GString: A Novel Approach for Efficient Search in Graph Databases

Haoliang Jiang[†] Haixun Wang[‡] Philip S. Yu[‡] Shuigeng Zhou[†]

[†]Dept. of Computer Sci. & Eng.
Fudan University, Shanghai 200433, China
{hljiang, sgzhou}@fudan.edu.cn

[‡]IBM T. J. Watson Research Center
19 Skyline Dr., Hawthorne, NY 10532, USA
{haixun, psyu}@us.ibm.com

Abstract

Graphs are widely used for modeling complicated data, including chemical compounds, protein interactions, XML documents, and multimedia. Information retrieval against such data can be formulated as a graph search problem, and finding an efficient solution to the problem is essential for many applications. A popular approach is to represent both graphs and queries on graphs by sequences, thus converting graph search to subsequence matching. State-of-the-art sequencing methods work at the finest granularity – each node (or edge) in the graph will appear as an element in the resulting sequence. Clearly, such methods are not semantic conscious, and the resulting sequences are not only bulky but also prone to complexities arising from graph isomorphism and other problems in searching. In this paper, we introduce a novel sequencing method to capture the semantics of the underlying graph data. We find meaningful components in graph structures and use them as the most basic units in sequencing. It not only reduces the size of resulting sequences, but also enables semantic-based searching. In this paper, we base our approach on chemical compound databases, although it can be applied to searching other complicated graphs, such as protein structures. Experiments demonstrate that our approach outperforms state-of-the-art graph search methods.

1. Introduction

Graphs are used for modeling complex data structures in increasingly many applications, including multimedia [9], bioinformatics [5], etc. A fundamental operation on graph data is graph search, which can be described as follows: *given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a graph query q, find all $g_i \in D$ such that q is a subgraph of g_i*. It is time-consuming to scan the whole graph database D and check whether q is a subgraph of any graph $g_i \in D$. The complexity is partially due to subgraph isomorphism, a decision problem known to be NP-complete. Furthermore, in real life applications such as searching for chemical compounds, subgraph isomorphism checking must tolerate some disparities, for example, minor differences in nodes, edges, and structures.

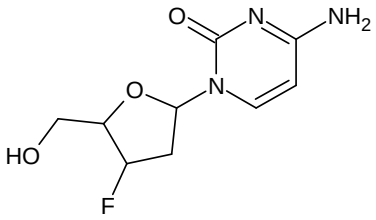
To fight the complexity, a variety of graph indices have been introduced. The basic idea of many approaches is to break a graph into paths (e.g. GraphGrep [11]) or fragments (e.g. gIndex [14]), which are utilized as filtering features in graph search. However, these approaches have the following drawbacks:

- Even small graphs may create a combinatorial explosion of paths and fragments, which translate to a large index space.
- Not every path or fragment is a meaningful feature as far as the application is concerned. On the other hand, when we break down a graph, we risk losing information about the global structure, which may be more meaningful to the application, and may take much computation, if possible, to rebuild from the pieces.

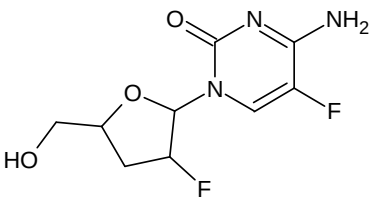
We demonstrate this problem in the domain of searching chemical compound databases. Figure 1(a) and 1(b) show two chemical compounds with highly similar structures. It is desirable that features extracted from the two graphs can capture the similarity.

GraphGrep [11] uses paths as features. To represent the compound in Figure 1(a), it records 96 occurrences of 28 different paths (of length $\leq$ 3), which are shown in Figure 1(d). If not for the limited variety (mostly carbon and hydrogen) of atoms in organic compounds, GraphGrep may introduce an even larger feature space. GIndex [14] tries to reduce features by indexing only *discriminative fragments*. However, the similarity of the two compounds is poorly captured by the enumerations of paths and fragments. In order to reveal the similarity, a lot of "join" operations among many features have to be conducted. This creates another dimension of combinatorial explosion.

On the other hand, in organic chemistry, compounds are named concisely using IUPAC nomenclature. For example, as shown in Figure 1(c), the two compounds are



(a)



(b)

Compound	IUPAC nomenclature
(a)	2',3'-dideoxy-3'-fluorocytidine
(b)	Cytidine,2',3'-dideoxy-2',5-difluoro

(c)

Path	Quantity	Path	Quantity
C	9	O	2
F	1	OH	1
N	3	OH-C	1
C-OH	1	C-O	3
O-C	3	C-N	6
N-C	6	C-F	1
F-C	1	C-C	12
OH-C-C	1	C-C-OH	1
C-C-F	2	F-C-C	2
C-C-N	4	N-C-C	4
C-N-C	6	N-C-N	4
N-C-O	3	O-C-N	3
C-O-C	2	O-C-C	3
C-C-O	3	C-C-C	8

(d)

Figure 1. Two chemical compounds in three different representations

named *2',3'-dideoxy-3'-fluorocytidine* and *Cytidine,2'3'-dideoxy,5-difluoro* respectively. From these concise names, domain experts know exactly the chemical structures they represent. However, for non-domain experts or for computers, such names are independent from the structures they represent, and they certainly reveal no similarity among the structures. Thus, it is impossible to base structural similarity search on names in IUPAC nomenclature, unless all relevant domain knowledge is encoded in a machine readable format and integrated into the search process.

In this paper, we present a novel approach that embeds graph structural semantics into concise presentations of the graphs, so that graph search can be conducted in an efficient and meaningful manner, once such presentations are properly indexed. Our approach is inspired by search in chemical compound databases. The chemical semantics in the graphs give queries and answers a hidden context. For example, if a user queries for *c-c-c-c*, unless explicitly requested by the user, it does not make much sense to return benzene, which consists of a cycle of 5 carbon atoms, because the two structures have very different chemical characteristics.

Thus, we focus on basic structures that have semantic meaning in the context of organic chemistry, and use them as indexing features. We consider chemical compounds consist of three basic patterns: *Line*, *Star*, and *Cycle*. Our approach can in fact accommodate any number of basic patterns, but we think that these three basic patterns are sufficient to depict the structures of chemical compounds. Certainly, for graphs in other domains (for example, graph representations of 3-dimensional protein structures), we may resort to totally different basic structures or patterns.

Once the basic structures are decided, we can transform the graph representation of a chemical compound to a string. We perform the same transformation for query graphs. Thus, the graph search problem is converted into a string matching problem. We create efficient indices on strings to support efficient string matching. To the best of our knowledge, our approach is the first work that encodes semantic meaning of complex graphs in a certain domain into strings of a small number of basic structures. Experimental results on real world databases show that our approach outperforms GraphGrep [11] and GIndex [15] in terms of index size, construction time, candidate answer set size, subgraph search accuracy and efficiency, and comparing with C-tree [4], our approach is advantageous in all evaluated performance measures except for index construction time.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 gives the preliminaries of graph search problem. Section 4 focuses on how to construct string representations of graphs. Section 5 introduces the matching, querying and indexing techniques. Experimental Results are presented in Section 6. We conclude and highlight future work in Section 7.

2 Related Work

Several indexing techniques have been developed for querying structured data. Various indexing methods have been developed for XML data, including DataGuide [10], A(k)-index [8], ViST [13, 12], etc. In particular, ViST represents both XML documents and XML queries in structure-encoded sequences, and implements query evaluation by subsequence matching, which avoids expensive joins of intermediary results.

Path-based indexing approaches [3, 11] are proposed to support queries on general graphs. GraphGrep [11] repre-

sents the state-of-the-art technique of *path-based graph indexing*. For each graph, it enumerates all paths up to length l_p in the graph, and records the number of occurrences of each path. Queries are processed in two steps. In the filtering step, it finds a set of candidate graphs which contains paths in the query structure, and requires the counts of such paths are beyond the threshold specified in the query. In the verification step, each candidate graph is examined by subgraph isomorphism to obtain the final results.

However, path-based approaches may not be suitable for complex graph queries due to the following reasons:

1. Paths are too simple: Global structural information is lost when graphs are disassembled into paths, and search based on path matching often leads to huge number of false positives.

2. Paths are too many: A graph may contain an exponenial number of paths. As a result, GraphGrep must put a retriction on the length of the paths it enumerates.

GIndex [15] uses frequent graph patterns instead of paths as index features. Using frequent graph patterns reduces index space and improves the filtering rate. However, GIndex still shares certain disadvantages of GraphGrep. First, it does not support similarity queries. Second, it must first find frequent paths/fragments, which itself is an expensive task in terms of space and time. Third, paths and fragments only carry local structure information of graph. Loss of information may lead to reduction of filtering rate.

C-tree [4] is a new index structure for graph queries. It is based on the concept of *graph closure*. The index structure Closure-tree (C-tree) is similar to tree structures used in spatial access methods, e.g., R-trees [1]. An approximation technique called *pseudo subgraph isomorphism* is developed to avoid costly full subgraph isomorphism tests. C-tree also support *k*NN similarity queries.

3 Preliminaries

Let $G = (V, E)$ denote a graph, where V is a set of vertices and E is a set of edges. A graph database $D = \{G_1, G_2, \ldots, G_m\}$ is a collection of graphs. In this paper, as we use chemical compound database to demonstrate our approach, we focus on undirected graphs where vertexes are atoms and edges represent bonds between atoms.

To perform similar match, we need a measure to quantify the similarirty between two graphs. Bunke and Shearer [2] use the maximum common subgraph to measure structure similarity. Researchers also introduce the concept of graph edit distance and graph alignment distance (corresponding to string edit distance and alignment distance). The matching of two graphs can be regarded as a result of three edit operations: insertion, deletion, and relabeling. According to the substructure similarity search, each of these operations relaxes the query graph by removing or relabeling one edge (insertion does not change the query graph). Without loss of generality, we take the *percentage of maximum retained edges* [15] in the query graph as the similarity measure.

Furthermore, graph query processing is usually carried out in two steps:

1. *Index construction*, which is a preprocessing step. It consists of extracting features in graph databases, designing suitable representations of these features, and organizing/storing these representations efficiently.

2. *Query processing*, which consists of two sub-steps: (1) Search, which compares the features of a query graph q with the features of the graphs in the database to create a *candidate query answer set* C_q; (2) Verification, which checks each graph g in C_q against q to see whether g indeed contains q or a similar version of q.

For graph query processing, the *query response time* is $T_{search} + |C_q| \times T_{isotest}$, where T_{search} is the time spent in the search step, and $T_{isotest}$ is the average time of subgraph isomorphism testing, which is conducted over query q and graphs in C_q. In the verification step, it takes $|C_q| \times T_{isotest}$ to prune false positives in C_q. Usually the verification time dominates the Query Response Time. Since the computational complexity of $T_{isotest}$ is NP-Complete. Approximately, the value of $T_{isotest}$ does not change too much with the difference of query. Thus, the key to reducing query response time is to minimize the size of the candidate answer set, $|C_q|$.

4 GString

In this section, we describe our method of transforming a graph into a string representation, or a GString, that captures the semantics in the underlying graph data.

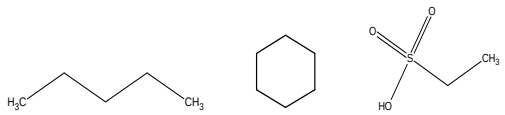
4.1 Basic Structures

Given a set of graphs, we want to find features that can embody the semantics in the graphs. For chemical compounds, we employ three types of basic graph structures: *Line*, *Cycle*, and *Star*.

1. *Line* denotes a structure consisting of a series of vertices connected end to end. Figure 2(a) shows a *Line* structure with 5 vertices.

2. *Cycle* denotes a structure consisting of a series of vertices that form a close loop. Figure 2(b) shows a *Cycle* structure with 6 vertices.

3. *Star* denotes a structure where a *core vertex* directly connects to several vertexes. To qualify as a *Star* structure, the fan out of the core vertex must be above a user-defined threshold c. As shown in Figure 2(c), the *Star* structure has a "S" vertex as the *core vertex* which has a fanout of 4.



(a) Line (b) Cycle (c) Star

Figure 2. Basic Structures

For a graph g, we first extract all *Cycle* structures in g, then we extract all *Star* structures, and finally, we identify the remaining structures as either *Line* structures or as attachments (see Section 4.2) to the extracted basic structures. The algorithms for the extractions are straightforward and are omitted here due to lack of space.

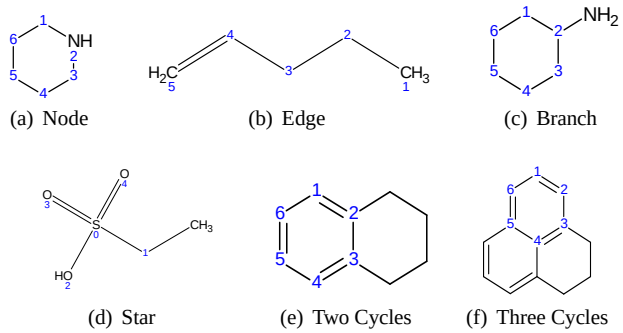
4.2 Annotations on Basic Structures

For chemical compound graphs, we assume each node by default is a *carbon*, and each connection (edge) by default represents a saturated or a *single bound*. We ignore *hydrogen* atoms, as their existence can be derived from the local structure. To change the defaults, we need to "annotate" the graph.

1. `Node annotation.` A non-carbon node is annotated by the name of the atom. For example, in Figure 3(a), a *nitrogen* atom replaces the default *carbon* atom at position 2, we annotate it as $\langle$*node 2 N*$\rangle$.

2. `Edge annotation.` In Figure 3(b), two carbon atoms form a *double bound*. We annotate it as $\langle$*edge 4 5 d*$\rangle$, which means the bond between the atoms at position 4 and 5 is a double bound[1].

3. `Branch annotation.` In Figure 3(c), the default *carbon* at position 2 is replaced by a branch that connects to a nitrogen. We annotate it as $\langle$*branch 2 N*$\rangle$. Here, we define a *branch* as an *edge* that is attached to a basic structure at one end and open at the other end.



(a) Node (b) Edge (c) Branch

(d) Star (e) Two Cycles (f) Three Cycles

Figure 3. Annotation Examples

[1]We use 'd' for double bound, 't' for triple bound.

Note that annotations on basic structures alone cannot express the compounds shown in Figure 3(e) and Figure 3(f). Rather, they are considered as consisting of multiple basic structures (*Cycles*). We discuss their sequence representations in Sec. 4.4.

Also note that in the annotation, each node is associated with a unique number for positioning. We devise a consistent way of numbering so that a basic structure can be reconstructed from its sequence representation. First, we find a *beginning node* in the basic structure. Second, we decide the *direction* of numbering starting from the *beginning node*. In the first phase, if the decision involves more than one candidate nodes, we pick the simplest node, where node simplicity is measured by the number of annotations at the node (we break ties arbitrarily). Specifically, we number nodes in the three types of basic structures as follows:

1. For *Line*, we choose the simpler end node as the first node, and we number the nodes along the line sequentially by $2, 3, \cdots$. Figure 3(b) shows the numbering result of a Line structure.

2. For *Cycle*, we choose the simplest node (say A) as the beginning node. Between two of A's neighboring nodes, we choose the simpler one (say B) as the second node. Then we number the nodes in the Cycle structure along the direction $A \rightarrow B$. Figure 3(a) and Figure 3(c) are numbering results of two Cycle structures.

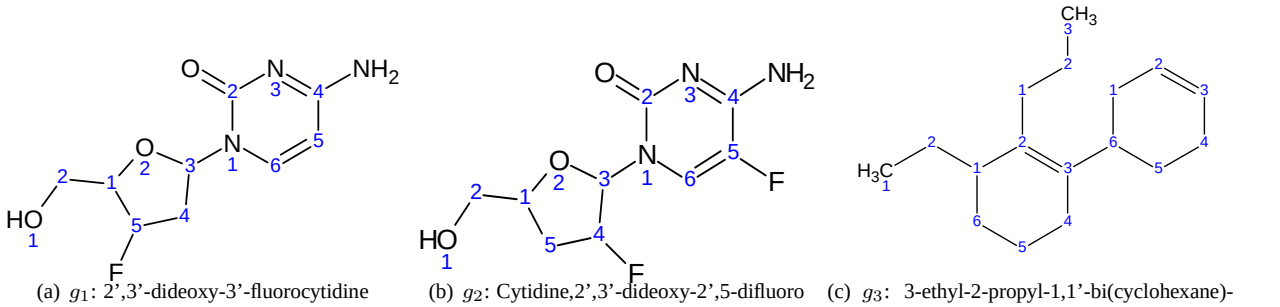
3. For *Star*, the core node is numbered 0. The remaining nodes in the star structure are numbered as in the *Cycle* structure. Figure 3(d) is the numbering result of a Star structure.

Figure 4 shows the numbering of the two chemical compounds in Figure 1.

4.3 GString Syntax

We convert a basic structure to a GString after numbering using the grammar shown in Table 1. Given a basic structure, its GString has 3 components: type, size, and a set of annotations (called edits). For *Line* or *Cycle*, size is the number of vertices in the structure. For *Star*, size indicates the fanout of the central vertex.

For example, benzene is represented by *Cycle 6*, which means benzene is a cycle structure consisting of 6 carbons connected in single bounds. Unlike benzene, chemical compounds shown in Figure 3 require annotations. We append the annotations at the end of the string for the default structure. For example, the compound in Figure 3(a) is represented by string *Cycle 6* $\langle$*node 2 N*$\rangle$, which describes *pyridine* as a cycle structure of 6 carbons where the second carbon is substituted by a nitrogen. The graph in Figure 3(b) is represented by *Line 5* $\langle$*edge 4 5 d*$\rangle$, which describes it as

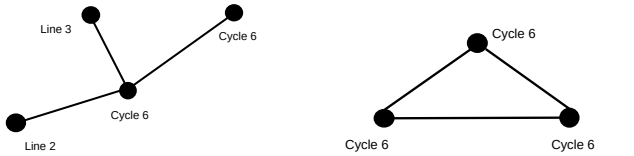


(a) g_1: 2',3'-dideoxy-3'-fluorocytidine (b) g_2: Cytidine,2',3'-dideoxy-2',5-difluoro (c) g_3: 3-ethyl-2-propyl-1,1'-bi(cyclohexane)-1,3'-diene

Figure 4. Example compounds

String	→	*Type Size Edits*
Type	→	**Line**|**Cycle**|**Star**
Size	→	**number**
Edits	→	*Edit Edits*|ϵ
Edit	→	⟨**node** *Pos Label*⟩|
		⟨**branch** *Pos Label*⟩|
		⟨**edge** *Pos Pos Edgelabel*⟩|
		⟨**c** *Pos Pos*⟩
Pos	→	**number**
Label	→	**atom**
Edgelabel	→	**d**|**t**

Table 1. Grammar for basic structures



(a) Compound of Figure 4(c) (b) Compound of Figure 3(f)

Figure 5. Graphs are simplified after each basic structure is collapsed into a single node.

a line structure except that the two carbons at position 4 and 5 have a double bound between them. Similarly, aniline in Figure 3(c) is represented by *Cycle 6* ⟨*branch 2 N*⟩. Some structures require more than one annotation. For example, the chemical compound *ethyl sulfate*, which is shown in Figure 3(d), is a star structure. It is represented as *Star 4* ⟨*node 0 S*⟩ ⟨*node 2 O*⟩ ⟨*node 3 O*⟩ ⟨*node 4 O*⟩ ⟨*edge 0 3 d*⟩ ⟨*edge 0 4 d*⟩ ⟨*branch 1 C*⟩.

As shown in the grammar, besides the *node*, *branch*, *edge* types of edits, there is yet another type: *connection*, or c. It is used to describe how basic structures in a compound connect to each other, which we detail in Section 4.4.

4.4 From Basic Structures to Compounds

Given a chemical compound, we break it into a set of connected basic structures and represent each of them using a GString. Next, we thread the basic structures together to create a global representation of the chemical compound.

Path extractions The goal of introducing the basic structures is to enable us to reduce the complexity of the original graph while retaining its semantics. Specifically, we collapse each basic structure in the graph to a single node. As a result, the first two structures in Figure 4 become a single path of 3 nodes, where one node represents a *Line*, the other two represent a *Cycle* each. The compound in Figure 4(c) becomes a branching structure of four nodes, which is shown in Figure 5(a). The compound in Figure 3(f) becomes a triangle structure in Figure 5(b).

We then invoke an algorithm similar to GraphGrep [11] to extract paths from the collapsed graph. Since the graph has been greatly simplified, the usual high complexity of GraphGrep diminishes. For instance, only one unique path is extracted from the first two structures in Figure 4, and three unique paths are extracted from the third one. The problem becomes tractable, while no semantics is lost.

Connections The collapsed graph shows whether two basic structures are connected, but it does not reveal how the connection is made. While the semantics of each basic structure is encoded into a sequence using the method described above, the semantics of the connection has not been addressed. We now describe how to encode a connection so that basic structures can form complex compounds.

Let $\langle b_1, \cdots, b_i, b_{i+1}, \cdots, b_n \rangle$ denote a list of basic structures on an extracted path. We use syntax ⟨*c Position1 Position2*⟩ to describe a connection from basic structure b_i to b_{i+1}, where *Position1* is the connecting position in b_i, and *Position2* is the connecting position in b_{i+1}. For example, in the compound shown in Figure 4(a), the 1st basic structure is *Line 2* ⟨*node 1 O*⟩. Its second node connects to the 2nd basic structure *Cycle 5* ⟨*node 2 O*⟩ at its first node. Then, we append ⟨*c 2 1*⟩ to the 1st basic structure, and its GString becomes *Line 2* ⟨*node 1 O*⟩ ⟨*c 2 1*⟩. Similarly, the

2nd basic structure connects to the 3rd basic structure at vertex 3 with distance 1. The GString of the 2nd basic structure becomes *Cycle 5* ⟨*node 2 O*⟩ ⟨*branch 5 F*⟩ ⟨*c 3 1*⟩.

Following the above procedure, we convert the first two compounds in Figure 4 to GStrings in Example 1. We can do the same for each path extracted from the third compound. We omit the details for lack of space.

Example 1 (GStrings for compounds of Figure 4)

G_1 : *Line 2*⟨*node 1 O*⟩⟨*c 2 1*⟩
Cycle 5⟨*node 2 O*⟩⟨*branch 5 F*⟩⟨*c 3 1*⟩
Cycle 6⟨*node 1 N*⟩⟨*node 3 N*⟩⟨*edge 3 4 d*⟩⟨*edge 5 6 d*⟩
⟨*branch 2 O*⟩⟨*branch 4 N*⟩

G_2 : *Line 2*⟨*node 1 O*⟩⟨*c 2 1*⟩
Cycle 5⟨*node 2 O*⟩⟨*branch 4 F*⟩⟨*c 3 1*⟩
Cycle 6⟨*node 1 N*⟩⟨*node 3 N*⟩⟨*edge 3 4 d*⟩⟨*edge 5 6 d*⟩
⟨*branch 2 O*⟩⟨*branch 4 N*⟩⟨*branch 5 F*⟩

In Figure 3(e) and 3(f), we showed two compounds that consist of multiple cycles. We can use the connection construct to describe the two compounds in similar ways. However, there are two subtle issues. First, more than one nodes may involve in the connection. In this case, we may add multiple connection constructs in the sequence. Second, the connections in Figure 3(e) and 3(f) are different from those in Figure 4 in that the connection is not made through a single bound, but rather, through the sharing of a same node. One alternative is to introduce a *sharing* construct, for example, ⟨*s Position1 Position2*⟩. We, instead, choose to overload the syntax of ⟨*c Position1 Position2*⟩ in our approach to represent sharing. The reason is that introducing a new construct for this case makes little improvement in the filtering power during chemical compound search.

5 Graph Query using GStrings

In this section, we demonstrate how *GStrings* are used to support meaningful subgraph queries.

5.1 Overview

Our goal is to index on the major features of a graph, and use the index to support subgraph queries. For this purpose, after obtaining *GStrings* from graphs, we extract representative characteristics from the *GStrings*, and use these characteristics as filtering features. In addition, to support approximate subgraph queries in our framework, we allow certain disparities in matching the characteristics.

5.2 Summary String

We extract certain features from a GString, and index these features. The features must capture the most important semantics of the graph, while leaving out minor details so that the index structure is compact and can be accessed efficiently.

For graphs that represent chemical compounds, the most important characteristics are captured by the basic structures (Cycle, Star, and Line) they contain. Our feature extraction leaves out certain details in the *Edits*, such as whether a connection between two atoms is a double bound connection or not.

The extracted features form a *summary string*. Each basic structure in a GString is transformed into a summary string by using the following rules:

- The *Type* and *Size* of each GString basic structure remain intact.
- For *Edit* of type **node**, **branch**, **edge**, we record the number of each type in the GString.

The idea is to use Type and Size for building the index structure, while using $[n_n, n_b, n_e]$ for additional filtering (e.g., through the use of a similarity function). Thus, for a basic structure, its summary string is in the form of:

$$\text{Type Size } [n_n, n_b, n_e]$$

where n_n, n_b and n_e are the number of node, branch, edge types of edits.

For example, the second basic structure in the compound shown in Figure 4(a) is represented in GString as *Cycle 5* ⟨*node 2 O*⟩ ⟨*branch 5 F*⟩ ⟨*c 3 1*⟩. Its summary string is simply *Cycle 5 [1 1 0]*.

A GString contains multiple basic structures. We turn each basic structure into a summary string using the above procedure, and concatenate the results into a complete summary string. Example 2 gives the summary strings for the three compounds shown in Figure 4. Note that for each extracted path, we also index its reverse path in the index structure. For example, g_{12} is the same path as g_{12} but in reverse order. As a result, we have a total of 10 summary strings for the 3 compounds in Figure 4.

Example 2 (Summary strings for compounds in Fig. 4)

g_{11} : *Line 2*$[1,0,0]$, *Cycle 5*$[1,1,0]$, *Cycle 6*$[2,2,1]$
g_{12} : *Cycle 6*$[2,2,1]$, *Cycle 5*$[1,1,0]$, *Line 2*$[1,0,0]$
g_{21} : *Line 2*$[1,0,0]$, *Cycle 5*$[1,1,0]$, *Cycle 6*$[2,3,2]$
g_{22} : *Cycle 6*$[2,3,2]$, *Cycle 5*$[1,1,0]$, *Line 2*$[1,0,0]$
g_{31} : *Line 2*$[0,0,0]$, *Cycle 6*$[0,0,1]$, *Line 3*$[0,0,0]$
g_{32} : *Line 2*$[0,0,0]$, *Cycle 6*$[0,0,1]$, *Cycle 6*$[0,0,1]$
g_{33} : *Line 3*$[0,0,0]$, *Cycle 6*$[0,0,1]$, *Cycle 6*$[0,0,1]$
g_{34} : *Line 3*$[0,0,0]$, *Cycle6* $[0,0,1]$, *Line 2*$[0,0,0]$
g_{35} : *Cycle 6*$[0,0,1]$, *Cycle 6*$[0,0,1]$, *Line 2*$[0,0,0]$
g_{36} : *Cycle 6*$[0,0,1]$, *Cycle 6*$[0,0,1]$, *Line 3*$[0,0,0]$

5.3 Summary String Index

We design a suffix tree based index structure to facilitate the *summary string* matching process. The index structure consists of two parts: a suffix tree, which indexes the

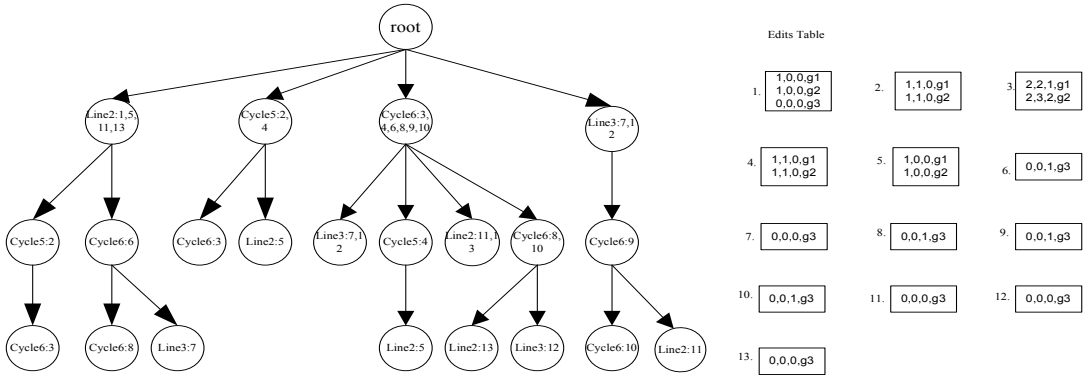


Figure 6. The enhanced suffix tree index

(Type, Size) part in the summary string, and an edit table, which stores the $[n_n, n_b, n_e]$ annotation associated with each (Type, Size).

For each summary string, we insert all of its suffixes into the suffix tree. For instance, g_{11}, one of the two summary strings of the first chemical compound in Figure 4, contains 3 elements, which means it has 3 suffixes.

S_1 : *Cycle 6*$[2, 2, 1]$
S_2 : *Cycle 5*$[1, 1, 0]$, *Cycle 6*$[2, 2, 1]$
S_3 : *Line 2*$[1, 0, 0]$, *Cycle 5*$[1, 1, 0]$, *Cycle 6*$[2, 2, 1]$

We insert the 3 suffixes into a tree structure, and we do the same for each of the extracted paths of the other compounds in Figure 4. The result index structure for compounds g_1, g_2, and g_3 is shown in Figure 6. In the tree, each node contains the following information: [`Type, Size`: $t_1, t_2, \cdots$], where t_i is a shared Edit Table that stores annotation information of those elements that match the node.

Algorithm 1 outlines the indexing procedure. It first inserts an entire path into the trie, creating new nodes whenever necessary. Once the path is inserted, we essentially have found (or created) for each element e_i in the path an edit table t_i to store the $[n_n, n_b, n_e]$ information of e_i. When we insert suffixes of the path, these edit tables are reused and added into the nodes that match the elements in the suffixes.

Input: a set of paths $P = \{p_1, \cdots, p_n\}$
Output: a Trie that records all the suffixes of P
Trie $\leftarrow$ an empty tree;
for *each* $p_i = \langle e_1, \cdots, e_n \rangle \in P$ **do**
  assume id = ID of the graph that contains p_i;
  `/* insert` p_i `into Trie */`;
  for *each* $e_i \in p_i$ **do**
    if $e_i \in p_i$ *matches node* n *in Trie* **then**
      $t_i \leftarrow$ the first Edit Table of n;
    else
      create a new node n in Trie for n;
      create an Edit Table t_i for n;
    insert entry $[n_n, n_b, n_c, id]$ into t_i;
  for *each suffix* $s_k = \langle e_k, \cdots, e_n \rangle$ *of* p_i **do**
    `/* insert` s_k `into Trie */`;
    for *each* $e_i \in s_k$ **do**
      if $e_i \in s_k$ *matches node* n *in the Trie* **then**
        add t_i into the set of n's Edit Tables;
      else
        create a new node n for e_i;
        add t_i into the set of n's Edit Tables;

Algorithm 1: Indexing

5.4 Subgraph Query

Given a query graph, we derive its GString and summary string. Then, we match the summary string against the suffix-tree of the graph database.

An element of a *summary string* matches a node in the suffix-tree if the following conditions are satisfied: i) their $Type$s match; ii) their $Size$s are equal if they are *Cycle*s, or the $Size$ in the query is no more than the $Size$ in the node; and iii) the counts of corresponding types of $Edits$ (n_n, n_b, n_e) in the query are no larger than those in the node. Clearly, the above matching rules already reflect many semantic concerns in searching chemical compounds. For example, it dictates that in order for two *Cycle* structures to match, they must be of the same size. However, for *Line*s and *Star*s, this restriction can be relaxed. Also, it uses n_n, n_b and n_e for filtering. We certainly can use any similarity function in the matching process.

Let us use the chemical compound in Figure 7 as a query. The database D contains three compounds g_1, g_2, and g_3 shown in Figure 4, and its index structure is shown in Fig-

ure 6. It is clear that g_2 contains q. However, because the structures of g_1 and g_2 are highly similar, it is hard to use the paths or fragments extracted from the original graphs to distinguish them: The computation cost will be very high because only after a large number of joins will the matching algorithm discover that they are not the same at a higher level.

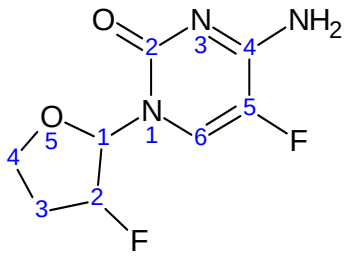


Figure 7. Query example

In our approach, the query graph is converted into a *GString* consisting of two basic structures:

Example 3 (GString for the query graph in Fig. 7)
Cycle 5 ⟨*node 5 O*⟩ ⟨*branch 2 F*⟩ ⟨*c 1 1*⟩
Cycle 6 ⟨*node 1 N*⟩ ⟨*node 3 N*⟩ ⟨*edge 3 4 d*⟩ ⟨*edge 5 6 d*⟩
⟨*branch 2 O*⟩ ⟨*branch 4 N*⟩ ⟨*branch 5 F*⟩

The *summary string* used for searching in the suffix tree is *Cycle 5 [1 1 0]Cycle 6 [2 3 2]*. Then, g_1 will be filtered out because it has $n_b = 2$ in the second *Cycle*, while the query asks for $n_b = 3$, which means the query graph cannot be a subgraph of g_1. On the other hand, g_2 passes the filtering, and after a comparison of their GString representations as well as a subgraph isomorphism test [7], it comes out as a solution. Algorithm 2 outlines the query procedure.

Note that when comparing *GString*s, we allow the $Position$ item in $Edit$ to be different. For example, g_2 and query graph q are different on numbering of their first *Cycles*. Our matching approach recognizes that the two GStrings *Cycle 5* ⟨*node 5 O*⟩ ⟨*branch 2 F*⟩ in the query and *Cycle 5* ⟨*node 2 O*⟩ ⟨*branch 4 F*⟩ in g_2 refer to the same *Cycle* structure. The difference in positions is caused by the numbering method.

6 Experimental Results

In this section, we evaluate the performance of GString on both subgraph queries and similarity queries through a series of experiments.

We use an NCI/NIH AIDS Antiviral Screen Dataset [6] in our experiments. This dataset contains around 43,000 chemical compounds. We generate the vertex-labelled graphs from the molecule structures and omit Hydrogen atoms. The graphs have an average number of 25 vertices and 27 edges, and a maximum number of 222 vertices and 251 edges. A major portion of the vertices are C, O and N. The total number of distinct labels is 62.

Input: paths $P = \{p_1, \cdots, p_n\}$ extracted from a query
Input: $Trie$, the index structure
Output: R, graphs that may contain the query
$R \leftarrow$ universe;
for $p_i \in P$ **do**
  match p_i along a path in $Trie$;
  a match is found if (Type, Size) matches and annotations in the Edit Table match, according to some similarity function;
  $R \leftarrow R \cap$ documents that match every path;
return R;

Algorithm 2: Searching

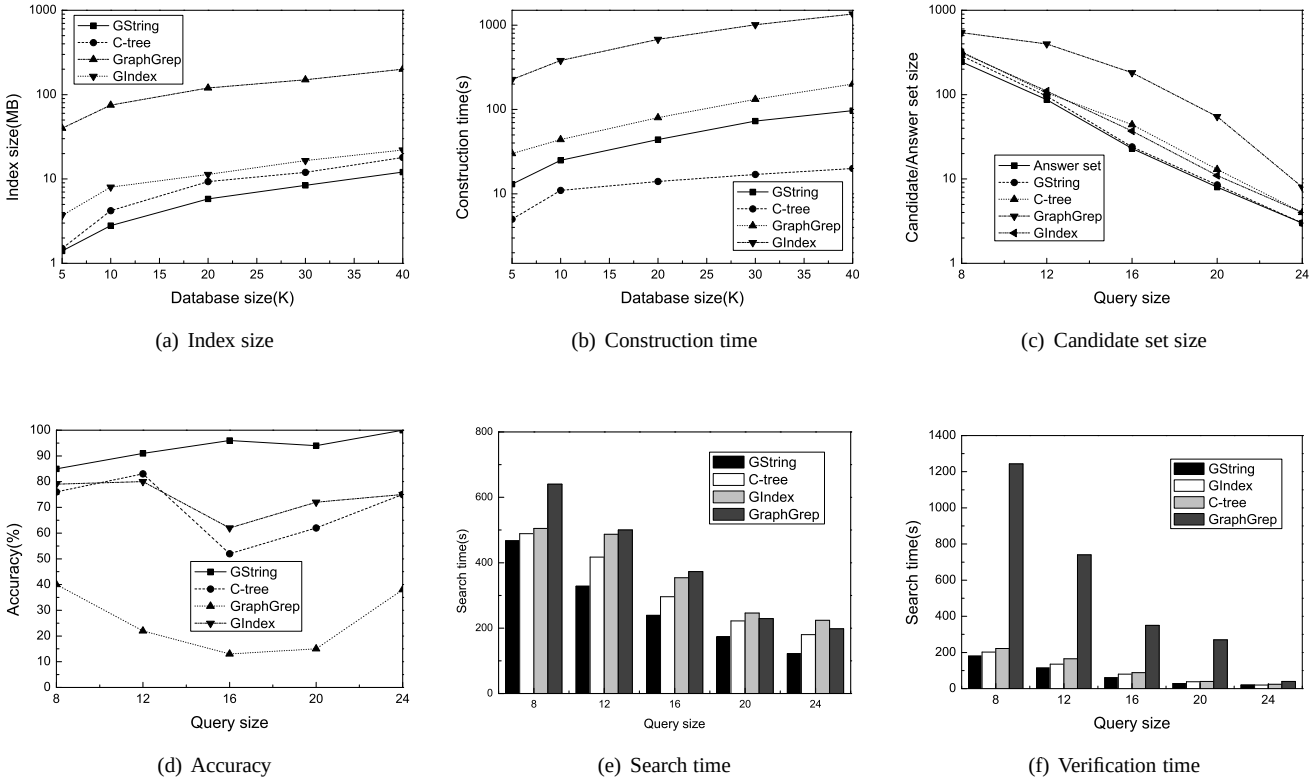
For subgraph queries, the performance of GString is compared with GraphGrep, GIndex and C-tree. GraphGrep, GIndex, GString are all implemented in C++ and compiled with gcc/g++. C-tree is implemented in Java and complied with Sun JDK 1.5.0. All experiments are conducted on a PC with CPU of AMD Sempron 1.4GHZ, 512MB memory, and running Linux2.6/Fedora 5.

In experiments, GraphGrep, GIndex and C-tree all need some parameters. We choose the default or the suggested values. For GraphGrep, there are two parameters: the length of path l_p and the length of fingerprint f_p. We set $l_p = 4$ and $f_p = 256$. For GIndex, we set the maximum fragment size max_L to 10, the minimum discriminative ratio γ_{min} to 2.0, and the maximum support Θ to $0.1N$. The size-increasing support function $\varphi(l)$ is 1 if $l < 4$; in all other cases, $\varphi(l)$ is $\sqrt{\frac{t}{maxL}}\Theta$. For C-tree, we set the minimum number of child nodes m=20, the maximum number M=$2m - 1$, the pseudo subgraph isomorphism level to 1. The NBM method [4] is used to compute graph closures.

6.1 Index Size and Construction Time

Fig. 8(a) shows the varying of index size with respect to the database size. We set database size to 5K, 10K, 20K, 30K, 40K respectively, which are obtained by randomly selecting graphs from the original dataset. As shown in the Fig. 8(a), the index sizes of GString are around 20 times smaller than that of GraphGrep; GString outperforms C-tree and GIndex by 7%-30% and 50% respectively. GraphGrep has to enumerate every path up to length lp and uses them as index features. In the worst case, the index size for GraphGrep grows exponentially with respect to lp. On the contrary, the index size of GString is coherently proportional to the database size.

Fig. 8(b) is the changing of index construction time with respect to the database size. Though, index construction of



(a) Index size (b) Construction time (c) Candidate set size

(d) Accuracy (e) Search time (f) Verification time

Figure 8. Performance Results

GString is a little slower than C-tree, taking consideration of index size, accuracy and search time, we validate that GString makes a good balance.

6.2 Results of Subgraph Queries

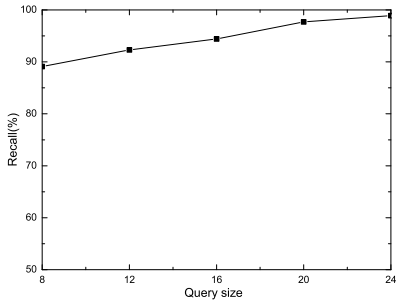
Experiments of subgraph queries are conducted as follows. We fix the database size to 20K and vary the query sizes from 8 to 24 in terms of the number of vertices. For each query size, we generate a set of 500 queries and average the results. Each query is generated by randomly selecting a graph form the database and randomly extracting several connected basic structures from the graph.

Fig. 8(c) shows the trend of candidate answer set size with respect to the query size. The candidate set sizes of the four approaches all decrease as the query size increases, because fewer graphs contain long queries. Nevertheless, GString has smaller candidate answer set than the other three approaches have, given the query size from 8 to 24. For large query sizes, the candidate set of GString is nearly one order of magnitude smaller than that of GraphGrep. Fig. 8(d) illustrates the accuracy of candidate answers, which is evaluated as the percentage of matching graphs in the candidate answers set. As shown in the Fig. 8(d), GString can select candidates with very high accuracy.

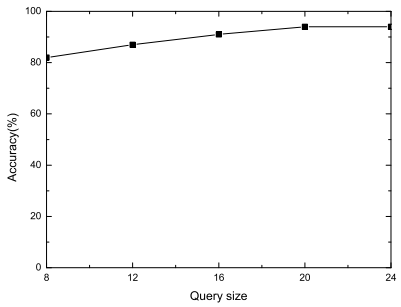
The query time, which consists of search and verification time, are shown in Fig. 8(e) and Fig. 8(f) respectively. GString outperforms C-tree and GraphGrep on search time around 10%-40% because of the high efficient suffix tree and linear complexity string pattern matching, while C-tree have to perform expensive graph computation, and GraphGrep accesses a quite large index. The verification time of GString is less than that of C-tree and GraphGrep, which is the direct result of smaller candidate sets generated by GString.

6.3 Results of Similarity Queries

Experiments of similarity queries are carried out as follows. We fix the database size to 5K and vary the query sizes from 8 to 24 in terms of the number of vertices. For each query size, we generate a set of 500 queries and average the results. Each query is generated by randomly selecting a graph from the database, randomly selecting and extracting several connected basic structures from the graph and randomly adding branches, relabelling of nodes and edges. We apply the maximal common graph algorithm



(a) Recall



(b) Accuracy

Figure 9. Recall and Accuracy

to verify substructure similarity, and set *relaxation ratio* to 10%.

Considering that GraphGrep and Gindex do not support substructure similarity search and the k-NN similarity query defined by C-tree is not compatible to the relaxation ratio, so in the following experiments we just evaluate GString.

GString may lose some final answers because it focuses on the basic structure, but this kind of loss is reasonable. For example, if the query graph Q is a cycle of 10 carbon atoms, and a substructure of a certain graph G in the database is a chain of carbo atoms with length 9, even though the relaxation ratio is 10%, it is unlikely that G should be selected because of the differences between their chemical characteristics.

But we still evaluate the recall percentage of GString. Fig. 9(a) shows that the answer loss of GString is in a acceptable level. Fig. 9(b) shows GString keeps high accuracy in substructure similarity search.

7 Conclusion

Graph indexing plays a critical role for efficient query processing over graph databases, which have gained increasing use and acceptance in bioinformatics, XML, and other applications involving complex structures. In this paper, we introduce GString, a semantic-based approach to index chemical compound database. Our method converts a graph into its string representation and then uses sophisticated string indexing techniques to support subgraph query. As we have demonstrated in our experiments, GString constructs compact index structures, and produces highly accurate and concise answers, both in an efficient manner.

8 Acknowledgments

We thank D. Shasha and R. Giugno for providing GraphGrep, H. He and A. K.Singh for providing C-tree, and X. Yan for providing Gindex. This work was supported by National Natural Science Foundation of China (NSFC) under grants no. 60373019 and no. 90612007. Shuigeng Zhou is also supported by Shuguang Scholar Program of Shanghai Education Development Foundation.

References

[1] A.Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[2] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition*, 19:255–259, 1998.

[3] C.A.James, D.Weininger, and J.Delany. Daylight theory manual daylight version 4.82. *Daylight Chemical Information Systems*, 2003.

[4] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.

[5] J. Huan, D. Bandyopadhyay, W. Wang, and J. Snoeyink. Comparing graph representation of protein structure for mining familiy-specific residue-based packing motifs. *Journal of Computational Biology (JCB)*, 12(6):657–671, 2005.

[6] N. C. Institue. http://dtp.nci.nih.gov/.

[7] J.Ullman. An alogrithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42, 1976.

[8] R. Kaushik, P.Shenoy, P.Bohannon, and E. Gudes. Exploting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, pages 129–140, 2002.

[9] J. Lee, J. Oh, and S. Hwang. Strg-index: Spatio-temporal region graph indexing for large video databases. In *SIGMOD*, pages 718–729, 2005.

[10] R.Goldman and J.Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[11] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[12] H. Wang and X. Meng. On the sequencing of tree structures for XML indexing. In *ICDE*, pages 372–383, 2005.

[13] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, 2003.

[14] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[15] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.