

Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-039

Discovering Frequent Geometric Subgraphs

Michihiro Kuramochi and George Karypis

October 21, 2004

Discovering Frequent Geometric Subgraphs^{*}

Michihiro Kuramochi¹ and George Karypis²

*Department of Computer Science & Engineering,
Digital Technology Center & Army HPC Research Center
University of Minnesota, MN 55455, USA*

Abstract

Data mining-based analysis methods are increasingly being applied to datasets derived from science and engineering domains that model various physical phenomena and objects. In many of these datasets, a key requirement for their effective analysis is the ability to capture the relational and geometric characteristics of the underlying entities and objects. Geometric graphs, by modeling the various physical entities and their relationships with vertices and edges, provide a natural method to represent such datasets. In this paper we present gFSG, a computationally efficient algorithm for finding frequent patterns corresponding to geometric subgraphs in a large collection of geometric graphs. gFSG 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 its performance using a large database of over 20,000 chemical structures, and our results show that it requires relatively little time, can accommodate low support values, and scales linearly with the number of transactions.

Key words: graph mining; pattern discovery; geometric subgraphs

^{*} This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, ACI-0133464 and ACI-0312828, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, by the Army High Performance Computing Research Center (AHPARC) under the auspices of the Department of the Army, Army Research Laboratory (ARL) under Cooperative Agreement numbers DAAH04-95-C-0008 and DAAD19-01-2-0014, and by the Digital Technology Center at the University of Minnesota. The content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

¹ Email: kuram@cs.umn.edu.

² Corresponding author. Email: karypis@cs.umn.edu.

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 [3,2,54,19,4,51]. Over the years, these frequent patterns have been used extensively to discover association rules, to extract prevalent patterns that exist in the datasets, and to build effective clustering and classification algorithms. Nevertheless, as data mining techniques have been increasingly applied to non-traditional domains, such as scientific, spatial and relational datasets, situations tend to occur in 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.

In recent years, labeled topological graphs have emerged as a promising abstraction to capture the characteristics of these datasets [32,22,26,30,25,27,35,17,28]. In this approach, each object to be mined is represented via a separate graph or the entire set of objects and their relations are represented via a single large graph. The vertices of these graphs correspond to the entities in the objects and the edges correspond to the relations between them. This graph-based modeling can directly capture many of the sequential, topological, and other relational characteristics of scientific datasets and allow us to solve problems that we could not solve previously. For example, graphs can be used to directly model the key topological characteristics of chemical structures (e.g., chemical compounds, protein molecules, *etc.*). Vertices in these graphs will correspond to different atoms or amino acids and the edges will correspond to atoms connected via bonds, or amino acids that are connected in the protein’s backbone or are connected via non-covalent bonds (i.e., contact points) in their 3D structure.

This paper focuses on the 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 are embedded in a sufficiently large number of graphs. 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. A prototypical example of such patterns are the two- and three-dimensional pharmacophores, which are used extensively in virtual screening methods for drug design [7,20,6]. These patterns correspond to certain chemical substructures that are present in chemical compounds and whose conserved geometry is critical for the compound’s ability to bind to a particular drug target.

Despite the importance of the problem, there has been limited work in develop-

ing general purpose algorithms to find such patterns as most of the existing research has focused on finding patterns only in topological graphs [30,25,17,29,48,8,49,23]. The notable exceptions are the work by Wang et al. proposed several algorithms for automated finding of interesting substructures in chemical or biomolecule domain [43,41] 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 [9]. 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, which follows the level-by-level approach made popular by the Apriori [3] algorithm, and incorporate numerous computationally efficient algorithms for (i) computing isomorphism between geometric subgraphs that are rotation, scaling and translation invariant, (ii) candidate generation, and (iii) frequency counting. In addition, GFSG incorporates an iterative pattern shape optimization algorithm whose goal is to identify the geometric shape of patterns that lead to the highest support. Experimental results using a large database of over 20,000 chemical structures show that GFSG requires relatively little time, can accommodate low support values, and scales linearly on the number of transactions.

The rest of this paper is organized as follows. Section 2 provides some background definitions and introduces the notation that is used through-out the paper. Section 3 formally defines the problem solved in this paper and discusses the various challenges associated with it. Section 4 surveys the related research in this area. Section 5 provides a detailed description of GFSG and the various algorithms used for subgraph isomorphism, candidate generation, and frequency counting. Section 6 presents a detailed experimental evaluation on a variety of chemical compound datasets. Finally, Section 7 provide some concluding remarks and discusses future research directions.

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 has a label $l(e) \in L_E$. L_E and L_V denote the sets of edge and vertex labels, respectively. These edge- and vertex-labels are not necessarily unique. That means more than one edge or vertex may have the same label. 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 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 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.

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

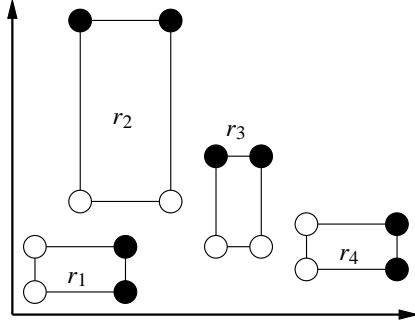


Fig. 1. Sample isomorphic geometric graphs.

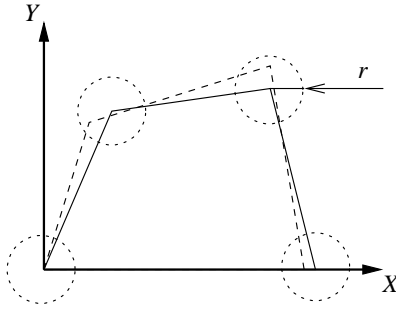


Fig. 2. Tolerance r .

amounts of error, and in many cases, we are interested in finding 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 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.

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 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 $\sigma|D|\%$ 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 σ 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 (i.e., edges) between some of the entities (i.e., 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 allows 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 [30]. In such cases, any frequent subgraph discovery algorithm needs to correctly identify how a particular subgraph maps to the vertices and edges of each graph transaction and can only be done by solving many instances of the subgraph isomorphism problem, which has been shown to be in NP-complete [16].

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

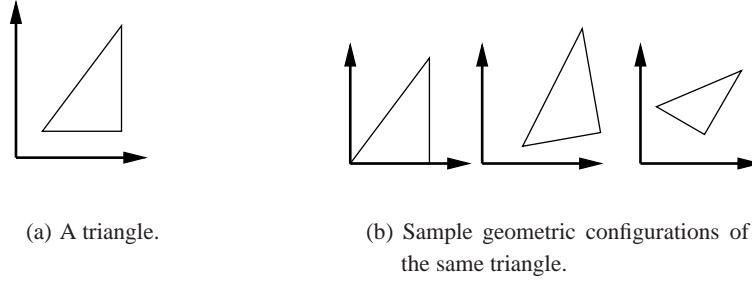


Fig. 3. A triangle and its geometric configurations under rotation and translation.

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 occur when we apply 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 is a clear definition of what is 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 (this is further discussed in Section 5.5).

4 Related Research

A number of different algorithms have been developed to find frequent patterns corresponding to frequent topological subgraphs in graph databases. Developing such algorithms is particularly challenging and their execution is computationally intensive, as graph and/or subgraph isomorphisms play a key role throughout the computations. For this reason, a considerable amount of work has been focused on approximate algorithms [50,22,27,33] that use various heuristics to prune the search space. However, a number of exact algorithms

have been developed [12,30,25,17,29,48,8,39,49,23,31] that are guaranteed to find all subgraphs that satisfy certain minimum support or other constraints. Among them, algorithms developed in the last three years [30,25,17,29,48,8,49,23,31], have been shown to achieve reasonably good performance and scalability. The enabling factors to the computational efficiency of these schemes have been (i) the development of efficient candidate subgraph generation schemes that reduce the number of times the same candidate subgraph is being generated, (ii) the use of efficient canonical labeling schemes to represent the various subgraphs; and (iii) the use of various techniques developed by the data-mining community to reduce the number of times subgraph isomorphism computations need to be performed.

Relatively little research has been done for finding frequent subgraphs in geometric graph databases. The most notable exception is the work by Jason T. L. Wang et al. [44,40,41,45]. In their most recent work [45], they propose an algorithm to find frequent substructures from a set of three dimensional graphs. Their approach starts with identifying blocks, or non-decomposable rigid substructures, and counts the frequency of them by geometric hashing [46]. By restricting the definition of a pattern to just those blocks, they do not have to perform any candidate generation. However, this approach cannot find arbitrary frequent geometric subgraphs and depending on the application domain, the assumption that the frequent subgraphs will have such block-based structure may not be applicable. An alternate approach was proposed by Parthasarathy and Coatney [34] that represents each subgraph as a list of pairs of vertices associated with a in-between distance. To reduce the size of vertex pair lists, their approach introduces a cut-off radius and ignores vertex-pairs with longer distances. To handle errors and noise, it discretizes distances using an appropriately chosen unit distance, and allows inexact matches using a technique called recursive fuzzy hashing which acts as a voting scheme.

In addition to the work on frequent subgraph discovery, researchers has recently focused on the related but different problem of mining trees to discover frequently occurring subtrees [42,5,53,10,47]. In particular, two similar algorithms have been recently developed by Abe et al. [1] and Zaki [53] that operate on rooted ordered trees and find all frequent subtrees. A rooted ordered tree is a tree in which one of its vertices designated as its root and the order of branches from every vertex is specified. Because rooted ordered subtrees are in a special class of graphs, the inherent computational complexity of the problem is dramatically reduced as both graph and subgraph isomorphism problems for trees can be solved in polynomial time. Cong et al. [11] also proposed an algorithm to find frequent subtrees from a set of tree transactions, which allows wildcards on edge- and vertex-labels. Their algorithm first finds a set of frequent paths which may contain wildcards, allowing inexact match on both the structure as well as the edge and vertex labels.

Finally, recent work on classification has shown that frequent subgraphs found by those algorithms can be used as effective features for other data mining tasks (e.g., [13,24]).

5 gFSG—Frequent Geometric Subgraph Discovery Algorithm

To solve the problem of finding the frequently occurring r -tolerant geometric subgraphs we developed an algorithm called gFSG. gFSG was designed to operate on a database of geometric graphs (either 2D or 3D) and find all subgraphs according to the problem statement described in Section 3. gFSG follows the level-by-level structure of the Apriori algorithm [3] and shares many characteristics with the previously developed frequent subgraph discovery algorithm for topological graphs [30]. The motivation behind this choice is the fact that the level-by-level structure of Apriori requires the smallest number of subgraph isomorphism computations during frequency counting, as it allows it to take full advantage of the downward closed property of the minimum support constraint and achieves the highest amount of pruning when compared with the most recently developed depth-first-based approaches such as dEclat [54], Tree Projection [2], and FPgrowth [19]. In fact, despite the extra overhead due to candidate generation that is incurred by the level-by-level approach, recent studies have shown that because of its effective pruning, it achieves comparable performance with that achieved by the various depth-first-based approaches, as long as the data set is not dense or the support value is not extremely small [21,18]. At the same time, the relatively simple algorithmic structure of this approach, allows us to focus on the non-trivial aspects of operating on geometric graphs.

To ensure that gFSG can correctly operate on geometric graphs and find frequent r -tolerant geometric subgraphs the input database must satisfy the following two conditions. First, the closest distance between any pair of points in each graph should be at most $2r$; second, the database does not contain frequent subgraphs that are $2r$ -tolerant geometrically isomorphic to each other. The first condition allows us 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 and/or undercount the frequency of some of the discovered patterns.

In addition gFSG provides two basic approaches for constructing the shape of the frequent geometric pattern. The first approach uses an arbitrarily selected embedding of a graph as its representative geometric shape, whereas the second approach employs an iterative shape optimization phase that tries

Table 1

Notations 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

to greedily select as its representative, a geometric shape that will maximize the frequency of the corresponding subgraph. These two approaches provide different performance-coverage trade-offs. The first approach is faster but it may fail to identify some of the frequent subgraphs, whereas the second approach is somewhat slower, but in general, finds a larger number of frequent geometric subgraphs. However, regardless of the method, due to their inherently heuristic nature, both of them may miss some of the patterns, especially as the value of r increases.

The high-level structure of gFSG is shown in Algorithm 1. (The notation used in this algorithm and in the rest of this paper is explained in Table 1.) 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, if desired, optimizes the shape of each pattern’s representative (Lines 10–11), and finally prunes subgraphs that do not satisfy the minimum support constraint (Line 12).

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, determine their frequency, and optimize their shape.

5.1 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. 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,

Algorithm 1 gFSG($D, s, adjust_type, N$) (Frequent Geometric Subgraph)

gFSG($D, s, adjust_type, N$)

- 1: $F^1, F^2, F^3 \leftarrow$ all frequent geometric subgraphs of size 1, 2 and 3 in D
- 2: $k \leftarrow 4$
- 3: **while** $F^{k-1} \neq \emptyset$ **do**
- 4: $C^k \leftarrow$ gFSG-GEN(F^{k-1})
- 5: **for each** candidate $g^k \in C^k$ **do**
- 6: $\triangleright g^k.S$ is the current set of supporting transactions.
- 7: $g^k.S \leftarrow$ COUNT-FREQUENCY(g^k, D)
- 8: **if** $|g^k.S| < s$ **then**
- 9: **continue**
- 10: **if** $adjust_type \neq \text{None}$ **then**
- 11: ADJUST-SHAPE($g^k, D, adjust_type, N$)
- 12: $F^k \leftarrow \{g^k \in C^k \mid g^k.count \geq sD\}$
- 13: $k \leftarrow k + 1$
- 14: **return** F^1, F^2, \dots, F^{k-2}

COUNT-FREQUENCY(g^k, D)

- 1: $S \leftarrow \emptyset$
- 2: **for each** transaction $t \in D$ **do**
- 3: **if** candidate g^k is included in t **then**
- 4: $S \leftarrow S \cup \{t\}$
- 5: **return** S

ADJUST-SHAPE($g^k, D, adjust_type, N$)

- 1: **for** $i = 1 \dots N$ **do**
- 2: compute the average of vertex coordinates of g^k across $g^k.S$.
- 3: **if** $adjust_type = \text{Simple}$ **then**
- 4: $g^k.S \leftarrow$ COUNT-FREQUENCY(g^k, D)
- 5: **else if** $adjust_type = \text{Support}$ **then**
- 6: $S' \leftarrow$ COUNT-FREQUENCY(g^k, D)
- 7: **if** $S' = g^k.S$ **then**
- 8: **return**
- 9: $g^k.S \leftarrow S'$
- 10: **else if** $adjust_type = \text{DWC}$ **then**
- 11: **if** g^k fails the downward closure check **then**
- 12: **return**
- 13: $g^k.S \leftarrow$ COUNT-FREQUENCY(g^k, D)

we can first identify the possible set of geometric transformations that 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 these approaches and found that the latter is more efficient as it allows us to terminate many of these mapping attempts earlier (i.e., quicker *miss-matches*). Furthermore, in contrast to the



Fig. 4. Edges for the basis of the local coordinate system.

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. Note that our description assumes that we are interested in geometric isomorphism that include all three transformations: rotation, scaling, and translation.

5.1.1 Transform and Map Approach

Each geometric graph has its own coordinate system or reference frame. When we check the geometric isomorphism between g_1 and g_2 , both should be in the same coordinate system. However, there are infinitely many possible local coordinate systems that 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 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 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

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

```

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 above algorithm 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. As discussed in the beginning of Section 5, gFSG requires that the minimum distance between any two pairs of vertices in the input graphs to be greater than $2r$, which ensures that the above restrictions are satisfied and thus, the correctness of this algorithm.

The complexity of Algorithm 2 depends on the size of the input geometric graphs. The number of possible geometric configuration is in $O(|V_1|^2)$ and $O(|V_1|^3)$ for the two and 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 overall time complexity of GEOMETRIC-ISOMORPH is in $O(|V|^2|E|)$ and $O(|V|^3|E|)$ for two- and three-dimensional patterns, respectively. 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.

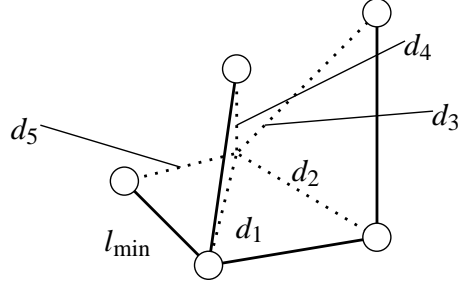


Fig. 5. The normalized sum of distances from the center.

5.1.2 Using Topological and Geometric Descriptors to Speedup the Computations

To further reduce the overall time spent in checking whether two graphs are geometrically isomorphic or not, gFSG computes various *descriptors* that capture certain topological properties and geometric transform invariants. Geometric transform invariants are certain quantities computed from a geometric graph that remain the same no matter how the original graph is rotated, scaled, or translated. The key idea behind this approach is to use these descriptors to quickly eliminate pairs of graphs that cannot be isomorphic to each other by simply checking to see whether their respective descriptors match or not. Since both the topological properties and the geometric transform invariants remain the same regardless of the geometric configuration of a particular graph, these descriptors need to be computed only once. Even though this approach does not decrease the worst-case asymptotic complexity, in most cases it leads to dramatic speedup as they are very effective in pruning the overall search space of possible isomorphisms.

For each geometric subgraph g , gFSG computes three descriptors. The first two are the distribution of vertex- and edge-labels, respectively, and the third is the normalized sum-of-distances between the vertices and the center of the graph. Specifically, the normalized sum-of-distances is given by

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

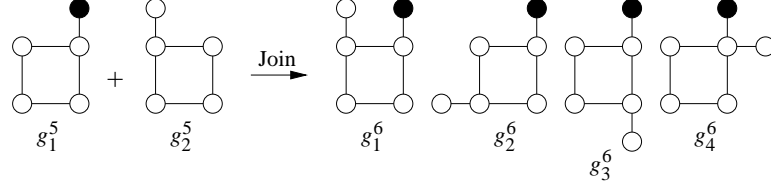
where $c(v)$ is the coordinate of vertex v , $c(c)$ is the coordinate of the g 's center point (i.e., $c(c) = (\sum_{v \in V} c(v))/|V|$), and l_{\min} is the length of g 's shortest edge (i.e., $l_{\min} = \min_{e \in E}(\text{length of } e)$) This definition is illustrated in Figure 5 for which the normalized sum-of-distances is given by $d = (d_1 + \dots + d_5)/l_{\min}$. Note that the sum-of-distances to the center is rotation and translation invariant, and by dividing this distance with l_{\min} , the resulting quantity becomes scaling invariant as well. Also, because the normalized sum-of-distances has the same dimension as distance, we use the same coordinate matching tolerance r for checking the equality between two normalized sums.

In addition to the above properties, a number of additional topological properties and geometric transform invariants can be used to increase the set of descriptors (e.g., vertex degree distributions, number of size-two paths and their label distributions, *etc.*). In our experiments, we found that incorporating such more complicated properties do not dramatically improve the performance as the cost of computing and comparing them outweigh the incremental savings in direct isomorphism-related computations.

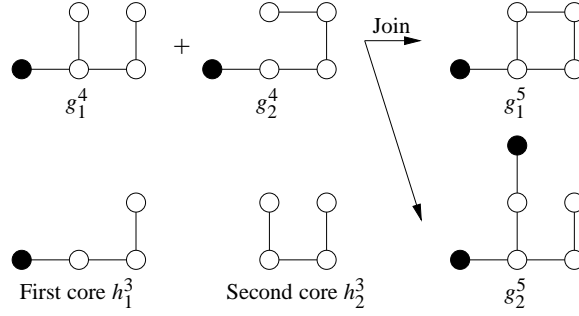
5.2 Generating Size One, Two, and Three Frequent Subgraphs

The first step in GFSG is to determine the frequent size one, two, and three r -tolerant geometric subgraphs using a direct enumeration approach. This is done primarily for two reasons. First, direct enumeration, if done efficiently, can significantly reduce the amount of time required to find size two and three subgraphs over an approach that uses the general candidate-generation and frequency-counting framework. This is consistent with observations of previous studies performed in the context of frequent itemsets and sequences. Second, GFSG’s candidate generation scheme (that will be described later in Section 5.3) obtains a candidate $(k + 1)$ -subgraph g^{k+1} by joining two distinct frequent k -subgraphs g_i^k and g_j^k that share a common $(k - 1)$ -subgraph h^{k-1} . To uniquely determine the geometry of the candidate subgraph, h^{k-1} must represent a rigid body. Thus, h^{k-1} must have two non-collinear vertices for 2D graphs, and three non-collinear vertices for 3D graphs. Consequently, the size k subgraphs that are joined should at least of size two or three for 2D and 3D, respectively.

GFSG performs the direct enumeration of these subgraphs by making extensive use of the various descriptors described in Section 5.1.2 to quickly eliminate the various subgraphs that are not frequent. Specifically, the frequent subgraphs are determined by performing two passes over the input graphs. During the first pass, GFSG records the descriptors of size two and three subgraphs that are present in every transaction and determines their frequencies in terms of the number of supporting transactions. During the second pass, for each size two and three subgraph whose descriptor’s frequency satisfies the minimum support requirement, it checks to see if it is r -tolerant geometric isomorphic to a previously encountered subgraph. If it is, it then updates the frequency of the corresponding subgraph, otherwise it creates a new *candidate* subgraph whose frequency is one. Our internal experimentation showed that this two-pass approach is very effective in dramatically reducing the number of subgraphs that need to be considered during the second pass.



(a) Multiple automorphisms of a core.



(b) Multiple cores.

Fig. 6. Three different cases of candidate joining.

5.3 Candidate Generation

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, the core itself can have multiple automorphisms, each potentially leading 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 $k - 1$. This case is illustrated in Figure 6(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 6(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 $k - 1$.

The overall algorithm for candidate generation (GFSG-GEN) is shown in Algorithm 3. For each pair of frequent subgraphs that share the same core it calls GFSG-JOIN (Line 6) to generate all possible candidates of size $k + 1$. Then, for

Algorithm 3 GFSG-GEN(F^k) (Candidate Generation)

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

each of these candidates, it checks to see if they have already been generated (i.e., they are already in C^{k+1}), and discards them if they have. Otherwise, it inserts them in C^{k+1} as long as all of its k -subgraphs are frequent (i.e., satisfy the downward closure property). The actual joining procedure (GFSG-JOIN) is shown in Algorithm 4. Given a pair of k -subgraphs g_1^k and g_2^k that share the same core $(k - 1)$ -subgraph h^{k-1} , GFSG-JOIN first generates the set M of all possible automorphisms of h^{k-1} and then identifies the pair of edges e_1 and e_2 from g_1^k and g_2^k , respectively that are not part of the core. Then, for each automorphism in M , it generates a candidate $(k + 1)$ -subgraph by adding the edges e_1 and e_2 .

5.4 Frequency Counting

Once candidate subgraphs have been generated, GFSG computes their frequency. In the context of the original Apriori algorithm, the frequency counting is performed efficiently by storing the candidate itemsets in a hash-tree data structure and then scanning each transaction to determine which of the itemsets in the hash-tree it supports. However, developing such an algorithm for frequent subgraphs is challenging (if not impossible) because there is no natural way to build a hash-tree-like structure for graphs. For this reason, GFSG's frequency counting approach considers one candidate subgraph at-a-time and tries to determine the transactions that it is contained in. Within this framework, GFSG implements three different approaches, which are described

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  $M$ 
```

in the rest of this section, that offer different time-space trade-offs.

5.4.1 Counting by Geometric Subgraph Isomorphism

In the first approach, for each subgraph GFSG scans each one of the graph transactions in the input dataset and determines 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, GFSG uses a descriptor-based approach (as it was done in the case of graph isomorphism in Section 5.1.2) to quickly detect whether a particular candidate subgraph can exist in a transaction or not. This is done by using an additional geometric transform invariant, referred to as the *edge-angle list*, that takes into account the multiset of angles between each pair of edges incident on the same vertex. Specifically, let $\angle e_i e_j$ denote the angle formed by two connected edges e_i and e_j . Then, the edge-angle list $\text{eal}(g)$ of a geometric graph g is the multiset given by

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

Note that edge-angles are invariant against translation, scaling, and rotation. To illustrate this definition, consider the graph g shown in Figure 7. This graph has three different pairs of edges that are incident on the same vertex, namely, $e_1 e_2$, $e_2 e_3$, and $e_3 e_1$, which result to an edge-angle list of $\text{eal}(g) = \{\theta_1, \theta_1, \theta_2\}$ (in this example $\angle e_1 e_2 = \angle e_2 e_3$).

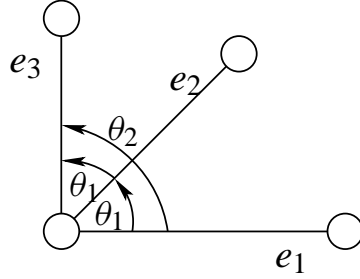


Fig. 7. A star-shaped geometric graph with three edges.

Given a transaction t and a candidate subgraph g , it follows directly from the geometric transform invariant nature of edge-angles that if g is contained in t , then $\text{eal}(g) \subseteq \text{eal}(t)$. Thus, by comparing the edge-angle lists, GFSG can easily detect cases where geometric subgraph isomorphism does not hold, without actually performing subgraph isomorphism. Note that equality between two angles is determined with a certain threshold as the vertex coordinate matching.

5.4.2 TID Lists

Determining the geometric subgraph isomorphism described in Section 5.4.1 is expensive and for this reason we also developed a frequency counting approach that uses Transaction ID (TID) lists, proposed by [15,38,55,52,54]. In this approach for each frequent subgraph GFSG keeps a list of transaction identifiers that support it. Now when the frequency of g^{k+1} is computed, GFSG first computes the intersection of the TID lists of its frequent k -subgraphs. If the size of the intersection is below the support, g^{k+1} is pruned, otherwise it computes the frequency of g^{k+1} using subgraph isomorphism by limiting the search to only 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 below 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 [52], 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.

5.4.3 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, GFSG initially identifies the set of frequent edge-angles by exhaustive enumeration. Then, for each frequent edge angle, it creates a list of transaction IDs that contain an instance of the edge angle. Let $\text{tid}(\theta)$ denote a list of transaction ID's that contain an instance of the edge angle θ . Suppose a candidate geometric graph g has an edge-angle list $\text{eal}(g) = \{\theta_1, \theta_2, \dots, \theta_n\}$. To compute the frequency of g , GFSG proceeds in a fashion similar to the TID-list intersection approach (Section 5.4.2) but this time it computes the intersections of the TID-lists of the various edge-angles that g contains. That is, it computes $l = \cap_i \text{tid}(\theta_i)$. The sequence in which these intersections are performed is based on the increasing order of the length of the angle-specific TID lists. This allows it to quickly detect the cases in which the final intersection will not be sufficiently large. It is easy to see that (i) if g is frequent, then $|l| \geq \sigma|D|$, and (ii) l contains all the transactions that can potentially support g .

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.

5.5 Iterative Shape Adjustment

Through-out our description of GFSG, we assumed that the geometric shape of a frequent pattern is determined when the corresponding candidate is first created by joining two smaller frequent subgraphs (the details are described in Section 5.3). As discussed in Section 3, the shape of the resulting candidate may not necessarily be optimal in the sense that it may not represent the geometric configuration that leads to the highest frequency. As a result, some of its occurrences may be missed with respect to the r -distance and the frequency of the pattern will tend to be smaller than what it should be.

To alleviate this problem GFSG implements a simple, and yet powerful mechanism to *adjust* the shape of a candidate subgraph so that to maximize its occurrence frequency. The overall idea behind this approach is to perform the frequency counting phase multiple times, each time incrementally adjusting the shape of the candidate subgraph so that to represent the consensus pat-

tern of its supporting embeddings. Specifically, for each candidate subgraph the frequency counting operation (i.e., the scanning of the input database looking for occurrences of the candidate) is repeated multiple times. For every iteration, the set of occurrences of the candidate subgraph are identified and the mean of the coordinates of the vertices is computed across these multiple occurrences. The resulting set of vertex-coordinates become the *adjusted* shape of the candidate subgraph. The motivation behind this approach is that by averaging the shape we are able to obtain a better overall representative of the various occurrences, which it can lead to higher occurrence frequency. Note that in this approach, the final geometric configuration of a pattern will not correspond to a particular embedding, but it will correspond to a *centroid*-like structure.

Within the above framework, gFSG implements three different schemes for determining when this iterative shape adjustment process will terminate. The first scheme, referred to as *simple adjustment* (SA), terminates the overall process after performing a user-specified fixed number N of iterations. The second scheme, referred to as *supporting transaction monitoring* (STM), extends the simple adjustment scheme by keeping track of the changes in the set of supporting transactions and allows for the early termination of the overall process as soon as the supporting set of a pattern does not change in two successive iterations. Finally, the third scheme, referred to as *downward closure check* (DWC), extends the supporting transaction monitoring scheme by terminating the overall process when the resulting candidate subgraph fails to satisfy the downward closure property. This is because due to the accumulated adjustments, the shape of a candidate subgraph may change significantly from its initial configuration. Once the change becomes large, it may happen that the modified candidate does not satisfy the downward closure property in terms of the given r -tolerance threshold. Depending on the application, obtaining such patterns may not be desirable.

6 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) [14]. 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

Table 2
runtime with scaling, rotation and translation.

σ %	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	$\#f$
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

σ : the minimum support threshold [%].

t : Runtime in seconds.

l : The size of largest frequent subgraphs.

$\#f$: The total number of frequent subgraphs discovered.

edges labels (bond types).

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

6.1 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 found the 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. The results from these experiments are shown in Tables 2 and 3, 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 size. For most values of support, the amount of time required on the database with 20,000 transactions is 15–30

Table 3
Runtime with rotation and translation only.

σ %	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	$\#f$
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

σ : the minimum support threshold [%].

t : Runtime in seconds.

l : The size of largest frequent subgraphs.

$\#f$: The total number of frequent subgraphs discovered.

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.

6.2 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. 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 the compounds. Then, we selected 5,000 compounds from four different locations of the sorted list, so that each dataset would have different average transaction sizes. This resulted in four datasets whose average transaction size were 14, 19, 23, and 28 edges, respectively. Because the chemical compounds are taken from the sorted order, the size of almost all the transactions is the same as that of 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
Runtime and average transaction size.

σ %	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	$\#f$
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

σ : the minimum support threshold [%].

t : Runtime in seconds.

l : The size of largest frequent subgraphs.

$\#f$: The total number of frequent subgraphs discovered.

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 runtime for finding the patterns when the average transaction size is 28, is about ten times longer than the runtime 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 larger search space, and is consistent with the time increases for other pattern discovery algorithms, such as those for finding frequent itemsets [36] and sequential patterns [37]. 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.

6.3 Effectiveness of Shape Adjustment

Table 6 shows the effect of the different iterative shape adjustment methods described in Section 5.5 for different values of the maximum number of allowed adjustment iterations (N). Two datasets are used in these experiments, one with 5,000 chemical compounds and the other with 10,000 compounds from the DTP dataset each.

From these results we can see that the iterative shape adjustment increases the total number of frequent subgraphs that are discovered by GFSG. For example, from the dataset with 5,000 compounds, 404 frequent subgraphs were found without any adjustment, while all three adjustment methods enabled GFSG to find more than 540 compounds—an increase of 34%. Likewise, from

Table 5

Runtime for the same datasets shown in Table 4, without scaling.

σ %	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	$\#f$
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

σ : the minimum support threshold [%].

t : Runtime in seconds.

l : The size of largest frequent subgraphs.

$\#f$: The total number of frequent subgraphs discovered.

the dataset with 10,000 compounds, there was at least a 25% increase in the number of frequent subgraphs for all the adjustment methods. Also, these results show that the maximum allowed number of adjustment iterations does not significantly increase either the overall execution time or the number of discovered frequent patterns. For example, for the dataset with 10,000 compounds and the supporting transaction monitoring scheme, the number of frequent patterns found with $N = 10$ (1,699) is only 5% greater than that found with $N = 2$ (1,637). Similar results hold for the other methods and dataset. Finally, the overall increase in runtime associated with performing these shape optimization iterations is quite modest. In all the examples of Table 6 the majority of the improvements can be obtained with only a 5%–25% increase in runtime.

7 Conclusion

In this paper we presented an algorithm, gFSG, for finding frequently occurring geometric subgraphs in large graph databases, which can be used to discover recurrent patterns in scientific, spatial, and relational datasets. These patterns can correspond to either exact occurrences or occurrences that are translation, rotation, and/or scaling invariant, and can accommodate a user-specified tolerance on how the coordinates of the various vertices are matched. In addition, we presented an iterative shape refinement framework that makes it possible to optimize the discovered patterns; thus, increasing their frequency and the number of patterns that get discovered. Our experimental evaluation showed that gFSG can scale reasonably well to very large graph databases provided that graphs contain sufficiently many different labels of edges and

Table 6
Effect of three shape adjustment methods.

Dataset	σ [%]	Adjustment	N	t [sec]	$\#f$		
DTP 5,000 compounds	2.0	None	—	124	404		
			SA	2	155	541	
				5	209	562	
		10		248	558		
		STM	2	138	543		
			5	187	542		
			10	233	536		
		DWC	2	138	543		
			5	187	542		
			10	248	557		
		DTP 10,000 compounds	1.0	None	—	1107	1295
					SA	2	1180
5	1509					1638	
10	1720			1678			
STM	2			1026	1618		
	5			1383	1666		
	10			1676	1699		
DWC	2			1123	1631		
	5			1461	1628		
	10			1737	1683		

σ : the minimum support threshold [%].

None: No adjustment.

Simple: Simple shape adjustment.

Support: Adjustment based on supporting transaction monitoring.

DWC: Adjustment based on the downward closure check.

N : The number of iterations for shape adjustment.

l : The size of largest frequent subgraphs.

t : Runtime in seconds.

$\#f$: The total number of frequent subgraphs discovered.

vertices.

Despite gFSG’s reasonable success in discovering frequently occurring geometric subgraphs, we believe that it represents a first attempt in developing computationally efficient algorithms for geometric graphs and can be improved along a number of different directions. For example, as discussed in Section 3, the notion of r -tolerant frequently occurring patterns dramatically changes the characteristics of the pattern discovery problem as it now becomes critical to properly identify the representation of the pattern. gFSG’s heuristic shape optimization approaches, though effective, are not guaranteed to identify all patterns that can have a sufficiently large number of embeddings. A precise solution to this problem may require the identification of the pattern shapes that maximize the number of discovered patterns, which in turn may require solving a certain instance of a maximum independent set problem. At the same, gFSG’s notion of r -tolerant patterns can be extended to allow additional types of tolerances by allowing certain degree of approximate- or near-matching (e.g., skipping certain vertices and/or edges). Finally, vertical or projection-based mining approaches can also be investigated. Recent research in the context of finding patterns in topological graph has shown that such approaches are in general faster than their horizontal counterparts [48,23,31].

References

- [1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized substructure discovery for semi-structured data. In *Proc. the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD-2002)*, LNAI 2431, pages 1–14. Springer-Verlag, August 2002.
- [2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.
- [3] 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 International Conference on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, September 1994.
- [4] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th International Conference on Data Engineering (ICDE)*, pages 3–14. IEEE Press, 1995.
- [5] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proc. of the 2nd SIAM International Conference on Data Mining (SDM'02)*, pages 158–174, 2002.
- [6] J. Bajorath. Integration of virtual and high throughput screening. *Nature Review Drug Discovery*, 2002.
- [7] H. Bohm and G. Schneider. *Virtual Screening for Bioactive Molecules*. Wiley-VCH, 2000.
- [8] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.
- [9] 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 International Conference on Computational Molecular Biology*, 1999.
- [10] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *Proc. of the 3rd IEEE International Conference on Data Mining (ICDM'03)*, pages 509–512, 2003.
- [11] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. of the 2nd SIAM International Conference on Data Mining (SDM-2002)*, 2002.
- [12] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press, 1998.
- [13] M. Deshpande, M. Kuramochi, and G. Karypis. Frequent sub-structure based approaches for classifying chemical compounds. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM)*, 2003.

- [14] DTP/2D and 3D structural information. ftp://dtpsearch.ncifcrf.gov/jan02_2d.bin.
- [15] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of the 15th IEEE International Conference on Data Engineering*, March 1999.
- [16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
- [17] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In *Proc. of the 5th International Conference on Discovery Science*, 2002.
- [18] B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, University of Limburg, Diepenbeek, Belgium, December 2002.
- [19] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.
- [20] J. S. Handen. The industrialization of drug discovery. *Drug Discovery Today*, 7(2):83–85, January 2002.
- [21] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining—a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.
- [22] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the SUBDUE system. In *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, pages 169–180, 1994.
- [23] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proc. of 2003 IEEE International Conference on Data Mining (ICDM'03)*, 2003.
- [24] J. Huan, W. Wang, A. Washington, J. Prins, R. Shah, and A. Tropsha. Accurate classification of protein structural families using coherent subgraph analysis. In *Proceedings of the 9th Pacific Symposium on Biocomputing (PSB)*, 2004.
- [25] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.
- [26] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of computational biology*, 3(2):289–306, 1996.
- [27] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in HIV data. In *Proc. of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-01)*, pages 136–143, 2001.
- [28] M. Kuramochi, M. Deshpande, and G. Karypis. *Data Mining: Next Generation Challenges and Future Directions*, chapter Mining Scientific Data Sets using Graphs. AAAI Press, 2004. in press.
- [29] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*. in press.

- [30] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, November 2001.
- [31] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *Proc. of the 2004 SIAM International Conference on Data Mining (SDM04)*, 2004.
- [32] E. M. Mitchell, P. J. Artymiuk, D. W. Rice, and P. Willett. Use of techniques derived from graph theory to compare secondary structure motifs in proteins. *Journal of Molecular Biology*, 212:151–166, 1989.
- [33] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, Edmonton, AB, Canada, July 2002.
- [34] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, pages 362–369, 2002.
- [35] J. W. Raymond. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *J. Chem. Inf. Comput. Sci.*, 42:305–316, 2002.
- [36] M. Seno and G. Karypis. LPMiner: An algorithm for finding frequent itemsets using length decreasing support constraint. In *Proc. of 2001 IEEE International Conference on Data Mining (ICDM)*, November 2001.
- [37] M. Seno and G. Karypis. SLPMiner: An algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.
- [38] 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 International Conference on Management of Data*, pages 22–33, May 2000.
- [39] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, pages 458–465, 2002.
- [40] J. T. L. Wang, Q. Ma, D. Shasha, and C. H. Wu. Application of neural networks to biological data mining: A case study in protein sequence classification. In *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2000)*, pages 305–309, Boston, MA, August 2000.
- [41] 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.
- [42] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12:353–371, 2000.
- [43] 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.
- [44] X. Wang, J. T. L. Wang, D. Shasha, B. Shapiro, S. Dikshitulu, I. Rigoutsos, and K. Zhang. Automated discovery of active motifs in three dimensional molecules.

- In *Proc. the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 89–95, Newport Beach, CA, August 1997.
- [45] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, I. Rigoutsos, and K. Zhang. Finding patterns in three dimensional graphs: Algorithms and applications to scientific data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):731–749, July/August 2002.
 - [46] H. Wolfson and I. Rigoutsos. Geometric hashing: An overview. *IEEE Computational Science and Engineering*, 4(4):10–21, 1997.
 - [47] Y. Xiao, J.-F. Yao, Z. Li, and M. H. Dunham. Efficient data mining for maximal frequent subtrees. In *Proc. of the 3rd IEEE International Conference on Data Mining (ICDM'03)*, pages 379–386, 2003.
 - [48] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. of 2002 IEEE International Conference on Data Mining (ICDM)*, 2002.
 - [49] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, 2003.
 - [50] K. Yoshida and H. Motoda. CLIP: Concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, 1995.
 - [51] M. J. Zaki. Fast mining of sequential patterns in very large databases. Technical Report 668, Department of Computer Science, Univeristy of Rochester, 1997.
 - [52] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):372–390, 2000.
 - [53] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, July 2002.
 - [54] M. J. Zaki and K. Gouda. Fast vertical mining using difsets. Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute, 2001.
 - [55] 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.