Finding Patterns in Three Dimensional Graphs: Algorithms and Applications to Scientific Data Mining*

Xiong Wang† Jason T. L. Wang‡ Dennis Shasha§
Bruce A. Shapiro¶ Isidore Rigoutsos‖ Kaizhong Zhang**

Abstract

This paper presents a method for finding patterns in three dimensional (3D) graphs. Each node in a graph is an undeformable or atomic unit and has a label. Edges are links between the atomic units. Patterns are rigid substructures that may occur in a graph after allowing for an arbitrary number of whole-structure rotations and translations as well as a small number (specified by the user) of edit operations in the patterns or in the graph. (When a pattern appears in a graph only after the graph has been modified, we call that appearance “approximate occurrence.”) The edit operations include relabeling a node, deleting a node and inserting a node. The proposed method is based on the geometric hashing technique, which hashes node triplets of the graphs into a 3D table and compresses the label triplets in the table. To demonstrate the utility of our algorithms, we discuss two applications of them in scientific data mining.

First, we apply the method to locating frequently occurring motifs in two families of proteins pertaining to RNA-directed DNA Polymerase and Thymidylate Synthase, and use the motifs to classify the proteins. Then, we apply the method to clustering chemical compounds pertaining to aromatic, bicyclickanes and photosynthesis. Experimental results indicate the good performance of our algorithms and high recall and precision rates for both classification and clustering.

Index Terms—KDD, classification and clustering, data mining, geometric hashing, structural pattern discovery, biochemistry, medicine.

---

*This work was supported in part by NSF grants IRI-9531548, IRI-9531554, IIS-9988345, IIS-9988636, and by the Natural Sciences and Engineering Research Council of Canada under Grant No. OGP0046373.
†Department of Computer Science, California State University, Fullerton, CA 92834, USA (wang@ecs.fullerton.edu). Part of the work of this author was done while he was with Department of Computer and Information Science, New Jersey Institute of Technology.
‡Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA (jason@cis.njit.edu). Part of the work of this author was done while visiting Courant Institute of Mathematical Sciences, New York University.
§Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA (shasha@cs.nyu.edu).
¶Laboratory of Experimental and Computational Biology, Division of Basic Sciences, National Cancer Institute, National Institutes of Health, Frederick, MD 21702, USA (bshapiro@ncifcrf.gov).
‖IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA (rigouts0@us.ibm.com).
**Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7 (kzhang@csd.uwo.ca).
1 Introduction

Structural pattern discovery finds many applications in natural sciences, computer-aided design and image processing [8, 33]. For instance, detecting repeatedly occurring structures in molecules can help biologists to understand functions of the molecules. In these domains, molecules are often represented by three-dimensional (3D) graphs. The tertiary structures of proteins, for example, are 3D graphs [5, 9, 18]. As another example, chemical compounds are also 3D graphs [21].

In this paper we study a pattern discovery problem for graph data. Specifically we propose a geometric hashing technique to find frequently occurring substructures in a set of 3D graphs. Our study is motivated by recent advances in the data mining field, where automated discovery of patterns, classification and clustering rules is one of the main tasks. We establish a framework for structural pattern discovery in the graphs, and apply our approach to classifying proteins and clustering compounds. While the domains chosen here focus on biochemistry, our approach can be generalized to other applications where graph data occur commonly.

1.1 3D Graphs

Each node of the graphs we are concerned with is an undecomposable or atomic unit and has a 3D coordinate. Each node has a label, which is not necessarily unique in a graph. Node labels are chosen from a domain-dependent alphabet \( \Sigma \). In chemical compounds, for example, the alphabet includes the names of all atoms. A node can be identified by a unique, user-assigned number in the graph. Edges in the graph are links between the atomic units. In the paper, we will consider 3D graphs that are connected. For disconnected graphs, we consider their connected components [16].

A graph can be divided into one or more rigid substructures. A rigid substructure is a subgraph in which the relative positions of the nodes in the substructure are fixed, under some set of conditions of interest. Note that the rigid substructure as a whole can be rotated (we refer to this as a “whole-structure” rotation or simply a rotation when the context is clear). Thus, the relative position of a node in the substructure and a node outside the substructure can be changed under the rotation. The precise definition of a “substructure” is application dependent. For example, in chemical compounds, a ring is often a rigid substructure.

Example 1. To illustrate rigid substructures in a graph, consider the graph \( G \) in Fig. 1. Each node is associated with a unique number, with its label being enclosed in parentheses. Table 1 shows the 3D coordinates of the nodes in the graph with respect to the Global Coordinate Frame. We divide the graph into two rigid substructures: \( \text{Str}_0 \) and \( \text{Str}_1 \). \( \text{Str}_0 \) consists of nodes numbered 0, 1, 2, 3, 4, 5 as well as edges connecting the nodes (Fig. 2(a)). \( \text{Str}_1 \) consists of nodes numbered 6, 7, 8, 9, 10 as well as edges connecting them (Fig. 2(b)). Edges in the rigid substructures are represented by boldface links. The edge \( \{5, 6\} \) connecting the two rigid substructures is represented by a lightface link, meaning that the two

\[ \text{More precisely, the 3D coordinate indicates the location of the center of the atomic unit.} \]
substructures are rotatable with respect to each other around the edge. Note that a rigid substructure is not necessarily complete. For example, in Fig. 2(a), there is no edge connecting the node numbered 1 and the node numbered 3.

![Graph G](image)

**Fig. 1. A graph G.**

<table>
<thead>
<tr>
<th>Node identifier</th>
<th>Node label</th>
<th>Node coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>(1.0178, 1.0048, 2.5101)</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>(1.2021, 2.0410, 2.0030)</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>(1.3960, 2.9864, 2.0006)</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>(0.7126, 2.0490, 3.1921)</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>(0.7610, 2.7125, 3.0124)</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>(1.0097, 3.6478, 2.2660)</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>(1.1329, 4.5002, 2.2024)</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>(1.5309, 3.2020, 1.7191)</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>(1.4529, 6.1015, 1.5712)</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>(1.0356, 6.0030, 2.2820)</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>(0.7359, 5.0571, 2.6857)</td>
</tr>
</tbody>
</table>

Table 1. Identifiers, labels and global coordinates of the nodes of the graph in Fig. 1.

We attach a local coordinate frame $SF_0$ ($SF_1$, respectively) to substructure $Str_0$ ($Str_1$, respectively). For instance, let us focus on the substructure $Str_0$ in Fig. 2(a). We attach a local coordinate frame to $Str_0$ whose origin is the node numbered 0. This local coordinate frame is represented by three basis points $P_{b_1}$, $P_{b_2}$ and $P_{b_3}$, with coordinates $P_{b_1}(x_0, y_0, z_0)$, $P_{b_2}(x_0 + 1, y_0, z_0)$ and $P_{b_3}(x_0, y_0 + 1, z_0)$, respectively. The origin is $P_{b_1}$ and the three basis vectors are $\vec{V}_{b_1,b_2}$, $\vec{V}_{b_2,b_3}$, and $\vec{V}_{b_1,b_3} \times \vec{V}_{b_1,b_2}$. Here, $\vec{V}_{b_1,b_2}$ represents the vector starting at point $P_{b_1}$ and ending at point $P_{b_2}$. $\vec{V}_{b_1,b_2} \times \vec{V}_{b_1,b_3}$ stands for the cross product of the two corresponding vectors. We refer to this coordinate frame as Substructure Frame 0, or $SF_0$. Note that the basis vectors of $SF_0$ are orthonormal. That is, the length of each vector is 1 and the angle between any two basis vectors has 90 degrees. Also note that, for any node numbered $i$ in the substructure $Str_0$ with

---

2Such an edge is referred to as a rotatable edge. If two rigid substructures cannot be rotated with respect to each other around the edge connecting them, that edge is a non-rotatable edge.
global coordinate $P_i(x_i, y_i, z_i)$, we can find a local coordinate of the node $i$ with respect to $SF_0$, denoted $P'_i$, where

$$P'_i = \vec{V}_{b, i} = (x_i - x_0, y_i - y_0, z_i - z_0)$$

(1)

1.2 Patterns in 3D Graphs

We consider a pattern to be a rigid substructure that may occur in a graph after allowing for an arbitrary number of rotations and translations as well as a small number (specified by the user) of edit operations in the pattern or in the graph. We allow three types of edit operations: relabeling a node, deleting a node and inserting a node. Relabeling a node $v$ means to change the label of $v$ to any valid label that differs from its original label. Deleting a node $v$ from a graph means to remove the corresponding atomic unit from the 3D Euclidean space and make the edges touching $v$ connect with one of its neighbors $v'$. Inserting a node $v$ into a graph means to add the corresponding atomic unit to the 3D Euclidean space and make a node $v'$ and a subset of its neighbors become the neighbors of $v$.

We say graph $G$ matches graph $G'$ with $n$ mutations if by applying an arbitrary number of rotations and translations as well as $n$ node insert, delete or relabeling operations, (i) $G$ and $G'$ have the same size, $^4$ (ii) the nodes in $G$ geometrically match those in $G'$, i.e., they have coinciding 3D coordinates, and (iii) for each pair of geometrically matching nodes, they have the same label. A substructure $P$ approximately occurs in a graph $G$ (or $G$ approximately contains $P$) within $n$ mutations if $P$ matches some subgraph of $G$ with $n$ mutations or fewer where $n$ is chosen by the user.

Example 2. To illustrate patterns in graphs, consider the set $S$ of three graphs in Fig. 3(a). Suppose

---

$^3$Here, exactly which subset of the neighbors is chosen is unimportant, as the proposed geometric hashing technique hashes nodes only, ignoring edges among the nodes. Notice that when a node $v$ is inserted or deleted, the nodes surrounding $v$ do not move, i.e., their coordinates remain the same. Note also that we do not allow multiple edit operations to be applied to the same node. Thus, for example, inserting a node with label $m$ followed by relabeling it to $n$ is considered as inserting a node with label $n$. The three edit operations are extensions of the edit operations on sequences; they arise naturally in graph editing [10] and molecule evolution [25]. As shown in Section 4, based on these edit operations, our algorithm finds useful patterns that can be used to classify and cluster 3D molecules effectively.

$^4$The size of a graph is defined to be the number of nodes in the graph, since our geometric hashing technique considers nodes only. Furthermore, in our target applications, e.g. chemistry, nodes are atomic units and determine the size of a compound. Edges are links between the nodes and have a different meaning from the atomic units; as a consequence, we exclude the edges from the size definition.
only exactly coinciding substructures (without mutations) occurring in at least two graphs and having size greater than 3 are considered as "patterns." Then $S$ contains one pattern shown in Fig. 3(b). If substructures having size greater than 4 and approximately occurring in all the three graphs within one mutation (i.e. one node delete, insert or relabeling is allowed in matching a substructure with a graph) are considered as "patterns," then $S$ contains one pattern shown in Fig. 3(c).

![Graphs and Patterns](image)

Fig. 3. (a) The set $S$ of three graphs; (b) the pattern exactly occurring in two graphs in $S$; (c) the pattern approximately occurring, within one mutation, in all the three graphs.

Our strategy to find the patterns in a set of 3D graphs is to decompose the graphs into rigid substructures and then use geometric hashing [14] to organize the substructures and then to find the frequently occurring ones. In [35], we applied the approach to the discovery of patterns in chemical compounds under a restricted set of edit operations including node insert and node delete, and tested the quality of the patterns by using them to classify the compounds. Here, we extend the work in [35] by (i) considering more general edit operations including node insert, delete and relabeling; (ii) presenting the theoretical foundation and evaluating the performance and efficiency of our pattern-finding algorithm; (iii) applying the discovered patterns to classifying 3D proteins, which are much larger and more complicated in topology than chemical compounds; and (iv) presenting a technique to cluster 3D graphs based on the patterns occurring in them.

Specifically, we conducted two experiments. In the first experiment, we applied the proposed method to
locating frequently occurring motifs (substructures) in two families of proteins pertaining to RNA-directed DNA Polymerase and Thymidylate Synthase, and used the motifs to classify the proteins. Experimental results showed that our method achieved a 96.4% precision rate. In the second experiment, we applied our pattern-finding algorithm to discovering frequently occurring patterns in chemical compounds chosen from the Merck Index pertaining to aromatic, bicyclic alkanes and photosynthesis. We then used the patterns to cluster the compounds. Experimental results showed that our method achieved 99% recall and precision rates.

The rest of the paper is organized as follows. Section 2 presents the theoretical framework of our approach and describes the pattern-finding algorithm in detail. Section 3 evaluates the performance and efficiency of the pattern-finding algorithm. Section 4 describes the applications of our approach to classifying proteins and clustering compounds. Section 5 discusses related work. Section 6 concludes the paper.

2 Pattern-Finding Algorithm

2.1 Terminology

Let $\mathcal{S}$ be a set of 3D graphs. The occurrence number of a pattern $P$ is the number of graphs in $\mathcal{S}$ that approximately contain $P$ within the allowed number of mutations. Formally, the occurrence number of a pattern $P$ with respect to mutation $d$ and set $\mathcal{S}$, denoted $\text{occur}_{\mathcal{S}}^d(P)$, is $k$ if there are $k$ graphs in $\mathcal{S}$ that contain $P$ within $d$ mutations. For example, consider Fig. 3 again. Let $\mathcal{S}$ contain the three graphs in Fig. 3(a). Then $\text{occur}_{\mathcal{S}}^1(P_1) = 2; \text{occur}_{\mathcal{S}}^1(P_2) = 3$.

Given a set $\mathcal{S}$ of 3D graphs, our algorithm finds all the patterns $P$ where $P$ approximately occurs in at least $\text{Occur}$ graphs in $\mathcal{S}$ within the allowed number of mutations $\text{Mut}$ and $|P| \geq \text{Size}$, where $|P|$ represents the size, i.e., the number of nodes, of the pattern $P$. ($\text{Mut}$, $\text{Occur}$ and $\text{Size}$ are user-specified parameters.) One can use the patterns in several ways. For example, biologists or chemists may evaluate whether the patterns are significant; computer scientists may use the patterns to classify or cluster molecules as demonstrated in Section 4.

Our algorithm proceeds in two phases to search for the patterns: (1) find candidate patterns from the graphs in $\mathcal{S}$; and (2) calculate the occurrence numbers of the candidate patterns to determine which of them satisfy the user-specified requirements. We describe each phase in turn below.

2.2 Phase (1) of the Algorithm

In phase (1) of the algorithm, we decompose the graphs into rigid substructures. Dividing a graph into substructures is necessary for two reasons. First, in dealing with some molecules such as chemical compounds in which there may exist two substructures that are rotatable with respect to each other, any graph containing the two substructures is not rigid. As a result, we decompose the graph into substructures having no rotatable components and consider the substructures separately. Second, our algorithm hashes node-triplets within rigid substructures into a 3D table. When a graph as a whole is too large, as
in the case of proteins, considering all combinations of three nodes in the graph may become prohibitive. Consequently, decomposing the graph into substructures and hashing node-triplets of the substructures can increase efficiency. For example, consider a graph of 20 nodes. There are \( \binom{20}{3} = 1140 \) node-triplets. On the other hand, if we decompose the graph into five substructures, each having four nodes, then there are only \( 5 \times \binom{4}{3} = 20 \) node-triplets.

There are several alternative ways to decompose 3D graphs into rigid substructures, depending on the application at hand and the nature of the graphs. For the purposes of exposition, we describe our pattern-finding algorithm based on a partitioning strategy. Our approach assumes a notion of atomic unit which is the lowest level of description in the case of interest. Intuitively, atomic units are fundamental building elements, e.g. atoms in a molecule. Edges arise as bonds between atomic units. We break a graph into maximal size rigid substructures (recall that a rigid substructure is a subgraph in which the relative positions of nodes in the substructure are fixed). To accomplish this, we use an approach similar to [16] that employs a depth-first search algorithm, referred to as DFB, to find blocks in a graph. The DFB works by traversing the graph in a depth-first order and collecting nodes belonging to a block during the traversal, as illustrated in the example below. Each block is a rigid substructure. We merge two rigid substructures \( B_1 \) and \( B_2 \) if they are not rotatable with respect to each other; that is, the relative position of a node \( n_1 \in B_1 \) and a node \( n_2 \in B_2 \) is fixed. The algorithm maintains a stack, denoted \( STK \), which keeps the rigid substructures being merged. Fig. 4 shows the algorithm, which outputs a set of rigid substructures of a graph \( G \). We then throw away the substructures \( P \) where \( |P| < Size \). The remaining substructures constitute the candidate patterns generated from \( G \). This pattern-generation algorithm runs in time linearly proportional to the number of edges in \( G \).

\begin{verbatim}
Procedure FindRigidSubstructures
Input: Graph G.
Output: A set of maximal size rigid substructures generated from G.
1. STK := \emptyset;
2. while G is not empty do
3. begin
4. locate the next block \( B_1 \) in G using the DFB algorithm;
5. delete \( B_1 \) from G;
6. if (STK is empty) or (\( B_1 \) and \( B_2 \) are not rotatable with respect to each other) then
7. push the nodes of \( B_1 \) into STK;
8. else begin
9. pop out all nodes in STK, merge them and output the resulting substructure;
10. push the nodes of \( B_1 \) into STK;
11. end;
12. end;
13. pop out all nodes in STK, merge them and output the resulting substructure;
\end{verbatim}

Fig. 4. Algorithm for finding rigid substructures in a graph.

\footnote{A block is a maximal subgraph that has no cut-vertices. A cut-vertex of a graph is one whose removal results in dividing the graph into multiple, disjointed subgraphs [16].}
**Example 3.** We use the graph in Fig. 5 to illustrate how the Find\_Rigid\_Substructures algorithm in Fig. 4 works. Rotatable edges in the graph are represented by lightface links; non-rotatable edges are represented by boldface links. Initially, the stack $STK$ is empty. We invoke DFB to locate the first block (Step 4). DFB begins by visiting the node numbered 0. Following the depth-first search, DFB then visits the nodes numbered 1, 2, and 5. Next, DFB may visit the node numbered 6 or 4. Without loss of generality, assume DFB visits the node numbered 6, and then the nodes numbered 10, 11, 13, 14, 15, 16, 18, and 17, in that order. Then, DFB visits the node numbered 14, and realizes that this node has been visited before. Thus, DFB goes back to the node numbered 17, 18, etc. until it returns to the node numbered 14. At this point, DFB identifies the first block, $B_1$, which includes nodes numbered 14, 15, 16, 17, and 18. Since the stack $STK$ is empty now, we push $B_1$ into $STK$ (Step 7).

![Graph](image)

**Fig. 5.** The graph used for illustrating how the Find\_Rigid\_Substructures algorithm works.

In iteration 2, we call DFB again to find the next block (Step 4). DFB returns the non-rotatable edge \{13, 14\} as a block, denoted $B_2$. The block is pushed into $STK$ (Step 7). In iteration 3, DFB locates the block $B_3$, which includes nodes numbered 10, 11, 12, and 13 (Step 4). Since $B_3$ and the top entry of the stack, $B_2$, are not rotatable with respect to each other, we push $B_3$ into $STK$ (Step 7). In iteration 4, we continue to call DFB and get the single edge \{6, 10\} as the next block, $B_4$ (Step 4). Since \{6, 10\} is rotatable and the stack $STK$ is nonempty, we pop out all nodes in $STK$, merge them and output the rigid substructure containing nodes numbered 10, 11, 12, 13, 14, 15, 16, 17, 18 (Step 9). We then push $B_4$ into $STK$ (Step 10).

In iteration 5, DFB visits the node numbered 7 and then 8, from which DFB goes back to the node numbered 7. It returns the single edge \{7, 8\} as the next block, $B_5$ (Step 4). Since \{7, 8\} is connected to the current top entry of $STK$, \{6, 10\}, via a rotatable edge \{6, 7\}, we pop out \{6, 10\}, which itself becomes a rigid substructure (Step 9). We then push \{7, 8\} into $STK$ (Step 10). In iteration 6, DFB returns the single edge \{7, 9\} as the next block, $B_6$ (Step 4). Since $B_6$ and the current top entry of $STK$, $B_5 = \{7, 8\}$, are not rotatable with respect to each other, we push $B_6$ into $STK$ (Step 7). In iteration 7, DFB goes back from the node numbered 7 to the node numbered 6 and returns the single edge \{6, 7\} as the next block, $B_7$ (Step 4). Since \{6, 7\} is rotatable, we pop out all nodes in $STK$, merge them and output the resulting rigid substructure containing nodes numbered 7, 8, and 9 (Step 9). We then push $B_7 = \{6, 7\}$ into $STK$ (Step 10).

---

6In practice, in graph representations for molecules, one uses different notation to distinguish non-rotatable edges from rotatable edges. For example, in chemical compounds, a double bond is non-rotatable. The different notation helps the algorithm to determine the types of edges.
In iteration 8, DFB returns the block \( B_8 = \{5, 6\} \) (Step 4). Since \( \{5, 6\} \) and \( \{6, 7\} \) are both rotatable, we pop out \( \{6, 7\} \) to form a rigid substructure (Step 9). We then push \( B_8 \) into \( STK \) (Step 10). In iteration 9, DFB returns the block \( B_9 \) containing nodes numbered 0, 1, 2, 3, 4, and 5 (Step 4). Since the current top entry of \( STK, B_9 = \{5, 6\} \), is rotatable with respect to \( B_9 \), we pop out \( \{5, 6\} \) to form a rigid substructure (Step 9) and push \( B_9 \) into \( STK \) (Step 10). Finally, since there is no block left in the graph, we pop out all nodes in \( B_9 \) to form a rigid substructure and terminate (Step 13).

### 2.3 Phase (2) of the Algorithm

Phase (2) of our pattern-finding algorithm consists of two subphases. In subphase A of phase (2), we hash and store the candidate patterns generated from the graphs in phase (1) in a 3D table \( \mathcal{H} \). In subphase B, we rehash each candidate pattern into \( \mathcal{H} \) and calculate its occurrence number. Notice that in the subphase B, one does not need to store the candidate patterns in \( \mathcal{H} \) again.

In processing a rigid substructure (pattern) of a 3D graph, we choose all three-node combinations, referred to as node-triplets, in the substructure and hash the node-triplets. We hash three-node combinations, because to fix a rigid substructure in the 3D Euclidean space one needs at least three nodes from the substructure and three nodes are sufficient provided they are not collinear. Notice that the proper order of choosing the nodes \( i, j, k \) in a triplet is significant and has an impact on the accuracy of our approach, as we will show later in the paper. We determine the order of the three nodes by considering the triangle formed by them. The first node chosen always opposes the longest edge of the triangle and the third node chosen opposes the shortest edge. For example, in the triangle in Fig. 6, we choose \( i, j, k \), in that order. Thus, the order is unique if the triangle is not isosceles or equilateral, which usually holds when the coordinates are floating point numbers. On the other hand, when the triangle is isosceles, we hash and store the two node-triplets \( [i, j, k] \) and \( [i, k, j] \), assuming that node \( i \) opposes the longest edge and edges \( \{i, j\}, \{i, k\} \) have the same length. When the triangle is equilateral, we hash and store the six node-triplets \( [i, j, k], [i, k, j], [j, i, k], [j, k, i], [k, i, j], \) and \( [k, j, i] \).

![Fig. 6. The triangle formed by the three nodes \( i, j, k \).](image)

The labels of the nodes in a triplet form a label-triplet, which is encoded as follows. Suppose the three nodes chosen are \( v_1, v_2, v_3 \), in that order. We maintain all node labels in the alphabet \( \Sigma \) in an array \( \mathcal{A} \). The code for the labels is an unsigned long integer, defined as \( ((L_1 \times \text{Prime} + L_2) \times \text{Prime}) + L_3 \), where \( \text{Prime} > |\Sigma| \) is a prime number, \( L_1 \), \( L_2 \) and \( L_3 \) are the indices for the node labels of \( v_1 \), \( v_2 \) and \( v_3 \), respectively, in the array \( \mathcal{A} \). Thus the code of a label-triplet is unique. This simple encoding scheme
reduces three label comparisons into one integer comparison.

**Example 4.** To illustrate how we encode label-triplets, consider again the graph $G$ in Fig. 1. Suppose the node labels are stored in the array $\mathcal{A}$ as shown in Table 2. Suppose $Prime$ is 1,009. Then, for example, for the three nodes numbered 2, 0 and 1 in Fig. 1, the code for the corresponding label-triplet is $((2 \times 1,009 + 0) \times 1,009) + 1 = 2,036,163$. 

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

Table 2. The node labels of the graph in Fig. 1 and their indices in the array $\mathcal{A}$.

### 2.3.1 Subphase A of Phase (2)

In this subphase, we hash the candidate patterns generated in phase (1) of the pattern-finding algorithm into a 3D table. For the purposes of exposition, consider the example substructure $Str_0$ in Fig. 2(a), which is assumed to be a candidate pattern. We choose any three nodes in $Str_0$ and calculate their 3D hash function values as follows. Suppose the chosen nodes are numbered $i, j, k$ and have global coordinates $P_i(x_i, y_i, z_i), P_j(x_j, y_j, z_j)$ and $P_k(x_k, y_k, z_k)$, respectively. Let $l_1, l_2, l_3$ be three integers where

\[
\begin{align*}
  l_1 &= ((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times Scale \\
  l_2 &= ((x_i - x_k)^2 + (y_i - y_k)^2 + (z_i - z_k)^2) \times Scale \\
  l_3 &= ((x_k - x_j)^2 + (y_k - y_j)^2 + (z_k - z_j)^2) \times Scale
\end{align*}
\]

Here $Scale = 10^p$ is a multiplier. Intuitively we round to the nearest $p$th position following the decimal point (here $p$ is the last accurate position) and then multiply the numbers by $10^p$. The reason for using the multiplier is that we want some digits following the decimal point to contribute to the distribution of the hash function values. We ignore the digits after the position $p$ because they are inaccurate. (The appropriate value for the multiplier can be calculated once data are given, as we will show in Section 3.)

Let

\[
\begin{align*}
  d_1 &= (l_1 + l_2) \mod Prime_1 \mod Nrow \\
  d_2 &= (l_2 + l_3) \mod Prime_2 \mod Nrow \\
  d_3 &= (l_3 + l_1) \mod Prime_3 \mod Nrow
\end{align*}
\]

$Prime_1, Prime_2$ and $Prime_3$ are three prime numbers and $Nrow$ is the cardinality of the hash table in each dimension. We use three different prime numbers in the hope that the distribution of the hash function values is not skewed even if pairs of $l_1, l_2, l_3$ are correlated. The node-triplet $[i, j, k]$ is hashed to the 3D bin with the address $h[d_1][d_2][d_3]$. Intuitively we use the squares of the lengths of the three edges connecting the three chosen nodes to determine the hash bin address. Stored in that bin are the graph
identification number, the substructure identification number, and the label-triplet code. In addition, we store the coordinates of the basis points \( P_{b_1}, P_{b_2}, P_{b_3} \) of Substructure Frame 0 (\( SF_0 \)) with respect to the three chosen nodes.

Specifically, suppose the chosen nodes \( i, j, k \) are not collinear. We can construct another local coordinate frame, denoted \( LF[i, j, k] \), using \( \vec{V}_{i, j}, \vec{V}_{i, k} \) and \( \vec{V}_{i, j} \times \vec{V}_{i, k} \) as basis vectors. The coordinates of \( P_{b_1}, P_{b_2}, P_{b_3} \) with respect to the local coordinate frame \( LF[i, j, k] \), denoted \( SF_0[i, j, k] \), form a 3 \( \times \) 3 matrix, which is calculated as follows (see Fig. 7):

\[
SF_0[i, j, k] = \begin{pmatrix}
\vec{V}_{i, b_1} \\
\vec{V}_{i, b_2} \\
\vec{V}_{i, b_3}
\end{pmatrix} \times A^{-1}
\]

where

\[
A = \begin{pmatrix}
\vec{V}_{i, j} \\
\vec{V}_{i, k} \\
\vec{V}_{i, j} \times \vec{V}_{i, k}
\end{pmatrix}
\]

Thus, suppose the graph in Fig. 1 has identification number 12. The hash bin entry for the three chosen nodes \( i, j, k \) is \((12, 0, Lcode, SF_0[i, j, k])\), where \( Lcode \) is the label-triplet code. Since there are 6 nodes in the substructure \( Str_0 \), we have \( \binom{6}{3} = 20 \) node-triplets generated from the substructure and therefore 20 entries in the hash table for the substructure.\(^7\)

\[\text{Fig. 7. Calculation of the coordinates of the basis points } P_{b_1}, P_{b_2}, P_{b_3} \text{ of Substructure Frame } 0 (SF_0) \text{ with respect to the local coordinate frame } LF[i, j, k].\]

\(^7\)One interesting question here is regarding the size limitation of patterns (rigid substructures) found by our algorithm. Suppose the graph identification number and the substructure identification number are both short integers of 2 bytes. \( Lcode \) is a long integer of 4 bytes. \( SF_0[i, j, k] \) is a 3 \( \times \) 3 matrix of floating point numbers, each requiring 4 bytes. Thus, each node-triplet requires 44 bytes. For a set of \( M \) patterns (substructures), each having \( T \) nodes on average, the hash table requires \( M \times \left( \frac{T}{3} \right) \times 44 \) bytes of disk space. Suppose there are 10,000 patterns, i.e. \( M = 10,000 \). If \( T = 20 \), the hash table requires about 502 Megabytes. If \( T = 100 \), the hash table requires over 71 Gigabytes. This gives an estimate of how large patterns can be in practice. In our case, the pattern sizes are between 5 and 13, so the size demands are modest. Note that the time and memory space needed also increase dramatically as patterns become large.
Example 5. To illustrate how we hash and store patterns in the hash table, let us consider Table 1 again. The basis points of $SF_0$ of Fig. 2(a) have the following global coordinates:

\[ P_{b1}(1.0178, 1.0048, 2.5101), \]
\[ P_{b2}(2.0178, 1.0048, 2.5101), \]
\[ P_{b3}(1.0178, 2.0048, 2.5101). \]

The local coordinates, with respect to $SF_0$, of the nodes numbered 0, 1, 2, 3 and 4 in substructure $Str_0$ of Fig. 2(a) are as follows:

\[ P_0'(0.0000, 0.0000, 0.0000), \]
\[ P_1'(0.1843, 1.0362, -0.5081), \]
\[ P_2'(0.3782, 1.9816, -0.5095), \]
\[ P_3'(0.0352, 1.0442, 0.6820), \]
\[ P_4'(0.2568, 1.7077, 0.5023). \]

Now, suppose $Scale$, $Prime_1$, $Prime_2$, $Prime_3$ are 10, 1, 0.09, 1, 033 and 1,051 respectively, and $N_row$ is 31. Thus, for example, for the nodes numbered 1, 2 and 3, the hash bin address is $h[25][12][21]$ and

\[ SF_0[1, 2, 3] = \begin{pmatrix}
-1.056731 & 0.357816 & 0.173981 \\
-0.875868 & 0.071949 & 0.907227 \\
-0.035973 & 0.417517 & 0.024041
\end{pmatrix} \quad (5) \]

As another example, for the nodes numbered 1, 4 and 2, the hash bin address is $h[24][6][9]$ and

\[ SF_0[1, 4, 2] = \begin{pmatrix}
0.369457 & -1.308266 & -0.242990 \\
-0.043584 & 0.857105 & -1.006793 \\
0.453285 & -0.343703 & -0.086982
\end{pmatrix} \quad (6) \]

Similarly, for the substructure $Str_1$, we attach a local coordinate frame $SF_1$ to the node numbered 6 as shown in Fig. 2(b). There are 10 hash table entries for $Str_1$, each having the form \( \{l, m, n\} \) where \( l, m, n \) are any three nodes in $Str_1$. □

Recall that we choose the three nodes $i$, $j$, $k$ based on the triangle formed by them—the first node chosen always opposes the longest edge of the triangle and the third node chosen opposes the shortest edge. Without loss of generality, let us assume that the nodes $i, j, k$ are chosen in that order. Thus, $\vec{V}_{i,j}$ has the shortest length, $\vec{V}_{i,k}$ is the second shortest and $\vec{V}_{j,k}$ is the longest. We use node $i$ as the origin, $\vec{V}_{i,j}$ as the X-axis and $\vec{V}_{i,k}$ as the Y-axis. Then construct the local coordinate frame $LF[i, j, k]$ using $\vec{V}_{i,j}$, $\vec{V}_{i,k}$ and $\vec{V}_{i,j} \times \vec{V}_{i,k}$ as basis vectors. Thus, we exclude the longest vector $\vec{V}_{j,k}$ when constructing $LF[i, j, k]$. Here is why.

The coordinates $(x, y, z)$ of each node in a 3D graph have an error due to rounding. Thus, the real coordinates for the node should be $(\vec{x}, \vec{y}, \vec{z})$, where $\vec{x} = x + \epsilon_1, \vec{y} = y + \epsilon_2, \vec{z} = z + \epsilon_3$ for three small decimal fractions $\epsilon_1, \epsilon_2, \epsilon_3$. After constructing $LF[i, j, k]$ and when calculating $SF_0[i, j, k]$, one may add or
multiply the coordinates of the 3D vectors. We define the **accumulative error** induced by a calculation \( C \), denoted \( \Delta(C) \), as

\[
\Delta(C) = |\mathbf{f} - \bar{f}|
\]

where \( \bar{f} \) is the result obtained from \( C \) with the real coordinates and \( f \) is the result obtained from \( C \) with rounding errors.

Recall that in calculating \( SF_0[i,j,k] \), the three basis vectors of \( LF[i,j,k] \) all appear in the matrix \( A \) defined in Equation (4). Let

\[
|\vec{V}_{i,j}| = |\vec{V}_{i,j}| + \delta_1, \quad |\vec{V}_{i,k}| = |\vec{V}_{i,k}| + \delta_2, \quad \text{and} \quad |\vec{V}_{j,k}| = |\vec{V}_{j,k}| + \delta_3.
\]

Let

\[
\delta = \max\{|\delta_1|, |\delta_2|, |\delta_3|\}.
\]

Notice

\[
|\vec{V}_{i,j} \times \vec{V}_{i,k}| = |\vec{V}_{i,j}| |\vec{V}_{i,k}| \sin \theta
\]

where \( |\vec{V}_{i,j}| \) is the length of \( \vec{V}_{i,j} \), and \( \theta \) is the angle between \( \vec{V}_{i,j} \) and \( \vec{V}_{i,k} \). Thus,

\[
\Delta(|\vec{V}_{i,j} \times \vec{V}_{i,k}|) = \frac{|\vec{V}_{i,j}^2||\vec{V}_{i,k}^2| \sin \theta - |\vec{V}_{i,j}| |\vec{V}_{i,k}| \sin \theta|}{|\vec{V}_{i,j}^2||\vec{V}_{i,k}^2| \sin \theta - |\vec{V}_{i,j}| |\vec{V}_{i,k}| \sin \theta}
\]

\[
= \frac{|(|\vec{V}_{i,j}| + \delta_1)(|\vec{V}_{i,k}| + \delta_2) \sin \theta - |\vec{V}_{i,j}||\vec{V}_{i,k}| \sin \theta|}{|(|\vec{V}_{i,j}| + \delta_1)(|\vec{V}_{i,k}| + \delta_2) - |\vec{V}_{i,j}||\vec{V}_{i,k}|| \sin \theta}
\]

\[
= \frac{|(|\vec{V}_{i,j}||\vec{V}_{i,k}| + |\vec{V}_{i,j}||\delta_2| + |\vec{V}_{i,k}||\delta_1| + |\delta_1||\delta_2|)| \sin \theta|}{|(|\vec{V}_{i,j}||\vec{V}_{i,k}| + |\vec{V}_{i,j}||\delta_2| + |\vec{V}_{i,k}||\delta_1| + |\delta_1||\delta_2|)| \sin \theta}
\]

\[
\leq (|\vec{V}_{i,j}| + |\vec{V}_{i,k}| + \delta) \delta
\]

\[
= U_1
\]

Likewise,

\[
\Delta(|\vec{V}_{i,j} \times \vec{V}_{j,k}|) \leq (|\vec{V}_{i,j}| + |\vec{V}_{j,k}| + \delta) \delta = U_2
\]

and

\[
\Delta(|\vec{V}_{i,k} \times \vec{V}_{j,k}|) \leq (|\vec{V}_{i,k}| + |\vec{V}_{j,k}| + \delta) \delta = U_3
\]

Among the three upperbounds \( U_1, U_2, U_3, U_1 \) is the smallest. It’s likely that the accumulative error induced by calculating the length of the cross product of the two corresponding vectors is also the smallest. Therefore we choose \( \vec{V}_{i,j}, \vec{V}_{i,k} \) and exclude the longest vector \( \vec{V}_{j,k} \) in constructing the local coordinate frame \( LF[i,j,k] \), so as to minimize the accumulative error induced by calculating \( SF_0[i,j,k] \).

### 2.3.2 Subphase B of Phase (2)

Let \( \mathcal{H} \) be the resulting hash table obtained in subphase A of phase (2) of the pattern-finding algorithm. In subphase B, we calculate the occurrence number of each candidate pattern \( P \) by rehashing the node-triplets of \( P \) into \( \mathcal{H} \). This way, we are able to match a node-triplet \( tri \) of \( P \) with a node-triplet \( tri' \) of another substructure (candidate pattern) \( P' \) stored in subphase A where \( tri \) and \( tri' \) have the same hash bin address. By counting the node-triplet matches, one can infer whether \( P \) matches \( P' \) and therefore whether \( P \) occurs in the graph from which \( P' \) is generated.

We associate each substructure with several counters, which are created and updated as illustrated by the following example. Suppose the two substructures (patterns) of graph \( G \) with identification number 12 in Fig. 1 have already been stored in the hash table \( \mathcal{H} \) in subphase A. Suppose \( i,j,k \) are three nodes in the
substructure $Str_0$ of $G$. Thus, for this node-triplet, its entry in the hash table is $(12, 0, Lcode, SF_0[i, j, k])$. Now, in subphase B, consider another pattern $P$; we hash the node-triplets of $P$ using the same hash function. Let $u, v, w$ be three nodes in $P$ that have the same hash bin address as $i, j, k$; that is, the node-triplet $[u, v, w]$ "matches" the node-triplet $[i, j, k]$. If the nodes $u, v, w$ geometrically match the nodes $i, j, k$ respectively, i.e., they have coinciding 3D coordinates after rotations and translations, we call the node-triplet match a true match; otherwise it is a false match. For a true match, let

$$SF_P = SF_0[i, j, k] \times \left( \begin{array}{c} \vec{V}_{u,v} \\ \vec{V}_{u,w} \\ \vec{V}_{u,v} \times \vec{V}_{u,w} \end{array} \right) + \left( \begin{array}{c} P_u \\ P_u \\ P_u \end{array} \right)$$ (7)

This $SF_P$ contains the coordinates of the three basis points of the Substructure Frame 0 ($SF_0$) with respect to the global coordinate frame in which the pattern $P$ is given. We compare the $SF_P$ with those already associated with the substructure $Str_0$ (initially none is associated with $Str_0$). If the $SF_P$ differs from the existing ones, a new counter is created, whose value is initialized to 1, and the new counter is assigned to the $SF_P$. If the $SF_P$ is the "same" as an existing one with counter value $Cnt$, then $Cnt$ is incremented by one. In general, a substructure may be associated with several different $SF_P$'s, each having a counter.

We now present the theory supporting this algorithm. Theorem 1 below establishes a criterion based on which one can detect and eliminate a false match. Theorem 2 below justifies the procedure of incrementing the counter values.

**Theorem 1.** Let $P_{c_1}$, $P_{c_2}$ and $P_{c_3}$ be the three basis points forming the $SF_P$ defined in Equation (7), where $P_{c_1}$ is the origin. $\vec{V}_{c_1,c_2}$, $\vec{V}_{c_1,c_3}$ and $\vec{V}_{c_1,c_2} \times \vec{V}_{c_1,c_3}$ are orthonormal vectors if and only if the nodes $u, v$ and $w$ geometrically match the nodes $i, j$ and $k$, respectively.

**Proof.** (If) Let $A$ be as defined in Equation (4) and let

$$B = \begin{pmatrix} \vec{V}_{u,v} \\ \vec{V}_{u,w} \\ \vec{V}_{u,v} \times \vec{V}_{u,w} \end{pmatrix}$$ (8)

Note that, if $u, v$ and $w$ geometrically match $i, j$ and $k$, respectively, then $|B| = |A|$, where $|B|$ ($|A|$ respectively) is the determinant of the matrix $B$ (matrix $A$, respectively). That is to say, $|A^{-1}|B| = 1$.

From Equation (3) and by the definition of the $SF_P$ in Equation (7), we have

$$SF_P = \begin{pmatrix} \vec{V}_{i,b_1} \\ \vec{V}_{i,b_2} \\ \vec{V}_{i,b_3} \end{pmatrix} \times A^{-1} \times B + \begin{pmatrix} P_u \\ P_u \\ P_u \end{pmatrix}$$ (9)

Thus, the $SF_P$ basically transforms $P_{b_1}, P_{b_2}$ and $P_{b_3}$ via two translations and one rotation, where $P_{b_1}, P_{b_2}$ and $P_{b_3}$ are the basis points of the Substructure Frame 0 ($SF_0$). Since $\vec{V}_{b_1,b_2}$, $\vec{V}_{b_1,b_3}$ and $\vec{V}_{b_2,b_3} \times \vec{V}_{b_2,b_3}$ are...
orthonormal vectors, and translations and rotations do not change this property [27], we know that \( \vec{V}_{c1,c2} \), \( \vec{V}_{c1,c3} \) and \( \vec{V}_{c1,c2} \times \vec{V}_{c1,c3} \) are orthonormal vectors. 

(Only if) If \( u, v \) and \( w \) do not match \( i, j \) and \( k \) geometrically while having the same hash bin address, then there would be distortion in the aforementioned transformation. Consequently, \( \vec{V}_{c1,c2}, \vec{V}_{c1,c3} \) and \( \vec{V}_{c1,c2} \times \vec{V}_{c1,c3} \) would no longer be orthonormal vectors. \( \square \)

**Theorem 2.** If two true node-triplet matches yield the same \( SF_P \) and the codes of the corresponding label-triplets are the same, then the two node-triplet matches are augmentable, i.e., they can be combined to form a larger substructure match between \( P \) and \( Str_0 \).

**Proof.** Since three nodes are enough to set the \( SF_P \) at a fixed position and direction, all the other nodes in \( P \) will have definite coordinates under this \( SF_P \). When another node-triplet match yielding the same \( SF_P \) occurs, it means that geometrically there is at least one more node match between \( Str_0 \) and \( P \). If the codes of the corresponding label-triplets are the same, it means that the labels of the corresponding nodes are the same. Therefore the two node-triplet matches are augmentable. \( \square \)

Fig. 8 illustrates how two node-triplet matches are augmented. Suppose the node-triplet \([3, 4, 2]\) yields the \( SF_P \) shown in the figure. Further suppose that the node-triplet \([1, 2, 3]\) yields the same \( SF_P \) as shown in Fig. 8. Since the labels of the corresponding nodes numbered 3 and 2 are the same, we can augment the two node-triplets to form a larger match containing nodes numbered 1, 2, 3, 4.

Thus, by incrementing the counter associated with the \( SF_P \), we record how many true node-triplet matches are augmentable under this \( SF_P \). Notice that in cases where two node-triplet matches occur due to reflections, the directions of the corresponding local coordinate systems are different, so are the \( SF_P \)’s. As a result, these node-triplet matches are not augmentable.

**Example 6.** To illustrate how we update counter values for augmentable node-triplet matches, let us consider the pattern \( P \) in Fig. 9. In \( P \), the nodes numbered 0, 1, 2, 3, 4 match, after rotation, the nodes numbered 5, 4, 3, 1, 2 in the substructure \( Str_0 \) in Fig. 2(a). The node numbered 0 in \( Str_0 \) does not appear in \( P \) (i.e. it is to be deleted). The labels of the corresponding nodes are identical. Thus, \( P \) matches \( Str_0 \).
with 1 mutation, i.e., one node is deleted.

Now, suppose in \( P \), the global coordinates of the nodes numbered 1, 2, 3 and 4 are
\[
\begin{align*}
P_1 & = (-0.269000, 4.153153, 2.911494), \\
P_2 & = (-0.317400, 4.749386, 3.253592), \\
P_3 & = (0.172100, 3.913515, 4.100777), \\
P_4 & = (0.306000, 3.244026, 3.433268).
\end{align*}
\]

Refer to Example 5. For the nodes numbered 3, 4 and 2 of \( P \), the hash bin address is \( h[25][12][21] \), which is the same as that of nodes numbered 1, 2, 3 of \( Str_0 \), and
\[
SF_P = \begin{pmatrix}
-0.012200 & 5.005500 & 4.474200 \\
0.987800 & 5.005500 & 4.474200 \\
-0.012200 & 4.298393 & 3.767093
\end{pmatrix}
\]
(10)

The three basis vectors forming this \( SF_P \) are
\[
\begin{align*}
\vec{V}_{c_1,c_2} & = (1.000000, 0.000000, 0.000000), \\
\vec{V}_{c_1,c_3} & = (0.000000, -0.707107, -0.707107), \\
\vec{V}_{c_2,c_3} & = (0.000000, 0.707107, -0.707107),
\end{align*}
\]
which are orthonormal.

For the nodes numbered 3, 1 and 4 of \( P \), the hash bin address is \( h[24][0][9] \), which is the same as that of nodes numbered 1, 4, 2 of \( Str_0 \), and
\[
SF_P = \begin{pmatrix}
-0.012200 & 5.005500 & 4.474200 \\
0.987800 & 5.005500 & 4.474200 \\
-0.012200 & 4.298393 & 3.767093
\end{pmatrix}
\]
(11)

These two true node-triplet matches have the same \( SF_P \), and therefore the corresponding counter associated with the substructure \( Str_0 \) of the graph 12 in Fig. 1 is updated to 2. After hashing all node-triplets
of \( P \), the counter value will be \( \left( \frac{5}{3} \right) = 10 \), since all matching node-triplets have the same \( SF_P \) as in Equation (10) and the labels of the corresponding nodes are the same. \( \square \)

Now, consider again the \( SF_P \) defined in Equation (7) and the three basis points \( P_{c_1}, P_{c_2}, P_{c_3} \) forming the \( SF_P \), where \( P_{c_1} \) is the origin. We note that for any node \( i \) in the pattern \( P \) with global coordinate \( P_i(x_i, y_i, z_i) \), it has a local coordinate with respect to the \( SF_P \), denoted \( P_i' \), where

\[
P_i' = \vec{V}_{c_1,i} \times E^{-1}
\]

Here, \( E \) is the base matrix for the \( SF_P \), defined as

\[
E = \begin{pmatrix}
\vec{V}_{c_1,c_2} \\
\vec{V}_{c_1,c_2} \\
\vec{V}_{c_1,c_2} \times \vec{V}_{c_1,c_3}
\end{pmatrix}
\]
and \( \vec{V}_{c_1,i} \) is the vector starting at \( P_{c_1} \) and ending at \( P_i \).

**Remark.** If \( \vec{V}_{c_1,c_2} \), \( \vec{V}_{c_1,c_3} \) and \( \vec{V}_{c_1,c_2} \times \vec{V}_{c_1,c_3} \) are orthonormal vectors, then \( |E| = 1 \). Thus, a practically useful criterion for detecting false matches is to check whether or not \( |E| = 1 \). If \( |E| \neq 1 \), then \( \vec{V}_{c_1,c_2} \), \( \vec{V}_{c_1,c_2} \times \vec{V}_{c_1,c_3} \) are not orthonormal vectors, and therefore the nodes \( u, v \) and \( w \) do not match the nodes \( i, j \) and \( k \) geometrically (cf. Theorem 1).

Intuitively, our scheme is to hash node-triplets and match the triplets. Only if one triplet \( tri \) matches another \( tri' \) do we see how the substructure containing \( tri \) matches the pattern containing \( tri' \). Using Theorem 1, we detect and eliminate false node-triplet matches. Using Theorem 2, we record in a counter the number of augmentable true node-triplet matches. The following theorem says that the counter value needs to be large (i.e., there are a sufficient number of augmentable true node-triplet matches) in order to infer that there is a match between the corresponding pattern and substructure. The larger the \( Mut \), the fewer node-triplet matches are needed.

**Theorem 3.** Let \( Str \) be a substructure in the hash table \( H \) and let \( G \) be the graph from which \( Str \) is generated. Let \( P \) be a pattern where \( |P| \geq Mut + 3 \). After rehashing the node-triplets of \( P \), suppose there is an \( SF_P \) associated with \( Str \) whose counter value \( Cnt > \Theta_P \) where

\[
\Theta_P = \left( \frac{N-1}{3} \right) = \frac{(N-1) \times (N-2) \times (N-3)}{6}
\]
and \( N = |P| - Mut \). Then \( P \) matches \( Str \) within \( Mut \) mutations (i.e., \( P \) approximately occurs in \( G \), or \( G \) approximately contains \( P \), within \( Mut \) mutations).

**Proof.** By Theorem 2, we increase the counter value only when there are true node-triplet matches that are augmentable under the \( SF_P \). Thus, the counter value shows currently how many node-triplets from the pattern \( P \) are found to match with the node-triplets from the substructure \( Str \) in the hash table. Note that \( \Theta_P \) represents the number of distinct node-triplets that can be generated from a substructure of size

17
If there are $N-1$ node matches between $P$ and $Str$, then $Cnt \leq \Theta_P$. Therefore, when $Cnt > \Theta_P$, there are at least $N$ node matches between $Str$ and $P$. This completes the proof.

Notice that our algorithm can not handle patterns with size smaller than 3, because the basic processing unit here is a node-triplet of size 3. This is reflected in the condition $|P| \geq Mut + 3$ stated in Theorem 3. Since $Mut$ can only be zero or greater than zero, this condition implies that the size of a pattern must be greater than or equal to 3.

**Example 7.** To illustrate how Theorem 3 works, let us refer to Example 6. Suppose the user-specified mutation number $Mut$ is 1. The candidate pattern $P$ in Fig. 9 has size $|P| = 5$. After rehashing the node-triplets of $P$, there is only one counter associated with the substructure $Str_0$ in Fig. 2(a); this counter corresponds to the $SF_P$ in Equation (10) and the value of the counter, $Cnt$, is 10. Thus, $Cnt$ is greater than $\Theta_P = (5 - 2)(5 - 3)(5 - 4)/6 = 1$. By Theorem 3, $P$ should match the substructure $Str_0$ within 1 mutation. This means that there are at least 4 node matches between $P$ and $Str_0$.

Thus, after rehashing the node-triplets of each candidate pattern $P$ into the 3D table $H$, we check the values of the counters associated with the substructures in $H$. By Theorem 3, $P$ approximately occurs in a graph $G$ within $Mut$ mutations if $G$ contains a substructure $Str$ and there is at least one counter associated with $Str$ whose value $Cnt > \Theta_P$. If there are less than $Occur$ graphs in which $P$ approximately occurs within $Mut$ mutations, then we discard $P$. The remaining candidates are qualified patterns. Notice that Theorem 3 provides only the “sufficient” (but not the “necessary”) condition for finding the qualified patterns. Due to the accumulative errors arising in the calculations, some node-triplets may be hashed to a wrong bin. As a result, the pattern-finding algorithm may miss some node-triplet matches and therefore miss some qualified patterns. In Section 3 we will show experimentally that the missed patterns are few compared with those found by exhaustive search.

**Theorem 4.** Let the set $S$ contain $K$ graphs, each having at most $N$ nodes. The time complexity of the proposed pattern-finding algorithm is $O(KN^3)$.

**Proof.** For each graph in $S$, phase (1) of the algorithm requires $O(N^2)$ time to decompose the graph into substructures. Thus the time needed for phase (1) is $O(KN^2)$. In subphase A of phase (2), we hash each candidate pattern $P$ by considering the combinations of any three nodes in $P$, which requires time $\binom{|P|}{3} = O(|P|^3)$. Thus the time needed to hash all candidate patterns is $O(KN^3)$. In subphase B of phase (2), we rehash each candidate pattern, which requires the same time $O(KN^3)$.  

---

\( ^9 \)Notice that we cannot use $Cnt = \Theta_P$ here. The reason is that we augment node-triplet matches only when they yield the same $SF_P$. In practice, it’s likely that two node-triplet matches could be augmented, but yield different $SF_P$’s due to the errors accumulated during the calculation of the $SF_P$’s. As a consequence, our algorithm fails to detect those node-triplet matches and update the counter values appropriately.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mut</td>
<td>1</td>
<td>Allowed mutation between a pattern and a graph</td>
</tr>
<tr>
<td>Occur</td>
<td>3</td>
<td>Minimum occurrence number of an interesting pattern</td>
</tr>
<tr>
<td>Size</td>
<td>6</td>
<td>Minimum size of an interesting pattern</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>0.01</td>
<td>Allowed error in comparing the entries of two coordinate matrices</td>
</tr>
<tr>
<td>Scale</td>
<td>10</td>
<td>the multiplier used in calculating the hash bin address</td>
</tr>
<tr>
<td>Prime$_1$</td>
<td>1,009</td>
<td>the 1st prime number used in calculating the hash bin address</td>
</tr>
<tr>
<td>Prime$_2$</td>
<td>1,033</td>
<td>the 2nd prime number used in calculating the hash bin address</td>
</tr>
<tr>
<td>Prime$_3$</td>
<td>1,051</td>
<td>the 3rd prime number used in calculating the hash bin address</td>
</tr>
<tr>
<td>Nrow</td>
<td>101</td>
<td>the cardinality of the hash table along each dimension</td>
</tr>
</tbody>
</table>

Table 3. Parameters in the pattern-finding algorithm and their base values used in the experiments.

3 Performance Evaluation

We carried out a series of experiments to evaluate the performance and the speed of our approach. The programs were written in the C programming language and run on a SunSPARC 20 workstation under the Solaris operating system version 2.4. Parameters used in the experiments can be classified into two categories: those related to data and those related to the pattern-finding algorithm. In the first category, we considered the size (in number of nodes) of a graph and the total number of graphs in a dataset. In the second category, we considered all the parameters described in Section 2, which are summarized in Table 3 together with the base values used in the experiments.

Two files were maintained: one recording the hash bin addresses and the other containing the entries stored in the hash bins. To evaluate the performance of the pattern-finding algorithm, we applied the algorithm to two sets of data: 1,000 synthetic graphs and 226 chemical compounds obtained from a drug database maintained in the National Cancer Institute. When generating the artificial graphs, we randomly generated the 3D coordinates for each node. The node labels were drawn randomly from the range A to E. The size of the rigid substructures in an artificial graph ranged from 4 to 10 and the size of the graphs ranged from 10 to 50. The size of the compounds ranged from 5 to 51.

In this section, we present experimental results to answer questions concerning the performance of the pattern-finding algorithm. For example, are all approximate patterns found, i.e., is the recall high? Are any uninteresting patterns found, i.e., is the precision high? In Section 4, we study the applications of the algorithm and intend to answer questions such as whether graphs having some common phenomenological activity (e.g. they are proteins with the same function) share structural patterns in common and whether these patterns can characterize the graphs as a whole.

3.1 Effect of Data-Related Parameters

To evaluate the performance of the proposed pattern-finding algorithm, we compared it with exhaustive search. The exhaustive search procedure works by generating all candidate patterns as in phase (1) of the pattern-finding algorithm. Then the procedure examines if a pattern $P$ approximately matches a substructure $Str$ in a graph by permuting the node labels of $P$ and checking if they match the node labels
of $Str$. If so, the procedure performs translation and rotation on $P$ and checks if $P$ can geometrically match $Str$.

The speed of the algorithms was measured by the running time. The performance was evaluated using three measures: recall ($RE$), precision ($PR$), and the number of false matches, $N_{fm}$, arising during the hashing process. (Recall that a false match arises, if a node-triplet $[u, v, w]$ from a pattern $P$ has the same hash bin address as a node-triplet $[i, j, k]$ from a substructure of graph $G$, though the nodes $u, v, w$ do not match the nodes $i, j, k$ geometrically, cf. Section 2.3.2.) Recall is defined as

$$RE = \frac{UP\ Found}{TotalP} \times 100\%$$

Precision is defined as

$$PR = \frac{UP\ Found}{P\ Found} \times 100\%$$

where $P\ Found$ is the number of patterns found by the proposed algorithm, $UP\ Found$ is the number of patterns found that satisfy the user-specified parameter values, and $TotalP$ is the number of qualified patterns found by exhaustive search. One would like both $RE$ and $PR$ to be as high as possible.

Fig. 10 shows the running times of the algorithms as a function of the number of graphs and Fig. 11 shows the recall. The parameters used in the proposed pattern-finding algorithm had the values shown in Table 3. As can be seen from the figures, the proposed algorithm is $10,000$ times faster than the exhaustive search method when the dataset has more than $600$ graphs while achieving a very high ($> 97\%$) recall. Due to the accumulative errors arising in the calculations, some node-triplets may be hashed to a wrong bin. As a result, the proposed algorithm may miss some node-triplet matches in subphase B of phase (2) and therefore can not achieve a $100\%$ recall. In these experiments, precision was $100\%$.

Fig. 12 shows the number of false matches introduced by the proposed algorithm as a function of the number of graphs. For the chemical compounds, $N_{fm}$ is small. For the synthetic graphs, $N_{fm}$ increases as the number of graphs becomes large. Similar results were obtained when testing the size of graphs for both types of data.
Fig. 11. Recall as a function of the number of graphs.

Fig. 12. Number of false matches as a function of the number of graphs.
3.2 Effect of Algorithm-Related Parameters

The purpose of this subsection is to analyze the effect of varying the algorithm-related parameter values on the performance of the proposed pattern-finding algorithm. To avoid the mutual influence of parameters, the analysis was carried out by fixing the parameter values related to data graphs—the 1,000 synthetic graphs and 226 compounds described above were used, respectively. In each experiment, only one algorithm-related parameter value was varied; the other parameters had the values shown in Table 3.

We first examined the effect of varying the parameter values on generating false matches. Since few false matches were found for chemical compounds, the experiments focused on synthetic graphs. It was observed that only \( \text{Nrow} \) and \( \text{Scale} \) affected the number of false matches. Fig. 13 shows \( N_{fm} \) as a function of \( \text{Scale} \) for \( \text{Nrow} = 101, 131, 167, 199 \), respectively. The larger the \( \text{Nrow} \), the fewer entries in a hash bin, and consequently the fewer false matches. On the other hand, when \( \text{Nrow} \) is too large, the running times increase substantially, since one needs to spend a lot of time in reading the 3D table containing the hash bin addresses.

Examining Fig. 13, we see how \( \text{Scale} \) affects \( N_{fm} \). Taking an extreme case, when \( \text{Scale} = 1 \), a node-triplet with the squares of the lengths of the three edges connecting the nodes being 12.4567 is hashed to the same bin as a node-triplet with those values being 12.0000, although the two node-triplets do not match geometrically, cf. Section 2.3.1. On the other hand, when \( \text{Scale} \) is large (e.g. \( \text{Scale} = 10,000 \)), the distribution of hash function values is less skewed, which reduces the number of false matches. It was observed that \( N_{fm} \) was largest when \( \text{Scale} \) was 100. This happens because with this \( \text{Scale} \) value, inaccuracy was being introduced in calculating hash bin addresses. A node-triplet being hashed to a bin might generate \( k \) false matches where \( k \) is the total number of node-triplets already stored in that bin—\( k \) would be large if the distribution of hash function values is skewed.

Figures 14, 15, and 16 show the recall as a function of \( \text{Size} \), \( \text{Mut} \), and \( \text{Scale} \), respectively. In all three figures, precision is 100%. From Fig. 14 and Fig. 15, we see that \( \text{Size} \) and \( \text{Mut} \) affect recall slightly. It was also observed that the number of interesting patterns drops (increases, respectively) significantly as
Size (Mut, respectively) becomes large. Fig. 16 shows that the pattern-finding algorithm yields a poor performance when Scale is large. With the data we tested, we found that setting Scale to 10 is the best overall for both recall and precision.

In sum, the value of Scale has a significant impact on the performance of our algorithm. The choice of Scale is domain and data dependent. In practice, how would one select a value of Scale for a particular database? Recall that the coordinates \((x, y, z)\) of each node in a 3D graph have an error due to rounding, cf. Section 2.3.1. The real coordinates for the node at \((x_i, y_i, z_i)\) should be \((\bar{x}_i, \bar{y}_i, \bar{z}_i)\), where \(\bar{x}_i = x_i + \epsilon_{x_i}, \bar{y}_i = y_i + \epsilon_{y_i}, \bar{z}_i = z_i + \epsilon_{z_i}\). Similarly, the real coordinates for the node at \((x_j, y_j, z_j)\) should be \((\bar{x}_j, \bar{y}_j, \bar{z}_j)\), where \(\bar{x}_j = x_j + \epsilon_{x_j}, \bar{y}_j = y_j + \epsilon_{y_j}, \bar{z}_j = z_j + \epsilon_{z_j}\). The real value of \(l_1\) in Equation (2) should be

\[
\bar{l}_1 = ((\bar{x}_i - \bar{x}_j)^2 + (\bar{y}_i - \bar{y}_j)^2 + (\bar{z}_i - \bar{z}_j)^2) \times \text{Scale}
\]

\[
= ((x_i + \epsilon_{x_i} - x_j - \epsilon_{x_j})^2 + (y_i + \epsilon_{y_i} - y_j - \epsilon_{y_j})^2 + (z_i + \epsilon_{z_i} - z_j - \epsilon_{z_j})^2) \times \text{Scale}
\]

\[
= ((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times \text{Scale}
\]
Recall (%)

\[ +2 \times ((x_i - x_j)(e_{z_i} - e_{z_j}) + (y_i - y_j)(e_{y_i} - e_{y_j}) + (z_i - z_j)(e_{z_i} - e_{z_j})) \times \text{Scale} \]
\[ + ((e_{x_i} - e_{x_j})^2 + (e_{y_i} - e_{y_j})^2 + (e_{z_i} - e_{z_j})^2) \times \text{Scale} \]
\[ = l_1 + e_{i_1} \]

where \( e_{i_1} \) is an accumulative error,
\[ e_{i_1} = 2 \times ((x_i - x_j)(e_{x_i} - e_{x_j}) + (y_i - y_j)(e_{y_i} - e_{y_j}) + (z_i - z_j)(e_{z_i} - e_{z_j})) \times \text{Scale} \]
\[ + ((e_{x_i} - e_{x_j})^2 + (e_{y_i} - e_{y_j})^2 + (e_{z_i} - e_{z_j})^2) \times \text{Scale} \]

Since we used \( l_1 \) to calculate the hash bin addresses, it is critical that the accumulative error would not mislead us to a wrong hash bin. Assuming that the coordinates are accurate to the \( n \)th digit after the decimal point, i.e. to \( 10^{-n} \), the error caused by eliminating the digits after the \( (n + 1) \)th position is no larger than \( 0.5 \times 10^{-n} \). Namely, for any coordinates \((x, y, z)\), we have \( e_x \leq 0.5 \times 10^{-n}, e_y \leq 0.5 \times 10^{-n}, \) and \( e_z \leq 0.5 \times 10^{-n} \). Thus,
\[ e_{i_1} \leq 2 \times ((|x_i - x_j||e_{x_i} - e_{x_j}| + |y_i - y_j||e_{y_i} - e_{y_j}| + |z_i - z_j||e_{z_i} - e_{z_j}|) \times \text{Scale} \]
\[ + ((|e_{x_i} - e_{x_j}|^2 + |e_{y_i} - e_{y_j}|^2 + |e_{z_i} - e_{z_j}|^2) \times \text{Scale} \]
\[ \leq 2 \times (|x_i - x_j|(|e_{x_i}| + |e_{x_j}|) + |y_i - y_j|(|e_{y_i}| + |e_{y_j}|) + |z_i - z_j|(|e_{z_i}| + |e_{z_j}|)) \times \text{Scale} \]
\[ + ((|e_{x_i}| + |e_{x_j}|)^2 + (|e_{y_i}| + |e_{y_j}|)^2 + (|e_{z_i}| + |e_{z_j}|)^2) \times \text{Scale} \]
\[ \leq 2 \times (|x_i - x_j| + |y_i - y_j| + |z_i - z_j|) \times 2 \times 0.5 \times 10^{-n} \times \text{Scale} \]
\[ + 3 \times (2 \times 0.5 \times 10^{-n})^2 \times \text{Scale} \]
\[ = (|x_i - x_j| + |y_i - y_j| + |z_i - z_j|) \times 2 \times \text{Scale} \times 10^{-n} + 3 \times \text{Scale} \times 10^{-2n} \]

Assume that the range of the coordinates is \([-M, M]\). We have \(|x_i - x_j| \leq 2M, |y_i - y_j| \leq 2M, \) and \(|z_i - z_j| \leq 2M\). Thus,
\[ e_{i_1} \leq 12 \times M \times \text{Scale} \times 10^{-n} + 3 \times \text{Scale} \times 10^{-2n} \]
The second term is obviously negligible in comparison with the first term. In order to keep the calculation of hash bin addresses accurate, the first term should be smaller than 1. That is, \textit{Scale} should be chosen to be the largest possible number such that

\[ 12 \times M \times \text{Scale} \times 10^{-n} < 1 \]

or

\[ \text{Scale} < \frac{1}{12 \times M} \times 10^n \]

In our case, the coordinates of the chemical compounds used were accurate to the 4th digit after the decimal point and the range of the coordinates was [-10, 10]. Consequently,

\[ \text{Scale} < \frac{1}{12 \times 10} \times 10^4 = \frac{500}{6} \]

Thus, choosing \textit{Scale} = 10 is good, as validated by our experimental results. Notice that \textit{Scale} can be determined once the data are given. All we need to know is how accurate the data are, and what the ranges of their coordinates are, both of which are often clearly associated with the data.

Figures 17 and 18 show the recall and precision as a function of \( \epsilon \). It can be seen that when \( \epsilon \) is 0.01, precision is 100\% and recall is greater than 97\%. When \( \epsilon \) becomes smaller (e.g. \( \epsilon = 0.0001 \)), precision remains the same while recall drops. When \( \epsilon \) becomes larger (e.g. \( \epsilon = 10 \)), recall increases slightly while precision drops. This happens because some irrelevant node-triplet matches were included, rendering unqualified patterns returned as an answer. We also tested different values for \textit{Occur}, \textit{Nrow}, \textit{Prime}_1, \textit{Prime}_2 and \textit{Prime}_3. It was found that varying these parameter values had little impact on the performance of the proposed algorithm.

Finally, we examined the effect of sensor errors, which are caused by the measuring device (e.g. X-ray crystallography), and are proportional to the values being measured. Suppose that the sensor error is \( 10^{-n} \) of the real value for any coordinates \((x, y, z)\), i.e. \( \epsilon_x = x \times 10^{-n} \), \( \epsilon_y = y \times 10^{-n} \), and \( \epsilon_z = z \times 10^{-n} \). We have \( x = x \times (1 + 10^{-n}) \), \( y = y \times (1 + 10^{-n}) \), and \( z = z \times (1 + 10^{-n}) \). Let \( \gamma_1 \) represent the sensor error.
induced in calculating $l_1$ in Equation (2). In a way similar to the calculation of the accumulative error $e_i$, in Equation (15), we obtain

$$\gamma_i = ((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times 2 \times 10^{-n} \times Scale$$

and

$$+((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times 10^{-2n} \times Scale$$

Let us focus on the first term since it is more significant. Clearly, the proposed approach is not feasible in the environment with large sensor errors. Taking an example, suppose that the sensor error is $10^{-2}$ of the real value. Again, in order to keep the calculation of hash bin addresses accurate, the accumulative error should be smaller than 1. That is,

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times 2 \times 10^{-2} \times Scale < 1$$

or

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2) \times Scale < 50$$

Even when $Scale = 1$, the Euclidean distance between any two nodes must be smaller than 7, since the square of the distance, viz. $(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)$, must be less than 50. Most data sets, including all the chemical compounds we used, do not satisfy this restriction.

To illustrate how this would affect recall, consider a substructure $P$ already stored in the hash table. We add a $10^{-2}$ sensor error to the coordinates of a node $v$ in $P$. Call the resulting substructure $P'$. Then we use $P'$ as a pattern to match the original substructure $P$ in the hash table. Those node-triplets generated from the pattern $P'$ that include the node $v$ would not match the corresponding node-triplets in the original substructure $P$. Supposing the pattern $P'$ has $N$ nodes, the total number of node-triplets that include the node $v$ is $\binom{N-1}{2}$. Thus, we would miss the same number of node-triplet matches. However, if all other node-triplets that do not include the node $v$ are matched successfully, we would still have $\binom{N}{3} - \binom{N-1}{2} = \binom{N-1}{3}$ node-triplet matches. According to Theorem 3, there would be $N - 1$ node matches between the pattern $P'$ and the original substructure $P$ (i.e. $P'$ matches $P$ with 1
mutation). This example reveals that, in general, if we allow mutations to occur in searching for patterns and the number of nodes with sensor errors equals the allowed number of mutations, the recall will be the same as the case in which there are no sensor errors and we search for exactly matched patterns without mutations. On the other hand, if no mutations are allowed in searching for patterns and sensor errors occur, the recall will be zero.

4 Data Mining Applications

One important application of pattern finding involves the ability to perform classification and clustering as well as other data mining tasks. In this section, we present two data mining applications of the proposed algorithm in scientific domains: classifying proteins and clustering compounds.\textsuperscript{10}

4.1 Classifying Proteins

Proteins are large molecules, comprising hundreds of amino acids (residues) [18, 33]. In each residue the \( C_{\alpha} \), \( C_{\beta} \) and N atoms form a backbone of the residue [17]. Following [28], we represent each residue by the three atoms. Thus, if we consider a protein as a 3D graph, each node of the graph is an atom. Each node has a label, which is the name of the atom and is not unique in the protein. We assign a unique number to identify a node in the protein, where the order of numbering is obtained from the Protein Data Bank (PDB) [1, 2], accessible at http://www.rcsb.org.

In the experiments, we examined two families of proteins chosen from PDB pertaining to RNA-directed DNA Polymerase and Thymidylate Synthase. Each family contains proteins having the same functionality in various organisms. We decompose each protein into consecutive substructures, each substructure containing 6 nodes. Two adjacent substructures overlap by sharing the two neighboring nodes on the boundary of the two substructures (see Fig. 19). Thus, each substructure is a portion of the polypeptide chain backbone of a protein where the polypeptide chain is made up of residues linked together by peptide bonds. The peptide bonds have strong covalent bonding forces that make the polypeptide chain rigid. As a consequence, the substructures used by our algorithm are rigid. Notice that in the proteins there are other atoms such as O and H (not shown in Fig. 19) lying between two residues. Since these atoms are not as important as \( C_{\alpha} \), \( C_{\beta} \) and N in determining the structure of a protein, we do not consider them here. Table 4 summarizes the number of proteins in each family, their sizes and the frequently occurring patterns (or motifs) discovered from the proteins. The parameter values used were as shown in Table 3. In the experiments, 2,784 false matches were detected and eliminated during the process of finding the motifs. That is, there were 2,784 times in which a node-triplet \([u, v, w]\) from a pattern was found to have the same hash bin address as a node-triplet \([i, j, k]\) from a substructure of a protein, though the nodes \( u, v, w \) did not match the nodes \( i, j, k \) geometrically, cf. Section 2.3.2.

To evaluate the quality of the discovered motifs, we applied them to classifying the proteins using the 10-way cross-validation scheme. That is, each family was divided into 10 groups of roughly equal size.

\textsuperscript{10}Another application of the proposed algorithm to flexible chemical searching in 3D structural databases can be found in [34].
Specifically, the RNA-directed DNA Polymerase family, referred to as family 1, contained five groups each having 5 proteins and five groups each having 4 proteins. The Thymidylate Synthase family, referred to as family 2, contained seven groups each having 4 proteins and three groups each having 3 proteins. Ten tests were conducted. In each test, a group was taken from a family and used as test data; the other nine groups were used as training data for that family. We applied our pattern-finding algorithm to each training dataset to find motifs (the parameter values used were as shown in Table 3). Each motif $M$ found in family $i$ was associated with a weight $d$ where

$$d = r_i - r_j$$

for $1 \leq i, j \leq 2, \ i \neq j$

<table>
<thead>
<tr>
<th>Family</th>
<th>Number of proteins</th>
<th>Maximum protein size</th>
<th>Minimum protein size</th>
<th>Number of motifs</th>
<th>Motif size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNA-directed DNA Polymerase</td>
<td>45</td>
<td>1,812</td>
<td>146</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>Thymidylate Synthase</td>
<td>37</td>
<td>2,128</td>
<td>1,000</td>
<td>33</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4. Statistics concerning the proteins and motifs found in them.
Here $r_i$ is $M$'s occurrence number in the training dataset of family $i$. Intuitively, the more frequently a motif occurs in its own family and the less frequently it occurs in the other family, the higher its weight is. In each family we collected all the motifs having a weight greater than one and used them as the characteristic motifs of that family.

When classifying a test protein $Q$, we first decomposed $Q$ into consecutive substructures as described above. The result was a set of substructures, say, $Q^1, \ldots, Q^p$. Let $n_i^k, 1 \leq i \leq 2, 1 \leq k \leq p$, denote the number of characteristic motifs in family $i$ that matched $Q^k$ within one mutation. Each family $i$ obtained a score $N_i$ where

$$N_i = \frac{\sum_{k=1}^{p} n_i^k}{m_i}$$

and $m_i$ is the total number of characteristic motifs in family $i$. Intuitively, the score $N_i$ here was determined by the number of the characteristic motifs in family $i$ that occurred in $Q$, divided by the total number of characteristic motifs in family $i$. The protein $Q$ was classified into the family $i$ with maximum $N_i$. If the scores were 0 for both families (i.e., the test protein did not have any substructure that matched any characteristic motif), then the “no-opinion” verdict was given. This algorithm is similar to those used in [30, 33] to classify chemical compounds and sequences.

As in Section 3, we use recall ($RE_c$) and precision ($PR_c$) to evaluate the effectiveness of our classification algorithm. Recall is defined as

$$RE_c = \frac{TotalNum - \sum_{i=1}^{2} NumLoss^i}{TotalNum} \times 100\%$$

where $TotalNum$ is the total number of test proteins and $NumLoss^i$ is the number of test proteins that belong to family $i$ but are not assigned to family $i$ by our algorithm (they are either assigned to family $j$, $j \neq i$, or they receive the “no-opinion” verdict). Precision is defined as

$$PR_c = \frac{TotalNum - \sum_{i=1}^{2} NumGain^i}{TotalNum} \times 100\%$$

where $NumGain^i$ is the number of test proteins that do not belong to family $i$ but are assigned by our algorithm to family $i$. With the 10-way cross validation scheme, the average $RE_c$ over the ten tests was 92.7% and the average $PR_c$ was 96.4%. It was found that 3.7% test proteins on average received the “no-opinion” verdict during the classification. We repeated the same experiments using other parameter values and obtained similar results, except that larger $Mut$ values (e.g., 3) generally yielded lower $RE_c$.

The binary classification problem studied here is concerned with assigning a protein to one of two families. This problem arises frequently in protein homology detection [31]. The performance of our technique degrades when applied to the $n$-ary classification problem, which is concerned with assigning a protein to one of many families [32]. For example, we applied the technique to classifying three families of proteins and the $RE_c$ and $PR_c$ dropped to 80%.

4.2 Clustering Compounds

In addition to classifying proteins, we have developed an algorithm for clustering 3D graphs based on the patterns occurring in the graphs and have applied the algorithm to grouping compounds. Given a
collection $S$ of 3D graphs, the algorithm first uses the procedure depicted in Section 2.2 to decompose the graphs into rigid substructures. Let $\{Str_p\}_{p=0,1,...,N-1}$ be the set of substructures found in the graphs in $S$ where $|Str_p| \geq Size$. Using the proposed pattern-finding algorithm, we examine each graph $G_q$ in $S$ and determine whether each substructure $Str_p$ approximately occurs in $G_q$ within $Mut$ mutations. Each graph $G_q$ is represented as a bit string of length $N$, i.e., $G_q = (b_q^0, b_q^1,...,b_q^{N-1})$, where

$$b_q^p = \begin{cases} 1 & \text{if } Str_p \text{ occurs in } G_q \text{ within } Mut \text{ mutations} \\ 0 & \text{otherwise} \end{cases}$$

For example, consider the two patterns $P_1$ and $P_2$ in Fig. 3 and the three graphs in Fig. 3(a). Suppose the allowed number of mutations is 0. Then $G_1$ is represented as 10, $G_2$ as 01, and $G_3$ as 10. On the other hand, suppose the allowed number of mutations is 1. Then $G_1$ and $G_3$ are represented as 11, and $G_2$ as 01.

The distance between two graphs $G_x$ and $G_y$, denoted $d(G_x, G_y)$, is defined as the Hamming distance [12] between their bit strings. The algorithm then uses the well known average-group method [13] to cluster the graphs in $S$, which works as follows.

Initially, every graph is a cluster. The algorithm merges two nearest clusters to form a new cluster, until there are only $K$ clusters left where $K$ is a user-specified parameter. The distance between two clusters $C_1$ and $C_2$ is given by

$$\frac{1}{|C_1||C_2|} \sum_{G_x \in C_1, G_y \in C_2} |d(G_x, G_y)|$$

(16)

where $|C_i|$, $i = 1, 2$, is the size of cluster $C_i$. The algorithm requires $O(N^2)$ distance calculations where $N$ is the total number of graphs in $S$.

We applied this algorithm to clustering chemical compounds. Ninety eight compounds were chosen from the Merck Index that belonged to three groups pertaining to aromatic, bicycalkanes and photosynthesis. The data was created by the CORINA program that converted 2D data (represented in SMILES string) to 3D data (represented in PDB format) [22]. Table 5 lists the number of compounds in each group, their sizes and the patterns discovered from them. The parameter values used were $Size = 5$, $Occur = 1$, $Mut = 2$; the other parameters had the values shown in Table 3.

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of compounds</th>
<th>Minimum compound size</th>
<th>Maximum compound size</th>
<th>Number of patterns</th>
<th>Minimum pattern size</th>
<th>Maximum pattern size</th>
</tr>
</thead>
<tbody>
<tr>
<td>aromatic</td>
<td>36</td>
<td>12</td>
<td>42</td>
<td>58</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>bicycalkanes</td>
<td>26</td>
<td>16</td>
<td>40</td>
<td>53</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>photosynthesis</td>
<td>36</td>
<td>31</td>
<td>44</td>
<td>114</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 5. Statistics concerning the chemical compounds and patterns found in them.

To evaluate the effectiveness of our clustering algorithm, we applied it to finding clusters in the compounds. The parameter value $K$ was set to 3, as there were three groups. As in the previous sections, we
use recall ($RE_r$) and precision ($PR_r$) to evaluate the effectiveness of the clustering algorithm. Recall is defined as

$$RE_r = \frac{TotalNum - \sum_{i=1}^{K} NumLoss_i}{TotalNum} \times 100\%$$

where $NumLoss_i$ is the number of compounds that belong to group $G_i$, but are assigned by our algorithm to group $G_j$, $i \neq j$, and $TotalNum$ is the total number of compounds tested. Precision is defined as

$$PR_r = \frac{TotalNum - \sum_{i=1}^{K} NumGain_i}{TotalNum} \times 100\%$$

where $NumGain_i$ is the number of compounds that do not belong to group $G_i$, but are assigned by our algorithm to group $G_i$. Our experimental results indicated that $RE_r = PR_r = 99\%$. Out of the 98 compounds, only one compound in the photosynthesis group was assigned incorrectly to the bicyclicalkanes group. We experimented with other parameter values and obtained similar results.\textsuperscript{11}

5 Related Work

There are several groups working on pattern finding (or knowledge discovery) in molecules and graphs. Conklin et al. [4, 5, 6], for example, represented a molecular structure as an image, which comprised a set of parts with their 3D coordinates and a set of relations that were preserved for the image. The authors used an incremental, divisive approach to discover the “knowledge” from a dataset, that is, to build a subsumption hierarchy that summarized and classified the dataset. The algorithm relied on a measure of similarity among molecular images that was defined in terms of their largest common subimages.

In [8], Djoko et al. developed a system, called SUBDUE, that utilized the minimum description length principle to find repeatedly occurring substructures in a graph. Once a substructure was found, the substructure was used to simplify the graph by replacing instances of the substructure with a pointer to the discovered substructure. In [7], Dehaspe et al. used DATALOG to represent compounds and applied data mining techniques to predicting chemical carcinogenicity. Their techniques were based on Mannila and Toivonen’s algorithm [15] for finding interesting patterns from a class of sentences in a database.

In contrast to the above work, we use the geometric hashing technique to find approximately common patterns in a set of 3D graphs without prior knowledge of their structures, positions, or occurrence frequency. The geometric hashing technique used here originated from the work of Lamdan and Wolfsen for model based recognition in computer vision [14]. Several researchers attempted to parallelize the technique based on various architectures, such as the Hypercube and the Connection Machine [3, 19, 20]. It was observed that the distribution of the hash table entries might be skewed. To balance the distribution of the hash function values, delicate rehash functions were designed [20]. There were also efforts exploring the uncertainty existing in the geometric hashing algorithms [11, 26].

In 1996, Rigoutsos et al. employed geometric hashing and magic vectors for substructure matching in a database of chemical compounds [21]. The magic vectors were bonds among atoms; the choice of them was domain dependent and was based on the type of each individual graph. We extend the work in [21]

\textsuperscript{11}The Occur value was fixed at 1 in these experiments because of the fact that all the compounds were represented as binary bit strings.
by providing a framework for discovering approximately common substructures in a set of 3D graphs, and applying our techniques to both compounds and proteins. Our approach differs from [21] in that instead of using the magic vectors, we store a coordinate system in a hash table entry. Furthermore, we establish a theory for detecting and eliminating false matches occurring in the hashing.

More recently, Wolfson and his colleagues also applied geometric hashing algorithms to protein docking and recognition [23, 24, 29]. They attached a reference frame to each substructure, where the reference frame is an orthonormal coordinate frame of arbitrary orientations, established at a “hinge” point. This hinge point can be any point or a specific point chosen based on chemical considerations. The 3D substructure can be rotated around the hinge point. The authors then developed schemes to generate node-triplets and hash table indices. However, based on those schemes, false matches cannot be detected in the hash table and must be processed in a subsequent, additional verification phase. Thus, even if a pattern appears to match several substructures during the searching phase, one has to compare the pattern with those substructures one by one to make sure they are true matches during the verification phase. When mutations are allowed, which were not considered in [23, 24, 29], a brute-force verification test would be very time-consuming. For example, in comparing our approach with their approach in performing protein classification as described in Section 4, we found that both approaches yield the same recall and precision, though our approach is 10 times faster than their approach.

6 Conclusion

In this paper we have presented an algorithm for finding patterns in 3D graphs. A pattern here is a rigid substructure that may occur in a graph after allowing for an arbitrary number of rotations and translations as well as a small number of edit operations in the pattern or in the graph. We used the algorithm to find approximately common patterns in a set of synthetic graphs, chemical compounds and proteins. This yields a kind of alphabet of patterns. Our experimental results demonstrated the good performance of the proposed algorithm and its usefulness for pattern discovery. We then developed classification and clustering algorithms using the patterns found in the graphs, and applied them to classifying proteins and clustering compounds. Empirical study showed high recall and precision rates for both classification and clustering, indicating the significance of the patterns.

We have implemented the techniques presented in the paper and are combining them with the algorithms for acyclic graph matching [36] into a toolkit. We use the toolkit to find patterns in various types of graphs arising in different domains and as part of our tree and graph search engine project. The toolkit can be obtained from the authors and is also accessible at http://www.cis.njit.edu/~jaso/sigmod.html on the Web.

In the future, we plan to study topological graph matching problems. One weakness of the proposed geometric hashing approach is its sensitivity to errors as we discussed in Section 3.2. One has to tune the Scale parameter when the accuracy of data changes. Another weakness is that the geometric hashing approach doesn’t take edges into consideration. In some domains, edges are important despite the nodes matching. For example, in computer vision, edges may represent some kind of boundaries and have to be considered.
in object recognition. (Notice that when applying the proposed geometric hashing approach to this domain, the node label alphabet could be extended to contain semantic information, such as color, shape, etc.) We have done preliminary work in topological graph matching, in which our algorithm matches edge distances and the software is accessible at http://cs.nyu.edu/cs/faculty/shasha/papers/agm.html. We plan to combine geometric matching and topological matching approaches to finding patterns in more general graphs.

Acknowledgments

We thank the anonymous reviewers for their constructive suggestions that helped improve both the quality and the presentation of this paper. We also thank Dr. Carol Venanzi, Bo Chen, and Song Peng for useful discussions and for providing chemical compounds used in the paper.

References


