A Platform Based on the Multi-dimensional Data Model for Analysis of Bio-Molecular Structures

Srinath Srinivasa

Indian Institute of Information Technology
"Innovator" Towers, ITPL,
Whitefield Road,
Bangalore 560066, INDIA
sri@iiitb.ac.in

Sujit Kumar

C L Infotech Pvt. Ltd
214, 1st main, 7th Block,
Koramangala,
Bangalore 560095, INDIA
sujit@clinf.com

Abstract

A platform called AnMol for supporting analytical applications over structural data of large biomolecules is described. The term "biomolecular structure" has various connotations and different representations. AnMol reduces these representations into graph structures. Each of these graphs are then stored as one or more vectors in a database. Vectors encapsulate structural features of these graphs. Structural queries like similarity and substructure are transformed into spatial constructs like distance and containment within regions. Query results are based on inexact matches. A refinement mechanism is supported for increasing accuracy of the results. Design and implementation issues of AnMol including schema structure and performance results are discussed in this paper.

Keywords: Biomolecular structures, Vectorization of structure, OLAP, Layered star schema

1 Introduction

Biomolecules like proteins are large molecular structures having hundreds to thousands of atoms each. Databases of biomolecular structures hold valuable information for activities like drug design, understanding of metabolic processes, etc. However, such information is not immediately relevant from the data; which brings in a need for analytical applications over such databases.

There are many existing applications that perform specific activities over molecular structures. For example, the SCOP [14], FSSP [6] and CATH [1] databases maintain structure-based classifications of protein molecules. While this would be useful for structural similarity searches, there is still a need for a more general *platform* for supporting many analytical queries in a uniform fashion. An analytical process may require other operations like clustering based on common substructures, finding different mutations of a structure, etc. in addition to structural similarity searches.

The objective of our work is to design such a computational platform, or an OLAP engine for molecular structures. This platform is named AnMol.

The design goals of a platform meant for supporting analysis is different from that of databases that store specific information like similarity measures. Analysis has to support searches that look for trends over a large dataset. For instance, collective properties like finding clusters of molecules satisfying a required property is more important than finding precise structural differences between a pair of molecules. Such queries have to be supported in an interactive fashion. AnMol hence incorporates searches based on *inexact* structure matching, but having a capability of interactive response times and refinement mechanisms for query results.

Biomolecules and their representation: A typical large biomolecule like a protein, comprises of hundreds to thousands of atoms. Most of these atoms are hydrocarbons with carbon and hydrogen occurring most frequently. There may also be other atoms like nitrogen, sulfur and phosphorus found in the molecules.

Adjacent atoms in a molecule are connected by covalent bonds. In addition to covalent bonds there may be "non-bonded" interactions between atoms that may

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

be far away in the covalent structure graph. Non-bonded interactions are responsible for folding the molecule and giving it a characteristic 3D structure. Usually, the 3D structure is "rigid" in the sense that interatomic distances between any two atoms in the molecule is the same regardless of the orientation of the molecule.

Biomolecular structures are described using many different characterizations. For example, proteins are described as a sequence of amino acids (called the *primary* structure), as a map of structural elements like α-helices and β-sheets (called the *secondary* structure), as a set of structural motifs formed by connecting the structural elements (called the *tertiary* structure), or as a set of 3D coordinates identifying the positions of its atoms (called the *quaternary* structure).

AnMol converts each of these different characterizations into one or more labeled undirected graphs. The graphs are generated such that they are "structure dominated" [15]. This means that the number of labels are much smaller than the number of nodes in the graph. Also, the number of edges are much smaller than $n(n-1)/2$, where n is the number of nodes. This means that there is a large diversity in the connectivity structure of the graphs and a small diversity in the labels.

Once the graphs are obtained, AnMol then uses a concept called "vectorization of structure" to encapsulate structural features of these graphs into one or more vectors. Vectorization enables AnMol to support many operations that are typical to OLAP applications.

Structural queries like similarity and substructure are mapped to spatial queries like vector distance and containment within regions. The results of such queries would be based on *inexact* structure matching. A refinement mechanism is provided by which the user or application can progressively refine query results. There is a tradeoff between the maximum accuracy that can be obtained and the time taken to add molecular structures. The database administrator can control this using a configurable parameter.

Currently, an AnMol database holding 457 proteins is implemented on a single Pentium III, 700 MHz PC, having 64 MB of RAM. While addition of protein structures is performed in batch mode, queries are interactive.

2 Related Literature

Molecular structure of proteins and other biomolecules is usually available in the form of 3D (x, y, z) coordinates for each atom in the molecule. An example is the PDB record available from the Protein Data Bank [13].

A commonly used mechanism of comparing two molecular structures is to calculate the root-mean-square distance (RMSD) between corresponding points of two PDB records. These points of correspondence could be all atoms, the C^α atoms that signify amino acids, centers of mass of amino acids, etc. Establishing correspondence may be a semi-automatic process where some manual intervention may be required. In the DALI algorithm [7] correspondence is established by first computing a distance matrix for each molecule that shows the all-pairs distances between its atoms. Two molecules are compared by aligning their distance matrices. The matrices are aligned by moving one over the other until the largest submatrix is found where the difference between the corresponding distance elements is minimum. Such an algorithm is computationally intensive and may not be able to support interactive queries.

The FSSP database [6] uses the DALI algorithm to maintain an exhaustive all-against-all 3D structure comparison between proteins in the protein data bank. Here, distances are computed offline and the database statically maintains RMSD values across pairs of proteins. Correspondences are established against C^α atoms for RMSD calculation.

But as discussed in the earlier section, supporting analytical tools may require more than a static all-against-all distance data. Distance calculation may be just one step in a larger query like: "Find all molecules that contain fragment g, and are close to the molecule p."

The notion of distance in AnMol is different from that of RMSDs. The vector space in AnMol is not the 3D physical space that is used in the above approaches. Also, AnMol does not require costly and semi-automatic structural alignments. In Section 5, the AnMol distances between members of a randomly selected sample is compared against their RMSD values obtained from FSSP.

A number of algorithms meant for sequences have been tried for protein structures by representing proteins as sequences (for example, primary structure, and sequences from secondary structures). Some examples are Hammel and Patel [5] and Wang et al. [16].

Lamdan and Wolfson [10] propose a technique called *geometric hashing* that enables fast comparison of 3D geometric structures. Here, a geometric structure is represented as a vector of features that is independent of any rigid transformations applied to the structure. The feature vector is placed in a hash table and structural congruence is determined by vectors that hash to the same bucket in the hash table.

Leibowitz et al. [11] use geometric hashing to align many molecular structures and discern commonly occurring substructures.

Wang et al. [17] also use geometric hashing to mine for patterns in 3D graphs. A pattern is defined as a frequently occurring substructure across graphs that is obtained by an arbitrary number of rigid structure transformations like rotations and translations, and a bounded number of edit operations over member graphs.

To contrast with geometric hashing, vectorization in AnMol does not require a 3D geometric structure. Vectorization is performed on graphs represented by nodes and edges. In fact, 3D structures are first converted to graphs before hashing. Hashing graphs rather than 3D structures enables many non-structural but interconnected features of molecules to be hashed using the same algorithm. For example, a hydrogen bond graph shows the occurrences of hydrogen bonds among atoms in a protein molecule. Proteins having similar patterns of hydrogen bonds would be interesting even if their 3D structure vary considerably. Similarly, sequences are considered as a special case of graphs and are handled in an analogous manner.

A vector in AnMol however, cannot regenerate the original graph unlike the geometric hashing model of Lamdan and Wolfson [10]. Two or more graph structures can hash to the same vector. But AnMol uses a *hierarchy* of vector spaces, where a graph is hashed onto vectors at one or more levels. If two or more structurally different graphs hash to the same point at a level l, the probability that they continue to hash to the same vector keeps progressively decreasing as we move up the hierarchy above l.

In the SUBDUE [2, 8] database, a concept called hierarchical conceptual clustering is used to maintain graph structures. Here the entire database is represented as a single graph. Graph nodes appear at different hierarchical levels where a node high in the hierarchy actually represents a substructure made of lower level nodes. Conceptual clustering involves replacing frequently occurring subgraphs with a single node to form a concept hierarchy.

AnMol incorporates a concept called hierarchical vector spaces and compression, which are somewhat analogous to that in SUBDUE. However, the core concept in AnMol is of vector spaces, that can support not only substructure, but also nearness searches in a uniform fashion.

The GraphGrep [4] model developed by Guigno and Shasha supports retrieval based on *exact* subgraph isomorphism. Complexity of subgraph isomorphism is moved to an offline "preparation" phase which is a one-time database preparation operation. The preparation phase involves storing member graphs as a set of paths of length l_p, where l_p is a configurable parameter. Query graphs are also converted to a set of paths, just like the member graphs. Subgraph isomorphism is performed by sequence based searches across paths. While GraphGrep provides results based on exact matches, enumerating the set of all paths from member graphs takes up super-polynomial time. This overhead is justified by saying that determining paths out of member graphs is a one-time, offline operation.

AnMol differs from GraphGrep in that it has only polynomial running time both for preparation and operation. However, queries are based on inexact matches.

3 AnMol Vectorization Model

3.1 Overall architecture

AnMol is an extension of our earlier project called GRACE [9, 15]. The underlying concept of vectorization is the same as in GRACE. However, some vital shortcomings of GRACE have been rectified in AnMol.

Shortcomings in GRACE that have been rectified in AnMol are as follows:

1. GRACE required two phases of activity: an *initialization* phase and an *operation* phase. In the initialization phase, a few candidate graphs were sampled to identify dimensions and create vector spaces. In the operation phase, the database was populated by vectorizing graphs based on the identified dimensions. If a given graph did not contain a specific dimension, its projection was taken to be 0; and if the graph contained a new dimension, it was silently ignored. The reason for such a design stemmed from concerns about efficiency of query handling. The GRACE schema organized tables in the form: $(d_1, d_2, \ldots, d_n, name)$ where $d_1, d_2, \ldots d_n$ are dimension names and *name* is the name of the protein. The schema was indexed using KD-trees which enabled fast retrieval based on spatial constraints.

   However, silently ignoring new dimensions results in loss of information, and the database relies heavily on the set of sample structures being representative enough. There is no simple way of ensuring this, given the large and intricate structure of biomolecules.

   AnMol on the other hand adds new dimensions on the fly, without the need for an initialization phase. This is enabled by the "layered star schema" introduced in Section 4.

2. GRACE had another shortcoming with higher-level vector spaces. As explained later in this section, graphs are vectorized at many levels. At the lowest level, a given graph is treated as a vector of structural features. The graph is then rewritten by replacing every occurrence of a structural feature with a new labeled node. This creates a graph at a "higher level" which can be further vectorized in the same fashion.

   However, there are many ways in which a graph can be rewritten, resulting in a number of different "views" at the higher level. In order to handle different views, GRACE created many vector spaces. The vector spaces were organized such that there was as little overlap in dimensions as

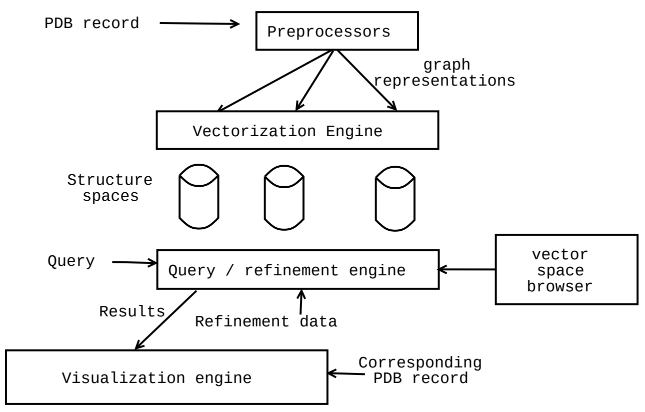


Figure 1: AnMol Architecture

possible among vector spaces. This was to avoid the problem of sparse tables comprising of a large number of zeroes. The layered star-schema avoids this altogether by changing the way in which the space is stored. In AnMol, every level has only one vector space.

3. GRACE supported only substructure queries, while AnMol supports two more kinds of structural queries.

Figure 1 shows the overall AnMol architecture. The input to AnMol is the PDB record. Although the vectorization engine is not PDB specific, currently preprocessors that have been written, all expect PDB records.

The file format of the PDB record is available from the content guide [12]. A number of data items present in the record are interesting from an analysis point of view. However, here we shall concentrate only on the quaternary molecular structure.

Molecular structure is described in PDB records by assigning x, y and z coordinates to atoms. AnMol preprocessors read this input and generates one or more labeled undirected graphs.

These graphs capture various views of the molecular structure. The *vectorization engine* hashes each of these graphs into one or more vectors. The vectors generated for each graph are stored in a *structure space* which is a hierarchy of vector spaces for graphs of a particular type.

Analytical queries are handled based on the data in these vector spaces. The PDB records are not consulted once they have been vectorized, except for visualization.

Queries are handled both in a textual form and a visual form using a visual browser. With the visual browser, vector spaces can be browsed by selecting any three dimensions from the set of all dimensions in them. For visualizing individual molecules, AnMol resorts to freely available visualization tools like Protein Explorer[1] that reads data directly from the PDB file.

[1] http://www.umass.edu/microbio/rasmol/

3.2 Preprocessing

Preprocessing involves converting a PDB record into a labeled undirected graph. Each graph so generated depicts a particular view of the molecular structure. Preprocessors for generating the following kinds of graphs have been built.

Covalent bond graph: This preprocessor takes a PDB record and generates a graph where nodes are atoms in the molecule and edges are covalent bonds among atoms. Nodes are labeled by their atomic symbol and edges are labeled either `b` for "backbone" or `o` for "other". The backbone represents the protein "main chain" which connects the different amino acids that make up the protein. Protein primary structure is the sequence of amino acids in the order that they appear on the backbone.

Proximity graph: This preprocessor takes each atom in the PDB record as a node and connects it to all other atoms that lie within a radius of some preset parameter r. It then removes all edges that could denote covalent bonds. Such a graph can capture some non-bonded interactions that contribute to the 3D structure of the molecule. Identifying non-bonded interactions is a major area of research among molecular biologists, and as such the proximity graph is only a crude mechanism of identifying such interactions. Just because two atoms are spatially near, does not necessarily imply the existence of a non-bonded interaction between them. Nevertheless, the proximity graph captures useful information about the 3D structure of the molecule.

Bond-angle graph: The bond-angle graph represents each covalent bond as graph nodes and the angle between adjacent bonds as edges. The node labels are concatenations of atom symbols on either end of the bond they represent. For instance, a bond connecting a nitrogen atom and a carbon atom is labeled `(N.C)`. Edge labels are the angles between adjacent bonds. In order to keep the number of labels small, angles are divided into a small number of regions. For instance, instead of storing the exact angle between bonds, the preprocessor can be configured to just label the quadrant in which the angle falls in.

Torsion-angle graph: Torsion angle is the angle between two adjacent planes formed by four adjacent atoms. Figure 2 schematically depicts a torsion angle. Torsion angle is usually computed only for atoms on the backbone in protein molecules. The torsion angle graph hence is a sequence comprising of a series of planes where adjacent plane labels are separated by a torsion angle label. Torsion angles are usually measured across amino

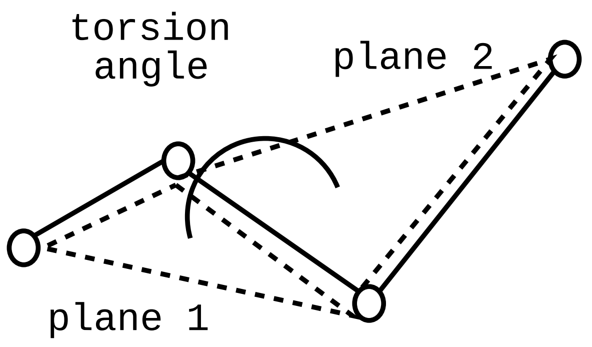


Figure 2: Schematic description of torsion angle

acids rather than any three atoms. In the torsion angle graph, node labels are amino acid symbols and edge labels are regions that classify the torsion angles.

3.3 Vectorization

After the PDB record is preprocessed, one or more labeled undirected graphs are obtained. The vectorization step hashes these graphs onto one or more vectors each.

We use the dotty syntax used in Graphviz [3] for representing undirected graphs. Each kind of graph received from the preprocessing phase represents a different structural characterization of the molecule. They are all vectorized separately and placed in different structure spaces.

Let $G = (V, E, V_L, E_L)$ be an output graph from the vectorization phase. Here V is the set of nodes in the graph, E is the set of edges, V_L is a set of node labels and E_L is a set of edge labels. Any given node $v \in V$ has a node label $v_l \in V_L$ associated with it. Similarly, any edge $e \in E$ has an edge label $e_l \in E_L$ associated with it.

For vectorization, it is very much desirable that $|V_L| \ll |V|$ and $|E_L| \ll |E|$. The preprocessing phase ensures that this property is satisfied. Some kinds of graphs like the covalent-bond graph, are easily amenable to the above property. In such a graph, $|E_l| = 2$ (either `b` or `o`), representing a bond on the protein "backbone" or "other". Node labels are the atom symbols. Some of the frequently occurring node labels are: `N` for nitrogen, `CA` for C^α atoms that denote an amino acid on the backbone, `CP` for the carboxyl carbon that occurs immediately after C^α on the backbone, and `C` for other carbon atoms. The number of node labels is usually between 5 and 10. The number of nodes and edges are usually of the order of a few hundreds to thousands.

Vectorization comprises of two operations: *parsing* and *compression*.

The parsing function takes a graph and creates a vector out of the graph. Parsing takes place at different *levels*. Initially, parsing starts at level 0. After a vector is created at level 0, the input graph is *compressed* to result in zero or more graphs at level 1. Each of these graphs are then parsed and compressed. The process can proceed until such a level where it is no longer possible to compress graphs any further. In reality through, a configurable parameter called `maxlevels` is set that determines the maximum number of levels in the database.

Level 0 is consulted for generating an initial response to a user's query. All levels above level 0 are used for *refinement* queries that are used to make query results more accurate. `maxlevels` is hence a tradeoff between the time saved during addition and the number of refinements possible.

A vector space is defined as $H = \langle D, \mathcal{V} \rangle$ where, D is a set of "dimensions" and $\mathcal{V}$ is a set of "vectors". Any vector $v \in \mathcal{V}$ is defined as an associative array comprising of elements of the form $(d \Rightarrow p)$, where $d \in D$ is a dimension name and p is the "projection" of v on dimension d. Let the term $dims(H)$ denote the set of all dimensions in H, $(D = dims(H))$; the term $vec(H)$ denote the set of all vectors in vector space H, $(\mathcal{V} = vec(H))$; and the term $proj(G, H, d)$ denote the projection of vector G on dimension d in vector space H. In an analogous way, we shall use the term $dims(G)$, where G is a graph, to mean the set of all dimensions found when G is parsed, $vec(G)$ to be an associative array of the form $(d \Rightarrow p)$ showing the number of occurrences of each dimension d in G, and $proj(G, d)$ to mean the projection of G on dimension d in $dims(G)$.

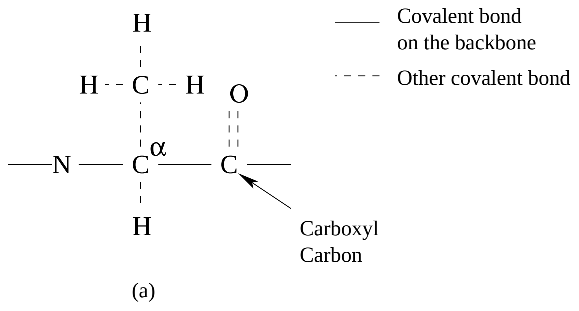
The algorithm for parsing a graph G at any given level l is as follows:

Algorithm Parse (Graph G, Level l)

1. Scan the graph G, and identify all substructures of the form $v_a v_b e$, where v_a and v_b are node labels and e is an edge label. The term $v_a v_b e$ identifies a substructure made of two nodes having labels v_a and v_b respectively, being connected by an edge having label e. We shall use the term $v_a \overset{e}{\leftrightarrow} v_b$ to denote such a substructure. Let $dims(G)$ be the set of all such labels thus found.
2. For any dimension $v_a v_b e$, let $proj(G, v_a v_b e)$ be the number of such labeled substructures found.
3. Let S_l be the vector space in the DBMS for level l. If G is the first graph to be inserted at level l, then $S_l = (\{\}, \{\})$. Insert G into S_l as follows.
    (a) Merge $dims(G)$ to $dims(S_l)$.
    $dims(S_l) = dims(S_l) \cup dims(G)$
    (b) Add G to the set of vectors in S_l.
    $vec(S_l) = vec(S_l) \cup vec(G)$.
    (Note that for any $d \in dims(S_l)$ and $d \notin dims(G)$, $proj(G, d)$ is assumed to be 0).

endAlgorithm Parse.

As is evident from the algorithm, parsing creates vectors that index different "minimal" substructures of the graph. The substructures describe two nodes



N:CA:b	=>	1
CA:CP:b	=>	1
CP:O:o	=>	1
CA:H:o	=>	1
CA:C:o	=>	1
C:H:o	=>	3

(b)

Figure 3: A graph fragment and its vector

connected by an edge. An edge is the simplest structural feature of a graph.

It can be easily shown that the dimensions created from the above algorithm are linearly independent of one another. No dimension can be expressed in terms of other dimensions.

Lemma: No two dimensions created by algorithm Parse, can be expressed in terms of one another.

Proof: First, note that every dimension at a given level are of the same *size*. They are made of exactly two node labels and one edge label. This precludes the possibility of any dimension being contained within another.

Second, dimensions of a vector space are created by a union operation (step 3a), out of the dimensions found in the current graph being parsed. The graph is undirected, therefore $v_a v_b e$ is considered identical to $v_b v_a e$ during union and are not repeated. This precludes the possibility two dimensions being identical.

Hence, no dimension can be expressed in terms of other dimensions at the same level.

Figure 3 shows an example of vectorization. 3(a) shows a graph fragment comprising of labeled nodes. Edges are labeled either b or o. The corresponding vector created by the above algorithm is shown in 3(b).

If $|V_L| = v$ and $|E_L| = p$, then there can be a maximum of $p \cdot v^2$ dimensions. Since v and p are usually small numbers, and since not all combinations of node and edge labels occur, the average number of dimensions is much smaller. In the implementation, after addition of 457 proteins, and performing a proximity graph preprocessing, the number of dimensions at different levels from 0 to 6 were as follows: 12, 39, 122, 195, 288, 281, 6.

Parsing requires examining each edge in a graph. Parsing hence has a complexity of $O(|E|)$.

But as is evident from the algorithm, the parsing function indexes very simple substructures: namely two nodes connected by an edge. Such an algorithm loses interconnectivity information across edges. This information is obtained when the graph is *compressed* to yield a graph at a higher level of abstraction.

The compression process is as follows: the procedure progressively replaces every occurrence of a substructure of the form $v_a \overset{e}{\leftrightarrow} v_b$ with a single node labeled $v_a v_b e$. Edges incident on either the node labeled v_a or the node labeled v_b are now made incident on the new node $v_a v_b e$. The process proceeds until it is no longer possible to replace any substructure with a node. Once this is done, any uncompressed nodes from the previous level and its incident edges are discarded from the compressed graph.

The structure of the resulting compressed graph is dependent upon the order in which substructures are replaced by nodes. If a graph resulted in n dimensions at a particular level, the number of ways in which it can be compressed is greater than even $O(n!)$.

In AnMol not all permutations of dimensions are considered for compression. Different compressed graphs usually have a significant overlap in structural features.

In order to minimize the number of compressions and overlap, a heuristic is adopted that makes only $|dims(G)|$ passes on the graph. Initially, dimensions of the graph G are ordered in decreasing order of the projection of G onto them. Using this decreasing sequence, G is compressed. Once a compressed graph is obtained, the dimension sequence is rotated such that the first dimension goes to the end. G is then compressed again using this new sequence. This process is repeated $|dims(G)|$ times to get $m \leq |dims(G)|$ compressed graphs.

The algorithm for compression is given as follows:

```
**Algorithm** Compress (Graph G, Level l)
1. $dimarray \leftarrow$ sort $dims(G)$ in decreasing order based on $proj$
2. **for** i $\leftarrow$ 0 to $|dims(G)| - 1$
do
    1.1 j $\leftarrow$ i; $G' \leftarrow G$
    1.2 repeat
    1.3 Compress $G'$ based on $dimarray[j]$ by replacing every occurrence of substructure described by $dimarray[j]$ with a node having label $dimarray[j]$
    1.5 Set the level of each new node created as $l + 1$
    1.6 j $\leftarrow$ (j+1) **mod** $|dims(H)|$
    1.7 until j == i
    1.8 Remove all nodes whose level is $l$ including the edges connected to them
    1.9 If $G'$ contains at least one edge connecting nodes at level $l + 1$ then add $G'$ to the list of graphs at level $l + 1$, else discard $G'$
**endfor**
**endalgorithm**
```

From the above algorithm it is evident that the com-

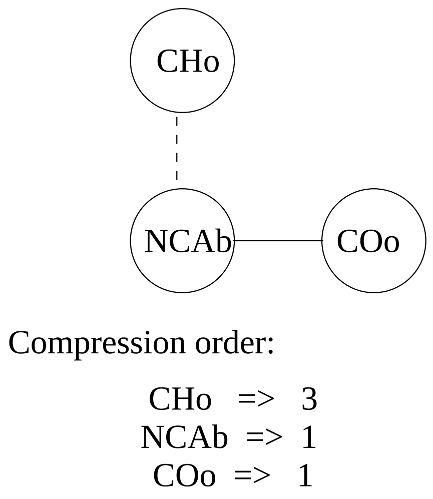


Figure 4: A compressed graph fragment based on Figure 3

pression process may create more than one compressed graphs at the next level. These different graphs are called different "views" of the original graph. All these views are given different subscripts and added into the vector space at the higher level. Sometimes a view may result in a graph with no edges. Such a graph cannot be vectorized further and is ignored. Figure 4 depicts a compressed graph formed from the fragment of Figure 3. The figure also shows the sequence of dimensions used for obtaining this compressed graph.

For any graph G, the number of views that can be generated is $O(|dims(G)|)$. The number of views is sensitive to V_L and V_E. In order to prevent too many views at a single level, the following rules are adopted:

1. For any generated view G_1, if there exists another generated view G_2 at the same level such that $dims(G_1) \subseteq dims(G_2)$, and $\forall d \in dims(G_1)$, $proj(G_1, d) \leq proj(G_2, d)$, then G_1 is discarded.
2. A configurable parameter called `maxviews` limits the number of views in a single level.

Observe that in level 0, each graph will have exactly one view. Also, for any graph G if there exists a vector for G at level l, there will be at least one vector for G in all levels below l.

The compression process effectively reduces the size of the input graph by half. Hence if the initial graph had $|E|$ edges, the number of levels created would be $O(log(|E|))$. In practice, `maxlevels` limits the number of levels created.

3.4 Handling queries

AnMol supports three kinds of queries: *nearness*, *substructure* and *mutation* queries. Queries from many analytical tasks usually require a combination of the above query types. Definitions of each of the query types is given as follows:

Nearness: In this query, a molecule is given as input, and query returns molecules that are close in structure to the input molecule. Nearness is defined based on the vector distance between the query vector q and the candidate graph vector v. For any vector space H, vector distance is given by: $\sqrt{\Sigma_{d \in dims(H)}(proj(q, H, d) - proj(v, H, d))^2}$.

Substructure: In this query, a structural fragment is given and the query returns molecular structures that most likely contain the given fragment. If q is the vector of the query fragment, then q is *probably* a substructure of candidate vector v, if for every vector space H, $\forall d \in dims(H), proj(v, H, d) \geq proj(q, H, d)$.

Mutations: Given a molecule as input, mutation queries return molecular structures that share a significant percentage of structural features with the input molecule. Mutation queries are similar to nearness queries, except that nearness does not consider molecular structures that do not have *all* structural features of the input molecule. Calculation of mutation queries are explained further down in this section.

Queries are handled in an interactive fashion. The first time a query is given, only level 0 is searched. Subsequently, the user or application may choose to refine query results, in which higher levels are searched.

Let G be a graph that is obtained after preprocessing the query structure. In order to answer the query, G is first parsed to create a vector at level 0. Query results now depend on searching the neighborhood of G's query vector. To search neighborhoods, a configurable parameter called "window length" (denoted by w) is used. For nearness and mutation queries $w/2$ is the radius around the vector of G which is searched. For substructure queries, the vector of G is the starting point from which a hypercube of length w on each side is searched for query results.

Let the term $w(d, H, G)$ denote the set of all points lying within the region characterized by w along dimension d in vector space H, given graph G as the query. Depending on the type of query, G is either at the center of the region or at the beginning.

For nearness and substructure queries in any vector space H, query results would be points that satisfy $w(d, H, G)$ for all $d \in dims(H)$. Hence the query result set is calculated as: $\bigcap_{d \in dims(H)} w(d, H, G)$. A ranking procedure then assigns ranks to each vector in the query result. Ranking algorithms are discussed further below.

For mutation queries, it is necessary to consider points that lie in the vicinity of G, but they may not share all features in common. As long as graphs share a significant number of dimensions in common with G we consider them for the result. In the *weak nearness* form of the mutation query, a union of results from all dimensions are taken. Hence, query result is given by: $\bigcup_{d \in dims(H)} w(d, H, G)$. The results are then ranked based on the number of dimensions in which they lie inside $w(d, H, G)$.

In the *strict mutation* form of the mutation query, query results are calculated as: $\bigcup_{d \in dims(H)} w(d, H, G) - \bigcap_{d \in dims(H)} w(d, H, G)$. Strict mutation means that the candidate is **not** near in *all*, but is near in *at least* one dimension. Candidates that are near in all dimensions are returned as part of nearness queries, which are explicitly removed from strict mutation.

Ranking query results: Ranking query results has two aspects: (a). ranking the query and (b). ranking the refinements. Also, the ranking algorithm varies depending on the type of the query.

For nearness queries, ranks are assigned at level 0, by sorting query results in increasing order of their distances with the query graph. During refinements, the distance metric for each query result is set as the minimum of the current distance measure and the distance obtained after refinement. Query results are then sorted again based on the new distance measures and ranks assigned accordingly.

For substructure queries ranking proceeds as follows. Given a query Q, at level 0 all its query results are assigned $rank = 0$. After each compression of the query graph Q, a set of query results are returned. For every graph G that has been returned after refinement, if G exists in the original set of query results, then the rank of G is increased by the level number after refinement.

Hence, if a graph matches at a higher level of abstraction, it gets a greater rank than another graph which matches at a lower level of abstraction. Similarly, the greater the number of views of a graph that match the query, the greater is its rank.

Note that in nearness queries, results are ordered in ascending order of their ranks while in substructure queries, results are ordered in descending order of their ranks.

For mutation queries, at level 0 the results are ordered based on the amount of dimensional overlap they have with the query. Ranks are assigned based on this list. If two or more graphs have the same amount of dimensional overlap, the tie is broken using the distance metric. After every refinement, the list of dimensional overlap is adjusted by setting it to the maximum of the current overlap and the overlap obtained after refinement. The query set is ordered again based on the new overlap measures and ranked.

In mutation queries, results are again ordered in increasing order of their ranks.

3.5 Query APIs

Query APIs are supported by AnMol that enables applications to embed AnMol queries into programs. As of now, only searches are supported this way; addition of new molecules has to be done manually.

The three types of queries namely nearness, substructure and mutation are available as functions named `near()`, `contains()` and `mutation()` respectively. These functions are in the form of methods of a Java class called `AnMol`.

There are two forms of the above functions: (a). a form that acts on the entire structure space and (b). a form that acts on a particular scope; typically the results of another query.

Hence, the function `near(G)` returns molecules that are near to G from the vector space at level 0. A function of the form `near(G',near(G))` returns a set of molecules that are near G' among the set of points that are near G.

Similar to `near()` the other two functions `contains()` and `mutation()` also has two analogous forms. The search scope specified in the second parameter can be any of the searches. Hence, the function `contains(G',near(G,mutation(P)))` searches for mutations of the query molecule P, that are structurally similar to G and contain the fragment G'.

Refinement is supported by the `refine()` function. This function takes the result of a query or another refinement as parameter, and performs the next refinement over it. For example `refine(near(G))` refines the results of the `near()` function once. The term `near(G,refine(refine(contains(H))))` searches for molecules containing H, refines the results twice and then searches among these results for molecules that are near G.

The above functions are part of a class called `AnMol`. Hence their invocation would be more like `A.near()`, `A.contains()`, etc., where `A` is an object of type `AnMol`.

The query molecule structure that is provided is either in the form of a file name pointing to a file in the dotty format [3]; or to a character buffer describing the structure in dotty format.

Query results would be returned in an object of a class called `QueryResults`. This object maintains the state of the query like the query type and the number of refinements completed. The return type of all the four functions defined above is `QueryResults`.

The `AnMol` class also contains other methods like `setStructureSpace()` that identifies a structure space for answering queries; `setWindowLength()` that sets the window length parameter; and `rank()` that takes a `QueryResults` object as parameter and assigns a rank for each query result. A window length of ∞ can be set by giving a negative parameter to `setWindowLength()`. Note that, while a window length of ∞ is useful for substructure queries, it would be of no use to mutation or nearness searches. If window length is set to ∞, then nearness and mutation searches return all vectors in the structure space.

For a query that requires searching within the results of another query, the process proceeds by first searching the structure space and obtaining results. An intersection is then computed between these results and those specified in the scope. Such a strategy is not costly because the biggest task in answering queries is in parsing and compressing the query graph. This has to be done anyway regardless of whether the structure space is searched or the former query results are searched.

4 Schema Design and Implementation

Databanks of biomolecular structures are usually read-only. Addition to the databanks are rare in comparison to queries. And modification of an existing structure almost never happens.

Exploiting the above, the schema of AnMol uses a number of materialized views to improve query performance. The schema of AnMol has various components. These are as follows:

- The *layered star-schema* which models the structure space for answering spatial queries
- The *vector tables* which hold vectors for each added molecule
- The *distance tables* which hold distance information between pairs of molecules in each vector space

Layered Star Schema: The structure space in AnMol is a *hierarchy* of vector spaces where member graphs are represented at different levels.

Also, all three queries on the structure space that were introduced require searching for points within ranges. In order to facilitate such queries, a variant of the star schema is designed called the *layered* star schema.[2]

Figure 5 schematically shows a layered star schema. At the center of a schema is a table called the "well" of the schema. This is analogous to the fact table of a star schema; but has certain differences. The well maintains facts about member graphs. Each member graph is given a numerical id, which acts as the primary key into the well.

Other fields like graph name, the file name of its PDB record, data about the molecule, etc. are stored in the other columns of the well. Unlike the fact table of a star schema, the well does not hold any dimension information.

In addition to the well, the layered star schema consists of a set of dimension tables. Each dimension table has a layer associated with it. The first column in a

[2]A simple table design like $(graph, level, dimension, projection)$ was discarded since it requires very complex nested SQL statements for performing range searches and needs a number of passes on the table. Handling mutation queries on such a table is especially tricky.

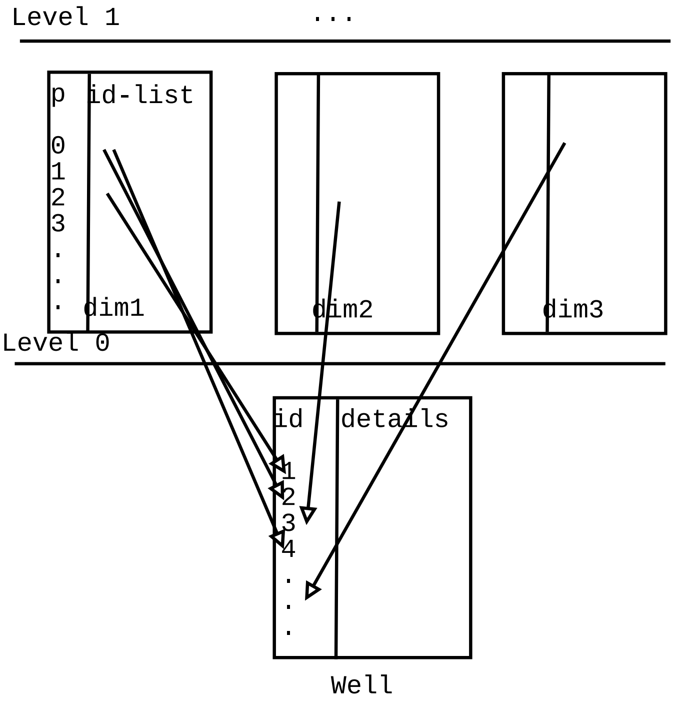


Figure 5: Schematic diagram of the layered star schema

dimension table is a projection value. This is used as the primary key. Each projection value present in a dimension table points to a set of one or more graph ids that are ordered incrementally. For example, in a dimension table named D at level l, if a row with key value = 5 points to the sequence $8, 12, 13$, it means that graphs having ids $8, 12$ and 13 have a projection of 5 on dimension D in level l. The list of ids stored for a projection points to their respective records in the well.

Such a set up enables efficient execution plans for substructure, nearness and mutation searches. All these queries involve extraction of a set of ids from each dimension based on a window. This can be quickly performed since dimension table records are ordered according to projections. The set of ids for a given projection are stored in increasing order. This enables a fast computation of intersection of query results across all dimensions.

The layered star schema also allows new dimensions to be added into the schema whenever they are encountered while parsing graphs.

The Vector Table: A *vector table* is available for each level in a structure space. The vector table holds vectors for individual graphs. They can be searched efficiently given a graph id to obtain its corresponding vector. They are useful when a query of the form "Show all graphs near Protein-B" is encountered. Here "Protein-B" is a graph that is already in the database.

It would be very inefficient to parse and compress the query graph when its vector is already present. However, using just the layered star schema, the vector for "Protein-B" cannot be efficiently retrieved. For such situations, the vector table is consulted.

The vector table at any given level, consists of three columns $(id, dim, proj)$, where id is a graph id, dim is a dimension name, and $proj$ is the projection of id onto dim. The table is indexed with respect to id enabling fast retrieval of the vector for a given graph.

The Distance Table: A distance table is also a materialized view of a vector space. A distance table is associated with each level, where all-pairs vector distances are maintained. This is used for ranking query results according to distance. The distance table comprises of three columns $(id_A, id_B, dist_{AB})$, where id_A and id_B are graph ids, and $dist_{AB}$ is the distance between the graphs. Rows are inserted such that id_A is always smaller than id_B, which helps in eliminating redundant records (since $dist(A, B) = dist(B, A)$) and establishes a policy for formulating distance queries (the graph with the smaller id in a nearness search always has to be searched in the first column).

5 Performance Evaluation

Performance evaluation of an AnMol implementation was performed for the following criteria:

Load time: Here, loading times (involving parsing, compression and addition into database) for protein molecules were measured for loading their covalent-bond graphs. The objective is to compare graph size and its loading time. Covalent-bond graphs were chosen because they were usually the largest graphs in terms of the number of edges generated by any of the preprocessors.

Figure 6 shows loading time as a function of the number of edges. The measurements are shown as points which are fitted with a smooth curve that shows a growth rate of about $10x$ with respect to the number of edges.

Query Time: Query response times varied depending on whether the input query graph was already present in the database or it needed to be parsed and compressed. The major time-consuming operations are parsing and compression. Times taken for searching the database and computing intersections were much smaller. Query response time did not vary significantly with respect to the type of queries, since all queries require parsing and compression.

If the query graph was already present in the database, query response time was typically between 200 ms for small queries to 2 seconds for large queries. When the query graph was not present in the database, each compression and parsing operation *for a given level* needed anywhere between 300 ms to 15 seconds. The complete response time for a user interaction ranged from 500 ms to 17 seconds.

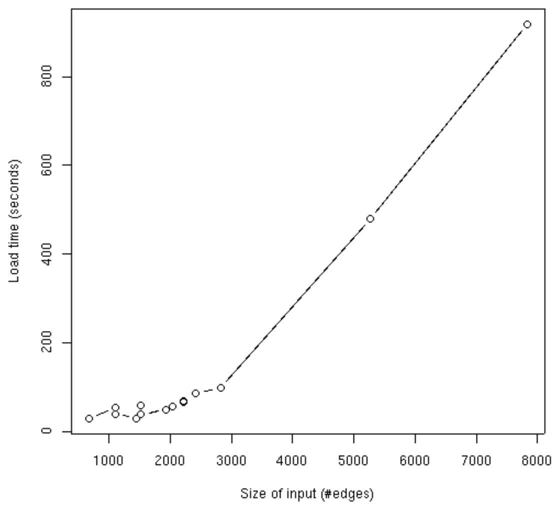


Figure 6: Loading time as a function of the number of edges

Comparison of Nearness with RMSD: In order to benchmark effectiveness of the nearness search, a comparison was made against RMSDs.

It should be noted that the concept of distance in RMSD and in AnMol are very different. RMSD computation involves establishing a correspondence between two molecules and computing their displacements with respect to one another. RMSD computation depends to a large extent on the establishment of correspondence. When the FSSP database was consulted, RMSD values for several pairs were missing. For example, no RMSD value was available between proteins $1a7j$ and $1a8y$. But, AnMol returns some distance value between any two graphs. This distance computation is based on the respective vectors that have been created for the graphs. The projection of any dimension that is absent in a graph is taken as 0.

Correspondence between the metrics is determined by computing RMSD and AnMol distances among a set of randomly selected pairs and sorting the pairs according to increasing distance measures. The correspondence measure is then the length of the largest alignment of pairs between the two lists. For example, consider four randomly selected molecules A, B, C and D. Let the list AB, BC, CD, AD, AC, BC depict a sequence showing increasing values of RMSDs. Suppose, after ordering the above based on AnMol distances, we get the sequence AB, BC, AD, AC, BC, CD. In this, except for CD, all other pairs are in the correct relative positions (i.e. $AB \geq BC \geq AD \ldots$). Hence the correspondence is $\frac{5}{6} = 0.826$.

We found that correspondence between RMSD and AnMol distances depends on the type of preprocessing performed. For instance, AnMol distances between covalent-bond graphs had very little correspondence

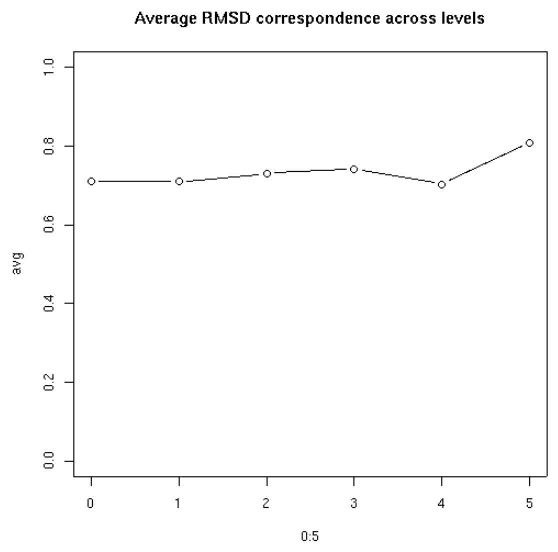


Figure 7: Correspondence between RMSD distances and AnMol distances

with RMSD.

For comparison with RMSD, proximity graphs were used. In the proximity graph preprocessor, only backbone atoms (N, CA and CP) were chosen. This is because, RMSD values available from FSSP were computed by establishing correspondence on only the backbone atoms.

Figure 7 shows the average correspondence obtained between AnMol distances and RMSD across different levels. The averages were computed over 26 experiments, where each experiment involved aligning sequences of up to 28 pairs.

It should be noted that, the proximity graph has lesser information than what is needed for RMSD calculation. It depicts distance between two atoms, but does not depict their relative orientation. This is a factor that affects RMSD values. AnMol is not proposed to be a replacement for RMSDs for calculating precise structural differences. As mentioned in the design goals, fast clustering of similar molecules is more desirable than computing precise structural differences. Benchmarking was performed against RMSD mainly due to the lack of any other suitable benchmark.

Performance of substructure queries: We maintained a window length of ∞ for substructure queries. This gave a recall value of 1 for all queries that were tested. Precision values varied. It depended on the size of the query graph and the number of refinements.

In order to evaluate substructure matches, a sequence of amino acids was chosen randomly from some protein and its corresponding structure was given as input. Substructure results were then verified by performing a corresponding sequence search on the primary structures of proteins that were returned by the query.

Size of query (edges)	Precision
11426	1
7834	1
6805	0.957
6789	1
5294	1
2544	0.895
2263	0.878
2249	0.906
1099	0.607
122	0.744
111	0.865

Table 1: Precision values for some substructure queries

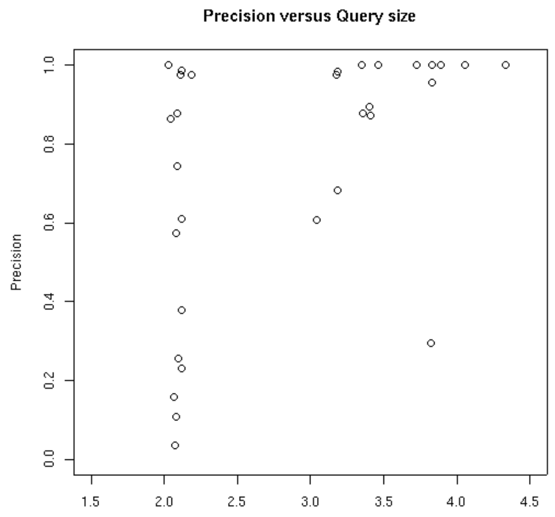


Figure 8: Scatter diagram depicting correlation between query size and precision obtained

If the query was small, either all proteins contained the structure or the query graph could not be compressed beyond a few levels. If the query structure was fairly large, then only one protein contained the query structure. This was the protein from which the query structure was extracted. In the latter case, the query graph would usually be big enough to be compressed to a level where accurate results could be obtained. As a rule therefore, performance of substructure queries seemed to be best when the query graphs were large.

Table 5 shows precision metrics obtained for some query graphs of different sizes. MS Excel returned a correlation coefficient of 0.459 between size of the query graph and the precision obtained. Figure 8 shows a scatter diagram for the above correlation.

6 Conclusions

AnMol is a commercial implementation and is targeted towards small bio-technology firms.

The design goals of AnMol are to provide a fast and reliable platform for structural analysis of biomolecules which can be used even over low-end computational infrastructure. This is a pertinent need since most analytical tools for structural analysis require large computational infrastructure. Some of these tools can be utilized over the internet, hence obviating the need for high-end computing at the user's end. However, generic tools available over the internet may not suffice for the specific requirements of the user. AnMol is meant to fill this need and provide a platform for tools that end users can install locally and use, without investing in large computational infrastructure.

Currently AnMol is at an alpha stage of implementation with a beta version expected to be ready in the coming months. Performance evaluation is carried out by biologists who are on the AnMol team. Performance results for the tests carried out so far have been rated good to very good from an end user's point of view.

7 Acknowledgments

The authors would like to thank members of the AnMol team: Sudip Midya, Pradeep Patil, L. Balasundaraman, Anitha Y., Sreenu Babu and Vipul Srivatsava. The authors are also deeply grateful to the anonymous reviewers for their valuable comments that helped in improving the paper.

References

[1] CATH: Protein Structure Classification. *http://www.biochem.ucl.ac.uk/ bsm/cath/*

[2] D.J. Cook, L.B. Holder, S. Su, R. Maglothin, and I. Jonyer. *Structural Mining of Molecular Biology Data.* IEEE Engineering in Medicine and Biology special issue on Advances in Genomics, Vol. 20, No. 4, pages 67-74, 2001.

[3] E.R. Gansner, S.C. North. An Open Graph Visualization System and its Applications to Software Engineering. *http://www.research.att.com/sw/tools/ graphviz/GN99.pdf*

[4] R. Guigno, D. Shasha. GraphGrep: A Fast and Universal Method for Querying Graphs. *Proceedings of the 16th International Conference in Pattern recognition (ICPR)*, Quebec, Canada, August 2002.

[5] L. Hammel, J. Patel. Searching on the Secondary Structure of Protein Sequences. *Proc. of VLDB 2002*, Hong Kong, China, 2002.

[6] L. Holm and C. Sander. Fold classification based on Structure-Structure alignment of Proteins. *http://www2.ebi.ac.uk/dali/fssp/*

[7] L. Holm, C. Sander. *Protein Structure Comparison By Alignment of Distance Matrices.* Journal of Molecular Biology, Vol. 233, pages 123-138. 1993.

[8] I. Jonyer, D.J. Cook, and L.B. Holder. *Discovery and Evaluation of Graph-Based Hierarchical Conceptual Clusters.* Journal of Machine Learning Research, Vol. 2, pages 19-43, 2001.

[9] S. Kumar, S. Srinivasa. A Database System for Storage and Fast Retrieval of Structure Data: A Demonstration. *to appear in Proc. of ICDE 2003*, IEEE Computer Society Press, Bangalore, India, Mar 2003.

[10] Y. Lamdan, H.J. Wolfson. Geometric Hashing: A General and Efficient Model-Based Recognition Scheme. *Proc. of the IEEE Int'l Conf. on Computer Vision*, pages 238-249, 1988.

[11] N. Leibowitz, Z.Y. Fligelman, R. Nussinov, H.J. Wolfson. Multiple Structural Alignment and Core Detection by Geometric Hashing. *Proc. of the 7'th International Conference on Intelligent Systems in Molecular Biology*, Heidelberg, Germany, pages 169-177, 1999.

[12] PDB File Format Contents Guide Version 2.2. *http://www.rcsb.org/pdb/docs/format/ pdbguide2.2/guide2.2_frame.html*

[13] PDB: The Protein Data Bank. *http://www.pdb.org/*

[14] SCOP: Structural Classification of Proteins. *http://scop.mrc-lmb.cam.ac.uk/scop/*

[15] S. Srinivasa, S. Acharya, H. Agrawal, R. Khare. Vectorization of Structure to Index Graph Databases. *Proc. of IASTED Int'l Conf. on Information Systems and Databases (ISDB'02)*, Acta Press, Tokyo, Japan, Sep 2002.

[16] J.T.L. Wang, Q. Ma, D. Shasha and C.H. Wu. *New Techniques for Extracting Features from Protein Sequences.* IBM Systems Journal, Special Issue on Deep Computing for the Life Sciences, Vol. 40, No. 2, pages 426-441, 2001.

[17] 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 Trans. on Knowledge and Data Engineering, Vol. 14, No. 4, pages 731-749, 2002.