An Efficient Algorithm for Discovering Frequent Subgraphs

Michihiro Kuramochi and George Karypis, *Member, IEEE*
Department of Computer Science
University of Minnesota
4-192 EE/CS Building, 200 Union St SE
Minneapolis, MN 55455

{kuram, karypis}@cs.umn.edu

***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, one way of formulating the frequent pattern discovery problem is as 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 datasets. 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 200,000 graph transactions and scales linearly with respect to the size of the dataset.**

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

I. Introduction

EFFICIENT algorithms for finding frequent patterns—both sequential and non-sequential—in very large datasets have been one of the key success stories of data mining research [1], [2], [20], [36], [41], [49]. Nevertheless, as data mining techniques have been increasingly applied to non-traditional domains, there is a need to develop efficient and general-purpose frequent pattern discovery algorithms that are capable of capturing the strong spatial, topological, geometric, and/or relational nature of the datasets that characterize these domains.

In recent years, labeled topological graphs have emerged as a promising abstraction to capture the characteristics of these datasets. In this approach, each object to be analyzed is represented via a separate graph whose vertices correspond to the entities in the object and the edges correspond to the relations between them. Within that model, one way of formulating the frequent pattern discovery problem is as that of discovering subgraphs that occur frequently over the entire set of graphs.

The power of graphs to model complex datasets has been recognized by various researchers [3], [6], [10], [14], [19], [23], [26], [30], [37], [43], [46] as it allows us to represent arbitrary relations among entities and solve problems that we could not previously solve. For instance, consider the 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. In fact, within the context of chemical compound classification, such techniques have been used to mine chemical compounds and identify the substructures that best discriminate between the different classes [5], [11], [27], [42], and were shown to produce superior classifiers than more traditional methods [21].

Developing algorithms that discover all frequently occurring subgraphs in a large graph dataset is particularly challenging and computationally intensive, as graph and subgraph isomorphisms play a key role throughout the computations. In this paper we present a new algorithm, called FSG, for finding all connected subgraphs that appear frequently in a large graph dataset. Our algorithm finds frequent subgraphs using the 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 datasets; 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-

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, 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.

isomorphism computations.

We experimentally evaluated FSG on three types of datasets. The first two datasets correspond to various chemical compounds containing over 200,000 transactions and frequent patterns whose size is large, 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 datasets and find all frequently occurring subgraphs in reasonable amount of time and scales linearly with the dataset size. For example, in a dataset containing over 200,000 chemical compounds, FSG can discover all subgraphs that occur in at least 1% of the transactions in approximately one hour. Furthermore, our detailed evaluation using the synthetically generated graphs shows that for datasets that have a moderately large number of different vertex and edge labels, FSG is able to achieve good performance as the transaction size increases.

The rest of the paper is organized as follows. Section II provides some definitions and introduces the notation that is used in the paper. Section III 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 IV describes in detail the algorithm. Section V describes the various optimizations that we developed for efficiently computing the canonical label of the patterns. Section VI provides a detailed experimental evaluation of FSG on a large number of real and synthetic datasets. Section VII describes the related research in this area, and finally, Section VIII provides some concluding remarks.

II. Definitions and Notation

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.

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 $E_s \subseteq E$ and it 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,

TABLE I
Notation used throughout the paper

Notation	Description
k-subgraph	A connected subgraph with k edges (also written as a size-k subgraph)
G^k, H^k	(Sub)graphs of size k
$E(G)$	Edges of a (sub)graph G
$V(G)$	Vertices of a (sub)graph G
$\mathrm{cl}(G)$	A canonical label of a graph G
a, b, c, e, f	edges
u, v	vertices
$d(v)$	Degree of a vertex v
$l(v)$	The label of a vertex v
$l(e)$	The label of an edge e
$H = G - e$	H is a graph obtained by the deletion of edge $e \in E(G)$
$\mathcal{D}$	A dataset of graph transactions
$\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_N\}$	Disjoint N partitions of $\mathcal{D}$ (for i and j, $i \neq j$, $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ and $\bigcup_i \mathcal{D}_i = \mathcal{D}$)
T	A graph transaction
C	A candidate subgraph
$\mathcal{C}^k$	A set of candidates with k edges
$\mathcal{C}$	A set of all candidates
F	A frequent subgraph
$\mathcal{F}^k$	A set of frequent k-subgraphs
$\mathcal{F}$	A set of all frequent subgraphs
k^*	The size of the largest frequent subgraph in $\mathcal{D}$
L_E	A set of all edge labels in $\mathcal{D}$
L_V	A set of all vertex labels in $\mathcal{D}$

i.e., to determine whether or not G_2 is included in G_1. The *canonical label* of a graph $G = (V, E)$, $\mathrm{cl}(G)$, is defined to be a unique *code* (i.e., a sequence of bits, a string, or a sequence of numbers) that is invariant on the ordering of the vertices and edges in the graph [15]. As a result, two graphs will have the same canonical label if they are isomorphic. Examples of different canonical label codes and details on how they are computed are presented in Section V. Both canonical labeling and determining graph isomorphism are not known to be either in P or in NP-complete [15].

The *size* of a graph $G = (V, E)$ is defined to be equal to $|E|$. Given a size-k connected graph $G = (V, E)$, by *adding an edge* we will refer to the operation in which an edge $e = (u, v)$ is added to the graph so that the resulting size-$(k + 1)$ graph remains connected. Similarly, by *deleting an edge* we refer to the operation in which $e = (u, v)$ such that $e \in E$ is deleted from the graph and the resulting size-$(k - 1)$ graph remains connected. Note that depending on the particular choice of e, the deletion of the edge may result in deleting at most one of its incident vertices if that vertex has only e as its incident edge.

Finally, the notation that we will be using through-out the paper is shown in Table I.

III. Frequent Subgraph Discovery—Problem Definition

The problem of finding frequently occurring connected subgraphs in a set of graphs is defined as follows:

Definition 1 (Subgraph Discovery): Given a set of graphs $\mathcal{D}$ each of which is an undirected labeled graph, and a parameter σ such that $0 < \sigma \leq 1$, find all connected undirected graphs that are subgraphs in at least $\sigma|\mathcal{D}|$ of the input graphs. We will refer to each of the graphs in $\mathcal{D}$ as a *graph transaction* or simply *transaction* when the context is clear, to $\mathcal{D}$ as the *graph transaction dataset*, and to σ as the *support* threshold.

There are two key aspects in the above problem statement. First, 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. Second, we allow the graphs to be labeled, and as discussed in Section II, input graph transactions and discovered frequent patterns can contain multiple vertices and edges carrying 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 makes the problem of finding such frequently occurring subgraphs non-trivial. This is because in such cases, 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 [16].

IV. FSG—Frequent Subgraph Discovery Algorithm

In developing our frequent subgraph discovery algorithm, we decided to follow the level-by-level structure of the Apriori [2] algorithm used for finding frequent itemsets. The motivation behind this choice is the fact that the level-by-level structure of Apriori requires the smallest number of subgraph isomorphism computations during frequency counting, as it allows it to take full advantage of the downward closed property of the minimum support constraint and achieves the highest amount of pruning when compared with the most recently developed depth-first-based approaches such as dEclat [49], Tree Projection [1], and FP-growth [20]. In fact, despite the extra overhead due to candidate generation that is incurred by the level-by-level approach, recent studies have shown that because of its effective pruning, it achieves comparable performance with that achieved by the various depth-first-based approaches, as long as the data set is not dense or the support value is not extremely small [18], [22].

The overall flow of our algorithm, called FSG, is similar to that of Apriori, and works as follows. FSG starts by enumerating all frequent single- and double-edge subgraphs. Then, it enters its main computational phase, which consists of a main iteration loop. During each iteration, FSG first generates all candidate subgraphs whose size is greater than the previous frequent ones by one edge, and then counts the frequency for each of these candidates and prunes subgraphs that do no satisfy the support constraint. FSG stops when no frequent subgraphs are generated for a particular iteration. Details on how FSG generates the candidates subgraphs, and on how it computes their frequency are provided in Section IV-A and Section IV-B, respectively.

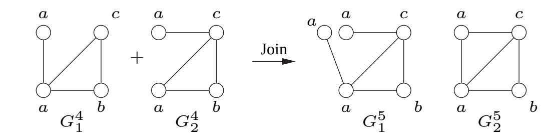
To ensure that the various graph-related operations are performed efficiently, FSG stores the various input graphs and the various candidate and frequent subgraphs that it generates using an adjacency list representation.

A. Candidate Generation

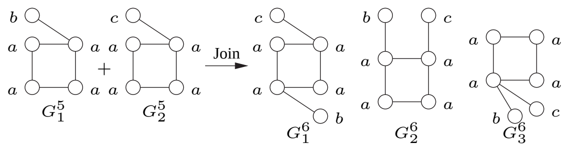
FSG generates candidate subgraphs of size $k+1$ by joining two frequent size-k subgraphs. In order for two such frequent size-k subgraphs to be eligible for joining they must contain the same size-$(k-1)$ connected subgraph. The simplest way to generate the complete set of candidate subgraphs is to join all pairs of size-k frequent subgraphs that have a common size-$(k-1)$ subgraph. Unfortunately, the problem with this approach is that a particular size-k subgraph, can have up to k different size-$(k-1)$ subgraphs. As a result, if we consider all such possible subgraphs and perform the resulting join operations, we will end up generating the same candidate pattern multiple times, and generating a large number of candidate patterns that are not downward closed. The net effect of this, is that the resulting algorithm spends a significant amount of time identifying unique candidates and eliminating non-downward closed candidates (both of which operations are non-trivial as they require to determine the canonical label of the generated subgraphs). Note that candidate generation approaches in the context of frequent itemsets, (e.g., Apriori [2]) do not suffer from this problem because they use a consistent way to order the items within an itemset (e.g., lexicographically). Using this ordering, they only join two size-k itemsets if they have the same $(k-1)$-prefix. For example, a particular itemset $\{A,B,C,D\}$ will only be generated once (by joining $\{A,B,C\}$ and $\{A,B,D\}$), and if that itemset is not downward closed, it will never be generated if only its $\{A,B,C\}$ and $\{B,C,D\}$ subsets were frequent.

Fortunately, the situation for subgraph candidate generation is not as severe as the above discussion seems to indicate and FSG addresses both of these problems by only joining two frequent subgraphs if and only if they share a certain, properly selected, size-$(k-1)$ subgraph. Specifically, for each frequent size-k subgraph F_i, let $\mathcal{P}(F_i) = \{H_{i,1}, H_{i,2}\}$ be the two size-$(k-1)$ connected subgraphs of F_i such that $H_{i,1}$ has the smallest canonical label and $H_{i,2}$ has the second smallest canonical label among the various connected size-$(k-1)$ subgraphs of F_i. We will refer to these subgraphs as the *primary subgraphs* of F_i. Note that if every size-$(k-1)$ subgraph of F_i is isomorphic to each other, $H_{i,1} = H_{i,2}$ and $|\mathcal{P}(F_i)| = 1$. FSG will only join two frequent subgraphs F_i and F_j, if and only if $\mathcal{P}(F_i) \cap \mathcal{P}(F_j) \neq \emptyset$, and the join operation will be done with respect to the common size-$(k-1)$ subgraph(s). The proof that this approach will correctly generate all valid candidate subgraphs is presented in Appendix . This candidate generation approach dramatically reduces the number of redundant and non-downward closed patterns that are generated and leads to significant performance improvements over the naive approach (originally implemented in [29]).

The actual join operation of two frequent size-k subgraphs F_i and F_j that have a common primary subgraph H is performed by generating a candidate size-$(k+1)$ subgraph that contains H plus the two edges that were deleted from F_i and F_j to obtain H. However, unlike the joining of itemsets



(a) By vertex labeling



(b) By multiple automorphisms of a single core

Fig. 1. Two cases of joining

in which two frequent size-k itemsets lead to a unique size-$(k+1)$ itemset, the joining of two size-k subgraphs may produce multiple distinct size-$(k+1)$ candidates. This happens for the following two reasons. First, the difference between the common primary subgraph and the two frequent subgraphs can be a vertex that has the same label. In this case, the joining of such size-k subgraphs will generate two distinct subgraphs of size $k+1$. Fig. 1(a) shows such an example, in which the pair of graphs G_a^4 and G_b^4 generates two different candidates G_a^5 and G_b^5. Second, the primary subgraph itself may have multiple automorphisms, and each of them can lead to a different size-$(k+1)$ candidate. In the worst case, when the primary subgraph is an unlabeled clique, the number of automorphisms is $k!$. An example for this case is shown in Fig. 1(b), in which the primary subgraph—a square of four vertices labeled with a—has four automorphisms resulting in three different candidates of size six. Finally, in addition to joining two different subgraphs, FSG also needs to perform self join. This happens, for example, when the two graphs G_i^k and G_j^k in Fig. 1 are identical. It is necessary because, for example, consider graph transactions without any labels. Then, there will be only one frequent size-1 subgraph and one frequent size-2 subgraph regardless of the support threshold, because those are the only allowed structures, and edges and vertices do not have labels assigned. In general, whenever $|\mathcal{F}^k| = 1$, self join is necessary to obtain a set of valid $(k+1)$-candidates.

B. Frequency Counting

The simplest way to determine the frequency of each candidate subgraph is to scan each one of the dataset transactions 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 [13], [40], [47]. In this approach for each frequent subgraph FSG keeps a list of transaction identifiers that support it. Now when FSG needs to compute the frequency of G^{k+1}, it first computes 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 FSG computes the frequency of G^{k+1} using subgraph isomorphism by limiting the 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, FSG is able to prune the candidate subgraph without performing any subgraph isomorphism computations. Second, when the intersection set is sufficiently large, FSG only needs to compute subgraph isomorphisms for those graphs that can potentially contain the candidate subgraph and not for all the graph transactions.

1) Reducing Memory Requirements of TID lists:

The computational advantages of TID lists come at the expense of higher memory requirements for maintaining them. To address this limitation we implemented a database-partitioning-based scheme that was motivated by a similar scheme developed for mining frequent itemsets [39]. In this approach, the database is partitioned into N disjoint parts $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_N\}$. Each of these sub-databases $\mathcal{D}_i$ is mined to find a set of frequent subgraphs $\mathcal{F}_i$, called *local frequent subgraphs*. The union of the local frequent subgraphs $\bar{\mathcal{C}} = \bigcup_i \mathcal{F}_i$, called *global candidates*, is determined and their frequency in the entire database is computed by reading each graph transaction and finding the set of subgraphs that it supports. The subset of $\bar{\mathcal{C}}$ that satisfies the minimum support constraint is output as the final set of frequent patterns $\mathcal{F}$. Since the memory required for storing the TID lists depends on the size of the database, their overall memory requirements can be reduced by partitioning the database in a sufficiently large number of partitions.

One of the problems with a naive implementation of the above algorithm is that it can dramatically increase the number of subgraph isomorphism operations that are required to determine the frequency of the global candidate set. In order to address this problem, FSG incorporates three techniques: (i) a priori pruning the number of candidate subgraphs that need to be considered; (ii) using bitmaps to limit the frequency counting of a particular candidate subgraph to only those partitions that this frequency has not already being determined locally; and (iii) taking advantage of the lattice structure of $\bar{\mathcal{C}}$ to check each graph transaction only against the subgraphs that are descendants of patterns that are already being supported by that transaction. The net effect of these optimizations is that, as shown in Section VI-A.1, the FSG's overall run-time increases slowly as the number of partitions increases.

The a priori pruning of the candidate subgraphs is achieved as follows. For each partition $\mathcal{D}_i$, FSG finds the set of local frequent subgraphs and the set of local negative border subgraphs[1], and stores them into a file S_i along with their associated frequencies. Then, it organizes the union of the local frequent and local negative border subgraphs across the various partitions into a lattice structure (called *pattern lattice*), by incrementally incorporating the information from each file S_i. Then, for each node v of the pattern lattice it computes an upper bound $f^*(v)$ of its occurrence frequency by adding the corresponding upper bounds for each one of the N partitions, $f^*(v) = f_1^*(v) + \cdots + f_P^*(v)$. For each partition $\mathcal{D}_i$, $f_i^*(v)$ is determined using the following equation:

$$f_i^*(v) = \begin{cases} f_i(v), & \text{if } v \in S_i \\ \min_u \left(f_i^*(u)\right), & \text{otherwise} \end{cases},$$

where $f_i(v)$ is the actual frequency of the pattern corresponding to node v in $\mathcal{D}_i$, and u is a connected subgraph of v that is smaller from it by one edge (i.e., it is its parent in the lattice). Note that the various $f_i^*(v)$ values can be computed in a bottom-up fashion by a single scan of S_i, and used directly to update the overall $f^*(v)$ values. Now, given this set of frequency upper bounds, FSG proceeds to prune the nodes of the pattern lattice that are either infrequent or fail the downward closure property.

V. Canonical Labeling

FSG relies on canonical labeling to efficiently check if a particular pattern satisfies the downward closure property of the support condition and to eliminate duplicate candidate subgraphs. Developing algorithms that can efficiently compute the canonical label of the various subgraphs is critical to ensure that FSG can scale to very large graph datasets.

Recall from Section II that the canonical label of a graph is nothing more than a *code* 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 defining the canonical label of a graph is as the string obtained by concatenating the upper triangular entries of the graph's adjacency matrix when this matrix has been symmetrically permuted so that this string becomes the lexicographically largest (or smallest) over the strings that can be obtained from all such permutations. This is illustrated in Fig. 2 that shows a graph G^3 and the permutation of its adjacency matrix[2] that leads to its canonical label "$aaazyx$". In this code, "aaa" was obtained by concatenating the vertex-labels in the order that they appear in the adjacency matrix and "zyx" was obtained by concatenating the columns of the upper triangular portion of the matrix. Note that any other permutation of G^3's adjacency matrix will lead to a code that is lexicographically smaller (or equal) to "$aaazyx$". If a graph has $|V|$ vertices, the complexity of determining its canonical label using this scheme is in $O(|V|!)$ making it impractical even for moderate size graphs.

In practice, the complexity of finding the canonical label of a graph can be reduced by using various heuristics to

[1]A local negative border subgraph is the one generated as a local candidate subgraph but does not satisfy the minimum threshold for the partition.

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

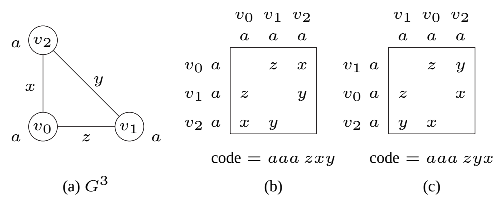


Fig. 2. Simple examples of codes and canonical adjacency matrices

narrow down the search space or by using alternate canonical label definitions that take advantage of special properties that may exist in a particular set of graphs [15], [31], [32]. In particular, the Nauty program [31] developed by Brendan McKay implements a number of such heuristics and has been shown to scale reasonably well to moderate size graphs. Unfortunately, Nauty does not allow graphs to have edge labels and as such it cannot be used directly by FSG. As a result we developed our own canonical labeling algorithm that incorporates some of the existing heuristics extended to vertex- and edge-labeled graphs as well as a number of new heuristics that are well-suited for our particular problem. Details of our canonical labeling algorithm are provided in the rest of this section.

Note that our canonical labeling algorithm operates on the adjacency matrix representation of a graph. For this reason, FSG converts its internal adjacency list representation of each candidate or frequent subgraph into its corresponding adjacency matrix representation, prior to computing its canonical label. Once the canonical label has been obtained, the adjacency matrix representation is discarded.

A. Vertex Invariants

Vertex invariants [15] are some inherent properties of the vertices that do not change across isomorphism mappings. An example of such an isomorphism-invariant property is the degree or label of a vertex, which remains the same regardless of the mapping (i.e., vertex ordering). 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. 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, respectively, 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. We

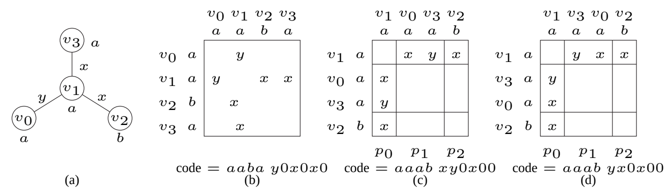


Fig. 3. A sample graph of size three and its adjacency matrices

have 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.

a) Vertex Degrees and Labels: This invariant partitions vertices into disjointed groups such that each partition contains vertices with the same label and the same degree. Fig. 3 illustrates the partitioning induced by this set of invariants for an example graph of size four. Based on their degree and their labels, the vertices are partitioned into three groups $p_0 = \{v_1\}$, $p_1 = \{v_0, v_3\}$ and $p_2 = \{v_2\}$ as shown in Fig. 3(c). Fig. 3 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 try only $1! \times 2! \times 1! = 2$ permutations, although the total number of permutations for four vertices is $4! = 24$.

b) Neighbor Lists: Invariants that lead to finer-grain partitioning can be created 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.

Fig. 4 illustrates the partitioning produced by also incorporating the *neighbor list* invariant on the graph of Fig. 4(a). Specifically, Fig. 4(b) shows the partitioning produced by the vertex degrees and labels, and Fig. 4(c) shows the partitioning that is produced by also incorporating neighboring lists. The neighbor lists are shown in Fig. 4(d). For this example we were able to reduce the number of permutations that needs to be considered from $4! \times 2!$ to $2!$.

c) Iterative Partitioning: Iterative partitioning generalizes the idea of the neighbor lists, by incorporating the partition information [15]. 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.

The effect of iterative partitioning is illustrated in Fig. 5. In this example graph, 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

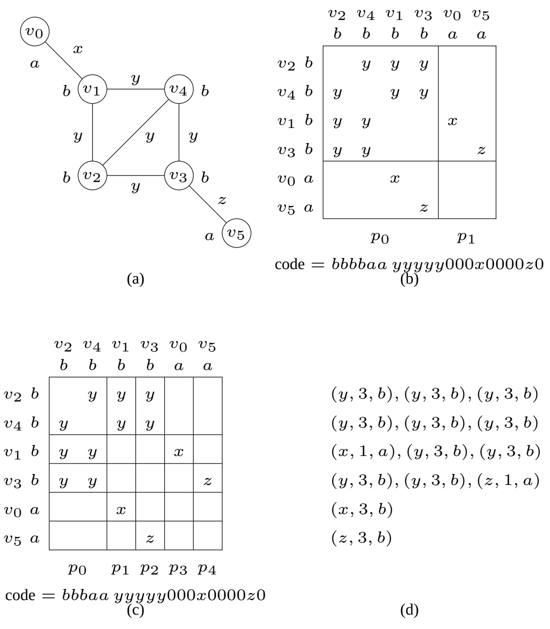

code = $bbbbaa\ yyyyy000x0000z0$
code = $bbbaa\ yyyyy000x0000z0$
$(y,3,b),(y,3,b),(y,3,b)$
$(y,3,b),(y,3,b),(y,3,b)$
$(x,1,a),(y,3,b),(y,3,b)$
$(y,3,b),(y,3,b),(z,1,a)$
$(x,3,b)$
$(z,3,b)$
(a)
(b)
(c)
(d)

Fig. 4. Use of neighbor lists

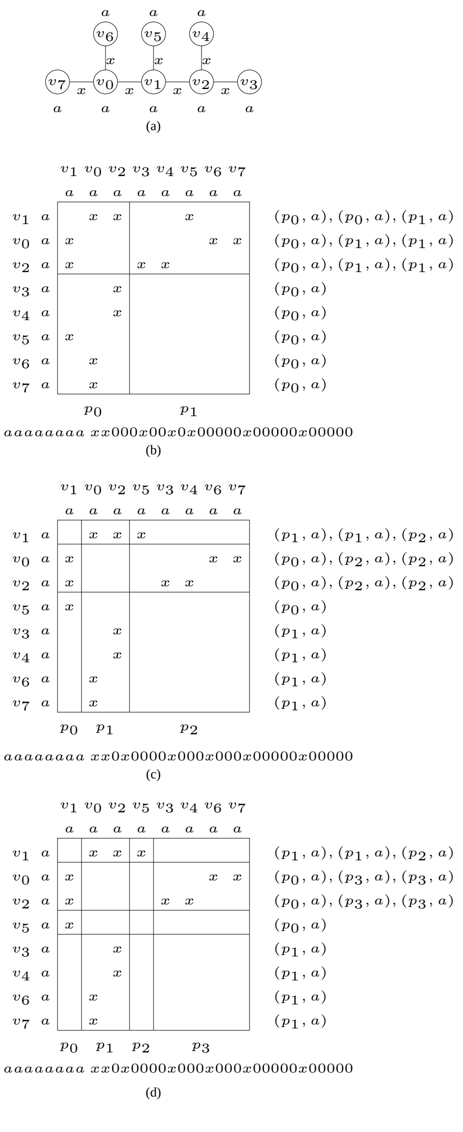

(a)
code = $aaaaaaaa\ xx000x00x0x00000x00000x00000$
(b)
code = $aaaaaaaa\ xx0x0000x000x000x00000x00000$
(c)
code = $aaaaaaaa\ xx0x0000x000x000x00000x00000$
(d)

Fig. 5. An example of iterative partitioning

partition they are sorted by their neighbor lists (Fig. 5(b)). The ordering of those partitions is based on the degrees and the labels of each vertex and its neighbors. 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 Fig. 5(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 Fig. 5(d).

B. Degree-based Partition Ordering

In addition to using the vertex invariants to compute a fine-grain partitioning of the vertices, the overall run-time 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 V-A 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.

C. Vertex Stabilization

Vertex stabilization is effective for finding isomorphism of graphs with regular or symmetric structures [31]. The key idea is to break the topological symmetry of a graph by forcing a particular vertex into its own partition, when the iterative partitioning leaves a large vertex partition which cannot be decomposed into smaller partitions anymore.

For example, consider a cycle $G = (V, E)$ of k edges where all the edges and the vertices have the same label. Each vertex is equivalent to any other since they are identical in terms of their degree, label, neighbors, and resulting partitions. As a result, a vertex cannot be distinguished from others and there will be only a singe partition containing all the k vertices. To obtain a canonical label under such a partitioning with the iterative partitioning only, it would require $O(k!)$ operations.

Vertex stabilization breaks such a regular structure by assuming that a particular vertex in a large partition with many equivalent vertices *is* different from the others. The selected vertex forms a new singleton partition for itself, which triggers for the rest of the vertices the successive iterative partitioning the details of which are described in Section V-A.0.c. Because we have chosen the vertex arbitrarily, we have to repeat the same process for the remaining vertices in the original partition. During the successive iterative partitioning, the vertex stabilization may be applied repeatedly if the iterative partitioning can not decompose a large partition effectively.

For example, in the case of a cycle with k edges, once a particular vertex v is chosen from the initial partition with all the k vertices, it breaks the symmetry and we immediately obtain $\lfloor (k-1)/2 \rfloor + 1$ partitions based on the distance from v to each vertex. Thus, the necessary number of permutations to compute the canonical label after this partitioning is $(\lfloor (k-1)/2 \rfloor + 1)!$. Because there are k such choices for the first vertex v, the entire computational complexity for the canonical labeling of G is bounded by $O(k(k/2)!)$ which is significantly smaller than $O(k!)$. Note that the vertex stabilization is not limited to cycles and that it is applicable to any types of graphs.

Once a partition becomes small enough, the straightforward permutation can be simpler and faster than vertex stabilization, in order to obtain a canonical label. Thus, our canonical labeling algorithm applies vertex stabilization only if the size of a vertex partition is greater than five.

For further details on vertex stabilization the readers should refer to a textbook on permutation groups such as [12].

VI. Experimental Evaluation

We experimentally evaluated the performance of FSG using actual graphs derived from the molecular structure of chemical compounds, and graphs generated synthetically. The first type of datasets allows us to evaluate the effectiveness of FSG for finding rather large patterns and its scalability to large real datasets, whereas the second one, a set of synthetic datasets, allows us 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 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+ (1.53 GHz) machines with 2 Gbytes main memory, running the Linux operating system. All the times reported are in seconds.

A. Chemical Compound Datasets

We derived graph datasets from two publicly available datasets of chemical compounds. The first dataset[3] contains 340 chemical compounds and was originally provided for the Predictive Toxicology Evaluation (PTE) Challenge [43], and the second dataset[4] contains 223,644 chemical compounds and

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

[4]http://dtp.nci.nih.gov/docs/3d_database/structural_information/structural_data.html

is available from the Developmental Therapeutics Program (DTP) at National Cancer Institute. From the description of chemical compounds in those two datasets, we created a transaction for a compound, a vertex for an atom, an edge for a bond. Each vertex has a vertex label assigned for its atom type and each edge has an edge label assigned for its bond type. In the PTE dataset there are 66 atom types and 4 bond types, and in the DTP dataset there are 104 atom types and 3 bond types. Each graph transaction obtained from the PTE and the DTP datasets has 27 and 22 edges on the average, respectively.

d) Results: Table II shows the results by FSG on four datasets derived from the PTE and DTP datasets. The first dataset was obtained by using all the compounds of the PTE dataset, whereas the remaining three datasets were obtained by randomly selecting 50,000, 100,000, and 200,000 compounds from the DTP dataset. There are three types of results shown in the table, the run-time in seconds (t), the size of the largest discovered frequent subgraph (k^*), and the total number of frequent patterns ($|\mathcal{F}|$) that were generated. The minimum support threshold was ranging from 10% down to 1.0%. Dashes in the table correspond to experiments that were aborted due to high computational requirements. All the results in this table were obtained using a single partition of the dataset.

FSG is able to effectively operate on datasets containing 200,000 transactions and discover all frequent connected subgraphs which occur in 1% of the transactions in approximately one hour. With respect to the number of transactions, the run-time scales almost linearly. For instance, with the 2% support, the run-time for 50,000 transactions is 263 seconds, whereas the corresponding run-time for 200,000 transactions is 1,343 seconds, an increase by a factor of 5.1. As the support decreases, the run-time increases reflecting the increase of the number of frequent subgraphs found from the input dataset. For example, with 200,000 transactions, the run-time for the 1% support is 4.2 times longer than that for the 3% support, and the number of found frequent subgraphs for the 1% support was 8.2 times more than that for the 3% support.

Comparing the performance on the PTE and DTP-derived datasets we notice that the run-time for the PTE dataset dramatically increases as the minimum support decreases, and eventually overtakes the run-time for most of the DTP-derived datasets. This behavior is due to the maximum size and the total number of frequent subgraphs that are discovered in this dataset (both of which are shown in Table II). For lower support values the PTE dataset contains both more and longer frequent subgraphs than the DTP-derived datasets do. This is due to the inherent characteristics of the PTE dataset because it contains larger and more similar compounds. For example, the PTE dataset contains 26 compounds with over 50 edges and the largest compound has 214 edges. Despite that, FSG requires 459 seconds for a support value of 2.0%, and is able to discover patterns containing over 22 edges.

1) Reducing Memory Requirement of TID lists: To evaluate the effectiveness of the database-partitioning-based approach (described in Section IV-B.1) for reducing the amount of memory required by TID lists (TID list memory), we performed a set of experiments in which we used two datasets derived from the DTP dataset containing 100,000 and 200,000 chemical compounds, respectively. For each dataset we used FSG to find all frequent patterns that occur in at least 1% of the transactions by partitioning the dataset in 2, 3, 4, 5, 10, 20, 30, 40, and 50 partitions. These results are shown in Table III. For each experiment, this table shows the total run-time, the maximum amount of TID list memory, and the maximum amount of memory required to store the pattern lattice (pattern lattice memory).

From these results we can see that the database-partitioning-based approach is quite effective in reducing the TID list memory, because it decreases almost linearly as the number of partitions. Moreover, the various optimizations described in Section IV-B.1 are quite effective in limiting the degradation in runtime of the resulting algorithm. For example, for the 200,000-compound dataset and 50 partitions, the runtime increases only by a factor of 3.4 over that for a single partition. Also, the pattern lattice memory increases slowly as the number of partitions increases, and unless the number of partitions is quite large, it is still dominated by the memory required to store the TID lists. Note that these results suggest that there is an optimal point for the number of partitions that leads to the least amount of memory, as the pattern lattice memory will eventually exceed the TID list memory as the number of partitions increases.

B. Synthetic Datasets

To evaluate the performance of FSG on datasets with different characteristics we developed a synthetic graph generator which can control the number of transactions $|\mathcal{D}|$, the average number of edges in each transaction $|T|$, the average number of edges $|I|$ of the potentially frequent subgraphs, the number of potentially frequent subgraphs $|\mathcal{S}|$, the number of distinct edge labels $|L_E|$, and the number of distinct vertex labels $|L_V|$ of the generated dataset. 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 [1], [2], [20].

The actual generator works as follows. First, it generates a set of $|\mathcal{S}|$ potentially frequent connected subgraphs called *seed patterns* whose size is determined by Poisson distribution with mean $|I|$. For each seed pattern, the topology and the labels of the edges and the vertices are chosen randomly. Each seed pattern has a weight assigned, which becomes a probability that the seed pattern is selected to be included in a graph transaction. The weights are calculated by dividing a random variable which obeys an exponential distribution with unit mean by the number of edges in the seed pattern, and the sum of the weights of all the seed patterns is normalized to one. We call this set $\mathcal{S}$ of seed patterns 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 frequent patterns.

Next, the generator creates $|\mathcal{D}|$ transactions. First, the generator determines the target size of each transaction, which is

TABLE II
RUN-TIME IN SECONDS FOR THE PTE AND DTP CHEMICAL COMPOUND DATASETS.

Support threshold	Run-time t[sec], Size of Largest Frequent Pattern k^*, and Number of Frequent Patterns $\|\mathcal{F}\|$											
	PTE $\|\mathcal{D}\| = 340$			DTP $\|\mathcal{D}\| = 50,000$			DTP $\|\mathcal{D}\| = 100,000$			DTP $\|\mathcal{D}\| = 200,000$		
[%]	t[sec]	k^*	$\|\mathcal{F}\|$	t[sec]	k^*	$\|\mathcal{F}\|$	t[sec]	k^*	$\|\mathcal{F}\|$	t[sec]	k^*	$\|\mathcal{F}\|$
10.0	3	11	844	74	9	351	156	9	360	337	9	373
9.0	3	11	977	80	9	400	169	10	420	366	10	442
8.0	4	11	1323	87	11	473	184	11	490	401	11	512
7.0	4	12	1770	94	11	562	200	11	591	437	11	635
6.0	6	13	2326	109	12	782	230	12	813	503	12	860
5.0	9	14	3608	122	12	1017	259	12	1068	570	12	1140
4.0	16	15	5935	146	13	1523	316	13	1676	705	13	1855
3.0	60	22	22758	186	14	2705	398	14	2810	894	14	3004
2.0	459	25	136927	263	14	5295	571	14	5633	1343	15	6240
1.0	—	—	—	658	16	19373	1458	16	20939	3776	17	24683

Note. Dashes indicate the computation was aborted because of the too long run-time.
$|\mathcal{D}|$: Number of transactions

TABLE III
RUN-TIME AND TID LIST MEMORY WITH PARTITIONING

$\|\mathcal{D}\|$	Run-time [sec]									
	Number of Partitions									
	1	2	3	4	5	10	20	30	40	50
100,000	1432	1878	2032	2189	2356	2924	3899	4842	6122	7459
200,000	3698	4494	5095	5064	5538	6418	7856	9516	11165	12670

$\|\mathcal{D}\|$	Maximum amount of memory for storing TID lists [Mbytes]									
	Number of Partitions									
	1	2	3	4	5	10	20	30	40	50
100,000	53.8	27.0	18.1	13.6	11.0	5.6	2.9	2.0	1.5	1.2
200,000	118	59.1	39.5	29.6	23.9	12.1	6.2	4.2	3.2	2.6

$\|\mathcal{D}\|$	Maximum amount of memory for storing pattern lattice [Mbytes]									
	Number of Partitions									
	1	2	3	4	5	10	20	30	40	50
100,000		1.4	1.5	1.5	1.6	1.9	2.5	3.2	3.8	4.3
200,000		1.7	1.8	1.8	1.8	2.0	2.4	2.8	3.2	3.6

Note. The two datasets are generated from the DTP dataset by sampling 100,000 and 200,000 chemical compounds. The minimum support $\sigma = 1.0\%$
Pattern lattice memory is left blank for a single partition because the lattice is not built.
$|\mathcal{D}|$: Number of transactions

a Poisson random variable whose mean is equal to $|T|$. Then, the generator selects a seed pattern from the seed pool, by rolling an $|\mathcal{S}|$-sided die. Each face of this die corresponds to the probability assigned to a seed pattern in the seed pool. If the size of the selected seed pattern fits in the target transaction size, the generator adds it to the transaction. If the size of the current intermediate transaction does not reach its target size, we keep selecting and putting another seed pattern into it. When adding the selected seed pattern makes the intermediate transaction size greater than the target transaction size, we add it for the half of the cases, and discard it and move onto the next transaction for the rest of the half. The generator adds a seed pattern into a transaction by connecting randomly selected pair of vertices, one from the transaction and the other from the seed pattern.

a) Results: Using this generator, we obtained a number of different datasets by varying the number of vertex labels $|L_V|$, the average size of the potentially frequent subgraphs $|I|$, and the average size of each transaction $|T|$, while keeping fixed the total number of transactions $|\mathcal{D}| = 10,000$, seed patterns $|\mathcal{S}| = 200$, and edge labels $|L_E| = 1$ respectively. Despite our best efforts in designing the generator, we observed that as both $|T|$ and $|I|$ increase, different datasets created under the same parameter combination lead to different run-

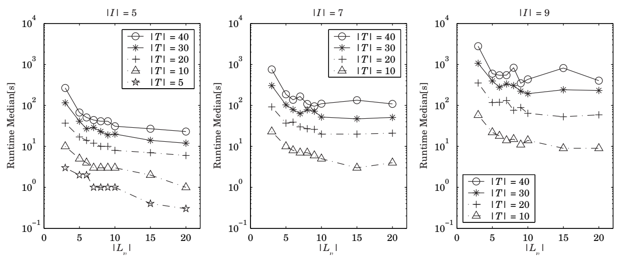


Fig. 6. Median of 10 run-times in seconds for synthetic data sets. $|T|$ is the average size of transactions, $|I|$ is the average size of seed patterns, and $|L_V|$ is the number of distinct vertex labels.

time because some may contain harder seed patterns (e.g., regular patterns with similar labels) than others do. To reduce this variability, we created ten different datasets for each parameter combination with different seeds for the pseudo random number generator and run FSG on all of them. The median of these run-times for each of the ten datasets is shown in Fig. 6. Note that these results were obtained using 2% as the minimum support threshold.

In general, the FSG's run-time decreases as the number of vertex labels $|L_V|$ increases, whereas it increases when the average size of the seed patterns $|I|$ or the average transaction size $|T|$ increases. These trends are consistent with the inherent characteristics of the datasets because of the following reasons: (i) As the number of vertex labels increases, the space of possible automorphisms and subgraph isomorphisms decreases—leading to faster candidate generation and frequency counting. (ii) As the size of the average seed pattern increases, because of the combinatorial nature of the problem, the total number of frequent patterns to be found from the dataset increases exponentially—increasing the overall run-time. (iii) As the size of the average transaction $|T|$ increases frequency counting by subgraph isomorphism becomes expensive, regardless of the size of candidate subgraphs. Moreover, the total number of frequent patterns to be found from the dataset also increases because more seed patterns can be put into each transaction. Both of these factors contribute in increasing the overall run-time.

VII. Related Work

Over the years, a number of different algorithms have been developed to find frequent patterns corresponding to frequent subgraphs in graph datasets. Developing such algorithms is particularly challenging and computationally intensive, as graph and subgraph isomorphisms play a key role throughout the computations. For this reason, a considerable amount of work has been focused on approximate algorithms [23], [28], [35], [46] that use various heuristics to prune the search space. However, a number of exact algorithms have been developed [5], [10], [17], [24], [25], [45] that guarantee to find all subgraphs that satisfy certain minimum support or other constraints.

Probably the most well-known heuristic-based approach is the SUBDUE system, originally developed in 1994, but has been improved over the years [8], [23]. SUBDUE finds patterns which can effectively compress the original input data based on the minimum description length principle, by substituting those patterns with a single vertex. To narrow the search-space and improve its computational efficiency, SUBDUE uses a heuristic beam search approach, which quite often results in failing to find subgraphs that are frequent. Nevertheless, despite its heuristic nature, its computational performance is considerably worse compared to some of the recent frequent subgraph discovery algorithms. Experiments reported in [17] for the PTE dataset [43], show that SUBDUE spends about 80 seconds on a Pentium III 900 MHz computer to find five most frequent substructures. In contrast, the FSG algorithm developed by our group [29], takes only 20 seconds on Pentium III 450 MHz to find all 3,608 frequent subgraphs that occur in at least 5% of the compounds.

A number of approaches for finding commonly occurring subgraphs have been developed in the context of inductive logic programming (ILP) systems [19], [33], [34], [38], [44], as graphs can be easily expressed using first-order logic. Each vertex and edge is represented as a predicate and a subgraph corresponds to a conjunction of such predicates. The goal of ILP-based approaches is to induce a set of rules capable of correctly classifying a set of positive and negative examples. In the case of graphs modeled by ILP systems, these rules usually correspond to subgraphs. Most ILP-based approaches are greedy in nature and use various heuristics to prune the space of possible hypotheses. Thus, they tend to find subgraphs that have high support and can act as good discriminators between classes. However, they are not guaranteed to discover all frequent subgraphs. A notable exception is the ILP system WARMR developed by Dehaspe and De Raedt [9] capable of finding all frequently occurring subgraphs. WARMR is not specialized for handling graphs, however, it does not employ any graph-specific optimizations and as such, it has high computational requirements.

In the last three years, three different algorithms have been developed capable of finding all frequently occurring subgraphs with reasonable computational efficiency. These are AGM by Inokuchi et al. [24], [25], the chemical substructure discovery algorithm developed by Borgelt and Berthold [5], and the gSpan algorithm developed by Yan and Han [45]. Among them, the early version of AGM [24] was developed prior to FSG, whereas the other algorithms were developed after the initial development of FSG [29].

AGM initially developed to find frequently induced subgraphs [24] and later extended to find arbitrary frequent subgraphs [25] discovers the frequent subgraphs using a breadth-first approach, and grows the frequent subgraphs one-vertex-at-a-time. To distinguish a subgraph from another, it uses a canonical labeling scheme based on the adjacency matrix representation. Experiments reported in [24] 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 the PTE dataset, as the minimum support threshold varied from 20% to 10%. Their modified algorithm [25] uses previously found embeddings of a frequent pattern in a transaction to save the subgraph isomorphism computation and improves the performance significantly at the expense of increased memory requirements.

The chemical substructure mining algorithm developed by Borgelt and Berthold [5], finds frequent substructures (connected subgraphs) using a depth-first approach similar to that used by dEclat [49] in the context of frequent itemset discovery. In this algorithm, once a frequent subgraph has been identified, it then proceeds to explore the input dataset for frequent subgraphs all of which contain the frequent subgraph. To reduce the number of subgraph isomorphism operations, it keeps the embeddings of previously discovered subgraphs and tries to extend the embeddings by one edge which is similar to the modified version of AGM [25]. In addition, since all the embeddings of the frequent subgraph are known, they project the original dataset into a smaller one by removing edges and vertices that are not used by any embeddings. Nevertheless, despite these optimizations, the reported speed of the algorithm is slower than that achieved by FSG. This is primarily due to two reasons. First, their candidate subgraph generation scheme does not ensure that the same subgraph is generated only once, as a result, they end up generating and determining the frequency of the same subgraph multiple times. Second, in chemical datasets, the same subgraph tends to have many embeddings (in the range of 20–200), as a result the cost of keeping track of them outweighs any benefits.

gSpan [45] finds the frequently occurring subgraphs also following a depth-first approach. Unlike the algorithm by Borgelt and Berthold, every time a candidate subgraph is generated, its canonical label is computed. If the computed label is the minimum one, the candidate is saved for further exploration of the depth search. If not, the candidate is discarded because there must be another path to the same candidate. By doing so, gSpan avoids redundant candidate generation. To ensure that these subgraph comparisons are done efficiently, they use a canonical labeling scheme based on depth-first traversals. In addition, gSpan does not keep the information about all previous embeddings of frequent subgraphs which saves the memory usage. However, all embeddings are identified on the fly, and use them to project the dataset in a fashion similar to that used by [5]. According to the reported performance in [45], gSpan and FSG are comparable on the PTE dataset, whereas gSpan performs better than FSG on synthetic datasets.

In addition to the work on frequent subgraph discovery, researchers has recently focused on the related but different problem of mining trees to discover frequently occurring subtrees. In particular, two similar algorithms have been recently developed by Asai et al. [4] and Zaki [48] that operate on rooted ordered trees and find all frequent subtrees. A rooted ordered tree is a tree in which one of its vertices is designated as its root and the order of branches from every vertex is specified. Because rooted ordered subtrees are in a special class of graphs, the inherent computational complexity of the problem is dramatically reduced as both graph and subgraph isomorphism problems for trees can be solved in polynomial time. Cong et al. [7] also proposed an algorithm to find frequent subtrees from a set of tree transactions, which allows wildcards on edge- and vertex-labels. Their algorithm first finds a set of frequent paths which may contain wildcards, allowing inexact match on both the structure as well as the edge and vertex labels.

VIII. Conclusions

In this paper we presented an algorithm, FSG, for finding frequently occurring subgraphs in large graph datasets, 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 [11]. Our detailed experimental evaluation shows that FSG can scale reasonably well to very large graph datasets 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 and candidate generation scheme, and its use of a TID list based approach for frequency counting. These three features combined, allow FSG to uniquely identify the various generated subgraphs, generate candidate patterns with limited degree of redundancy, and to quickly prune most of the infrequent subgraphs without having to resort to computationally expensive graph and subgraph isomorphism computations. Furthermore, we presented and evaluated a database-partitioning-based approach that substantially reduces FSG's memory requirement for storing TID lists with only a moderate increase in run-time.

Appendix

Correctness of FSG's Candidate Generation

Let C denote a connected size-$(k+1)$ subgraph which is to be generated as a valid candidate. A size-$(k+1)$ subgraph is a *valid candidate* if each of its connected size-k subgraphs is frequent. Let $\mathcal{F}(C) = \{F_i\}$ and $\mathcal{H}(C) = \{H_i\}$ denote sets of all connected size-k and size-$(k-1)$ subgraphs of C, respectively. For each $F_i \in \mathcal{F}(C)$, let c_i be the edge of C such that $F_i = C - c_i$. Likewise, for each $H_i \in \mathcal{H}(C)$, let a_i and b_i be the edges of C such that $H_i = C - a_i - b_i$. Let $\mathcal{H}^+(C) = \{H_i^+\}$ be the set of connected size-$(k-1)$ subgraphs of C such that for each H_i^+, there exists a pair of edges a_i^+ and b_i^+ that belong to C so that $H_i^+ = C - a_i^+ - b_i^+$ and both $C - a_i^+$ and $C - b_i^+$ are connected. Note that $\mathcal{H}^+(C) \subseteq \mathcal{H}(C)$ and it contains only those size-$(k-1)$ subgraphs of $\mathcal{H}(C)$ that regardless of the order in which the two edges are removed, the intermediate size-k subgraph remains connected. Let $H^* \in \mathcal{H}^+(C)$ denote a $(k-1)$-subgraph whose canonical label is the smallest among all the $(k-1)$-subgraphs in $\mathcal{H}^+(C)$. We will refer to H^* as the *pivotal core* of C. Let a^* and b^* be the edges deleted from C to obtain H^*, and we refer to a^* and b^* as the *pivotal edges*. Let F^{-a^*} and F^{-b^*} denote $C - a^*$ and $C - b^*$, respectively. We will refer to F^{-a^*} and F^{-b^*} as the *primary* frequent size-k subgraphs of C. Note that by construction, we have that $F^{-a^*} \in \mathcal{F}(C)$, $F^{-b^*} \in \mathcal{F}(C)$, and that H^* is a connected size-$(k-1)$ subgraph of both F^{-a^*} and F^{-b^*}.

Lemma 1: Given a connected size-$(k+1)$ valid candidate subgraph C, let H^*, a^*, b^* be the pivotal core and pivotal edges of C, respectively, and let F^{-a^*} and F^{-b^*} be the primary size-k subgraphs of C. Then, in each of the two primary size-k subgraphs of C, there exists at most one connected size-$(k-1)$ subgraph whose canonical label is smaller than that of the pivotal core H^*.

Proof: We prove the lemma only for F^{-a^*} and the same proof holds for F^{-b^*}.

Let H' be a connected size-$(k-1)$ subgraph of F^{-a^*} such that $\mathrm{cl}(H') < \mathrm{cl}(H^*)$. Note that since $F^{-b^*} \in \mathcal{F}(C)$, we have that $H' \in \mathcal{H}(C)$. Let a' and b' be the two edges of C that were deleted to obtain H', that is, $H' = C - a' - b'$. From the definition of H^*, we have that $H' \notin \mathcal{H}^+(C)$, otherwise we would have that $H^* = H'$. Without loss of generality, we assume that $C - a'$ is connected and that $C - b'$ is disconnected.

Now, since F^{-a^*} is a connected size-k subgraph of C that contains H', we know that F^{-a^*} will be either $C - a'$ or $C - b'$. However, because $C - b'$ is disconnected, we have that $F^{-a^*} = C - a'$, and because F^{-a^*} was initially obtained by deleting a^*, we have that $a' = a^*$. Thus, H' can be written as

$$H' = C - a^* - b', \tag{1}$$

where a^* is independent of H'. Moreover, because $C - b'$ is disconnected, b' must be a cut-edge that separates a^* from the rest of the graph.

Given the above, we can now show by contradiction that there exists only one connected size-$(k-1)$ subgraph of F^{-a^*}

whose canonical label is smaller than H^*. Assume that there exist two distinct connected size-$(k-1)$ subgraphs, H'_i and H'_j, such that $\text{cl}(H'_i) < \text{cl}(H^*)$ and $\text{cl}(H'_j) < \text{cl}(H^*)$. Let $H'_i = C - a'_i - b'_i$ and $H'_j = C - a'_j - b'_j$, and without loss of generality, assume that $C - a'_i$ and $C - a'_j$ are connected, and $C - b'_i$ and $C - b'_j$ are disconnected. Then, from Equation (1) we have that

$$H'_i = C - a'_i - b'_i = C - a^* - b'_i$$
$$H'_j = C - a'_j - b'_j = C - a^* - b'_j.$$

In order for $H'_i \neq H'_j$, we must have that $b'_i \neq b'_j$. However, because both b'_i and b'_j are cut-edges separating a^* from the rest of the graph, and because a^* can have only one such cut-edge (otherwise it cannot be separated by a single-edge deletion), we have that $b'_i = b'_j$. This is a contradiction, and thus $H'_i = H'_j$. ■

Using the above lemma, we can now prove the main theorem that shows that FSG's candidate generation approach, described in Section IV-A is correct.

Theorem 1: Given a connected size-$(k+1)$ valid candidate subgraph C, there exists a pair of connected size-k frequent subgraphs F_i and F_j such that $\mathcal{P}(F_i) \cap \mathcal{P}(F_j) \neq \emptyset$ that can be joined with respect to their common primary subgraph to obtain C.

Proof: Let $H^* = C - a^* - b^*$ be the pivotal core of C, and let $F^{-a^*} = C - a^*$ and $F^{-b^*} = C - b^*$. Since from Lemma 1 there exists at most one such common connected size-$(k-1)$ subgraph shared by F^{-a^*} and F^{-b^*} that has a smaller canonical label than H^*, it follows that $H^* \in \mathcal{P}(F^{-a^*})$ and $H^* \in \mathcal{P}(F^{-b^*})$; thus, $H^* \in \mathcal{P}(F^{-a^*}) \cap \mathcal{P}(F^{-b^*})$. Consequently, $F_i = F^{-a^*}$ and $F_j = F^{-b^*}$ are the desired size-k frequent subgraphs of C, and H^* is their common primary subgraph that leads to C. ■

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[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 Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, September 1994.

[3] Y. Amit and A. Kong. Graphical templates for model registration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(3):225–236, 1996.

[4] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. of the 2nd SIAM International Conference on Data Mining (SDM'02)*, pages 158–174, 2002.

[5] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

[6] 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.

[7] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. of the 2nd SIAM International Conference on Data Mining (SDM-2002)*, 2002.

[8] D. J. Cook and L. B. Holder. Graph-based data mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.

[9] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proc. of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 125–132. Springer-Verlag, 1997.

[10] 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 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press, 1998.

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

[12] J. D. Dixon and B. Mortimer. *Permutation Groups*, volume 163 of *Graduate Texts in Mathematics*. Springer-Verlag, 1996.

[13] 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.

[14] 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.

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

[16] 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.

[17] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In *Proc. of the 5th International Conference on Discovery Science*, 2002.

[18] B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, University of Limburg, Diepenbeek, Belgium, December 2002.

[19] J. Gonzalez, L. B. Holder, and D. J. Cook. Application of graph-based concept learning to the predictive toxicology domain. In *Proc. of the Predictive Toxicology Challenge Workshop*, 2001.

[20] 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.

[21] C. Hansch, P. P. Maolney, T. Fujita, and R. M. Muir. Correlation of biological activity of phenoxyacetic acids with hammett substituent constants and partition coefficients. *Nature*, 194:178–180, 1962.

[22] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining–a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

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

[24] 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.

[25] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.

[26] 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.

[27] R. D. King, S. H. Muggleton, A. Srinivasan, and M. J. E. Sternberg. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. In *Proc. of the National Academy of Sciences*, volume 93, pages 438–442, 1996.

[28] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in HIV data. In *Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-01)*, pages 136–143, 2001.

[29] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, November 2001.

[30] T. K. Leung, M. C. Burl, and P. Perona. Finding faces in cluttered scenes using random labeled graph matching. In *Proc. of the 5th IEEE International Conference on Computer Vision*, June 1995.

[31] B. D. McKay. Nauty users guide. http://cs.anu.edu.au/ bdm/nauty/.

[32] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[33] S. H. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3–4):245–286, 1995.

[34] S. H. Muggleton. Scientific knowledge discovery using Inductive Logic Programming. *Communications of the ACM*, 42(11):42–46, 1999.

[35] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *Proc. of the 8th ACM SIGKDD Internal Conference on Knowlege Discovery and Data Mining (KDD'2002)*, Edmonton, AB, Canada, July 2002.

[36] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of 2001 International Conference on Data Engineering (ICDE'01)*, pages 215–226, 2001.

[37] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *Knowledge and Data Engineering*, 9(3):435–447, 1997.

[38] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[39] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st Int. Conf. on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[40] 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.

[41] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th International Conference on Extending Database Technology (EDBT)*, volume 1057, pages 3–17, 1996.

[42] A. Srinivasan and R. D. King. Feature construction with inductive logic programming: a study of quantitative predictions of biological activity aided by structural attributes. *Data Mining and Knowledge Discovery*, 3(1):37–57, 1999.

[43] 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.

[44] A. Srinivasan, R. D. King, S. H. 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, 1997.

[45] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.

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

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

[48] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, July 2002.

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