# Discovering Frequent Geometric Subgraphs*

Michihiro Kuramochi and George Karypis

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

Technical Report 02-024

{kuram, karypis}@cs.umn.edu

Last updated on June 17, 2002

**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. We evaluated the performance of the algorithm using a large database of over 20,000 real two dimensional chemical structures, and 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, 20, 8, 3, 18]. We can use these itemsets for discovering association rules, for extracting prevalent patterns that exist in the datasets, or for classification. Nevertheless, as data mining techniques have been increasingly applied to non-traditional domains, such as scientific, spatial and relational datasets, situations tend to occur on which we can not apply existing itemset discovery algorithms, because these problems are difficult to be adequately and correctly modeled with the traditional market-basket transaction approaches.

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, 17, 10, 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

*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.

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 the 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 counterparts. 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 [16, 15], 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, 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. That means more than one distinct edges or vertices may have the same label. 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 $e = xy \in E$, $l(xy) = l(\phi(x)\phi(y))$. A graph $g = (V, E)$ is called ***automorphic*** if $g$ is isomorphic to itself via a non-identity mapping. 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 $\phi$ 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 $\phi$ 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, $\phi$ is called a ***geometric isomorphism*** between $g_1$ and $g_2$. Geometric automorphism is defined in an analogous fashion. Figure 1 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.

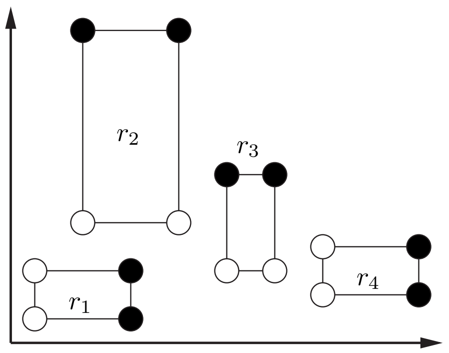


Figure 1: Sample isomorphic geometric graphs

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, equivalence of the two sets of coordinates is not straightforward. 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 $\phi$ 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 2. 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.

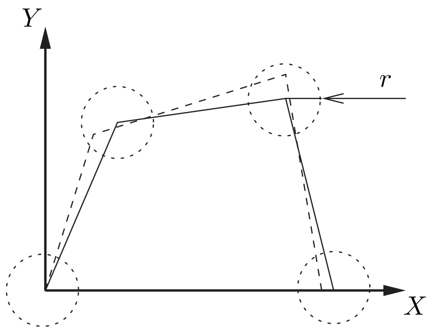


Figure 2: Tolerance $r$

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 Frequent Geometric Subgraph Discovery—Problem Definition

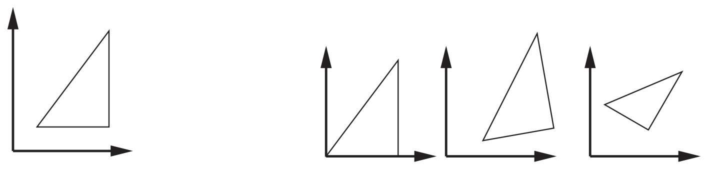
The input for the frequent geometric subgraph discovery problem is a set of graphs $D$, each of which is an undirected labeled geometric graph, a parameter $\sigma$ such that $0 < \sigma \leq 1.0$, a set of allowed geometric transforms out of rotation, scaling and translation, and a coordinate matching 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 $\sigma|D|\%$ of the input graphs.

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 $\sigma$ as the *support* threshold, and each of the discovered patterns as the *r-tolerant frequent geometric subgraph*.

There are four key aspects in the above problem statement. First, we are only interested in geometric 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 2, each graph (and discovered pattern) can contain vertices and/or edges with the same label. This greatly increases our modeling ability, as it allow us to find patterns involving multiple occurrences of the same entities and relations, but at the same time makes the problem of finding such frequently occurring subgraphs non-trivial [11]. In such cases, and 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 [7].

Third, 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 3 of a triangle. The triangle shown in Figure 3(a) has infinitely many geometric configurations, some of which are shown in Figure 3(b).


(a) A triangle

(b) Sample geometric configurations of the same triangle

Figure 3: A triangle and its geometric configurations under rotation and translation

Fourth, 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 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 [11]. The motivation behind using the level-by-level structure of the Apriori algorithm is that it achieves the highest amount of pruning compared with other algorithms such as *GenMax*, dEclat [20] and Tree Projection [1]. At the same time, the relatively simple algorithmic structure of this approach, allow us to focus on the non-trivial aspects of operating on geometric graphs.

The high level structure of our algorithm, called gFSG, is shown in Algorithm 1. Edges in the algorithm correspond to items in traditional frequent itemset discovery. Namely, as these algorithms increase the size of frequent itemsets by adding a single item at a time, our algorithm increases the size of frequent subgraphs by adding an edge one-by-one. 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.

In the rest of this section we describe the various algorithms used by gFSG to compute geometric graph isomorphism, generate the size one, two, and three frequent subgraphs, generate the candidate subgraphs, and computes their frequency.

**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:    $C^k \leftarrow$ gfsg-gen($F^{k-1}$)
7:    **for each** candidate $g^k \in C^k$ **do**
8:       $g^k$.count $\leftarrow 0$
9:       **for each** transaction $t \in D$ **do**
10:          **if** candidate $g^k$ is included in $t$ **then**
11:             $g^k$.count $\leftarrow g^k$.count $+ 1$
12:    $F^k \leftarrow \{g^k \in C^k \mid g^k.\text{count} \geq sD\}$
13:    $k \leftarrow k + 1$
14: **return** $F^1, F^2, \ldots, F^{k-2}$
```

Table 1: The meaning of the 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 principle, a geometric isomorphism between two graphs $g_1$ and $g_2$ can be computed in two different ways. First, we can identify all topological isomorphisms between $g_1$ and $g_2$, and then check each one of them to determine whether or not there is an allowable homogeneous geometric transformation that brings the corresponding vertices of the two graphs within an $r$ distance from each other (where $r$ is the coordinate matching tolerance). Alternatively, we can first identify the possible of 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.

In gFSG we experimented with both of those approaches, and we found that the latter approach is more efficient as it allow us for quicker *miss-matches*. Furthermore, in contrast to the purely topological graph isomorphism (used in the first approach) whose time complexity has not been proven to be in P or NP-complete, the second approach has the advantage of having a polynomial complexity. The details of this algorithm and additional optimizations are described in the rest of this section. Our description assumes 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 4(a) as the X axis), and in the three dimensional space, two connected non-collinear edges (edges $uv$ and $uw$ in Figure 4(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***.

Y
u
v
X
Z
Y
w
u
v
X

(a) 2D (b) 3D

Figure 4: Edges for the basis of the local coordinate system

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 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).

Algorithm 2 will correctly determine if two graphs are geometrically isomorphic or not provided that the distance between any pair of points in either $g_1$ or $g_2$ is greater than $2r$. This is because it pairs a vertex of $g_1$ to a single closest vertex of $g_2$, and it stops considering a particular geometric configuration as soon as a pairing is not $r$-tolerant. There may be a case in which there is an alternate pairing of the earlier vertices, that are $r$-tolerant but not necessarily the closest pairs, that will allow to find an overall $r$-tolerant pairing. Nevertheless, allowing for such bijections would substantially increase the complexity of the problem, without significantly increasing our modeling ability.

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 $\phi$. 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.

### 4.1.1 Using Simple Keys To Speedup The Computations

To further reduce the overall time spent in checking whether or not two graphs are geometrically isomorphic, gFSG employs a number of 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. Even though this approach does not decrease the worst-case asymptotic complexity, in practice it leads to dramatic speedup. 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. In addition to those, other properties such as vertex degree distributions, or the number of size-two paths and their label distributions can also be used. In our experiments, we found that

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

```
1: if $|V_1| \neq |V_2|$ or $|E_1| \neq |E_2|$ then
2:   return false
3: choose one arbitrary geometric configuration of $g_2$.
4: for each geometric configuration of $g_1$ do
5:   change the coordinates of all the vertices in $g_1$ according to the chosen geometric configuration.
6:   {assume $g_1$ and $g_2$ now share the same coordinate system}
7:   for each vertex $v \in V_1$ do
8:     find the closest vertex $u \in V_2$ from $v$ such that $l(u) = l(v)$
9:     if $\|c(v) - c(u)\| > r$ then
10:      break
11:    $\phi(v) \leftarrow u$
12:  if $\phi$ is a valid topological isomorphism between $g_1$ and $g_2$ then
13:    return true
14: return false
```

incorporating such more complicated properties do not dramatically improve the performance, because they lead to relatively long keys.

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 center and vertices. The normalized sum of distance is given by

$$d = \frac{1}{l_{\min}} \sum_{v \in V} \|c(v) - c(c)\|$$

where

$$l_{\min} = \min_{e \in E}(\text{length of } e)$$
$$c(c) = \frac{1}{|V|} \sum_{v \in V} c(v)$$

This invariant is useful for checking the identity during the candidate generation. For example, in Figure 5, the normalized sum for the graph with four edges is given by $d = (d_1 + \ldots + d_5)/l_{\min}$.

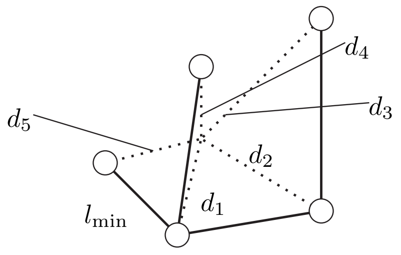


Figure 5: The normalized sum of distances from the center

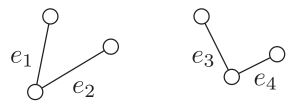
Because the normalized sum of distances has the same dimension as distances, we use the same coordinate matching tolerance $r$ for checking the equality between two normalized sums.

## 4.2 Generating Size One, Two And Three Frequent Subgraphs

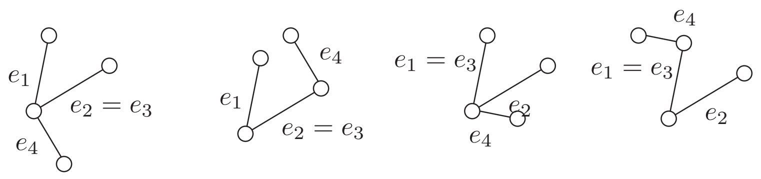
As breadth-first—or horizontal—algorithms for the traditional frequent itemset discovery start with enumerating singletons and doublet of items exhaustively before they actually begin the main iterative process of discovering frequent itemsets, our algorithm also starts with listing up all the frequent subgraphs of small size. Not only the size one and two, but also it exhaustively searches for the frequent size 3-subgraphs too and there are two distinct reasons for it.

One is the requirement for generating three dimensional candidates. If the input geometric graph transactions are described in the three dimensional space, the frequent patterns are also three dimensional geometric graphs. In the candidate generation phase which we will describe in Section 4.3, we obtain a candidate of size $k+1$ by joining two distinct frequent $k$-subgraphs that share a common subgraph of size $k-1$. To uniquely determine the merged geometric graph in the three dimensional space, that shared core must have at least three non-collinear vertices, that means we must start with at least frequent elementary geometric subgraphs of size three for the three dimensional space, and size two for the two dimensional space.

The other reason is the fact that we may be able to speed up the process of getting the size 3-subgraphs by the direct enumeration, rather than by merging two frequent 2-subgraphs, especially for the two dimensional cases. This is because in the two dimensional space with all the rotation, scaling and translation allowed, it is easy to merge two frequent subgraphs of size two into the one of size three, since all we need is to align one edge against another. For example, Figure 6(a) shows two frequent geometric subgraphs. If we focus on the edge $e_3$ in one of the size two graphs, those two size-2 subgraphs generate four candidates of size three as shown in Figure 6(b). The other edge $e_4$ will produce another four different candidates. As this example illustrates, potentially the number of size 3-candidates can be large.



(a) Two geometric 2-subgraphs



(b) Examples of size-3 candidates

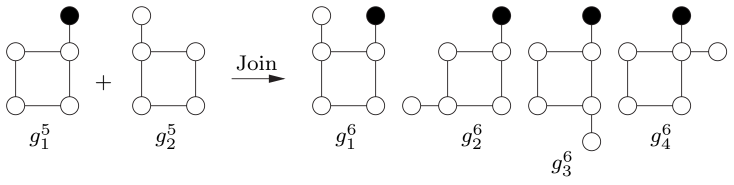
Figure 6: Two geometric 2-subgraphs and examples of generated candidates with rotation, scaling and translation

The key aspect of enumerating the size 1, 2 and 3 frequent subgraphs is to use the simple keys for the fast isomorphism checks, which are described in Section 4.1.1. The process consists of two phases. In the first phase, we record all the simple keys of size two or three graphs in every transaction and their frequencies in terms of the number of supporting transactions. By focusing on the simple keys of geometric subgraphs, we do not need to store all the possible coordinates of vertices. At the end of this phase, we can distinguish the simple keys of frequent subgraphs from the ones of infrequent subgraphs, and we will use those frequent simple keys in the next phase as a filter. In the second phase, we start over traversing all the transactions. This time we store all the necessary information for each of size two and three subgraphs including vertex coordinates and their frequencies. However, we do so only if the subgraph has one of the frequent simple keys identified during the first phase. Because of the filtering based on the frequent signatures, we can suppress the number of candidates we have to record during this process.

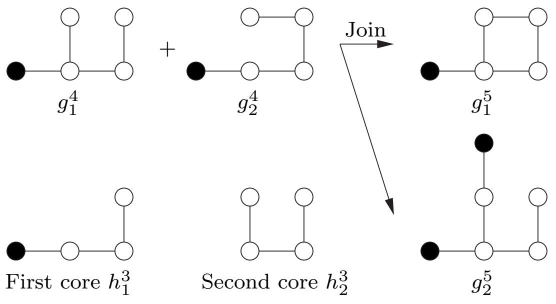
## 4.3 Candidate Generation

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*.

Unlike the joining of itemsets in which two frequent $k$-size itemsets lead to a unique $(k + 1)$-size itemset, the joining of two geometric subgraphs of size $k$ can lead to multiple distinct geometric subgraphs of size $k + 1$. This can happen because of two different reasons. First, there may be more than one automorphisms of a single core. The core itself may have multiple automorphisms, and each of them can lead to a different $(k + 1)$-candidate. In the worst case where a core of size $k - 1$ has a symmetric structure, the number of automorphisms can be at most $k - 1$. An example for this case is shown in Figure 7(a), in which the core—a square of 4 vertices—has more than one automorphism which result in four different candidates of size 6. Second, two frequent geometric subgraphs may have multiple geometric cores as depicted by Figure 7(b). Because every core has one fewer edge, for a pair of two $k$-subgraphs to be joined, the number of multiple cores is bounded by $k - 1$.



(a) Multiple automorphisms of a core



(b) Multiple cores

Figure 7: Three different cases of candidate joining

The overall algorithm for candidate generation is shown in Algorithm 3. For each pair of frequent subgraphs that share the same core, the gfsg-join is called at Line 6 to generate all possible candidates of size $k + 1$. For each of the candidates, the algorithm first checks if they are already in $C^{k+1}$. If they are not, then it verifies if all its $k$-subgraphs are frequent. If they are, gfsg-join then inserts it into $C^{k+1}$, otherwise it discards the candidate (Lines 7—17).

Algorithm 4 shows the joining procedure. Given a pair of $k$-subgraphs, first we determine whether those two subgraphs share the same core or not. We may discover more than one core at this step because of the second reason described above. Once we find such cores, for each of them, we generate all possible automorphisms of the core. This step corresponds to the first reason described above.

**Algorithm 3** gfsg-gen($F^k$) (Candidate Generation)

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

## 4.4 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 we implemented three different approaches for computing the frequency of each candidate geometric subgraph. These approaches offer different time-space trade-offs and are described in the rest of this section.

**Algorithm 4** gfsg-join($g_1^k, g_2^k, h^{k-1}$) (Join)

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

### 4.4.1 Counting By Geometric Subgraph Isomorphism

In the first approach, for each subgraph we scan each one of the graph transactions in the input dataset and determine if it is contained or not using geometric subgraph isomorphism. This operation requires to check each geometric configuration of the graph against a particular geometric configuration of the pattern, using an algorithm similar to that for geometric graph isomorphism.

To reduce the amount of time required for frequency counting using this approach, 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 $\text{eal}(g)$ of a geometric graph $g$ is a multiset where

$$\text{eal}(g) = \{\angle e_i e_j \mid \angle e_i e_j, \text{ such that two distinct edges } e_i,\, e_j \text{ share the same end point}\}$$

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 $\text{eal}(g) \subseteq \text{eal}(t)$. Equality of two angles is again determined with a certain threshold as the vertex coordinate matching. For example, the graph $g$ in Figure 8 has the following edge-angle list; $\text{eal}(g) = \{\angle e_1 e_2, \angle e_2 e_3, \angle e_3 e_1\} = \{a_1, a_1, a_2\}$, where $a_1 = \angle e_1 e_2$ and $a_2 = \angle e_1 e_3$. Note this is because $\angle e_1 e_2 = \angle e_2 e_3$.

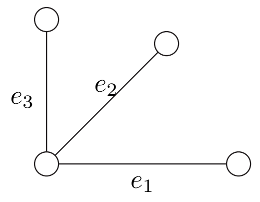


Figure 8: A star-shaped geometric graph with three edges

By using the comparison on the edge angle lists, we can easily detect cases where geometric subgraph isomorphism does not hold, without actually performing the subgraph isomorphism.

### 4.4.2 TID Lists

Determining the geometric subgraph isomorphism described in Section 4.4.1 is expensive and for this reason we also developed a frequency counting approach that uses Transaction ID (TID) lists, proposed by [6, 14, 21, 19, 20]. In this approach for each frequent subgraph we keep a list of transaction identifiers that support it. Now when we need to compute the frequency of $g^{k+1}$, we first compute the intersection of the TID lists of its frequent $k$-subgraphs. If the size of the intersection is below the support, $g^{k+1}$ is pruned, otherwise we compute the frequency of $g^{k+1}$ using subgraph isomorphism by limiting our search only to the set of transactions in the intersection of the TID lists. The advantages of this approach are two-fold. First, in the cases in which the intersection of the TID lists is bellow the minimum support level, we are able to prune the candidate subgraph without performing any subgraph isomorphism computations. Second, when the intersection set is sufficiently large, we only need to perform subgraph isomorphism for those graphs that can potentially contain the candidate subgraph and not for all the graph transactions.

However, the computational advantages of the TID lists come at the expense of higher memory requirements. In particular, when gFSG is working on finding the frequent patterns of size $(k+1)$, it needs to store in memory the TID lists for all frequent patterns of size $k$. Even though this approach can be extended to work in cases in which the amount of available memory is not sufficient [19], such an extension will require to perform multiple passes over the database, and we may not be able to get the same effect of pruning based on the downward closure property.

#### 4.4.3 Hybrid Approach

The last scheme that we developed can be thought of as a hybrid between the counting approach that uses subgraph isomorphisms and the one that uses TID lists.

In this approach, we initially identify the set of frequent edge-angles by exhaustive enumeration. Then, instead of checking the edge-angle lists of a candidate subgraph against that of each transaction, for each frequent edge angle, we create a list of transaction ID's that contain an instance of the edge angle. Let $\mathrm{tid}(a)$ denote a list of transaction ID's that contain an instance of the edge angle. For example, if a transaction $t = (V, E)$ is included in $\mathrm{tid}(a)$, then there is a pair of edges $e_i$ and $e_j \in E$ such that $\angle e_i e_j = a$. Suppose a candidate geometric graph $g$ has an edge-angle list $\mathrm{eal}(g) = \{a_1, a_2, \ldots, a_n\}$. To check the frequency of $g$, we perform the following operation successively. First, we initialize $l$ by $\mathrm{tid}(a_1)$, *i.e.*, $l = \mathrm{tid}(a_1)$. Second, we take the intersection of two TID lists $l$ and $\mathrm{tid}(a_2)$, and update $l$ by the intersection ($l = l \cap \mathrm{tid}(a_2)$). If $|l| < \sigma|D|$, that means $g$ does not have enough support. We repeat the process for the rest of edge angles $a_3, \ldots, a_n$. Every time we update $l$ by $l \cap \mathrm{tid}(a_i)$, we check the size of the resulting intersection since it is the frequency of the set of the edge angles, and if a geometric subgraph appears in $N$ transactions, then $|l| \geq N$.

When we take the intersections of $\mathrm{tid}(a_i)$, we start with the less frequent angles so that we may be able to detect the intersection may not be long enough at the earlier steps. Our experiments (not presented in this paper) showed that this approach is usually five times faster than the one based on subgraph isomorphism and only twice as slow as the one based on TID lists. However, it has the advantage of requiring substantially less memory than the TID-list based approach, and is the scheme that was used in all of our experiments.

## 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, scaling and translation invariant; whereas in the second set of experiments we are to find subgraphs that are only rotation and translation invariant. For both sets of experiments, we used different values of support ranging from 0.25% up to 5%, and set $r$ to 0.05.

Tables 2 and 3 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 patttern, and the total number of geometric subgraphs that were discovered.

Table 2: Running times in seconds for chemical compound data sets which are randomly chosen from the DTP dataset. The column with $\sigma$ 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.

| | Total Number of Transactions $D$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma$ | $D = 1000$ | | | $D = 2000$ | | | $D = 5000$ | | | $D = 10000$ | | | $D = 20000$ | | |
| % | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ |
| 5.0 | 8 | 6 | 119 | 14 | 6 | 113 | 34 | 6 | 114 | 75 | 5 | 117 | 179 | 6 | 111 |
| 4.5 | 9 | 6 | 137 | 20 | 6 | 138 | 45 | 6 | 139 | 83 | 5 | 132 | 209 | 6 | 126 |
| 4.0 | 10 | 6 | 168 | 22 | 6 | 157 | 52 | 6 | 160 | 96 | 6 | 151 | 244 | 6 | 154 |
| 3.5 | 12 | 6 | 206 | 30 | 6 | 209 | 57 | 6 | 184 | 110 | 6 | 185 | 281 | 6 | 182 |
| 3.0 | 14 | 7 | 236 | 35 | 6 | 246 | 73 | 7 | 236 | 126 | 6 | 217 | 321 | 6 | 224 |
| 2.5 | 20 | 7 | 314 | 55 | 7 | 329 | 85 | 7 | 287 | 150 | 6 | 259 | 357 | 7 | 268 |
| 2.0 | 26 | 7 | 415 | 72 | 7 | 430 | 124 | 7 | 404 | 205 | 7 | 352 | 522 | 7 | 359 |
| 1.5 | 48 | 7 | 687 | 107 | 7 | 613 | 218 | 8 | 630 | 410 | 7 | 552 | 842 | 7 | 526 |
| 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 |

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

Table 3: Running times in seconds for the same chemical compound data sets shown in Table 2, without scaling. The column with $\sigma$ shows the used minimum support (%), the column with $t$ is the running time in seconds, the column with $l$ show the size of the largest frequent subgraph discovered, and the column with $\#f$ is the total number of discovered frequent patterns.

| | Total Number of Transactions $D$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma$ | $D = 1000$ | | | $D = 2000$ | | | $D = 5000$ | | | $D = 10000$ | | | $D = 20000$ | | |
| % | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ |
| 5.0 | 4 | 6 | 90 | 7 | 6 | 80 | 19 | 6 | 92 | 39 | 6 | 96 | 79 | 4 | 69 |
| 4.5 | 4 | 6 | 105 | 7 | 6 | 89 | 20 | 6 | 101 | 40 | 6 | 107 | 89 | 6 | 84 |
| 4.0 | 4 | 6 | 116 | 8 | 6 | 106 | 23 | 6 | 121 | 45 | 6 | 122 | 82 | 4 | 92 |
| 3.5 | 5 | 6 | 154 | 12 | 6 | 146 | 26 | 6 | 144 | 60 | 6 | 150 | 93 | 4 | 111 |
| 3.0 | 8 | 6 | 203 | 18 | 6 | 197 | 32 | 6 | 177 | 69 | 6 | 187 | 109 | 4 | 143 |
| 2.5 | 10 | 6 | 250 | 24 | 6 | 251 | 47 | 6 | 236 | 105 | 6 | 238 | 155 | 4 | 191 |
| 2.0 | 14 | 6 | 371 | 38 | 6 | 356 | 62 | 6 | 321 | 128 | 6 | 321 | 216 | 5 | 269 |
| 1.5 | 26 | 7 | 610 | 61 | 7 | 559 | 79 | 6 | 457 | 162 | 6 | 436 | 380 | 6 | 421 |
| 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 |

## 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 5.0% to 0.25%. Tables 4 and 5 show the amount of time and the number of frequent patterns discovered in these two sets of experiments.

Table 4: 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 with $\sigma$ 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.

| | Average Transaction Size $T$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma$ | $T = 14$ | | | $T = 19$ | | | $T = 23$ | | | $T = 28$ | | |
| % | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ |
| 5.0 | 15 | 6 | 74 | 21 | 6 | 93 | 37 | 6 | 116 | 92 | 6 | 201 |
| 4.5 | 16 | 6 | 86 | 26 | 6 | 112 | 46 | 6 | 142 | 102 | 6 | 236 |
| 4.0 | 17 | 6 | 110 | 29 | 6 | 130 | 54 | 6 | 166 | 115 | 7 | 277 |
| 3.5 | 19 | 7 | 127 | 34 | 6 | 162 | 64 | 6 | 205 | 128 | 7 | 309 |
| 3.0 | 22 | 7 | 154 | 41 | 6 | 196 | 73 | 6 | 249 | 175 | 7 | 408 |
| 2.5 | 27 | 7 | 195 | 59 | 6 | 247 | 96 | 7 | 302 | 331 | 7 | 658 |
| 2.0 | 36 | 7 | 264 | 81 | 6 | 325 | 142 | 7 | 420 | 543 | 8 | 993 |
| 1.5 | 53 | 7 | 386 | 138 | 6 | 502 | 291 | 7 | 729 | 1002 | 8 | 1599 |
| 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 |

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 [12] and sequential patterns [13]. 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.

Table 5: Running times in seconds for the same chemical compound data sets shown in Table 4, without scaling.

| | Average Transaction Size $T$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma$ | $T = 14$ | | | $T = 19$ | | | $T = 23$ | | | $T = 28$ | | |
| % | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ | $t$[sec] | $l$ | $\#f$ |
| 5.0 | 11 | 5 | 68 | 14 | 6 | 79 | 20 | 5 | 97 | 36 | 5 | 128 |
| 4.5 | 12 | 5 | 78 | 17 | 5 | 97 | 26 | 5 | 122 | 46 | 5 | 158 |
| 4.0 | 13 | 5 | 93 | 17 | 5 | 113 | 29 | 5 | 148 | 54 | 5 | 182 |
| 3.5 | 15 | 6 | 118 | 20 | 5 | 136 | 32 | 5 | 182 | 64 | 6 | 213 |
| 3.0 | 17 | 6 | 148 | 26 | 5 | 175 | 47 | 6 | 258 | 81 | 6 | 268 |
| 2.5 | 20 | 6 | 189 | 35 | 5 | 237 | 64 | 6 | 338 | 142 | 6 | 428 |
| 2.0 | 27 | 7 | 270 | 49 | 5 | 324 | 81 | 7 | 418 | 288 | 6 | 792 |
| 1.5 | 33 | 8 | 389 | 84 | 5 | 473 | 107 | 7 | 583 | 481 | 8 | 1268 |
| 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 |

## 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 iterative 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, November 1998.

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

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

[4] L. P. Chew, D. Huttenlocher, K. Kedem, and J. Kleinberg. Fast detection of common geometric substructure in proteins. In *Proceedings of the 3rd ACM RECOMB International Conference on Computational Molecular Biology*, 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 International Conference on Data Engineering*, March 1999.

[7] 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.

[8] 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.

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

[10] 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.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 1st IEEE International Conference on Data Mining (ICDM)*, November 2001.

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

[13] M. Seno and G. Karypis. SLPMiner: An algorithm for finding frequent sequential patterns using length decreasing support constraint. Technical Report 02-023, Department of Computer Science, University of Minnesota, 2002.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

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