 3
4:  while F^{k-1} ≠ ∅ do
5:    C^k ← fsg-gen(F^{k-1})
6:    for each candidate G^k ∈ C^k do
7:      G^k.count ← 0
8:      for each transaction T ∈ D do
9:        if candidate G^k is included in transaction T then
10:         G^k.count ← G^k.count + 1
11:   F^k ← {G^k ∈ C^k | G^k.count ≥ σ|D|}
12:   k ← k + 1
13: return F^1, F^2, ..., F^{k-2}
```

In the rest of this section we describe how FSG generates the candidates subgraphs during each level of the algorithm and how it computes their frequency.

Table 1: Notation used throughout the paper

Notation	Description
$\boldsymbol{D}$	A dataset of graph transactions
T	A transaction of a graph in $\boldsymbol{D}$
k-(sub)graph	A (sub)graph with k edges
G^k, H^k	A k-(sub)graph
$\boldsymbol{C}^k$	A set of candidates with k edges
$\boldsymbol{C}$	A set of all candidates generated from $\boldsymbol{D}$
$\boldsymbol{F}^k$	A set of frequent k-subgraphs
$\boldsymbol{F}$	A set of all frequent connected subgraphs in $\boldsymbol{D}$

Notation	Description
k^*	The size of the largest frequent subgraph in $\boldsymbol{D}$
$\mathrm{cl}(G^k)$	A canonical label of a k-graph G^k
v_i	A unique identification of the ith vertex
$d(v_i)$	Degree of a vertex v_i
$l(v_i)$	The label of a vertex v_i
$l(e_i)$	The label of an edge e_i
L_E	A set of all edge labels contained in $\boldsymbol{D}$
L_V	A set of all vertex labels contained in $\boldsymbol{D}$

4.1 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 distinct subgraphs of size $k+1$. This can happen for the following three reasons. First, the difference between the shared core and the two subgraphs can be a vertex that has the same label in both k-subgraphs. In this case, the joining of such k-subgraphs will generate two distinct subgraphs of size $k+1$. Figure 1(a) shows such an example. The pair of graphs G_a^4 and G_b^4 generates two different candidates G_a^5 and G_b^5. Second, the core itself may have multiple automorphisms, and each of them can lead to a different $(k+1)$-candidate. In the worst case, when the core is an unlabeled clique, the number of automorphisms is $k!$. An example for this case is shown in Figure 1(b), in which the core—a square of four vertices labeled with a—has four automorphisms resulting in three different candidates of size 6. Third, two frequent subgraphs may have multiple cores as depicted by Figure 1(c). Because every core has one fewer edge, for a pair of two k-subgraphs to be joined, the number of multiple cores is bounded by $k-1$.

The overall algorithm for candidate generation is shown in Algorithm 2. For each pair of frequent subgraphs, fsg-gen start by detecting all the cores shared by the two frequent subgraphs. Then, for each pair of subgraphs and a shared core, fsg-join is called at Line 6 to generate all possible candidates of size $k+1$. Once a candidate is generated, the algorithm first checks if the candidate is already in $\boldsymbol{C}^{k+1}$. If it is not, then fsg-gen verifies if all its k-subgraphs are frequent. If they are, fsg-join then inserts the candidate into $\boldsymbol{C}^{k+1}$, otherwise it discards the candidate (Lines 7–16). Algorithm 3 shows the joining procedure. Given a pair of k-subgraphs, G_1^k and G_2^k, and a core of size $k-1$, first fsg-join determines the two edges, one included only in G_1^k and the other only in G_2^k which are not a part of the core. Next, fsg-join generates all the automorphisms of the core. Finally, for each set of the two edges, the core, and the automorphism, fsg-join integrates the two subgraphs G_1^k and G_2^k into one candidate of size $k+1$. It may generate at most two distinct candidates because of the situation depicted in Figure 1(a).

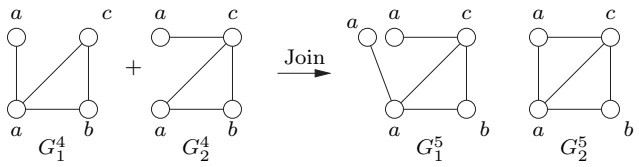
Note that in addition to joining two different subgraphs, we also need to perform self join, that is, the two graphs G_i^k and G_j^k in Algorithm 2 are identical. It is necessary because, for example, consider graph transactions without any labels. Then, we will have only one frequent 1-subgraph and 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 $\boldsymbol{F}^1$ and $\boldsymbol{F}^2$ where $|\boldsymbol{F}^1| = |\boldsymbol{F}^2| = 1$, to generate larger graphs of $\boldsymbol{C}^k$ and $\boldsymbol{F}^k$ for $k \geq 3$, the only way is the self join.

Algorithm 2 fsg-gen($\boldsymbol{F}^k$) (Candidate Generation)

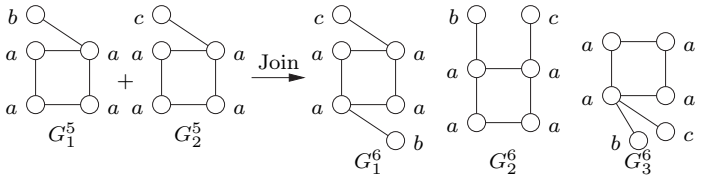
$\boldsymbol{C}^{k+1} \leftarrow \emptyset$
for each pair of $G_i^k, G_j^k \in \boldsymbol{F}^k, i \leq j$ such that $\mathrm{cl}(G_i^k) \leq \mathrm{cl}(G_j^k)$ **do**
  $\boldsymbol{H}^{k-1} \leftarrow \{H^{k-1} \mid$ a core H^{k-1} shared by G_i^k and $G_j^k\}$
  for each core $H^{k-1} \in \boldsymbol{H}^{k-1}$ **do**
    $\{\boldsymbol{B}^{k+1}$ is a set of tentative candidates$\}$
    $\boldsymbol{B}^{k+1} \leftarrow$ fsg-join(G_i^k, G_j^k, H^{k-1})
    for each $G_j^{k+1} \in \boldsymbol{B}^{k+1}$ **do**
      {test if the downward closure property holds}
      flag $\leftarrow$ true
      for each edge $e_l \in G_j^{k+1}$ **do**
        $H_l^k \leftarrow G_j^{k+1} - e_l$
        if H_l^k is connected and $H_l^k \notin \boldsymbol{F}^k$ **then**
          flag $\leftarrow$ false
          break
      if flag = true **then**
        $\boldsymbol{C}^{k+1} \leftarrow \boldsymbol{C}^{k+1} \cup \{G_j^{k+1}\}$
return $\boldsymbol{C}^{k+1}$

Algorithm 3 fsg-join(G_1^k, G_2^k, H^{k-1}) (Join)

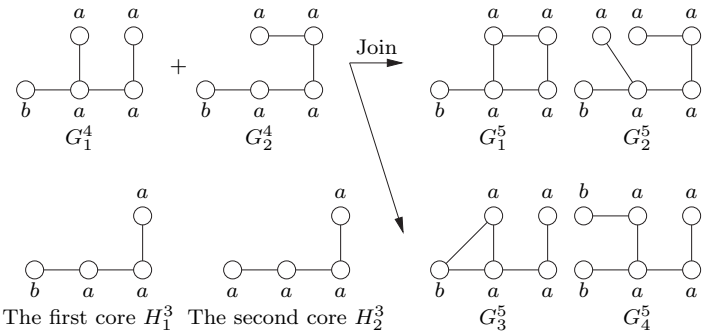
$e_1 \leftarrow$ the edge appears only in G_1^k, not in H^{k-1}
$e_2 \leftarrow$ the edge appears only in G_2^k, not in H^{k-1}
$M \leftarrow$ generate all automorphisms of H^{k-1}
$\boldsymbol{B}^{k+1} = \emptyset$
for each automorphism $\phi \in M$ **do**
  $\boldsymbol{B}^{k+1} \leftarrow \boldsymbol{B}^{k+1} \cup \{$all possible candidates of size $k+1$ created from a set of e_1, e_2, H^{k-1} and $\phi\}$
return $\boldsymbol{B}^{k+1}$



(a) By vertex labeling



(b) By multiple automorphisms of a single core



(c) By multiple cores

Figure 1: Three different cases of candidate joining

4.1.1 Efficient Implementation of Candidate Generation

The key computational steps in candidate generation are (1) core identification, (2) joining, and (3) using the downward closure property of the support condition to eliminate some of the generated candidates. A straightforward way of performing these tasks is as follows. A core between a pair of graphs G_i^k and G_j^k can be identified by creating each of the $(k-1)$-subgraphs of G_i^k by removing each of the edges and checking whether or not this subgraph is also a subgraph of G_j^k. Then, after generating all the possible automorphisms of the detected core, we join the two size k-subgraphs, G_i^k and G_j^k to obtain size $(k+1)$-candidates G_l^{k+1}, by integrating two edges, one only in G_i^k and the other only in G_j^k into the G^{k+1} which are not a part of the core. Finally, for each generated candidate of size $(k+1)$ we can generate each one of the k-size subgraphs by removing the edges and checking to see if they exist in $\boldsymbol{F}^k$. Unfortunately, the above approaches require the solution of various graph and subgraph isomorphism and automorphism problems, making it impractical for large databases and long patterns. To substantially reduce the complexity of candidate generation, FSG takes advantage of the frequent subgraph lattice and the canonical labeling representation of the subgraphs as follows.

The amount of time required to identify the core between two frequent subgraphs 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. The complexity of this approach is quadratic on the number of frequent subgraphs of size k (*i.e.*, $|\boldsymbol{F}^k|$), which can be prohibitively expensive if $|\boldsymbol{F}^k|$ is large. For this reason, for each frequent subgraph of size $k-1$, we maintain a list of child subgraphs of size k. Then, we only need to form every possible pair from the child list of every size $k-1$ frequent subgraph. This reduces the complexity of finding an appropriate pair of subgraphs to the square of the number of child subgraphs of size k. In the rest of the paper, we will refer to this as the *inverted indexing* scheme.

To speed up the computation of the automorphism step during joining, we cache previous automorphisms associated with each core and look them up instead of performing the same automorphism computation again. The saved list of automorphisms is discarded once $\boldsymbol{C}^{k+1}$ has been generated.

FSG uses canonical labeling to substantially reduce the complexity of the checking whether or not a candidate pattern satisfies the downward closure property of the support condition. There are two reasons for us to use canonical labeling for the candidate generation. The first one is we can use the canonical label repeatedly for comparison without the recalculation, once we obtain it. The second reason is, by regarding canonical labels as strings, we get the total order of graphs. Then it is easy for us to sort them in an array and to index one by binary search efficiently. The algorithm used for the canonical labeling is described in Section 5.

4.2 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 graph transactions in the input dataset 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 identifier (TID) lists, proposed by [10, 23, 30, 28, 29]. 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. The advantages of this approach are two-fold. First, in the cases where the intersection of the TID lists is bellow the minimum support level, we are able to prune the candidate subgraph without performing any subgraph isomorphism computations. Second, when the intersection set is sufficiently large, we only need to compute subgraph isomorphisms for those graphs that can potentially contain the candidate subgraph and not for all the graph transactions.

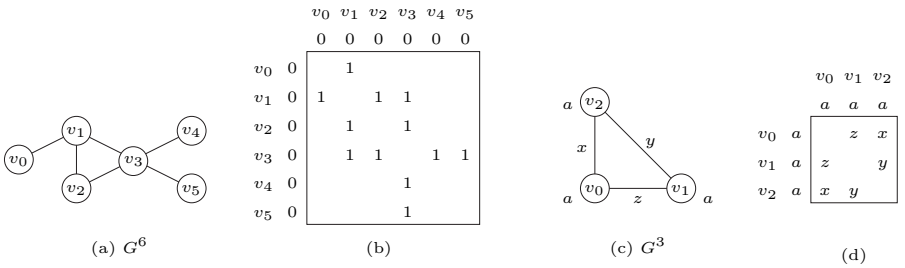
However, the computational advantages of the TID lists come at the expense of higher memory requirements. In particular, when FSG is working on finding the frequent patterns of size $(k+1)$, it needs to store in memory the TID lists for all frequent patterns of size k. FSG can be easily extended to work in cases where the amount of available memory is not sufficient for storing the TID lists of a particular level by adopting a depth-first approach for frequent pattern generation. Starting from a subset of subgraphs of size k, without generating all the rest of

size k subgraphs, we can proceed to larger size. In this way, we may not be able to get the same effect of pruning based on the downward closure property. Nevertheless, it is beneficial in terms of memory usage because at each phase we keep smaller number of subgraphs and their associated TID lists. This approach corresponds to vertical frequent itemset mining such as [28].

5 Canonical Labeling

The FSG algorithm relies on canonical labeling to efficiently perform a number of operations such as checking whether or not a particular pattern satisfies the downward closure property of the support condition, or finding whether a particular candidate subgraph has already been generated or not. Developing algorithms that can efficiently compute the canonical labeling of the various candidate and frequent subgraphs is critical to ensure that FSG can scale to very large graph datasets.

Recall from Section 2 that the canonical label of a graph is nothing more than a *code* (*i.e.*, a sequence of bits, a string, or a sequence of numbers) that uniquely identifies the graph such that if two graphs are isomorphic to each other, they will be assigned the same code. A simple way of assigning a code to a graph is to convert its adjacency matrix representation into a linear sequence of symbols. For example, this can be done by concatenating the rows or the columns of the graph's adjacency matrix one after another to obtain a sequence of zeros and ones or a sequence of vertex and edge labels. This is illustrated in Figure 2 in which an unlabeled graph G^6 is shown in Figure 2(a) and a labeled graph G^3 that has both vertex and edge labels is shown in Figure 2(c). Their adjacency matrices are shown in Figures 2(b) and (d) respectively. The symbol v_i in the figure is a vertex ID, not a vertex label, and blank elements in the adjacency matrix means there is no edge between the corresponding pair of vertices[2]. An adjacency matrix in Figure 2(b) for G^6 produces a code "00000101011000100010" by concatenating a list of six vertex labels "000000" and the upper triangle part of the columns one after another, "1", "01", "011", "0001" and "00010", assuming that each vertex has a label "0" and each edge has a label "1". Likewise, an adjacency matrix in (d) for the labeled graph G^3 produces a code "*aaazxy*". The first three letters "*aaa*" are the vertex labels in the order that they appear in the adjacency matrix. The next three letters "*zxy*" are again created from the upper triangle of the matrix by concatenating columns. Note that the labels of the vertices are incorporated in the code by listing them first prior to listing the contents of their columns.



(a) G^6 (b) (c) G^3 (d)

Figure 2: Simple examples of codes

Unfortunately, we can not directly use the codes derived from the adjacency matrices in Figures 2(b) or (d) as their canonical labels because they are dependent on the order of the vertices, *i.e.*, the permutation of the rows and the columns of an adjacency matrix. Different orderings of the vertices will give rise to different codes, which

[2]This notation will be used in the rest of the section.

violates the requirement for the canonical labels to be invariant with respect to isomorphism. One way to obtain isomorphism-invariant codes is to try every possible permutation of the vertices and its corresponding adjacency matrix, and to choose the ordering which gives lexicographically the largest, or the smallest code [21, 12]. The vertex ordering of the largest codes for the two graphs G^6 and G^3 are shown in Figure 3. The codes of the adjacency matrices in Figures 3(a) and (b) are "00000011110010000 1000", and "$aaazyx$", respectively. We can use these lexicographically largest codes as the canonical labels of the graphs.

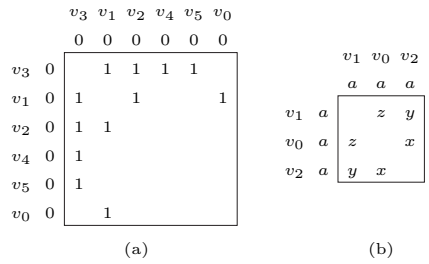


Figure 3: Canonical adjacency matrices

If the graph contains $|V|$ vertices, then the time complexity of the above method for determining its canonical label is in $O(|V|!)$, as we need to check each one of the $|V|!$ permutations of the vertices, before we can select the maximum code. Consequently, this canonical labeling approach is all but impractical for moderate size graphs. Fortunately, the complexity of finding a canonical labeling of a graph can, in practice, be substantially reduced by using the concept of *vertex invariants* that are described in the rest of this section.

5.1 Vertex Invariants

Vertex invariants [21] are some attributes or properties assigned to a vertex which do not change across isomorphism mappings. An example of such an isomorphism-invariant property is the degree or label of a vertex, which remain the same regardless of the mapping (*i.e.*, vertex ordering).

Vertex invariants can be used to reduce the amount of time required to compute a canonical label as follows. Given a graph, the vertex invariants can be used to partition the vertices of the graph into equivalence classes such that all the vertices assigned to the same partition have the same values for the vertex invariants. Because vertex invariants remain the same regardless of the ordering of edges and vertices, we can always create the same partitions no matter how the vertices of the input graph are ordered. Now, using these partitions we can define the canonical label of a graph to be the lexicographically largest code obtained by concatenating the columns of the upper triangular adjacency matrix (as it was done earlier), over all possible permutations of the vertices subject to the constraint that the vertices of each one of the partitions are numbered consecutively. Thus, the only modification over our earlier definition is that instead of maximizing over all permutations of the vertices, we only maximize over those permutations that keep the vertices in each partition together. Note that two graphs that are isomorphic will lead to the same partitioning of the vertices and they will be assigned the same canonical label.

If m is the number of partitions created by using vertex invariants, containing $p_1, p_2, \ldots, p_m$ vertices, then the number of different permutations that we need to consider is $\prod_{i=1}^{m}(p_i!)$, which can be substantially smaller than the $|V|!$ permutations required by the earlier approach. Of course, vertex invariants do not asymptotically change the computational complexity of canonical labeling [12], as we may not be able to compute a fine-grain partitioning of graph's vertices, which can happen for regular graphs.

Nevertheless, for most real-life graphs we can substantially reduce the amount of time required for canonical labeling provided that we can discover vertex invariants that lead to a large number of partitions. Toward this goal we have developed and incorporated in FSG three types of vertex invariants that utilize information about the degrees and labels of the vertices, the labels and degrees of their adjacent vertices, and information about their adjacent partitions. As our experimental results in Section 6 will show, these vertex invariants lead to substantial reductions in the overall runtime of canonical labeling and FSG. In the rest of this section we describe these vertex invariants and illustrate them using a non-trivial example.

5.1.1 Vertex Degrees and Labels

The first set of vertex invariants that we have incorporated in FSG's canonical labeling algorithm is based on the labels and degrees of the vertices. In this approach, the vertices are partitioned into disjointed groups such that each partition contains vertices with the same label and the same degree, and those partitions are sorted by the vertex degree and label in each partition.

Figure 4 illustrates the partitioning induced by this set of invariants for the graph of size four shown in Figure 4(a). Vertices v_0, v_1 and v_3 have the same vertex label a, and only v_2 has b. Also every edge has the same edge label E, except for the one with F between v_0 and v_1. The adjacency matrix of the graph, ordered in increasing lexicographical order on the vertex ID is shown in Figure 4(b). Based on their degree and their labels, the vertices can be partitioned into three groups $p_0 = \{v_1\}$, $p_1 = \{v_0, v_3\}$ and $p_2 = \{v_2\}$. The partition p_0 contains a vertex of degree three, p_1 contains two vertices whose degree is one and label is a, and p_2 is a partition of a vertex of degree one with the label b. This partitioning is illustrated in Figure 4(c). Also, Figure 4 shows the adjacency matrix corresponding to the partition-constrained permutation that leads to the canonical label of the graph. Using the partitioning based on vertex invariants, we had to try only $1! \times 2! \times 1! = 2$ permutations, although the total number of permutations for four vertices is $4! = 24$.

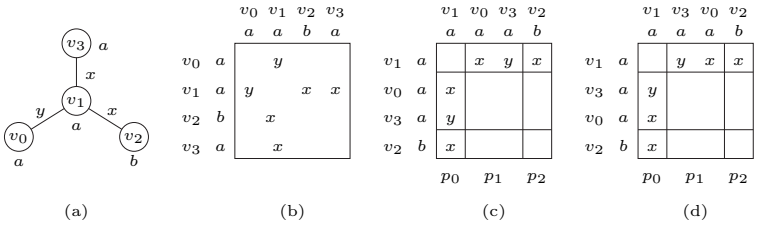


Figure 4: A sample graph of size three and its adjacency matrices

5.1.2 Neighbor Lists

So far, we only used the degree of a vertex to capture information about its adjacency structure. However, we can create better invariants by incorporating information about the labels of the edges incident on each vertex, the degrees of the adjacent vertices, and their labels. In particular, we describe an adjacent vertex v by a tuple $(l(e), d(v), l(v))$ where $l(e)$ is the label of the incident edge e, $d(v)$ is the degree of the adjacent vertex v, and $l(v)$ is its vertex label. Now, for each vertex u, we construct its neighbor list $\mathrm{nl}(u)$ that contains the tuples for each one of its adjacent vertices. Using these neighbor lists, we then partition the vertices into disjoint sets such that two vertices u and v will be in the same partition if and only if $\mathrm{nl}(u) = \mathrm{nl}(v)$. Note that this partitioning is performed within the partitions already computed by the previous set of invariants.

Figure 5 illustrates the partitioning produced by also incorporating the *neighbor list* invariant. In particular, Figure 5(a) shows a graph with seven edges, Figure 5(b) shows the partitioning produced by the vertex degrees and labels, and Figure 5(c) shows the partitioning that is produced by also incorporating neighboring lists. The neighbor lists are shown in Figure 5(d). In this particular example, we were able to reduce the search space size of the canonical labeling from $4! \times 2!$ to $2!$.

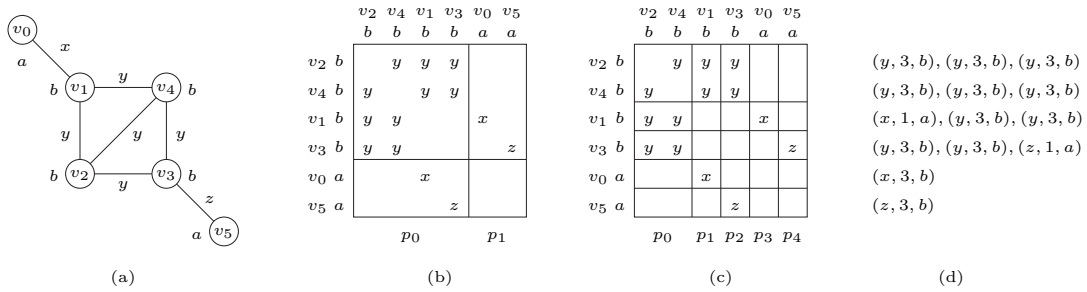


Figure 5: Use of neighbor lists

5.1.3 Iterative Partitioning

Iterative partitioning generalizes the idea of the neighbor lists, by incorporating the partition information [21]. This time, instead of a tuple $(l(e), d(v), l(v))$, we use a pair $(p(v), l(e))$ for representing the neighbor lists where $p(v)$ is the identifier of a partition to which a neighbor vertex v belongs and $l(e)$ is the label of the incident edge to the neighbor vertex v.

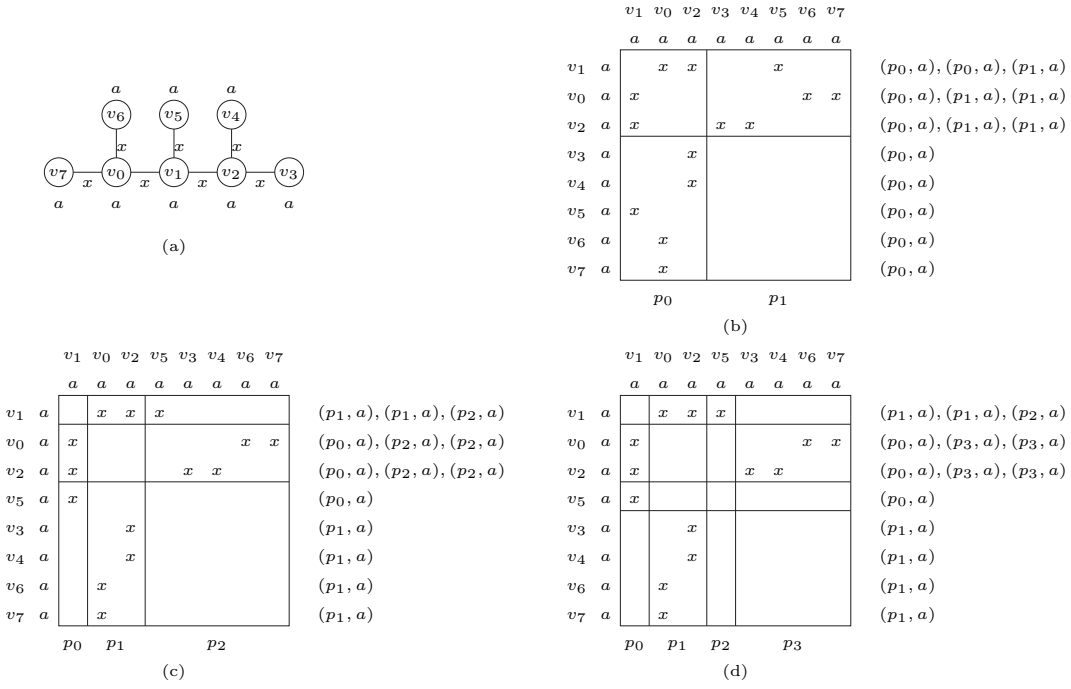


Figure 6: An example of iterative partitioning

We will illustrate the idea behind the iterative partitioning by the example shown in Figure 6. In the graph of Figure 6(a), all edges have the same label x and all vertices have the same label a. Initially the vertices are partitioned into two groups only by their degrees, and in each partition they are sorted by their neighbor lists as shown in Figure 6(b). The ordering of those partitions is originally based on the degrees and the labels of each vertex and its neighbors. Thus, we can uniquely determine this partition-ordering irrespective of how the vertices are ordered initially. Then, we split the first partition p_0 into two, because the neighbor lists of v_1 is different from those of v_0 and v_2. By renumbering all the partitions, updating the neighbor lists, and sorting the vertices based on their neighbor lists, we obtain the matrix as shown in Figure 6(c). Now, because the partition p_2 becomes non-uniform in terms of the neighbor lists, we again divide p_2 to factor out v_5, renumber partitions, update and sort the neighbor lists, and sort vertices to obtain the matrix in Figure 6(d).

5.2 Degree-based Partition Ordering

In addition to using the vertex invariants to compute a fine-grain partitioning of the vertices, the overall runtime of the canonical labeling can be further reduced by properly ordering the various partitions. This is because, a proper ordering of the partitions may allow us to quickly determine whether a set of permutations can potentially lead to a code that is smaller than the current best code or not; thus, allowing us to prune large parts of the search space.

Recall from Section 5.1 that we obtain the code of a graph by concatenating its adjacent matrix in a column-wise fashion. As a result, when we permute the rows and the columns of a particular partition, the code corresponding to the columns of the preceding partitions is not affected. Now, while we explore a particular set of within-partition permutations, if we obtain a prefix of the final code that is larger than the corresponding prefix of the currently best code, then we know that regardless of the permutations of the subsequent partitions, this code will never be smaller than the currently best code, and the exploration of this set of permutations can be terminated.

The critical property that allows us to prune such unpromising permutations is our ability to obtain a *bad* code prefix. Ideally, we will like to order the partitions in a way such that the permutations of the vertices in the initial partitions lead to dramatically different code prefixes, which it turn will allow us to prune parts of the search space. In general, the likelihood of this happening depends on the density (*i.e.*, the number of edges) of each partition, and for this reason we sort the partitions in decreasing order of the degree of their vertices.

For example, consider the graph shown in Figure 7(a), where there are two edge labels, E and F, and all the vertices have been assigned the same label. First, based on the vertex degree, we have two partitions, $p_0 = \{v_3, v_4, v_5\}$ and $p_1 = \{v_0, v_1, v_2\}$. If we sort the partitions in the ascending order of vertex degree, we have the adjacency matrix of Figure 7(b) at the starting point of the canonical labeling. Because the first partition p_0 does not have any non-zero elements in its upper triangle, the second partition p_1 plays the key role to determine the canonical label. In fact, the order of v_3, v_4 and v_5 does not change the code as long as we focus on the first partition p_0. Thus, we have to exhaustively compare all the vertex permutations in p_0. On the contrary, consider the case where we order the partitions based on their vertex degrees in the descending order. We have the matrix of Figure 7(c). At some point during the process of the canonical labeling by permuting the rows and the columns of the adjacency matrix, we will encounter the one shown in Figure 7(d).

When we compare the two matrices, 7(c) with 7(d), for example, we immediately know that 7(d) is *smaller* than 7(c), only by comparing the first partition and we can avoid further permutations in the second partition for the degree one vertices, such as $v_2v_0v_1v_3v_5v_4$ or $v_2v_0v_1v_4v_3v_5$. This is because as long as we have the ordering of $v_2v_0v_1$ in p_1, the matrix is always *smaller* than the one with $v_0v_1v_2$ no matter how the vertices in p_0 are ordered.

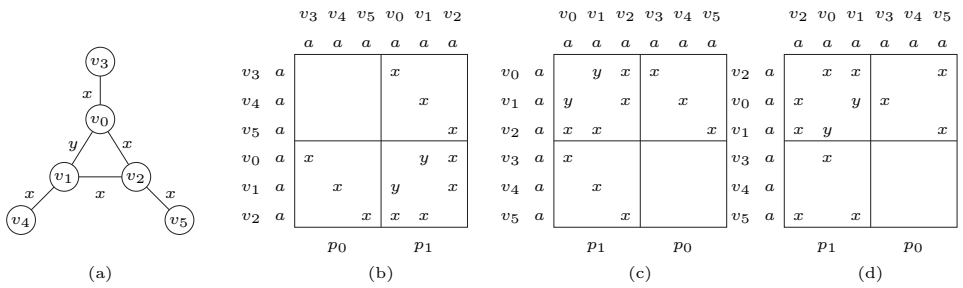


Figure 7: An example of degree-based partition ordering

6 Experimental Evaluation

We performed three sets of experiments to evaluate the performance of the FSG algorithm. In the first two set of experiments, we used datasets consisting of the molecular structure of various chemical compounds, and in the second set of experiments we used various datasets that were generated synthetically.

The primary goal of the first two experiments was to evaluate the impact of the various optimizations for candidate generation (described in Section 4.1.1) and for canonical labeling (described in Section 5), and to illustrate the effectiveness of FSG for finding rather large patterns and scale to very large real datasets. On the other hand, the primary goal for the third set of experiments was to evaluate the performance of FSG on datasets whose characteristics (*e.g.*, number of graph transactions, average graph size, average number of vertex- and edge-labels, and/or average length of patterns) differs dramatically; thus, providing insights on how well FSG scales with respect to these characteristics.

All experiments were done on dual AMD Athlon MP 1800+ (1.53GHz) machines with 2GB main memory, running the Linux operating system. All the times reported are in seconds.

6.1 Chemical Compound Dataset From PTE

The first chemical dataset that we used in our experiments was obtained from [20] and was originally provided for the Predictive Toxicology Evaluation Challenge [25]. The entire data set contains 340 chemical compounds and for each compound it lists the set of atoms that it is made off and how these atoms are connected together, *i.e.*, their bonds. Each atom is specified as a pair of *element-name* and *element-type*, and each bond is specified by its *bond-type*. The entire dataset contains 24 different element names[3], 66 different element types, and four different types of bonds.

We converted these compounds into graph transactions by representing each atom via a vertex and each bond by an edge connecting the corresponding vertices. The label of each vertex was derived from the pair of element-name and element-type, and the label of the edge was derived from the bond-type. This resulted in a dataset containing 340 transactions with 66 different vertex labels and four different edge labels. The average transaction size is 27.4 in terms of the number of edges, and 27.0 in terms of the number of vertices. There are 26 transactions that have more than 50 edges and vertices, and the largest transaction contains 214 edges and 214 vertices. Even though the dataset is rather small, the size of many of these compounds is quite large and also contains fairly large patterns—making it an especially challenging dataset and well-suited for illustrating the effect of the various optimizations incorporated in FSG.

[3]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.

6.1.1 Experimental Results

Table 2 shows the amount of time required by FSG to find all frequently occurring subgraphs using the various optimizations described in Sections 4.1.1 and 5, for different values of support. This table shows a total of five different sets of results, corresponding to increasing levels of optimization. The column labeled "Degree-Label Partitioning", corresponds to the version of FSG that uses only vertex degrees and their labels as the vertex invariants for canonical labeling (described in Section 5.1.1). The column labeled "Inverted Index" corresponds to the scheme that also incorporates the inverted-index approach for determining the pairs of candidate to be joined together (described in Section 4.1.1). The column labeled "Partition Ordering" corresponds to the scheme that also orders the partitions produced by the vertex invariants in decreasing order of their degrees (described in Section 5.2). The column labeled "Neighbor List" corresponds to the scheme that also incorporates the invariants based on neighbor lists (described in Section 5.1.2). Finally, the column labeled "Iterative Partitioning" corresponds to the scheme that also incorporates vertex invariants based on the iterative partitioning approach (described in Section 5.1.3). Because iterative partitioning is especially effective for large subgraphs, we apply it only if the number of vertices in a subgraph is greater than eleven. The last three columns show the size of the largest pattern that was discovered, the number of candidate patterns that were generated, and the total number of the frequent patterns that were discovered, respectively. The performance of FSG was evaluated using a total of fifteen different values for the support, ranging from 10% down to 2%. Dashes in the table correspond to experiments that were aborted due to high computational requirements.

Table 2: Comparison of various optimizations using the chemical compound dataset

Support	Running Time[s] with Optimizations					Largest		Frequent
σ	Degree-Label	Inverted	Partition	Neighbor	Iterative	Pattern	Candidates	Patterns
[%]	Partitioning	Index	Ordering	List	Partitioning	Size k^*	$\lvert \boldsymbol{C} \rvert$	$\lvert \boldsymbol{F} \rvert$
10.0	6	4	3	3	3	11	970	844
9.0	8	6	4	3	4	11	1168	977
8.0	22	13	6	5	5	11	1602	1323
7.5	29	15	7	6	6	12	1869	1590
7.0	45	23	10	7	7	12	2065	1770
6.5	138	59	17	9	9	12	2229	1932
6.0	1853	675	56	13	11	13	2694	2326
5.5	5987	1691	112	18	14	13	3076	2692
5.0	24324	7377	879	33	22	14	4058	3608
4.5	—	55983	4196	40	35	15	5533	4984
4.0	—	—	12363	126	51	15	6546	5935
3.5	—	—	—	697	152	20	14838	13816
3.0	—	—	—	3097	317	22	24064	22758
2.5	—	—	—	9329	537	22	33660	31947
2.0	—	—	—	—	3492	25	139666	136927

Looking at these results we can see that the various optimizations have a dramatic impact on the overall performance of FSG. There is a three orders of magnitude difference between the *lightly* optimized version of the algorithm that uses only invariants based on the degrees and labels of the vertices and the *highly* optimized version that uses efficient algorithms for candidate generation and sophisticated vertex invariants. The results also show that as the length of the patterns being discovered increases (as a result of decreasing the support), the more sophisticate algorithms for canonical labeling are essential in controlling the exponential complexity of the problem.

In terms of the effect of individual optimizations, we can see that the use of inverted indices during candidate generation improves the runtime by a factor of 2–4; the proper ordering of the various partitions is able to reduce

the runtime by a factor of 5–15 (especially for small support values); the invariants based on neighbor lists achieve one to two orders of magnitude improvement; and the iterative partitioning is able to further improve the runtime by an additional order of magnitude, especially for very small support values.

Overall, FSG was able to successfully operate for a support value as low as 2% and discover frequent patterns containing up to 24 vertices and 25 edges. However, as the results indicate, when the support threshold falls below 2.5%, both the amount of time required and the number of frequent subgraphs increase exponentially. Nevertheless, FSG does quite well even for 2.5% support as it requires only 537 seconds and is able to discover patterns containing over 22 edges. To put these results in perspective, previously reported results on the same dataset using the AGM algorithm which finds frequent *induced* subgraphs, required about 8 days for 10% and 40 minutes for 20% support on a 400MHz PC [17].

6.2 Chemical Compound Dataset From DTP

The second chemical dataset that we used contains a total of 223,644 chemical compounds and is available from the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) [9]. Using the same method described in Section 6.1, these compounds were converted into graphs by using vertices to model the atoms and edges to model their bonds. The various types of atoms were modeled as vertex labels and the various types of bonds were modeled as edge labels. Overall, there was a total of 104 distinct vertex labels (atom types) and three distinct edges labels (bond types).

This set of experiments was designed to evaluate the scalability of FSG with respect to the number of input graph transactions. Toward this goal we created six datasets with 10,000, 20,000, 30,000, 40,000, 50,000 and 100,000 transactions. Each graph transaction was randomly chosen from the original dataset of 223,644 compounds. This random dataset creation process resulted in datasets in which the average transaction size (the number of edges per transaction) was about 22.

6.2.1 Experimental Results

Table 3 shows the results obtained on the six DTP datasets, for different values of support ranging from 10% down to 2.5%. For each dataset, Table 3 shows the amount of time (t), the size of the largest discovered frequent subgraph (k^*), and the total number of frequent patterns ($|\boldsymbol{F}|$) that were generated. A dash ('—') indicates runs that did not finish within one hour.

Looking at these results we can see that FSG is able to effectively operate in datasets containing over 100,000 graph transactions, and discover all subgraphs that occur in at least 3% of the transactions in approximately fifteen minutes. Moreover, with respect to the number of transactions, we can see that the runtime of FSG scales linearly. For instance, for a support of value 4%, the amount of time required for 10,000 transactions is 54 seconds, whereas the corresponding time for 100,000 transactions is 517 seconds; an increase by a factor of 9.6. Also, as the support decreases up to a value of 3%, the amount of time required by FSG increases gradually, mirroring the increase in the number of frequent subgraphs that exist in the dataset. For instance, for 50,000 transactions, the amount time required for a support of 3% is 1.6 times higher than that required for a support of 4% and it finds 1.8 times more frequent subgraphs.

However, there is some discontinuity in the results obtained by FSG at support value of 2.5%. For 10,000 transactions FSG requires 381 seconds, whereas for 20,000, 30,000, 40,000 and 50,000 transactions, it requires somewhere in the range of 1,776 to 1,966 seconds. Moreover, for 100,000 transactions it takes over one hour. In analyzing why the runtime increase did not follow a pattern similar to that of the other support values we discovered

Table 3: Runtimes in seconds for chemical compound data sets which are randomly chosen from the DTP dataset. The column with σ shows the used minimum support (%), the column with t is the runtime in seconds, the column with k^* shows the size of the largest frequent subgraph discovered, and the column with $|\boldsymbol{F}|$ is the total number of discovered frequent patterns.

σ	Total Number of Transactions $\|\boldsymbol{D}\|$																	
	$\|\boldsymbol{D}\| = 10,000$			$\|\boldsymbol{D}\| = 20,000$			$\|\boldsymbol{D}\| = 30,000$			$\|\boldsymbol{D}\| = 40,000$			$\|\boldsymbol{D}\| = 50,000$			$\|\boldsymbol{D}\| = 100,000$		
[%]	t[sec]	k^*	$\|\boldsymbol{F}\|$	t[sec]	k^*	$\|\boldsymbol{F}\|$	t[sec]	k^*	$\|\boldsymbol{F}\|$	t[sec]	k^*	$\|\boldsymbol{F}\|$	t[sec]	k^*	$\|\boldsymbol{F}\|$	t[sec]	k^*	$\|\boldsymbol{F}\|$
10.0	24	9	267	49	9	270	72	9	273	99	9	277	124	9	276	257	9	283
9.0	27	9	335	54	9	335	80	9	335	110	9	335	137	9	336	289	10	356
8.0	30	10	405	59	11	402	88	11	399	119	11	404	150	11	425	311	11	426
7.0	34	11	546	67	11	559	99	11	554	135	11	557	167	11	559	351	11	588
6.0	40	12	713	77	12	723	113	12	714	152	12	734	188	12	733	391	12	764
5.0	44	12	939	84	12	951	122	12	937	165	12	960	205	12	967	432	12	1018
4.5	49	12	1129	91	12	1133	133	12	1118	178	12	1141	223	12	1154	467	12	1237
4.0	54	12	1431	108	12	1417	144	13	1432	203	12	1445	251	13	1471	517	13	1579
3.5	67	13	1896	118	13	1895	167	13	1912	221	13	1944	374	13	1969	666	13	2086
3.0	174	14	2499	231	13	2539	284	14	2530	346	14	2565	402	14	2603	923	14	2710
2.5	381	14	3371	1776	14	3447	1857	14	3393	1907	14	3440	1966	14	3523	—	—	—

that at that FSG generated a candidate of size 13 which is a cycle consisting of with a single vertex label and a single edge label (*i.e*, C_{13}). This is a very regular graph and the various vertex invariants described in Section 5 can not be used to partition the vertices. Consequently, the amount of time required by canonical labeling increases substantially. We are currently investigating methods that will allow FSG to effectively operate on such regular frequent subgraphs and some of these approaches are briefly described in Section 7.

6.3 Synthetic Datasets

To evaluate the performance of FSG on datasets with different characteristics we developed a synthetic dataset generator. The design of our generator was inspired by the synthetic transaction generator developed by the Quest group at IBM and used extensively to evaluate algorithms that find frequent itemsets [2, 1, 15, 22]. In the reminder of this section we first describe the synthetic graph generator followed by a detailed experimental evaluation of FSG on a wide range of synthetically generated datasets.

6.3.1 Dataset Generator

Our synthetic dataset generator produces a database of graph transactions whose characteristics can be easily controlled by the user. The generator allows us to specify the number of desired transactions $|\boldsymbol{D}|$, the average number of edges in each transaction $|T|$, the average size $|I|$ of the potentially frequent subgraphs in terms of the number of edges, the number of potentially frequent subgraphs $|\boldsymbol{S}|$ in the dataset, the number of distinct edge labels $|L_E|$, and the number of distinct vertex labels $|L_V|$. These parameters are summarized in Table 4.

A synthetic graph dataset is generated as follows. First, we generate a set of $|\boldsymbol{S}|$ potentially frequent connected subgraphs called *seed patterns* 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 subgraph is selected to be included in a transaction. The weights are calculated by dividing a random variable that obeys an exponential distribution with unit mean by the number of edges in the subgraph, and the sum of the weights of all the frequent subgraphs is normalized to one. We call this set $\boldsymbol{S}$ of frequent subgraphs a *seed pool*. The reason that we divide the exponential random variable by the number of edges is to reduce the chance that larger weights are assigned to larger seed patterns. Otherwise, once a large weight was assigned to a large seed pattern, the resulting dataset would contain an exponentially large number of

Table 4: Synthetic dataset parameters

Notation	Parameter
$\|\boldsymbol{D}\|$	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)
$\|\boldsymbol{S}\|$	The number of potentially frequent subgraphs
$\|L_E\|$	The number of edge labels
$\|L_V\|$	The number of vertex labels

frequent patterns.

Next, we generate $|\boldsymbol{D}|$ 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 $|\boldsymbol{S}|$-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 insert a seed into a transaction is to add a newly selected seed pattern to the transaction by connecting randomly selected pair of vertices, one from the transaction and the other from the seed pattern.

6.3.2 Experimental Results

Sensitivity on the Structure of the Graphs Using the synthetic dataset generator we obtained a number of different graph datasets by using different combinations of $|L_V|$, $|I|$, and $|T|$, while keeping $|\boldsymbol{D}|$, $|\boldsymbol{S}|$, and $|L_E|$ fixed. Table 5 summarizes the values for the various parameters that were used to derive these datasets. Note that in our datasets we used different vertex labels but only a single edge label. This is because in FSG both edge- and vertex-labels act as constraints to narrow down the search space of canonical labeling and graph/subgraph isomorphisms, and their impact can be evaluated by simply varying only one of them. Also, we did not obtain any datasets in which the average transaction size $|T|$ is smaller than the average size of the potentially frequent subgraphs $|I|$, because our generator cannot produce reasonable datasets for this type of parameter combinations.

Furthermore, our experience with generating synthetic graph datasets has been that as both $|T|$ and $|I|$ increase, the amount of time required to mine different randomly-generated dataset-instances obtained with the same set of parameters, can differ dramatically. This is because a particular dataset may contain some *hard* seed patterns (*e.g.*, regular patterns with similar labels), dramatically increasing the amount of time spent during canonical labeling. For this reason, instead of measuring the performance of FSG on a single dataset for each parameter combination, we created ten different datasets for each parameter combination using different seeds for the pseudo random number generator, and run FSG on all of them.

Table 6 shows the average, the median and the standard deviation of the amount of time required by FSG to find all frequent subgraphs for various datasets using a support of 2%. Note that these statistics were obtained

Table 5: Parameter settings

Parameter	Values
$\|\boldsymbol{D}\|$	10,000
$\|T\|$	5,10,20,30,40
$\|I\|$	3,5,7,9,11
$\|\boldsymbol{S}\|$	200
$\|L_E\|$	1
$\|L_V\|$	3,5,10,15,20
σ	2%

by analyzing the ten different runs for each particular combination of parameters. The actual runtimes for each particular parameter combination and for each one of the ten different datasets are plotted in Figure 8. Also, the entries in Table 6 that contain a dash correspond to problem instances in which we had to abort the computation because either FSG exhausted the available memory or it required a long amount of time.

A number of observations can be made by analyzing the results in Table 6 and Figure 8 that illustrate how FSG performs on different types of datasets.

The results in Table 6 and Figure 8 confirm that when $|I|$ is large, the runtime obtained by the ten different instances of each dataset differ widely (large standard deviations). As discussed earlier, this is because the generator sometimes may create *seed patterns* that are either regular or large, which significantly increases the complexity of canonical labeling. Consequently, the amount of time required by certain experiments may be higher than that required by what is perceived to be a simpler problem. For example, with $|L_V| = 5$ and $|I| = 11$, both the average and the median runtimes for $|T| = 30$ is longer than those of $|T| = 40$. For this reason, we will tend to focus on median runtimes, as we believe they represent a better measure of the amount of time required by FSG.

As the number of vertex labels $|L_V|$ increases, the amount of time required by FSG decreases. For example, while the runtime is 11 seconds for $|L_V| = 5$, $|I| = 5$ and $|T| = 10$, it decreases to 7 seconds for $|L_V| = 10$,$|I| = 5$ and $|T| = 10$. This is because as the number of vertex labels increases there are fewer automorphisms and subgraph isomorphisms, which translates to faster candidate generation and frequency counting. Also, by having a larger number of vertex labels, we can effectively prune the search space of isomorphism because distinct vertex labels act as constraints when we seek a mapping of vertices. Note that for $|I| = 5$ and $|T| = 10$, however, the runtime stops decreasing at $|L_V| = 15$, and at $|L_V| = 20$ there is no further improvement. This is because when $|L_V|$ is sufficiently large compared to the transaction size, the vertices in each graph tend to be assigned a unique vertex label, and any additional increases in $|L_V|$ does not improve the runtime any further. Of course, for different values of $|I|$ and $|T|$, there will be a different value of $|L_V|$ after which the performance improvements will diminish, and this can be seen by looking at the results in Table 6.

As the size of the average seed pattern $|I|$ increases, the runtime tends to increase. When $|L_V| = 5$ and $|T| = 20$, the runtime for $|I| = 7$ is 1.8 times longer than that for $|I| = 5$. Likewise, when $|L_V| = 15$ and $|T| = 20$, the runtime for $|I| = 7$ is 2.2 times longer than $|I| = 5$. This is natural because the size of the seed patterns determine the total number of frequent patterns to be discovered, and because of the combinatorial nature, the number of such patterns drastically increases even by having one additional edge in a seed pattern. We should note that this ratio of the runtimes with respect to $|I|$ does not necessarily decrease as $|L_V|$ increases, especially when $|I|$ is large. For example, the median of the running time is 3.0 times longer for $|L_V| = 15$, $|I| = 9$ and $|T| = 20$ than for $|L_V| = 15$,

Table 6: Runtimes in seconds for the synthetic data sets. For each set of the three parameters, the number of vertex labels $|L_V|$, the average seed size $|I|$ and the average transaction size $|T|$, we showed the average $\bar{t}$, the median t^* and the standard deviation STD of the runtimes over ten trials.

$\|L_V\|$	$\|I\|$	$\|T\|$	$\bar{t}$[sec]	t^*[sec]	STD
3	3	5	5	5	0
3	3	10	11	11	1
3	3	20	38	37	3
3	3	30	96	96	6
3	3	40	170	171	14
3	5	5	7	7	1
3	5	10	17	16	1
3	5	20	64	63	7
3	5	30	195	193	41
3	5	40	352	354	39
3	7	10	38	38	6
3	7	20	190	183	41
3	7	30	495	494	100
3	7	40	1464	1469	322
3	9	10	124	129	29
3	9	20	557	509	183
3	9	30	2208	1966	551
3	9	40	—	—	—
3	11	20	5200	2514	4856
3	11	30	—	—	—
3	11	40	—	—	—

$\|L_V\|$	$\|I\|$	$\|T\|$	$\bar{t}$[sec]	t^*[sec]	STD
5	3	5	4	4	0
5	3	10	8	8	0
5	3	20	21	20	1
5	3	30	41	41	2
5	3	40	70	68	6
5	5	5	5	5	0
5	5	10	11	11	1
5	5	20	32	32	4
5	5	30	65	63	9
5	5	40	118	118	14
5	7	10	20	17	4
5	7	20	59	57	10
5	7	30	155	132	65
5	7	40	316	278	121
5	9	10	46	44	14
5	9	20	222	182	126
5	9	30	990	594	903
5	9	40	1109	775	648
5	11	20	3657	646	6735
5	11	30	9348	8328	6919
5	11	40	8955	4815	8791

$\|L_V\|$	$\|I\|$	$\|T\|$	$\bar{t}$[sec]	t^*[sec]	STD
10	3	5	2	2	0
10	3	10	5	5	0
10	3	20	12	12	1
10	3	30	22	22	1
10	3	40	36	35	2
10	5	5	3	3	0
10	5	10	7	7	0
10	5	20	18	17	1
10	5	30	38	37	7
10	5	40	57	58	5
10	7	10	10	9	2
10	7	20	37	37	7
10	7	30	121	98	57
10	7	40	265	140	226
10	9	10	43	25	56
10	9	20	145	108	118
10	9	30	559	408	468
10	9	40	1109	746	800
10	11	20	427	267	301
10	11	30	7392	4050	7365
10	11	40	10915	10216	7451

$\|L_V\|$	$\|I\|$	$\|T\|$	$\bar{t}$[sec]	t^*[sec]	STD
15	3	5	2	2	0
15	3	10	4	4	0
15	3	20	9	9	0
15	3	30	16	15	1
15	3	40	24	23	0
15	5	5	3	3	0
15	5	10	6	5	1
15	5	20	15	14	3
15	5	30	32	25	18
15	5	40	42	40	7
15	7	10	10	9	2
15	7	20	57	31	72
15	7	30	75	73	20
15	7	40	129	121	36
15	9	10	27	13	39
15	9	20	200	92	211
15	9	30	446	368	289
15	9	40	1300	1384	682
15	11	20	1064	774	1409
15	11	30	4854	3443	4294
15	11	40	15402	6762	14135

$\|L_V\|$	$\|I\|$	$\|T\|$	$\bar{t}$[sec]	t^*[sec]	STD
20	3	5	2	2	0
20	3	10	4	4	0
20	3	20	9	9	0
20	3	30	14	14	1
20	3	40	22	21	1
20	5	5	3	3	0
20	5	10	5	5	0
20	5	20	16	14	6
20	5	30	26	27	3
20	5	40	44	40	11
20	7	10	14	9	12
20	7	20	36	32	11
20	7	30	92	81	29
20	7	40	159	133	105
20	9	10	25	18	20
20	9	20	254	197	179
20	9	30	807	419	1202
20	9	40	2290	1045	3439
20	11	20	1750	455	3944
20	11	30	8894	2713	11776
20	11	40	—	—	—

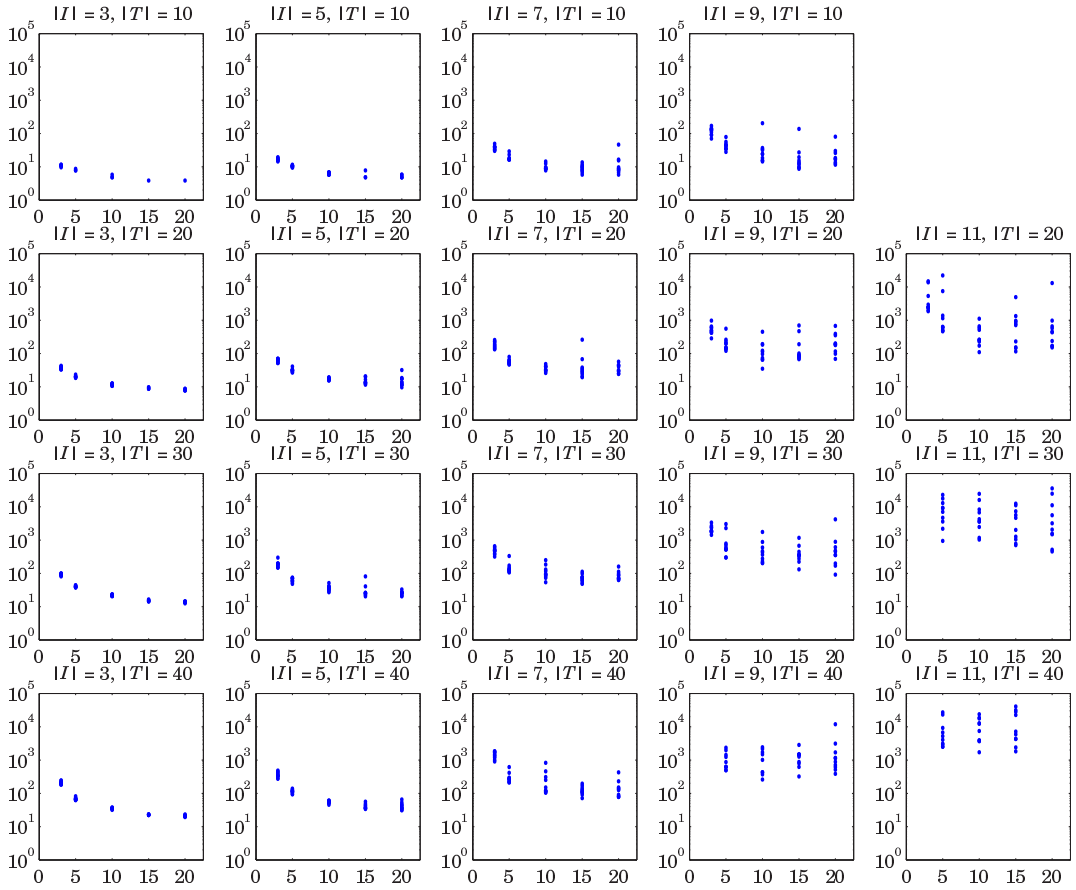


Figure 8: Runtimes in seconds for the synthetic data sets. The horizontal axis is for the number of vertex labels $|L_V|$ and the vertical axis is for the runtime spent by FSG. We plotted the ten runtimes for each set of the three parameters, the number of vertex labels $|L_V|$, the average seed size $|I|$ and the average transaction size $|T|$. The average and the median of the runtimes shown in Table 6 were calculated from those data points. The figures in the first, the second and the third row are the results obtained with the average transaction size $|T| = 10$, 20, 30, and 40, respectively. Likewise, the columns of the figures corresponds to the same average size of potential frequent subgraphs. The first, the second, the third and the fourth columns are for $|I| = 3$, 5, 7, 9 and 11.

$|I| = 7$ and $|T| = 20$, however, it is 6.1 times longer for $|L_V| = 20$, $|I| = 9$ and $|T| = 20$ than for $|L_V| = 20$, $|I| = 7$ and $|T| = 20$. This is due to the variability of the complexity of the datasets where both $|T|$ and $|I|$ are large.

As the size of the average transaction $|T|$ increases, the overall runtime increases as well. The relative increase is higher when $|L_V|$ is smaller. For example, by going from $|T| = 10$ to $|T| = 30$ with $|L_V| = 3$ and $|I| = 7$, the median runtime increases by a factor of 13, whereas for the same set of parameters when $|L_V| = 20$, the increase is by a factor of 9. The reason for that is also due to the fact that when there are large number of different vertex labels, canonical labeling and subgraph isomorphism is considerably faster. On the other hand, when $|L_V|$ is small and $|T|$ is large, the overall search space of these computational kernels increases dramatically as we cannot significantly prune the corresponding search spaces.

Finally, as $|I|$ increases the overall runtime also increases. Again the relative increase is smaller for larger values of $|L_V|$ and smaller values of $|T|$ due to the same reasons described above.

Database Size Scalability Finally, to determine the scalability of FSG when the number of transactions increases, we performed an experiment in which we used $|\boldsymbol{D}|$ = 10,000, 20,000, 40,000, 60,000, 80,000 and 100,000, $\sigma = 2\%$, $|L_V| = 10$, $|\boldsymbol{S}| = 200$, $|I| = 5$ and $|T|$ ranging from 5 to 40. For all those experiments, we used the same set of seed patterns, to ensure (up to a point) that the intrinsic complexity of the datasets remained the same.

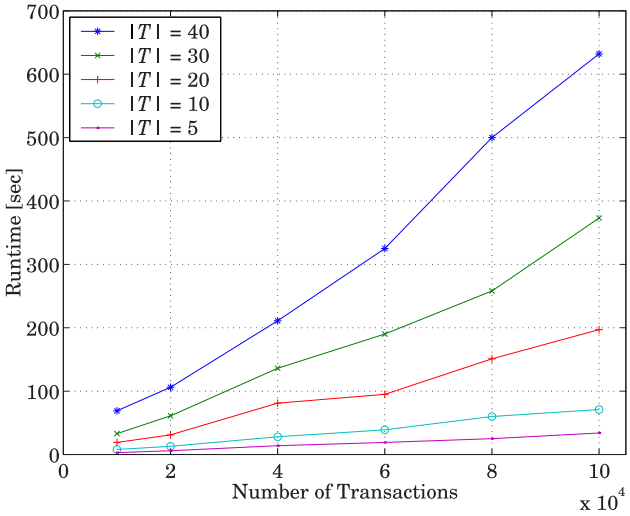


Figure 9: Scalability on the number of transaction

Figure 9 shows the amount of time required by FSG to find the frequent patterns in these datasets. As we can see from the figure, FSG scales linearly with the number of transactions, and is able to find all frequent subgraphs in a database with 100,000 graph transactions of size 40 in about 10 minutes. These results were also confirmed by the performance on the DTP datasets.

6.4 Summary of Performance Evaluation

The experiments presented in the previous sections provide some valuable insights on the performance characteristics of FSG and on the type of datasets that it can operate efficiently. In this section we summarize some of these findings.

FSG works better on graph datasets that have a large number of edge- and vertex-labels. During both candidate

generation and frequency counting, what FSG essentially does is to solve graph or subgraph isomorphism problems. In the absence of edge- and vertex-labels, determining these isomorphisms is considerably more expensive, because we can not use labeling information to constraint the space of possible vertex mappings. This is the reason why the performance of FSG is worse on the PTE dataset than on the synthetic datasets (once we take into account the difference in the number of transactions). 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 bond type. Essentially with $\sigma < 10\%$, this dataset becomes almost similar to the synthetic datasets where $|L_V| = 2$.

FSG's runtime depends heavily on the size of frequent subgraphs being discovered. For example, as the results on the synthetic datasets illustrate, when the patterns are small (*e.g.*, $|I| \leq 9$), FSG runs reasonably fast but when $|I| = 11$, FSG will not likely finish its computation in a reasonable amount of time. Similar observations can be made for the chemical dataset as well. For example, if we use $\sigma = 10\%$, FSG spends 3 seconds to get 844 frequent subgraphs, in which the largest subgraph contains 11 edges, and there are only 10 such frequent 11-subgraphs. As we decrease the support, however, the length of the discovered patterns increase considerably, and the amount of time increases at a much higher rate than the number of patterns. The key reason for this behavior is that as the length of the frequent subgraph increases, candidate generation and frequency counting become much more expensive. The asymptotic complexity of FSG for finding the subgraphs of length k (in the absence of a large number of vertex- and edge-labels) is exponential on k.

Another important factor is the size of a transaction. If the average size of the transactions becomes large, frequency counting by subgraph isomorphism becomes expensive, regardless of the size of candidate subgraphs. Note that traditional frequent itemset finding algorithms do not suffer from this type of problems, as frequency counting can be simply performed by taking the intersection of the itemset and the transaction.

Finally, despite the inherent complexity of finding frequent subgraphs, FSG scales linearly on the size of the input transaction database. This is because, the complexity of finding frequent subgraphs increases proportionally with the number of transactions provided that the characteristics of these transactions and the frequent patterns they contain remain the same as we scale up the input size.

7 Conclusion and Directions for Future Research

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. Such patterns can play an important role for understanding the nature of these datasets and can be used as input to other data-mining tasks [14, 8].

Our detailed experimental evaluation shows that FSG can scale reasonably well to very large graph databases provided that the graphs contain a sufficiently many different labels of edges and vertices. Key elements to FSG's computational scalability are the highly efficient canonical labeling algorithm that it employs and its use of a TID-list based approach for frequency counting. These two features combined, allow FSG to uniquely identify the various generated subgraphs and to quickly prune most of the infrequent subgraphs without having to resort to computationally expensive graph and subgraph isomorphism computations.

The performance of FSG can be improved in a number of different ways. Some of these approaches that we are currently investigating are the following. If the candidate patterns are regular and relatively large, canonical labeling still requires a significant amount of time. These computational requirements can be reduced by caching some of the canonical labeling computations. Such a caching will lead to improved performance as during candidate generation,

the same candidate pattern is being generated multiple times. Also, in many cases, determining whether or not two patterns are identical may be simpler by just using graph isomorphism (especially for very regular patterns). Thus, a hybrid approach that uses canonical labeling for easy patterns and isomorphism for regular patterns may lead to better performance. Also, FSG's performance can be substantially reduced for finding patterns in very sparse graphs by developing canonical labeling algorithms that explicitly takes this sparsity into account.

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 *Proc. 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 *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36, 1998.

[8] M. Deshpande and G. Karypis. Automated approaches for classifying structures. In *Proc. of the 2nd Workshop on Data Mining in Bioinformatics (BIOKDD '02)*, 2002.

[9] DTP/2D and 3D structural information. ftp://dtpsearch.ncifcrf.gov/jan02_2d.bin.

[10] 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.

[11] 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.

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

[13] 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.

[14] V. Guralnik and G. Karypis. A scalable algorithm for clustering sequential data. In *Proc. of the 1st IEEE International Conference on Data Mining (ICDM 2001)*, 2001.

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

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

[17] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, pages 13–23, Lyon, France, September 2000.

[18] 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.

[19] 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.

[20] ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Datasets/carcinogenesis/progol/carcinogenesis.tar.Z.

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

[22] M. Seno and G. Karypis. LPMiner: An algorithm for finding frequent itemsets using length decreasing support constraint. In *Proc. of the 1st IEEE International Conference on Data Mining (ICDM)*, November 2001.

[23] 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.

[24] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287, 1997.

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

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

[27] M. J. Zaki. Fast mining of sequential patterns in very large databases. Technical Report 668, Department of Computer Science, Univeristy of Rochester, 1997.

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

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

[30] 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.