Discovering Frequent Geometric Subgraphs*

Michihiro Kuramochi and George Karypis

Department of Computer Science/Army HPC Research Center
University of Minnesota, MN 55455
{kuram, karypis}@cs.umn.edu

Abstract

As data mining techniques are being increasingly applied to non-traditional domains, existing approaches for finding frequent itemsets cannot be used as they cannot model the requirement of these domains. An alternate way of modeling the objects in these data sets, is to use a graph to model the database objects. Within that model, the problem of finding frequent patterns becomes that of discovering subgraphs that occur frequently over the entire set of graphs. In this paper we present a computationally efficient algorithm for finding frequent geometric subgraphs in a large collection of geometric graphs. Our algorithm is able to discover geometric subgraphs that can be rotation, scaling and translation invariant, and it can accommodate inherent errors on the coordinates of the vertices. Our experimental results show that our algorithms requires relatively little time, can accommodate low support values, and scales linearly on the number of transactions.

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, 22, 8, 3, 20]. Nevertheless, as data mining techniques have been increasingly applied to non-traditional domains, such as scientific, spatial and relational datasets, situations tend to occur on which we can not apply existing itemset discovery algorithms, because these problems are difficult to be adequately and correctly modeled with the traditional market-basket transaction approaches.

Recently several approaches have been proposed for mining graphs in the context where the graphs are used to model relational, physical and scientific datasets [9, 19, 10, 12, 7, 11]. Modeling objects using graphs allows us to represent arbitrary relations among entities. 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 three dimensional orientation). Once these graphs have been created, recurrent substructures across different compounds become frequently occurring subgraphs.

This paper focuses on the related problem of finding frequently occurring geometric patterns in geometric graphs—graphs whose vertices have two or three dimensional coordinates associated with them. These patterns correspond to geometric subgraphs that have a sufficiently large support. Datasets arising in many scientific domains often contain such geometric information, and any patterns discovered in them are of interest if they preserve both the topological and the geometric nature of the pattern. Moreover, being able to directly find geometric patterns (as opposed using a post-processing step on the topological patterns), holds the promise of leading to algorithms that are significantly more scalable than their topological counter-parts. Despite the importance of the problem, there has been limited work in developing generic algorithms to find such patterns. The notable exceptions are the work by Wang et al. proposed several algorithms for automated finding of interesting substructures in chemical or biomolecule domain [18, 17], and the work by Chew et al. that proposed an approach to find common substructures in protein sequences using root mean squared (RMS) distance minimization [4]. However, these approaches are either computationally too expensive,

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

0-7695-1754-4/02 $17.00 © 2002 IEEE

or they find a restricted set of geometric subgraphs.

In this paper we present an algorithm called gFSG that is capable of finding frequently occurring geometric subgraphs in a large database of graph transactions. The key characteristic of gFSG is that it allows for the discovery of geometric subgraphs that can be rotation, scaling and translation invariant. Furthermore, to accommodate inherent errors on the coordinates of the vertices (either due to experimental measurements or floating point round-off errors), it allows for patterns in which the coordinates can match with some degree of tolerance. gFSG uses a pattern discovery framework that uses the level-by-level approach that was made popular by the Apriori [2] algorithm for finding frequent itemsets, and incorporate numerous computationally efficient algorithms for computing isomorphism between geometric subgraphs that are rotation, scaling and translation invariant, for candidate generation, and for frequency counting. Experimental results using a large database of over 20,000 real two dimensional chemical structures show that gFSG requires relatively little time, can accommodate low support values, and scales linearly on the number of transactions.

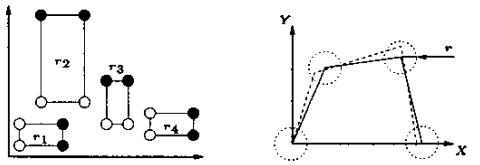
In the rest of the paper, we first defines basic notions and introduces notation. Then, we describe our problem setting of finding frequent geometric subgraphs, the outline and the details of the algorithm. Finally we experimentally evaluate our algorithm on a real dataset of chemical compounds and analyze the performance and the scalability of our algorithm.

2 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 vertex $v \in V$ has a label $l(v) \in L_V$, and each edge $e \in E$ is an unordered pair of vertices uv where $u, v \in V$. Each edge also a label $l(e) \in L_E$. L_E and L_V denote the sets of edge and vertex labels respectively. Those edge and vertex labels are not necessarily to be unique. If $|L_E| = |L_V| = 1$, then we call it an ***unlabeled graph***. If each vertex $v \in V$ of the graph has coordinates associated with it, in either the two or three dimensional space, we call it a ***geometric graph***. We will denote the coordinates of a vertex v by $c(v)$.

Two graphs $g_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$ are ***isomorphic***, denoted by $g_1 \sim g_2$, if they are topologically identical to each other, *i.e.*, there is a bijection $\phi : V_1 \mapsto V_2$ with $e = xy \in E_1 \leftrightarrow \phi(x)\phi(y) \in E_2$ for every edge $e \in E_1$ where $x, y \in V_1$. In the case of labeled graphs, this mapping must also preserve the labels on the vertices and edges, that means for every vertex $v \in V$, $l(v) = l(\phi(v))$ and for every edge $e = xy \in E$, $l(xy) = l(\phi(x)\phi(y))$. An ***automorphism*** of a graph $g = (V, E)$ is a bijection from the vertices in g to vertices in the same g. 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.*, to determine whether or not g_2 is included in g_1.

The notion of isomorphism and automorphism can be extended for the case of geometric graphs as well. A simple way of defining geometric isomorphism between two geometric graphs g_1 and g_2 is to require that there is an isomorphism ϕ that in addition to preserving the topology and the labels of the graph, to also preserve the coordinates of every vertex. However, since the coordinates of the vertices depend on the particular reference coordinate axes, the above definition is of limited interest. Instead, it is more natural to define geometric isomorphism that allows homogeneous transforms on those coordinates, prior to establishing a *match*. For the purpose of our work, we consider three basic types of geometric transformations: rotation, scaling and translation, as well as, their combination. In light of that, we define that two geometric graphs g_1 and g_2 are ***geometrically isomorphic***, if there exists an isomorphism ϕ of g_1 and g_2 and a homogeneous transform $\mathcal{T}$, that preserves the coordinates of the corresponding vertices, *i.e.*, $\mathcal{T}(c(v)) = c(\phi(v))$ for every $v \in V$. In this case, ϕ is called a ***geometric isomorphism*** between g_1 and g_2. Geometric automorphism is defined in an analogous fashion. Figure 1(a) shows some examples illustrating this definition. There are four geometric graphs drawn in this two dimensional example, each of which is a rectangle. Edges are unlabeled and vertex labels are indicated by their colors. The graphs $r_1 \sim r_2$ if all of the rotation, scaling and translation are allowed, and $r_1 \sim r_3$ if both rotation and translation are allowed, and $r_1 \sim r_4$ if translation is allowed.



(a) Sample isomorphic geometric graphs

(b) Tolerance r

Figure 1. Geometric isomorphism and r-tolerance

One of the challenges in using the above definition of geometric graph isomorphism is that it requires an exact match of the coordinates of the various vertices. Unfortunately, geometric graphs derived from physical datasets may contain small amounts of error, and in many cases, we are interested in find geometric patterns that are similar to, but slightly different from each other. To accommodate these requirements, we allow a certain amount of tolerance r when we establish a match between coordinates. That is, if $\| \mathcal{T}(c(v)) - c(\phi(v)) \| \leq r$ for every $v \in V$, we regard ϕ as a valid geometric isomorphism. We will refer to the parameter r as the ***coordinate matching tolerance***. A two

dimensional example is shown in Figure 1(b). We can think of an imaginary circle or sphere of a radius r centered at each vertex. After aligning the local coordinate axes of the two geometric graphs with each other, if a corresponding vertex in another graph is inside this circle or sphere, we consider that the two vertices are located at the same position. We will refer to these isomorphisms as ***r-tolerant geometric isomorphisms***, and will be the type of isomorphisms that will assume for the rest of this paper.

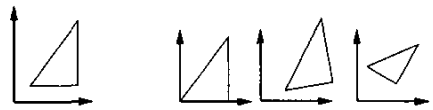
Finally, a graph is ***connected*** if there is a path between every pair of vertices in the graph. Given a graph $g = (V, E)$, a graph $g_s = (V_s, E_s)$ will be a ***subgraph*** of g if and only if $V_s \subseteq V$ and $E_s \subseteq E$. In a way similar to isomorphism, the notion of subgraph can be extended to ***r-tolerant geometric subgraphs*** in which the coordinates match after a particular homogeneous transform $\mathcal{T}$.

3 Problem Definition

The input for the frequent geometric subgraph discovery problem is a set of graphs D, each of which is an undirected labeled geometric graph, the minimum support σ such that $0 < \sigma \leq 1.0$, a set of allowed geometric transforms out of rotation, scaling and translation, and a coordinate matching tolerance r. The goal of the frequent geometric subgraph discovery is to find all connected undirected geometric graphs that have an r-tolerant geometric subgraph in at least $\sigma|D|$ graphs of the input dataset. We will refer to each of the graphs in D as a *geometric graph transaction* or simply a *transaction* when the context is clear, to D as the geometric graph transaction database, to σ as the *support* threshold, and each of the discovered patterns as the *r-tolerant frequent geometric subgraph*.

There are two key aspects in the above problem statement. First, we allow homogeneous transforms when we find instances of them in transactions. That is, a pattern can appear in a transaction in a shifted, scaled or rotated fashion. This greatly increases our ability to find interesting patterns. For instance in many chemical datasets, common substructures are at different orientation from each other, and the only way to identify them is to allow for translation and rotation invariant patterns. However, this added flexibility comes at a considerable increase in the complexity of discovering such patterns, as we need to consider all possible geometric configurations (a combination of rotation, scaling and translation) of a single pattern. For example, let us take a look at Figure 2 of a triangle. The triangle shown in Figure 2(a) has infinitely many geometric configurations, some of which are shown in Figure 2(b).

Second, we allow for some degree of tolerance when we try to establish a matching between the vertex-coordinates of the pattern and its supporting transaction. Even though this significantly improves our ability to find meaningful patterns and deal with measurement errors and errors due



(a) A triangle

(b) Sample geometric configurations of the same triangle

Figure 2. A triangle and its geometric configurations under rotation and translation

to floating point operations (that are occurred by applying the various geometric transforms), it dramatically changes the nature of the problem for the following reason. In traditional pattern discovery problems such as finding frequent itemsets, sequential patterns, and/or frequent topological graphs there was a clear definition of what was the pattern given its set of supporting transactions. On the other hand, in the case of r-tolerant geometric subgraphs, there are many different geometric representations of the same pattern (all of which will be r-tolerant isomorphic to each other). The problem becomes not only that of finding a pattern and its support, but also finding the right representative of this pattern. Note that this representative can be either an actual instance, or a composite of many instances. The selection of the right representative can have a serious impact on correctly computing the support of the pattern. For example, given a set of subgraphs that are r-tolerant isomorphic to each other, the one that corresponds to an *outlier* will tend to have a lower support than the one corresponding to the *center*. Thus, the exact solution of the problem of discovering all r-tolerant geometric subgraphs involves a *pattern optimization* phase whose goal is to select the right representative for each pattern, such that it will lead to the largest number of frequent patterns.

4 gFSG—Frequent Geometric Subgraph Discovery Algorithm

To solve the problem of finding the frequently occurring r-tolerant geometric subgraphs, as defined in Section 3, we developed an algorithm called gFSG. gFSG represents a first attempt for addressing this problem, and due to the complexity imposed by allowing a tolerance on how the different coordinates are matched, gFSG is not guaranteed to find all such frequent r-tolerant geometric subgraphs. In particular, gFSG uses a simple approach that is based on the first occurrence in determining the representative for each pattern, that may lead to under-counting the frequency of certain patterns. However, gFSG can be easily extended to perform a shape optimization for these representative patterns, and those extensions are described in Section 6.

In addition to that, to improve the performance of gFSG, it imposes two additional conditions that must be satisfied by the input database. First, the closest distance between any pair of points in each graph is least $2r$; and second,

the are no frequent subgraphs in the database that are $2r$-tolerant geometrically isomorphic to each other. Both of these conditions stem from the fact that we allow a tolerance to the mapping of the coordinates. The first condition allow as to efficiently compute geometric isomorphism between two graphs, whereas the second condition states that the frequent patterns in order to be distinguished as being different, they have to be reasonably far away from each other. If these conditions are not met, gFSG may fail to discover some patterns.

The gFSG algorithm follows the level-by-level structure of the Apriori algorithm used for finding frequent itemsets in market-basket datasets [2], and shares many characteristics with our previously developed frequent subgraph discovery algorithm for topological graphs [12]. The high level structure of our algorithm is shown in Algorithm 1. Edges in the algorithm correspond to items in traditional frequent itemset discovery. Our algorithm increases the size of frequent subgraphs by adding an edge one-by-one. gFSG initially enumerates all the frequent single, double and triple edge graphs. Then, based on the double and triple edge graphs, 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 6) of Algorithm 1. Next, it counts the frequency for each of these candidates, and prunes subgraphs that do no satisfy the support constraint (Lines 8–12). 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 gfsg(D, s) (Frequent Geometric Subgraph)

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

In the rest of this section we outline the algorithms used by gFSG to compute geometric graph isomorphism, generate the candidate subgraphs, and compute their frequency. Additional details on some of these algorithms can be found in [13].

Table 1. Notation used throughout the paper

Notation	Description
D	A dataset of graph transactions
t	A graph transaction in D
k-(sub)graph	A (sub)graph with k edges
g^k	A k-subgraph
C^k	A set of candidates with k edges
F^k	A set of frequent k-subgraphs

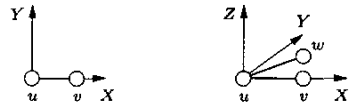
4.1 Geometric Graph Isomorphism

One of the key computational kernels used by gFSG is that of determining whether or not two geometric graphs are geometrically isomorphic to each other. This operation is used extensively when computing the size one, two, three frequent subgraphs and during candidate generation to essentially establish whether two patterns are identical or not.

In gFSG, a geometric isomorphism between two graphs g_1 and g_2 is computed by first identifying the possible geometric transformations that will map the vertices of g_1 within an r distance of the vertices of g_2, and then check each one of them to see if it preserves the topology (and the vertex and edge labels) of the two graphs. The details of this algorithm and additional optimizations are described in the rest of this section. Note that our description will assume that we are interested in geometric isomorphism that include all three transformations: rotation, scaling and translation.

Each geometric graph has its own coordinate system, or a reference frame. When we check the geometric isomorphism between g_1 and g_2, both should be in the same coordinate system. Nevertheless, there are infinitely many possible local coordinate systems we can choose, especially when we consider rotation invariant isomorphisms. Our algorithm limits this number by using a subset of the edges of the graph to define the coordinate axes. In the two dimensional space, it suffices to choose an edge and its direction to determine a local coordinate system (*e.g.*, the edge uv in Figure 3(a) as the X axis), and in the three dimensional space, two connected non-collinear edges (edges uv and uw in Figure 3(b)) form the XY plane and set the reference frame. These reference frames allow us to find translation and rotation invariant isomorphisms. To accommodate isomorphisms that are scale invariant, we can uniformly scale the graph such that one of these edges (*e.g.*, the one defining the X-axis) is of unit length. We will refer to each one of the graphs obtained by using the edge-defined reference frames as a ***geometric configuration***.

The algorithm for computing the geometric isomorphism is shown in Algorithm 2. First we check to see if g_1 and g_2 are of the same size, and if not, then the algorithm returns "false" indicating that these graphs are not isomorphic to each other. Then, the algorithm chooses an arbitrary geometric configuration for g_2 and tries to find a bijection between that configuration of g_2 and all possible geometric



(a) 2D (b) 3D

Figure 3. Edges for the basis of the local coordinate system

configurations of g_1. The bijection between a pair of geometric configurations is determined by iterating over each vertex of g_1 and pairing it with the closest vertex of g_2 with the same label that has not yet being paired. If at any given time, the pair of closest vertices are more than r-distance apart, the algorithm terminates the search for that configuration, as there is not an r-tolerant bijection between them. Once a bijection has been established, it is then checked to determine if it is a valid topological isomorphism (line 12–13).

The complexity of this algorithm is dependent on the size of the input geometric graphs. The number of possible geometric configuration is in $O(|V_1|^2)$ or $O(|V_1|^3)$ for the two or three dimensions respectively. Choosing the closest point out of $|V_2|$ vertices can be done in $O(|V_2|)$ time. It takes $O(|E_1|)$ steps to check the validity of a bijection ϕ. Therefore, the time complexity of geometric-isomorph is in $O(|V|^2|E|)$ for the two dimensions and $O(|V|^3|E|)$ for the three dimensions. Note that the expressions on the number of geometric configurations assume that g is dense. For most real-life problems, however, g will be sparse, dramatically reducing the overall complexity of this algorithm.

Algorithm 2 geometric-isomorph($g_1 = (V_1, E_1), g_2 = (V_2, E_2), r$) (Geometric Isomorphism)

```
1: if |V1| ≠ |V2| or |E1| ≠ |E2| then
2:    return false
3: choose one arbitrary geometric configuration of g2.
4: for each geometric configuration of g1 do
5:    change the coordinates of all the vertices in g1 according to the
      chosen geometric configuration.
6:    {assume g1 and g2 now share the same coordinate system}
7:    for each vertex v ∈ V1 do
8:       find the closest vertex u ∈ V2 from v such that l(u) = l(v)
9:       if ||c(v) − c(u)|| > r then
10:         break
11:      φ(v) ← u
12:   if φ is a valid topological isomorphism between g1 and g2 then
13:      return true
14: return false
```

To further reduce the overall time spent in checking whether or not two graphs are geometrically isomorphic, gFSG employs both topological properties and geometric transform invariants. The key idea is to first check those properties and invariants, and proceed computing an isomorphism only if these properties and invariants match. We will collectively refer to those topological properties and geometric transform invariants as *simple keys*.

The topological properties used by gFSG is the distribution of vertex and edge labels since they are easy to compute and check. Geometric transform invariants are values computed from a geometric graph which remain the same no matter how we rotate, scale or translate the original geometric graph. Because it does not change by those transforms, we only need to calculate the invariant once for each geometric graph regardless of its geometric configuration. In gFSG we used the normalized sum of distances between the geometric center of the graph and its vertices. Because the normalized sum of distances has the same dimension as distances, we use the same coordinate matching tolerance r for checking the equality between two normalized sums.

4.2 Candidate Generation

In the candidate generation phase, we create a set of candidates of size $k+1$, given frequent geometric k-subgraphs. Candidate geometric subgraphs of size $k+1$ are generated by joining two frequent geometric k-subgraphs. In order for two such frequent k-subgraphs to be eligible for joining they must contain the same geometric $(k-1)$-subgraph. We will refer to this common geometric $(k-1)$-subgraph among two k-frequent subgraphs as their *core*. The joining algorithm is basically the same as the one used in our previous work for finding topological frequent subgraphs [12] and we will not describe the details, which can be found in [12, 13].

4.3 Frequency Counting

Once candidate subgraphs have been generated, gFSG computes their frequency. 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 because there is no natural way to build the hash-tree for graphs.

In gFSG frequency counting is performed using a scheme that performs geometric graph isomorphism (using effective topological and geometric invariants) combined with the use of TID lists to reduce the set of graph transactions for which these isomorphisms need to be computed. The resulting algorithm makes it possible to efficiently count the frequency of the patterns and achieve a scalable memory usage.

In order to determine if a graph transaction contains a particular pattern, we need to check each geometric configuration of the graph against a particular geometric configuration of the pattern. This can be achieved using an algorithm similar to that for geometric graph isomorphism.

To reduce the execution time, we first use some topological properties and geometric transform invariants to quickly identify most of the miss-matches (as it was done in the case of graph isomorphism). The geometric transform invariants that we use for detecting whether or not a particular pattern can exist in a graph is based on edge-angle lists.

Let $\angle e_i e_j$ denote the angle formed by two connected edges e_i and e_j. Then, an edge-angle list eal(g) of a geometric graph g is a multiset where eal(g) = $\{\angle e_i e_j \mid \angle e_i e_j$, such that two distinct edges e_i, e_j share the same end point$\}$. We create the edge-angle list for each of the transactions and candidate subgraphs. Because edge angles are invariant against rotation, scaling and translation, if a geometric subgraph g is included in a transaction t, then eal(g) $\subseteq$ eal(t). Equality of two angles is again determined with a certain threshold as the vertex coordinate matching. For example, the graph g in Figure 4 has the following edge-angle list; eal(g) = $\{\angle e_1 e_2, \angle e_2 e_3, \angle e_3 e_1\} = \{a_1, a_1, a_2\}$, where $a_1 = \angle e_1 e_2$ and $a_2 = \angle e_1 e_3$. Note this is because $\angle e_1 e_2 = \angle e_2 e_3$. By using the comparison on the edge angle

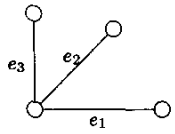


Figure 4. A star-shaped geometric graph with three edges

lists, we can easily detect cases where geometric subgraph isomorphism does not hold, without actually performing the subgraph isomorphism.

To limit the set of graph transactions that need to be checked while determining the frequency of a particular pattern, we use an approach motivated by the well-known technique of TID lists [6, 16, 23, 21, 22], but in a memory-efficient manner. We initially identify the set of frequent edge-angles over all graph transaction by exhaustive enumeration. Then, for each frequent edge angle, we create a list of transaction ID's that contain an instance of the edge angle. Let tid(a) denote a list of transaction ID's that contain an instance of the edge-angle a. For example, if a transaction $t = (V, E)$ is included in tid(a), then there is a pair of edges e_i and $e_j \in E$ such that $\angle e_i e_j = a$.

Now, let g be a candidate geometric graph and let eal(g) = $\{a_1, a_2, \ldots, a_n\}$ be its edge-angle list. The frequency of g is computed by checking to see if g is contained only in the graph transactions that are in the set corresponding to the intersection of the various tid(a_i) lists. Of course, if the intersection of these TID-lists is smaller than the minimum support, then the exact frequency of g is not even computed, as it is not frequent.

5 Experimental Evaluation

We experimentally evaluated the performance of gFSG using a set of real geometric graphs representing chemical compounds. In particular, we used a dataset containing 223,644 chemical compounds with their two dimensional coordinates that is available from the Developmental Therapeutics Program (DTP) at National Cancer Institute (NCI) [5]. These compounds were converted to geometric graphs in which the vertices correspond to the various atoms with their two dimensional coordinates and the edges correspond to the bonds between the atoms. The various atom types were modeled as vertex labels and the various types of bonds were modeled as edge labels. Overall, there are a total of 104 distinct vertex labels (atom types) and three distinct edges labels (bond types).

Note that even though the gFSG algorithm can find frequent geometric subgraphs in both two and three dimensional datasets, at the time of writing of this paper, we had only finished and optimized the two dimensional version of the code. For this reason our evaluation will only include two dimensional geometric graphs.

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.

5.1 Scalability With Respect To The Database Size

Our first set of experiments were designed to evaluate the scalability of gFSG with respect to the number of input graph transactions. Toward this goal we created five datasets with different number of transactions varying from 1,000 to 20,000. 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 23.

Using these datasets we performed two types of experiments. In the first experiment we used gFSG to find all frequently occurring geometric subgraphs that are rotation and translation invariant; whereas in the second set of experiments we used gFSG to find subgraphs that are also scaling invariant. For both sets of experiments, we used different values of support ranging from 0.25% up to 3%, and set r to 0.05.

Table 2 show the results obtained for the first and second set of experiments, respectively. For each individual experiment, these tables show the amount of time required to find the frequent geometric subgraphs patterns, the size of the largest discovered frequent pattern, and the total number of geometric subgraphs that were discovered.

There are three main observations that can be made from these results. First, gFSG scales linearly with the database

Table 2. Running times in seconds for chemical compound data sets which are randomly chosen from the DTP dataset. The column "Support" shows the used minimum support (%), the column with t is the running time in seconds, the column with l shows the size of the largest frequent subgraph discovered, and the column with $\#f$ is the total number of discovered frequent patterns.

Scaling	Support	Total Number of Transactions D														
		$D = 1000$			$D = 2000$			$D = 5000$			$D = 10000$			$D = 20000$		
	%	t[sec]	l	$\#f$	t[sec]	l	$\#f$	t[sec]	l	$\#f$	t[sec]	l	$\#f$	t[sec]	l	$\#f$
No	3.0	8	6	203	18	6	197	32	6	177	69	6	187	109	4	143
	2.0	14	6	371	38	6	356	62	6	321	128	6	321	216	5	269
	1.0	60	8	1124	134	7	1023	161	8	795	423	8	874	839	7	826
	0.5	263	8	3213	540	8	3009	622	9	2177	1514	9	2281	2465	7	2091
	0.25	790	10	10040	2051	10	10733	2224	10	6112	5351	9	6090	8590	10	5649
Yes	3.0	14	7	236	35	6	246	73	7	236	126	6	217	321	6	224
	2.0	26	7	415	72	7	430	124	7	404	205	7	352	522	7	359
	1.0	123	8	1393	315	8	1395	460	9	1189	1107	8	1295	1974	8	1019
	0.5	694	10	4960	1478	10	4623	2108	10	3593	4621	9	3869	9952	9	3354
	0.25	2043	13	14235	5674	12	15232	8972	12	11103	17421	9	10929	41895	11	11177

size. For most values of support, the amount of time required on the database with 20,000 transactions is 15–30 times larger than the amount of time required for 1,000 transactions. Second, as with any frequent pattern discovery algorithm, as we decrease the support the runtime increases and the number of frequent patterns increases. The overall increase in the amount of time tends to follow the increase in the number of patterns, indicating that the complexity of gFSG scales well with the number of frequent patterns. Third, comparing the scale invariant with the scale variant results, we can see that the latter is faster by almost a factor of two. This is because the number of discovered patterns is usually smaller, and each pattern has fewer supporting transactions, reducing the amount of time to compute their frequency.

5.2 Scalability With Respect To The Graph Size

Our second set of experiments was designed to evaluate the runtime of gFSG when the average size (*i.e.*, the number of edges) of each transaction increases. Again, using the whole set of chemical compounds, we created four different datasets by extracting 5,000 chemical compounds in the following way. First we sorted the original dataset based on the size of compounds. Then, we selected 5,000 compounds from four different locations of the sorted list, so that each dataset would have different transaction size. This resulted in four datasets whose average transaction size were 14, 19, 23 and 28. Because the chemical compounds are taken from the sorted order, almost all the transactions are in the same size as the average.

As with our earlier experiments, we used gFSG to find both scale invariant and scale variant patterns and we varied the minimum support from 3.0% to 0.25%. Tables 3 show the amount of time and the number of frequent patterns discovered in these two sets of experiments.

From these results we can see that as the average transaction size increases, the time required to find the frequent geometric subgraphs increases, as well. In most cases, this increase is at a higher rate than the corresponding increase on the size of each transaction. In general, the running time for finding the patterns when the average transaction size is 28, is about ten times longer than the running time for the average transaction size 14. This non-linear relation between the time complexity and the size of the transaction is due to the fact that the algorithm needs to explore a much higher search space, and is consistent with the time increases for other pattern discovery algorithms, such as those for finding frequent itemsets [14] and sequential patterns [15]. Nevertheless, gFSG is able to mine the largest dataset with a support of 0.25 in less than two hours. Also, comparing the scale invariant with the scale variant experiments, we can see that as before, finding the scale variant patterns is faster by about a factor of two.

6 Conclusion

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

One of the limitations of the current implementation of gFSG is it does not perform any shape optimizations on the representation of each geometric pattern. However, some simple, and yet powerful optimizations can be performed by using an approach motivated by k-means clustering. In this approach, the frequency counting phase will be performed multiple times. During each iteration, the candidate pattern will be incrementally adjusted, as its supporting set is being identified, to represent the centroid consensus pattern of the supporting set of graph transactions. In principle the itera-

Table 3. Running times in seconds for the four chemical compound data sets. Each dataset has a different average transaction size, from 14 to 28. The column "Support" shows the used minimum support (%), the column with t is the running time in seconds, the column with l shows the size of the largest frequent subgraph discovered, and the column with $\#f$ is the total number of discovered frequent patterns.

Scaling	Support	Average Transaction Size T											
		$T = 14$			$T = 19$			$T = 23$			$T = 28$		
	%	t[sec]	l	$\#f$	t[sec]	l	$\#f$	t[sec]	l	$\#f$	t[sec]	l	$\#f$
No	3.0	17	6	148	26	5	175	47	6	258	81	6	268
	2.0	27	7	270	49	5	324	81	7	418	288	6	792
	1.0	65	8	671	181	6	807	287	8	1164	1189	9	2975
	0.5	196	7	1613	610	7	2180	1002	9	3315	3454	11	8093
	0.25	497	9	3708	1726	9	5768	2622	12	8819	10172	12	24242
Yes	3.0	22	7	154	41	6	196	73	6	249	175	7	408
	2.0	36	7	264	81	6	325	142	7	420	543	8	993
	1.0	92	9	680	284	8	927	612	9	1385	2530	10	3936
	0.5	406	9	2072	1438	9	2859	3050	9	4620	9923	12	13178
	0.25	1226	10	5358	4997	10	8949	10824	12	15232	29686	14	38788

tive optimization on the shape of a pattern will stop, as soon as its support does not change in the course of an iteration. We are currently in the process of evaluating this algorithm and investigating efficient ways of implementing it.

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, 1998.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB*, 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th ICDE*, 1995.

[4] L. P. Chew, D. Huttenlocher, K. Kedem, and J. Kleinberg. Fast detection of common geometric substructure in proteins. In *Proc. of the 3rd ACM RECOMB*, 1999.

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

[6] B. Dunkel and N. Soparkar. Data organizatinon and access for efficient data mining. In *Proc. of the 15th IEEE ICDE*, 1999.

[7] S. Ghazizadeh and S. Chawathe. Discovering freuqent structures using summaries. Technical Report CS-TR-4364, Dept. of Computer Science, University of Maryland, 2002.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD*, 2000.

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

[10] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th PKDD*, 2000.

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

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 1st IEEE ICDM*, 2001.

[13] M. Kuramochi and G. Karypis. Discovering geometric frequent subgraphs. Technical Report 02-024, Dept. of Computer Science, University of Minnesota, 2002.

[14] M. Seno and G. Karypis. LPMiner: An algorithm for finding frequent itemsets using length decreasing support constraint. In *Proc. of the 1st IEEE ICDM*, 2001.

[15] M. Seno and G. Karypis. SLPMiner: An algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *Proc. of the 2nd IEEE ICDM*, 2002.

[16] 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*, 2000.

[17] J. T. L. Wang, Q. Ma, D. Shasha, and C. H. Wu. New techniques for extracting features from protein sequences. *IBM Systems Journal*, 40(2):426–441, 2001.

[18] X. Wang and J. T. L. Wang. Fast similarity search in three-dimensional structure databases. *Journal of Chemical Information and Computer Sciences*, 40(2):442–451, 2000.

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

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

[21] M. J. Zaki. Scalable algorithms for association mining. *IEEE TKDE*, 12(2):372–390, 2000.

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

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