# A Novel Spectral Coding in a Large Graph Database *

Lei Zou
Huazhong Univ. of Sci. & Tech.
Wuhan, China
zoulei@mail.hust.edu.cn

Lei Chen
Hong Kong Univ. of Sci. & Tech.
Hong Kong, China
leichen@cse.ust.hk

Jeffrey Xu Yu
The Chinese Univ. of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Yansheng Lu
Huazhong Univ. of Sci. & Tech.
Wuhan, China
lys@mail.hust.edu.cn

## ABSTRACT

Retrieving related graphs containing a query graph from a large graph database is a key issue in many graph-based applications, such as drug discovery and structural pattern recognition. Because sub-graph isomorphism is a NP-complete problem [4], we have to employ a *filter-and-verification* framework to speed up the search efficiency, that is, using an effective and efficient pruning strategy to filter out the false positives (graphs that are not possible in the results) as many as possible first, then validating the remaining candidates by subgraph isomorphism checking. In this paper, we propose a novel filtering method, a spectral encoding method, i.e. GCoding. Specifically, we assign a signature to each vertex based on its local structures. Then, we generate a spectral graph code by combining all vertex signatures in a graph. Based on spectral graph codes, we derive a necessary condition for sub-graph isomorphism. Then we propose two pruning rules for sub-graph search problem, and prove that they satisfy the no-false-negative requirement (no dismissal in answers). Since graph codes are in numerical space, we take this advantage and conduct efficient filtering over graph codes. Extensive experiments show that GCoding outperforms existing counterpart methods.

## 1. INTRODUCTION

As a popular data structure, graphs have been used to model many complex data objects and their relationships in the real world, for example, the chemical compounds [19] [9], entities in images [13] and social networks [1]. Due to the wide usage of graphs, it is quite important to retrieve related graphs containing a query graph from a graph database efficiently. For example, given a large chemical compound database, a chemist may want to find all chemical compounds having a particular sub-structure. Formally, we define this type of search as *sub-graph search*, that is *given a query graph Q, we need to find all data graphs $G_i$, where $G_i$ contains the query Q, namely, Q is sub-graph isomorphism to $G_i$*. Figure 1 shows a running example of sub-graph search, where a query graph $Q$ and a graph database with 4 graphs are listed. The number beside the vertex is vertex ID and the letter in the vertex is vertex label. Graph 003 should be returned as the result, since graph 003 contains query $Q$.

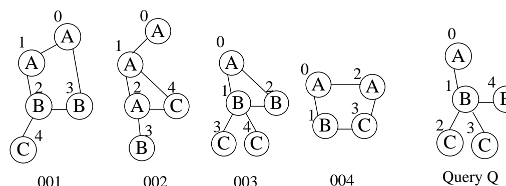



**Figure 1: Running Example**

However, it is not trivial to conduct sub-graph search efficiently since the sub-graph isomorphism itself is a NP-complete problem [4]. Thus, to speed up the search, we often use the filter-and-verification framework, that is, first employing an effective and efficient *pruning strategy* to remove the false positives (graphs that are not possible in the results), then checking the remaining candidates by sub-graph isomorphism. Since the latter step is computation expensive, the effectiveness of the pruning strategy is the key issue to improve the search efficiency. So far, there are many pruning strategies have been proposed [14, 22, 7, 24, 6, 3, 2, 25], which can be divided into the following two categories:

1. Data-mining based filtering. The approaches in this category apply data mining methods first to extract some features (sub-structures) from the graphs. After that, an inverted index is created for each feature. To answer a sub-graph query $Q$, $Q$ is represented as a set of features and all the graphs that may contain $Q$ are retrieved by examining the inverted index. The rational behind this type of filtering methods is that if some features of graph $Q$ do not exist in a data graph $G$, $G$ cannot contain $Q$ as its sub-graph. The filtering methods belonging to this category are gIndex [22], Treepi [24], FG-Index [2] and Tree+$\delta$ [25]. However, the ef-

*The work was done when the first author visited Hong Kong University of Science and Technology as a visiting scholar. The work was supported by Hong Kong RGC Grant No.611907, National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303000. The first and the fourth author were also partially supported by National Natural Science Foundation of China under Grant 70771043.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

fectiveness of these filtering methods depends on the quality of selected features. Moreover, as mentioned in [22] [24], the quality of the selected features may degrade over time after lots of insertions and deletions. In this case, we have to re-select features in the whole updated graph database, and re-build the index from scratch, which is quite time consuming. For example, according to the report by the SCI Finder[1], approximate 4,000 new compound structures are added each day. In this scenario, it is expensive to re-compute the index to handle updates from scratch.

2. Non-data mining based filtering. Different from the above methods, no data mining-based feature selections are needed in the second category, such as GraphGrep [14] and Closure-tree [7]. In GraphGrep, all pathes up to $maxL$ are enumerated. Then, we build inverted index for each path in GraphGrep. Given a query graph, all the distinct pathes in the query graph are searched in the inverted index for the data graphs contain those path as well. In Closure-tree, the basic idea is that if we can determine that there does not exist an injective function from vertices in graph $Q$ to ones in graph $G$, $Q$ cannot be sub-graph isomorphism to $G$. These two methods can handle the graph updates with less cost, since they do not rely on the effectiveness of selected features. However, current non-data mining approaches are either less effective in pruning power due to using simple paths (e.g. GraphGrep), or have to conduct expensive structure comparison (e.g. Closure-tree) in the filtering process, which makes the filtering step itself become expensive and degrades the filtering efficiency.

To address the above shortcomings, in this paper, we propose a spectral pruning strategy (that is *GCoding*) for sub-graph search problem. Specifically, we use a tree to represent the local structure associated with each vertex. Then, we assign the spectral signatures to vertices by mapping the local structure information into the numerical space based on spectral graph theory. We generate the graph code by combining all vertex signatures of a graph and conduct filtering step using both the graph code and local signatures. We prove that our filtering method based on spectral coding will not introduce false negatives, moreover, due to the light computation cost of graph code comparison, the graph coding is efficient and effective in removing false positives, which is also confirmed by the experiments. Most importantly, it is not necessary for *GCoding* to re-compute spectral signatures from scratch in order to handle insertion and deletion. Therefore, *GCoding* can work well in the graph database with frequent updates.

In summary, we made the following contributions in this paper:

1. We propose a novel spectral graph coding technique, *GCoding*, by encoding the structures of a graph into a numerical space. We design an efficient storage schema for storing encoded graphs and an efficient index structure, GCoding-tree.
2. Based on *GCoding* technique, we propose effective and efficient pruning strategies for sub-graph search problem, and prove that they satisfy *no-false-negatives* requirement (no dismissal in answers).
3. We have shown the superiority of GCoding compared to the existing methods through extensive experiments.

[1]SCI Finder: a research discovery tool that allows you to access the world's largest collection of biochemical, chemical, chemical engineering, medical, and other related information. http://www.cas.org/SCIFINDER/

The remainder of the paper is organized as follows: We discuss some background knowledge in Section 2. The graph coding method and pruning strategy are proposed in Section 3. The framework of sub-graph search is discussed in Section 4. We evaluate our method in the extensive experiments in Section 5. We discuss the related work in Section 6 in details. Finally, we conclude the paper in Section 7.

# 2. BACKGROUND

In this section, we briefly review the terminologies that we will use in this paper.

## 2.1 Graphs and Trees

Definition 2.1. ***Graph.*** *A labeled graph is always denoted as* $< V, E, L_v, L_e, F_v, F_e >$, *where (1)* $V$ *is the set of vertices; (2)* $E$ *is the set of edges; (3)*$L_v$ *is the set of vertex labels; (4)* $L_e$ *is the set of edge labels; (5)* $F_v$ *is a function:* $V \to L_v$ *that assigns labels to vertices; (6)* $F_e$ *is a function:* $E \to L_e$ *that assigns labels to edges.*

Definition 2.2. ***Sub-graph.*** *If a graph* $G'$ *whose vertices and edges form subsets of the vertices and edges of a given graph* $G$, $G'$ *is a sub-graph of* $G$.

Definition 2.3. ***Induced Sub-graph.*** *An induced sub-graph* $G'$ *is a subset of the vertices of a graph* $G$ *together with any edges whose endpoints are both in this subset.*

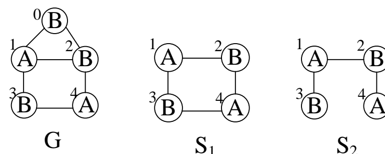



**Figure 2: Sub-graph and Induced Sub-graph**

In fact, *induced sub-graph* is a special case of *sub-graph*. In Figure 2, $S_1$ and $S_2$ are sub-graphs of graph $G$ respectively. $S_1$ is also an induced sub-graph of $G$. However, $S_2$ is not an induced sub-graph of $G$, since the edge $(3, 4)$ does not exist in $S_2$.

Definition 2.4. ***Graph Isomorphism.*** *Assume that we have two graphs* $G_1 < V_1, E_1, L_{1v}, L_{1e}, F_{1v}, F_{1e} >$ *and* $G_2 < V_2, E_2, L_{2v}, L_{2e}, F_{2v}, F_{2e} >$. $G_1$ *is graph isomorphism to* $G_2$, *if and only if there exists at least one bijective function* $f : V_1 \to V_2$ *such that: 1) for any edge* $uv \in E_1$, *there is an edge* $f(u)f(v) \in E_2$; *2)* $F_{1v}(u) = F_{2v}(f(u))$ *and* $F_{1v}(v) = F_{2v}(f(v))$; *3)* $F_{1e}(uv) = F_{2e}(f(u)f(v))$.

Definition 2.5. ***(Induced) Sub-graph Isomorphism.*** *Assume that we have two graphs* $G'$ *and* $G$, *if* $G'$ *is graph isomorphism to at least one (induced) sub-graph of* $G$ *under bijective function* $f$, $G'$ *is (induced) sub-graph isomorphism to* $G$ *under injective function* $f$.

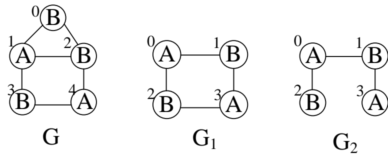



**Figure 3: (Induced) Sub-graph Isomorphism**

In Figure 3, $G_1$ is induced sub-graph isomorphism to $G$, since $G_1$ is graph isomorphism to $S_1$ in Figure 2, where $S_1$ is an induced subgraph of $G$. Similarly, $G_2$ is sub-graph isomorphism to $G$. Usually, for presentation convenience, when $G'$ is (induced) sub-graph isomorphism to $G$, we also say that $G'$ is (induced) sub-graph of $G$. Throughout the rest of the paper, we do not distinguish *(induced) sub-graph* and *(induced) sub-graph isomorphism* when the context is clear.

In this paper, all trees are *unlabeled rooted unordered* trees unless otherwise specified, which is defined as follows.

DEFINITION 2.6. ***Unlabeled Rooted Unordered Tree.*** *A rooted unordered tree* T *denoted as* $T = (V, v_0, E)$, *where (1) V is the set of nodes ; (2)* $v_0$ *is a distinguished node called the root that has no entering edges; (3) E is the set of edges in the tree. Note that, there are no predefined ordering among each set of siblings in rooted unordered tree.*

DEFINITION 2.7. ***Induced Subtree.*** *For a tree* $T$ *with node set* $V$ *and edge set* $E$ *, we say that a tree* $T'$ *with node set* $V'$ *and edge set* $E'$*, is an induced subtree of* $T$ *if and only if there exists at least one injective function* $f$*:* $V' \rightarrow V$ *such that: for any directed edge* $uv \in E'$ *(namely,* $u$ *is a parent of* $v$ *in tree* $T'$*), there is an directed edge* $f(u)f(v) \in E$*.*

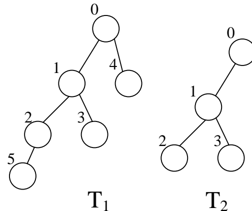



**Figure 4: Induced Subtree**

In Figure 4, tree $T_2$ is an induced subtree of $T_1$.

## 2.2 Matrices and Eigenvalues

As we all know, for any graph $G$, we can denote it by $\{0, 1\}$ adjacency matrix $M$. In this paper, we only consider non-directed graphs. Therefore, $M$ is a symmetric matrix. Given a $n \times n$ matrix $M$, there exist a column $n$-vector $\overrightarrow{v}$ and a scalar $\lambda$ such that

$$M\overrightarrow{v} = \lambda\overrightarrow{v}$$
$$\langle \overrightarrow{v}, \overrightarrow{v} \rangle = 1$$

, where $\langle \overrightarrow{v}, \overrightarrow{v} \rangle$ is the inner product of two vectors, which is defined as $\langle \overrightarrow{v}, \overrightarrow{v} \rangle = \sum_{i=1}^{n} v_i * v_i$. The $\overrightarrow{v}$ and $\lambda$ are called the normalized eigenvector (or simply eigenvector) and eigenvalue of $M$ respectively. The eigenvectors need to be normalized since otherwise there is an infinite number of eigenvalues that are obtained by scaling the eigenvectors.

For an $n \times n$ matrix, there are a total of $n$ such eigenvector and eigenvalue pairs, but they may not be distinct. The eigenvalues are usually denoted by $\lambda_1 ... \lambda_n$ ordered by their magnitude in non-ascending order. In spectral graph theory, there exists a close relation between the eigenvalues of a graph $G$ and its induced sub-graph $G'$, which is stated in the following theorem.

THEOREM 2.1. [18] *Given a graph G with n vertices and a graph* $G'$ *with m vertices (*$n \geq m$*), their adjacency matrices are denoted as A and B respectively. For matrix A, its eigenvalues are* $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$*. For matrix B, its eigenvalues are* $\beta_1 \geq \beta_2 \geq ... \geq \beta_m$*. If* $G'$ *is an* ***induced*** *sub-graph of G, then* $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$*, ( i = 1,..., m ).*

In fact, the above theorem is derived from *Interlacing Theorem* about the matrix $M$ and its principal sub-matrix [18, 15]:

THEOREM 2.2. ***(Interlacing Theorem)*** [18]. *Let A be a symmetric matrix with eigenvalues* $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$ *and let B be one of its principal sub-matrix. If the eigenvalues of B are* $\beta_1 \geq \beta_2 \geq ... \geq \beta_m$*, then* $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$*, ( i = 1,..., m ).*

In *Interlacing Theorem*, a principal sub-matrix $B$ of size $m$ for a matrix $A$ ($n \times n$) is formed by selecting $m$ rows and the corresponding $m$ columns from the matrix $A$ ($m \leq n$). Now, we can prove Theorem 2.1 as following:

*Proof.* (Sketch) If $G'$ is induced sub-graph of $G$, it is clear to know, the adjacency matrix $A$ must be a principle sub-matrix of matrix $B$. According to *Interlacing Theorem*, Theorem 2.1 holds. □

For a tree $T$ with $n$ vertices, we denote it by the adjacency matrix $M$. In tree $T$, if the node $i$ is a parent of node $j$, we set $M_{ij}$ =1 and $M_{ji}$=1. Therefore, $M$ is also a symmetric matrix. If tree $T'$ is an induced sub-tree of $T$, the above theorem also holds for $T$ and $T'$. Formally, we have the following theorem.

THEOREM 2.3. *Given a tree T with n vertices and a tree* $T'$ *with m vertices (*$n \geq m$*), their adjacency matrices are denoted as A and B respectively. For matrix A, its eigenvalues are* $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$*. For matrix B, its eigenvalues are* $\beta_1 \geq \beta_2 \geq ... \geq \beta_m$*. If* $T'$ *is an* ***induced*** *sub-tree of T, then* $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$*, ( i = 1,..., m).*

*Proof.* (Sketch) If $T'$ is induced sub-graph of $T$, it is clear to know, the adjacency matrix $A$ must be a principle sub-matrix of matrix $B$. According to *Interlacing Theorem*, Theorem 2.1 holds. □

It seems that we can directly use Theorem 2.1 as the filtering method to prune false positives in sub-graph search. Unfortunately, the answer is No due the following observation.

REMARK 2.1. *Given a graph G with n vertices and a graph* $G'$ *with m vertices (*$n \geq m$*), their adjacency matrices are denoted as A and B respectively. For matrix A, its eigenvalues are* $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$*. For matrix B, its eigenvalues are* $\beta_1 \geq \beta_2 \geq ... \geq \beta_m$*.*
*1) If* $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$*, ( i = 1,..., m ), we* ***cannot*** *guarantee* $G'$ *must be an induced sub-graph of G.*
*2) If* $G'$ *is a sub-graph (may* ***not*** *be induced) of G, we* ***cannot*** *guarantee that* $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$*, ( i = 1,..., m ).*

Observed from the above Remark 2.1, we know that Theorem 2.1 is only **necessary** (not sufficient) condition about graph $G$ and its **induced** sub-graph (not sub-graph) $G'$. However, in sub-graph search problem, we need to consider *sub-graphs*, not *induced sub-graphs*. Therefore, we cannot directly apply Theorem 2.1 as a pruning strategy, which, in fact, motivates us to find a necessary condition for sub-graph isomorphism according to spectral graph theory.

# 3. GRAPH ENCODING

As we know that when the query graph $Q$ is sub-graph isomorphism to a data graph $G$, for each vertex $v$ in $Q$, there must exist a corresponding vertex $v'$ in $G$. Furthermore, the local structure around $v$ in $Q$ should be preserved around $v'$ in $G$. Thus, by checking the existence of this mapping, we can remove false positives as many as possible. However, as we mentioned before, in the filtering process of sub-graph search, directly checking the mapping for each vertex and its local structure is very expensive operation, since it needs to perform structure comparison. Thus, in this paper, in the filtering process, we propose GCoding to reduce the checking cost significantly, due to numerical comparison instead of structure comparison.

In GCoding, we propose *Vertex Signature* based on spectral graph theory and develop *Graph Code* for each graph by combining all vertex signatures, which will be discussed in Section 3.1 and 3.2 respectively. Based on vertex signature and graph code, we propose two pruning rules for sub-graph search problem. Furthermore, we prove that the two pruning rules satisfy *no-false-negative* requirement.

## 3.1 Vertex Signature

In this work, we first represent the local structure around a vertex as a tree and then encode the information related to the tree to a numerical space, called *vertex signature*. The filtering will be conducted over these signatures. Algorithm 1 shows the steps to construct the tree for a vertex $v$ in $G$, which is called **Level-n Path Tree** and denoted as $LNPT(G, v, n)$. $LNPT(G, v, n)$ consists of all $n$-step simple pathes from $v$ in graph $G$. Here, we define a *simple path* as a path in a graph with no repeated vertices.

Given a graph 003 and query $Q$ in Figure 5, $LNPT(003, 0, 2)$ and $LNPT(Q, 0, 2)$ are both shown in Figure 5. Note that, for each $v$ in $G$, $LNPT(G, v, n)$ is unique since trees here are unordered (we do not differentiate the order between siblings).

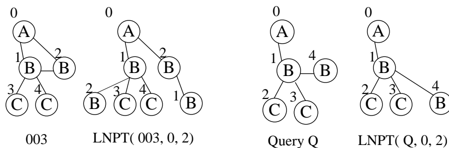



**Figure 5: Level-n Path Tree**

LEMMA 3.1. *Given two graphs* Q *and* G*,* Q *is sub-graph isomorphism to* G *under an injective function g. For each vertex* v *in graph* Q*, we have the* Level-n Path Subtree *around the vertex* v*, denoted by $LNPT(Q, v, n)$. We have a vertex $v'$ in graph* G*, where $v' = g(v)$. Then, $LNPT(Q, v, n)$ is an induced sub-tree of $LNPT(G, v', n)$.*

**Algorithm 1** Extracting Level-n Path Sub-tree

**Require:** Input: a graph $G$ and a vertex $v$ in $G$.
Output: level-n path sub-tree around the vertex $v$, denoted as $LNPT(G, v, n)$

1: Set the vertex $v$ as the root of $LNPT(v)$
2: Set the vertex set *Visited*$= v$.
3: **for** each neighbor $r$ of the vertex $v$ **do**
4: Insert the vertex $r$ as a child of $v$ in $LNPT(v)$.
5: Insert the vertex $r$ into the set *Visited*.
6: Call Function $Search(r, n)$.
7: **end for**

*Function*: Search(v,n)

1: $n = n$ - 1
2: **if** n == 0 **then**
3: **return**
4: **end if**
5: **for** each neighbor $r$ of the vertex $v$ **do**
6: **if** $r$ exists in the set *Visited* **then**
7: **return**
8: **end if**
9: insert the vertex $r$ as a child of $v$ in $LNPT(v)$.
10: insert the vertex $r$ into the set *Visited*.
11: Call Function $Search(r, n)$.
12: delete the vertex r from the set *Visited*.
13: **end for**

*Proof.* (Sketch) For each node $u$ in $LNPT(Q, v, n)$, according to the definition of $LNPT$, there must exist a path $vu$ in graph $Q$. Since graph $Q$ is sub-graph isomorphism to graph $G$, the path $vu$ in graph $Q$ must be preserved in graph $G$, which is corresponding to the path $v'u'$. Therefore, we can define an injective function $f$ from node $u$ in $LNPT(Q, v, n)$ to node $u'$ in $LNPT(G, v', n)$: $u' = f(u)$, where the path $vu$ is corresponding to the path $v'u'$. Under the injective function $f$, it is straightforward to prove that $LNPT(Q, v, n)$ is a induced sub-tree of $LNPT(G, v', n)$, according to the Definition 2.7. □

In Figure 5, $Q$ is a sub-graph of 003. Furthermore, vertex 0 in $Q$ is corresponding to vertex 0 in graph 003. Therefore, $LNPT(Q, 0, 2)$ is an induced sub-tree of $LNPT(003, 0, 2)$. We map the structure information of $LNPT$ into a numerical space. Specifically, for each vertex $v$ in graph $G$, we denote the local structure around $v$ (that is $LNPT(G, v, n)$) by $\{0, 1\}$ adjacency matrix $M$. Then, based on the eigenvalues of $M$, we assign *Vertex Topology Signature* to the vertex $v$. Formally, we have Definition 3.1 as follows.

DEFINITION 3.1. ***Vertex Topology Signature****. Given a graph* G *and a vertex* v $\in$ G*, the adjacency matrix of $LNPT(G, v, n)$ is denoted as $M$. Let eigenvalues of $M$ be $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$. The* Vertex Topology Signature *of the vertex v is defined as the sorted list topS(v)=[ $\lambda_1$,...,$\lambda_t$ ]. t is a specified parameter, where $t \leq m$*

Note that there are two parameters, that are $n$ and $t$, in Definition 3.1 about *topS*$(v)$. In experiments, we will discuss setting of $n$ and $t$. Here, without loss of generality, we set $n = 2$ and $m = 2$, which means that, for each vertex $v$ in graph $G$, we denote the local structure around $v$ by 2-level path tree (that is $LNPT(G, v, 2)$). When assigning *topS*$(v)$, we always choose the first two largest eigenvalues (spectral graph theory suggests that some largest eigenvalues determine greatly the graph's topological structure). For example, for vertex 0 in graph 003 in Figure 5,

$topS(0) = [2.10, 1.26]$. Similarly, for vertex 0 in query $Q$, $topS(0) = [2.0, 0.0]$

LEMMA 3.2. *Given two graphs* Q *and* G*,* Q *is sub-graph isomorphism to graph* G *under the injective function* $g$*. For any vertex* $v$ *in* Q*, its vertex topology signature is* $topS(v) = [\lambda_1, .., \lambda_i, ..., \lambda_t]$*. There exists a vertex* $v'$ *in graph* G*, where* $v' = g(v)$*. For* $v'$*, its topology signature is* $topS(v') = [\beta_1, ..., \beta_i, ..., \beta_t]$*. We can say that* $topS(v) \leqslant topS(v')$*, where* $\leqslant$ *is overloaded here to stand the relationship between* $topS(v)$ *and* $topS(v')$*, i.e.,* $\lambda_i \leqslant \beta_i$ *for* $i = 1, ..., t$*.*

*Proof.* According to Lemma 3.1, $LNPT(Q, v, n)$ is an induced sub-tree of $LNPT(G, v', n)$. Based on Theorem 2.3, it is straightforward to know that $\lambda_i \leqslant \beta_i$ for $i = 1, ..., t$. □

In Figure 5, query $Q$ is sub-graph isomorphism of graph 003. Furthermore, vertex 0 in $Q$ is corresponding to vertex 0 in 003. Therefore, $topS(0) = [2.0, 0.0]$ in $Q$ is "$\leq$" $topS(0) = [2.10, 1.26]$ in graph 003. Furthermore, in order to handle matchings of vertex labels, we propose the following definition.

DEFINITION 3.2. ***Vertex Signature***. *Given a graph* G *and a vertex* v $\in$ G*, the vertex signature of* $v$ *is a triplet* $< \mathrm{L}, \mathrm{N}, topS(v) >$*, where* L *is a length-X counter string to denote the vertex label, and* N *is a length-X counter string to denote the neighbor labels,* $topS(v) = [\lambda_1, ..., \lambda_i, ...\lambda_t]$ *is its* vertex topology signature *(defined in Definition 3.1).*

In Definition 3.2, the $L$ and $N$ parts denote the vertex label and vertex neighbor labels respectively. For each distinct label, we use hash function to set $m$ out of $X$ elements to 1. According to the vertex v's label, we set the value of $L$. For $N$, N[j]=$\sum L_i[j]$, $L_i$ is the v's neighbor label and $N[j]$ is j-th element of $N$. Figure 6 shows an example of computing the signature of vertex 0 in graph 003.

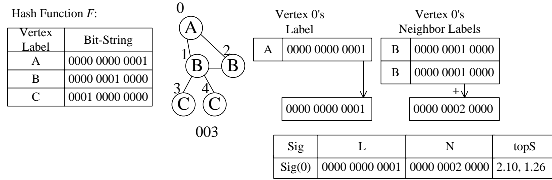



**Figure 6: Vertex Signature**

LEMMA 3.3. *Assume that a graph* Q *is sub-graph isomorphism to another graph* G *under the injective function* $g$*. For each vertex* v *in* Q*, its signature is* $< L_1, N_1, topS(v) >$*. In graph* $G$*, there is a vertex* $v'$*, where* $v'$=$g(v)$*. Its signature is* $< L_2, N_2, topS(v') >$*. The two vertex signatures satisfy the following conditions: 1)* $L_1$*[i] =* $L_2$*[i]; 2)* $N_1$*[i]* $\leqslant$ $N_2$*[i]; 3)* $topS(v) \leqslant topS(v')$*.*

*Proof.*(Sketch) a) According to sub-graph isomorphism, the labels of $v$ and $v'$ are the same with each other, i.e. $L_1[i] = L_2[i]$.
b) All neighbors of $v$ in graph $Q$ should be a *subset* of neighbors of $v'$ in graph $G$. According to the method about how to generate the counter string $N_1$ and $N_2$, it is straightforward to prove that condition 2) is true.
c) Condition 3) has been proved in Lemma 3.2. □

Given one vertex $v$ in graph $Q$ and one vertex $v'$ in $G$, if $sig(v)$ and $sig(v')$ satisfy three conditions in Lemma 3.3, we say $sig(v)$ is **compatible** to $sig(v')$. Based on Lemma 3.3, we can derive the necessary condition about a graph $G$ and its sub-graph $Q$, which is stated in following Theorem.

THEOREM 3.1. *Given a graph* $G$ *and its sub-graph* $Q$*, for each vertex* $v$ *in* $Q$*, we always can find a vertex* $v'$ *in* $G$*, where* $sig(v)$ *is* ***compatible*** *to* $sig(v')$*.*

*Proof.*Directly derived from Lemma 3.3. □

According to Theorem 3.1, if we can not find a $v'$ in graph $G$ that is compatible with a vertex $v$ in $Q$, we can conclude that $G$ does not contain $Q$ as its subgraph for sure. Therefore, we have the Pruning Rule 1 as follows.

**Pruning Rule 1.** Given two graphs $Q$ and $G$, for a vertex $v$ in $Q$, if we cannot find a vertex $v'$ in graph $G$, where $sig(v)$ is compatible to $sig(v')$, $Q$ cannot be sub-graph isomorphism to $G$.□

LEMMA 3.4. *Pruning Rule 1 satisfies* no-false-negative *requirement for sub-graph search problem.*

PROOF. (proof by contradiction) We assume that Pruning Rule 1 will lead to false-negative. Specifically, given a query $Q$, we assume the data graph $G^*$ is a correct result. However, $G^*$ is pruned by Pruning Rule 1. Namely, $G^*$ is a false negative.
Since $Q$ is a sub-graph of $G^*$, according to Theorem 3.1, for each vertex $v$ in query $Q$, there exists a compatible vertex $v'$ in the graph $G^*$. Therefore, according to Pruning Rule 1, $G^*$ will not be filtered out. It leads to contradiction to assumption. Thus, Lemma 3.4 is correct. □

The above pruning rule requires "vertex to vertex" comparison, which is computational expensive. Therefore, we need to design a more efficient pruning strategy for large graph databases, which is discussed in the next sub-section.

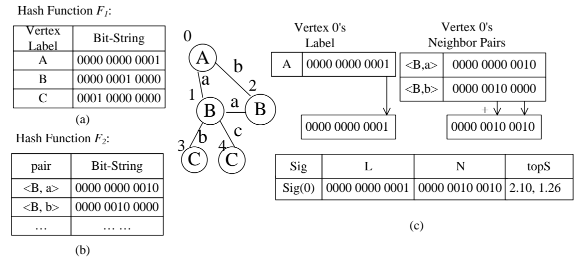



**Figure 7: Encoding Edge Label**

In experiments, we will handle vertex-labeled and edge-labeled graphs. In order to consider edge label, we illustrate the method by Figure 7.For a vertex $v$ in a graph $G$, we denote the neighbor vertex label and adjacent edge label as a pair $< vL, eL >$, where $vL$ is the neighbor vertex label and $eL$ is the corresponding adjacent edge label. For example, the vertex 0 in Figure 7 has two pairs, that are $< B, a >$ and $< B, b >$. We encode the pairs into the $N$ part of the vertex signature $Sig(0)$. Specifically, for each distinct pair, we also use the hash function to set $m$ out of $X$ elements to '1', as shown in Figure 7b. Then we set $N$ part of $Sig(0)$ by element-wise "ADD" operations. In fact, the method in

encoding adjacent edge labels is the same with that in encoding neighbor vertex labels. Furthermore, in order to obtain $L$ part of $Sig(0)$, like in Figure 6, we also map each vertex label into a hash value in Figure 7a. The $Sig(0)$ is shown in Figure 7c. It is also straightforward to know Lemma 3.3 and 3.4 and Theorem 3.1 still hold. For illustration convenience, we do not consider the adjacent edge labels in the following discussion.

## 3.2 Graph Code

Benefiting from the vertex signature, the local structure around a vertex has been mapped into the numerical space. In this subsection, we propose the graph coding technique by combining all the vertex signatures of a graph $G$, which maps the global structure of $G$ into the numerical space. In the new pruning strategy, we only need to compare two graph codes instead of the "vertex to vertex" comparison.

Graph 003 :

| VertexID | L | N | topS | |
|---|---|---|---|---|
| | | | $\lambda_1$ | $\lambda_2$ |
| 0 | 000000000001 | 000000020000 | 2.10 | 1.26 |
| 1 | 000000010000 | 000200010001 | 2.14 | 1.00 |
| 2 | 000000010000 | 000000010001 | 2.10 | 1.26 |
| 3 | 000100000000 | 000000010000 | 2.00 | 0.00 |
| 4 | 000100000000 | 000000010000 | 2.00 | 0.00 |

| GCode(003) | 000200020001 | 000200060002 | Seq1 | Seq2 |
|---|---|---|---|---|

| | |
|---|---|
| Seq1 | 2.14, 2.10, 2.10, 2.00, 200 |
| Seq2 | 1.26,1.26, 1.00, 0.00, 0.00 |

**Figure 8: Graph Code of graph 003**

Figure 8 shows the example of building graph code for graph 003 (that is GCode(003)). First, we compute all vertex signatures of graph 003. We also assume that we only consider the first two largest eigenvalues (that are $\lambda_1$ and $\lambda_2$) when computing vertex signatures. $GCode(003) =< L, N, Seq_1, Seq_2 >$. To obtain the $L$ (and $N$) part of $GCode(003)$, we combine the $L$ (and $N$) part of all vertex signatures by element-wise "ADD" operations. All $\lambda_1$ (and $\lambda_2$) are collected and ranked to form a non-ascending sequence $Seq_1$ (and $Seq_2$). Formally, we have the following definition about graph code.

DEFINITION 3.3. ***Graph Code*** (GCode for short) *The code of a graph $G$ with n vertices is a triplet, $GCode(G) =< \mathrm{L}, \mathrm{N}, SpecSeq(G) >$, where* L *and* N *are counter strings and $SpecSeq(G) = [Seq_1, Seq_2, ..., Seq_t]$ is a sequence of eigenvalue lists. Assume that graph* G *has* n *vertices, and each vertex $v_j$ is denoted as $sig(v_j) = < L_j, N_j, topS(v_j) = [\lambda_{j1}, .., \lambda_{jt}] >$, we have:*
*1) $L[i] = \sum_{j=0}^{j=n-1} L_j[i]$, where* L$[i]$ *is the i-th counter of* L.
*2)$N[i] = \sum_{j=0}^{j=n-1} N_j[i]$, where* N$[i]$ *is the i-th counter of* N.
*3) For all vertex topology signatures $topS(v_j)$, all $\lambda_{jk}$ are ranked according to non-ascending order to form the sorted list $Seq_k$, where $k = 1, ..., t$.*

All graph codes of running example are shown in Figure 9. Based on the Definition of 3.3, we have the following Theorem.

| | L | N | Seq1 * | Seq2 |
|---|---|---|---|---|
| Gcode(001) | 000200020001 | 000200050001 | 1.93, 1.90, 1.90, 1.73, 1.73 | 1.18, 1.18, 1.00, 1.00, 0.00 |
| Gcode(002) | 000100010003 | 000200010007 | 2.05, 2.05, 2.00, 1.73, 1.73 | 1.41, 1.21, 1.21, 0.00, 0.00 |
| Gcode(003) | 000200020001 | 000200060002 | 2.14, 2.10, 2.10, 2.00, 200 | 1.26,1.26, 1.00, 0.00, 0.00 |
| Gcode(004) | 000100010002 | 000200020004 | 1.73, 1.73, 1.73, 1.73 | 1.00, 1.00, 1.00, 1.00 |
| Gcode(Q) | 000200020001 | 000200050001 | 2.00, 2.00, 2.00, 2.00, 2.00 | 0.00, 0.00, 0.00, 0.00, 0.00 |

Note: Seq1 * : For each graph G, the length of Seq1 (or Seq2 ) is equal to vertex number of G.

**Figure 9: Graph Code Database**

THEOREM 3.2. *Given two graphs* Q *with $n_1$ vertices and* G *with $n_2$ vertices, where $n_1 \leq n_2$, their GCodes are denoted as $GCode(Q) =< QL, QN, [Seq_1(Q), Seq_2(Q), ..., Seq_t(Q)]>$ and $GCode(G) = < GL, GN, [Seq_1(G), Seq_2(G), ..., Seq_t(G)] >$. If* Q *is sub-graph isomorphism to graph* G*, $GCode(Q)$ and $GCode(G)$ satisfy the following conditions:*
*1)* QL$[i] \leq$ GL$[i]$*, where* QL$[i]$ *is i-th elment of* QL*;*
*2)* QN$[i] \leq$ GN$[i]$*, where* QN$[i]$ *is i-th element of* QN*;*
*3) $Seq_k(Q)[j] \leq Seq_k(G)[j]$, $k = 1, ..., t$ and $j = 0...n_1 - 1$.*

*Proof.* Since graph $Q$ is sub-graph isomorphism to graph $G$, it is necessary that each vertex $v$ in $Q$ has a corresponding vertex $u = g(v)$ in $G$ under some injective function $g$. In the following analysis, assume that $sig(v) = < vL, vN, topS(v) >$ and $sig(u) = < uL, uN, topS(u) >$.
a) According to Lemma 3.3, we know that $vL[i] = uL[i]$. Since the function $g$ is an injective function from vertices in $Q$ to vertices in $G$, it means that $\sum_{j_1=0}^{j_1=n_1-1} vL_{j_1} \leq \sum_{j_2=0}^{j_2=n_2-1} vL_{j_2}$
Therefore, $QL[i] \leq GL[i]$.
b) According to Lemma 3.3, we know that $vN[i] \leq uN[i]$. Since the function $f$ is an injective function from vertices in $Q$ to vertices in $G$, it means that $\sum_{j_1=0}^{j_1=n_1-1} vN_{j_1} \leq \sum_{j_2=0}^{j_2=n_2-1} vN_{j_2}$.
Therefore, $QN[i] \leq GN[i]$.
c) We prove the 3) by *contradiction.*
Assume that 3) does not hold, that is $SpecSeq(Q)_k[j] > SpecSeq(G)_k[j]$. $SpecSeq(Q)_k$ is a sorted list in a non-ascending order, therefore, $SpecSeq(Q)_k[i] \geq SpecSeq(G)_k[j]$, where $i = 0...j$. It means that there exist *j+1* vertices $v_i$ in graph $Q$ ($i$=0...j), whose $\lambda_k$ are larger than $SpecSeq_k(G)[j]$. Since graph $Q$ is sub-graph isomorphism to graph $G$, for each vertex $v_i$ in $Q$, it has a corresponding vertex $u_i$. According to condition 3) in Lemma 3.3, the $\lambda_k$ of vertex $v_i$ is no larger than that of vertex $u_i$. Thus, there must exist *j+1* vertices $u_i$ in graph $G$, whose $\lambda_k$ are larger than $SpecSeq(G)_k[j]$.
On the other hand, as we know, $SpecSeq(G)_k$ is also a non-ascending sorted list and $SpecSeq(G)_k[j]$ is the j-th largest one, which indicates that there exist at most $j$ vertices $u_i$ ($i$=0...j-1), whose $\lambda_k$ are no smaller than $SpecSeq(G)_k[j]$. This contradicts to the above conclusion. Therefore, condition 3) is correct. □

For example, in Figure 9, $Seq_1(Q)[0]$=2.00 > $Seq_1(001)[0]$ =1.93. Therefore, graph 001 is pruned safely, since $GCode(Q)$ and $GCode(001)$ cannot satisfy the third condition in Theorem 3.2. Similarly, graph 002 and 004 are also filtered out. Only graph 003 is left as a candidate, since $GCode(Q)$ and $GCode(001)$ satisfy all three conditions in Theorem 3.2. Actually, Theorem 3.2 is the necessary condition about $GCode(Q)$ and $GCode(G)$, if $Q$ is a sub-graph of $G$. Thus,

we have the following Pruning Rule 2.

**Pruning Rule 2.** Given two graphs $Q$ with $n_1$ vertices and $G$ with $n_2$ vertices ($n_1 \leq n_2$), their GCodes are denoted as $GCode(Q) =< QL, QN, [Seq_1(Q), Seq_2(Q), ..., Seq_t(Q)] >$, $GCode(G) =< GL, GN, [Seq_1(G), Seq_2(G), ..., Seq_t(G)] >$. If $GCode(Q)$ and $GCode(G)$ cannot satisfy the following three conditions, graph $Q$ cannot be sub-graph isomorphism to graph $G$.
1) $QL[i] \leq GL[i]$, where $QL[i]$ is i-th element of $QL$;
2) $QN[i] \leq GN[i]$, where $QN[i]$is i-th element of $QN$;
3) $Seq_k(Q)[j] \leq Seq_k(G)[j]$, k=1...m and j = 0...$n_1 - 1$. □

LEMMA 3.5. *Pruning Rule 2 satisfies* no-false-negative *requirement for sub-graph search problem.*

PROOF. Similar with Lemma 3.4, it can be proved according to Theorem 3.2 by contradiction. □

# 4. SUBGRAPH SEARCH

In GCoding, there are two processes: *offline* and *online*. In offline process, we compute graph code database. In online process, we conduct sub-graph search over graph code databases. We explain the details about two processes as follows.

In the offline process, for each graph $G_i$ in a graph database, we compute all vertex signatures in $G_i$ and combine them to obtain GCode based on the discussion in the above section. All graph codes of the running example is shown in Figure 9. Since the same vertex signature may be shared in different graphs, we build a vertex signature dictionary to store all distinct vertex signatures. For each graph ID, it has a list of pairs $< signatureID, count >$, where $signatureID$ is a pointer to some vertex signature in the dictionary, and $count$ denotes the number of this vertex signature in the graph. Note that, graph code database and vertex signature dictionary are easy for update maintenance. We only need to compute the vertex signatures and GCode for the inserted (or deleted) data graphs, and then insert them into (or delete them from) graph code database and vertex signature dictionary.

In the online process, given a query graph $Q$, we convert it to its GCode and use $GCode(Q)$ to search over the graph code database to filter false positive as many as possible before we conduct the expensive subgraph isomorphism checking. We formulate the online framework of sub-graph search in GCoding as follows:

1. We compute the vertex signatures and graph code of query $Q$.

2. In the filtering phase, we use pruning rules discussed in Section 3 to remove unqualified graphs, Rule 2 first then Rule 1 as shown in Figure 10.

3. To obtain the final results, we perform sub-graph isomorphism for each candidate.

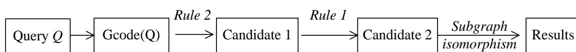



**Figure 10: Online Processing**

It is clear that using Rule 2 is much more efficient than that of Rule 1. Therefore, we designed an online framework shown in Figure 10, which first uses Rule 2 to prune most false positives in the 1st filter process and then use Rule 1 to filter out more false positives in the 2nd filter process.

Given a query graph $Q$ and its graph code $GCode(Q)$, using Rule 2, we can conduct pairwise comparison between $GCode(Q)$ and each $GCode(G_i)$ in the graph code database. Though the pairwise comparison between $GCode(Q)$ and $GCode(G_i)$ is not an expensive task, linear scan is not scalable for large graph databases. Therefore, an efficient index structure is needed for reducing the number of pairwise comparisons.

Similar to S-tree [16] for indexing the signature files, we develop our GCode-Tree. For each $GCode(G_i) =< L_i, N_i, SepcSeq(G_i) >$, we extract the first two elements, i.e. $L$ and $N$, to build a GCode-Tree. GCode-Tree is a balanced tree, where each node has at least $m$ children ($m \leq 2$), and at most M children ((M+1)/2 $\geq$ m). Assume that the intermedin node (directory node) $I$ in GCode-Tree has $k$ child nodes $C_i$, we set $I.L[j]$ = Max($C_i.L[j]$), $i$=1...$k$, where $I.L[j]$ is the j-th counter of $I.L$. Similarly, we set $I.N[j]$ = Max($C_i.N[j]$). The insertion and deletion operations of GCode-Tree are analogous to those of S-tree, we omit the details about these dynamic operations. For the running example given in Figure 1, the corresponding GCode-Tree is shown in Figure 11.

Given a query graph $Q$ and its $GCode$, $GCode(Q) =< L, N, SpecSeq(Q) >$, we start traversing the GCode-tree from the root. For each intermedin node $I =< L, N >$, if there exists some $i$, where $GCode(Q).L[i] > I.L[i]$ or $GCode(Q).N[i] > I.N[i]$, all descendants of $I$ can be pruned safely. For example, given the query graph $Q$, $GCode(Q)$ shown in Figure 9, for the node $I_1$ in GCode-Tree, $GCode(Q)$ $.L[3] = 2 > I_1.L[3] = 1$, all descendants of $I_1$ are not in the result set. It means that graph 001 and 002 can be pruned safely. Therefore, by scanning the GCode-Tree, we can prune all graph codes that do not satisfy the condition 1) or 2) of Pruning Rule 2. For each remaining graph code $GCode(G_i)$, we check whether $GCode(Q)$ and $GCode(G_i)$ satisfy the condition 3) of Rule 2. Then, in the second step of the filtering process, we perform "vertex-to-vertex" comparison filtering based on Rule 1.

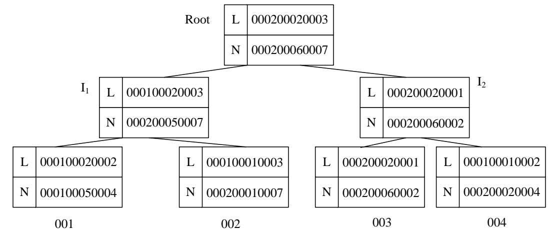



**Figure 11: GCode-Tree**

# 5. EXPERIMENTS

In this section, we evaluate the performance of our method, i.e. GCoding, for sub-graph search. gIndex [22], Closure-Tree [7], FG-Index [2] and Tree+$\delta$ [25] are chosen to compare with our methods. Our methods are implemented in standard C++ with STL library support and compiled with

gcc/g++. All experiments are done on a P4 1.7GHz machine of 1G RAM running Linux.

## 5.1 Datasets and Setting

In experiments, we consider vertex-labeled and edge-labeled graphs. Because the implementation of Closure-tree does not handle edge labels, we use the same technique mentioned by [3], that is to insert an additional vertex for each edge to encode this information. Since the edge and vertex labels were drawn from disjoint sets, there could be no ambiguity between edges and vertices. This enables a performance comparison.

1) **AIDS Antiviral Screen Dataset** This dataset is available publicly on the website of the Developmental Therapeutics Program [2]. We generate 10,000 connected and labeled graphs from the molecule structures and omit Hydrogen atoms. The graphs have an average number of 24.80 vertices and 26.80 edges, and a maximum number of 214 vertices and 217 edges. A major portion of the vertices are C, O and N. The total number of distinct vertex labels is 62, and the total number of distinct edge labels is 3. We refer to this dataset as AIDS dataset. Each query set $Qm$ has 1000 connected query graphs and query graphs in $Qm$ are connected size-$m$ graphs (the edge number in each query is $m$), which are extracted from some data graphs randomly, such as $Q4$, $Q8$, $Q12$, $Q16$, $Q20$ and $Q24$.

2) **Large Graph Database** We download the "NCI 127K Connection Tables " dataset from the same web site with AIDS dataset. The dataset contains about 127K compounds. We randomly choose 100,000 compounds to form the large graph database. The graphs have an average number of 18.6 vertices and 19.7 edges, and a maximum number of 100 vertices and 111 edges. The total number of distinct vertex labels and edge labels are 77 and 3 respectively. We refer it as NCI dataset. Using the same method in AIDS dataset, we generate three query sets, that are Q25, Q20 and Q15. Each query set has 1,000 query graphs.

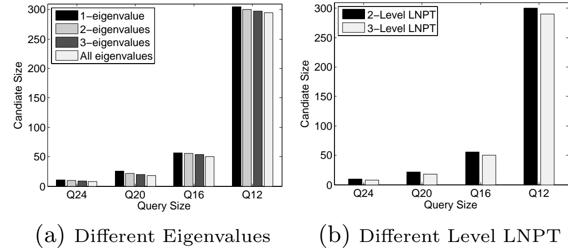
3) **Synthetic Dataset** The synthetic dataset is generated by a synthetic graph generator provided by authors of [12]. The synthetic graph dataset is generated as follows: First, a set of $S$ seed fragments are generated randomly, whose size is determined by a Poisson distribution with mean $I$. The size of each graph is a Poisson random variable with mean $T$. Seed fragments are then randomly selected and inserted into a graph one by one until the graph reaches its size. Parameter $V$ and $E$ denote the number of distinct vertex labels and edge labels respectively. The cardinality of the graph dataset is denoted by $D$. We generate the graph database using the same parameters with gIndex in [22]: D=10,000, S=200, I=10, T=50, V=4, E=1. The large synthetic dataset, used in scalability test in subsection 5.3.5, has all the same parameters except for D=100,000.

In gIndex, Closure-tree, FG-Index and Tree+$\delta$ algorithms, we choose the default or the suggested values for parameters according to [22, 7, 2, 25].

In GCoding, for vertex signature (see Definition 3.2) and GCode (see Definition 3.3), the $L$ and $N$ part are the hash value of vertex label and neighbor labels. The longer the $L$ and $N$ are, the less conflicts among hash values. On the other hand, it is straightforward to know that longer $L$ and $N$ means lager space cost. Eventually, the $L$ and $N$ part are analogous to signature file [11], which are used to denote the set of elements. In order to obtain a tradeoff between hash conflicts and space cost, we use the analysis in [11] to guide setting the length of $L$ and $N$. In experiments, we set $|L|$=30 and $|N|$=30.

As we know, in definitions about the vertex signature and GCode (see Definition 3.2 and 3.3), there are two important parameters, i.e. $n$ and $t$. $n$ means using $n$-level LNPT, and $t$ means that we always choose the $t$ largest eigenvalues. We evaluate the effects of $n$ and $t$ in the following experiments.

## 5.2 Effect of $n$ and $t$




(a) Different Eigenvalues (b) Different Level LNPT

**Figure 12: The Pruning Power at (a) Different Eigenvalues and (b) Different Level LNPT in AIDS Dataset**

Figure 12(a) shows the performance in different eigenvalues. Observed from the Figure 12(a), we find that choosing three or more eigenvalues cannot lead to more improvement in pruning power. Moreover, choosing more eigenvalues means the larger graph code database. Therefore, we always choose the two largest eigenvalues, i.e. $\lambda_1$ and $\lambda_2$ in our experiments.

As we know, $LNPT(G, v, 1)$ is the level-1 path tree around vertex $v$. Actually, all nodes in $LNPT(G, v, 1)$ are all $v$'s neighbor vertices. The 1st level $LNPT$ information has been coded into the "$N$" part of vertex signatures and graph codes. Therefore, we should use 2 or more levels $LNPT$. However, choosing more levels means that the corresponding matrix is larger, and it needs more time to compute eigenvalues for vertex signatures and graph codes. As show in Figure 12(b), choosing 3-levels does not lead to significant improvement in pruning power. Therefore, we use 2-level $LNPT$ in GCoding in our experiments.

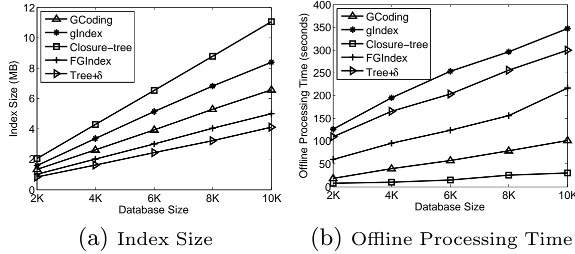
## 5.3 Performance Study

### 5.3.1 Cost of Offline Processing

We first evaluate the performance of GCoding in the offline process, such as the size of "graph code database together with Indexes" and the processing time of generating GCode for all the graphs.

Figure 13(a) shows the index size in different methods. For the GCoding method, we refer to "graph code database together with Indexes" as "index". With the increase of sizes of datasets, from 2K to 10K, the index size of GCoding is smaller than gIndex, Closure-Tree and lager than FGIndex and Tree+$\delta$. In experiments, our index file of GCoding is stored as "ASCII" format. The index size of GCoding can be optimized by "binary" format file. The index file of gIndex is stored in "ASCII" file. The index files in FGIndex and GCoding are stored in "binary" file. Figure 13(b) shows the offline processing time in different methods. Closure-tree is the fastest. Our methods is slower than Closure-tree. In GCoding, in order to compute the eigenvalues of adjacency-matrix, we implement Jacobi method [5], which is an itera-

[2] $http://dtp.nci.nih.gov/$




(a) Index Size (b) Offline Processing Time

**Figure 13: (a) Index Size and (b) Offline Processing Time in AIDS dataset**

**Table 1: Evaluating Multi-step Pruning Strategy of GCoding in AIDS dataset**

| | Q24[1] | | Q20 | | Q16 | | Q12 | | Q8 | | Q4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Cand.[2] | Time[3] (sec.) | Cand. | Time | Cand. | Time | Cand. | Time | Cand. | Time | Cand. | Time |
| Compute Graph Codes | | 16.20 | | 12.89 | | 9.59 | | 6.42 | | 3.98 | | 1.75 |
| 1-st filtering | 46.30 | 2.30 | 96 | 3.15 | 293 | 5.30 | 1033 | 6.10 | 2511.1 | 10.50 | 5527 | 25.80 |
| 2-nd filtering | 9.00 | 9.91 | 24.67 | 13.62 | 55.87 | 29.79 | 271.29 | 47.58 | 916.98 | 55.82 | 3315 | 57.55 |
| Total filtering time | | 28.41 | | 29.66 | | 44.68 | | 60.10 | | 70.30 | | 85.10 |
| Total Response Time | 3.1 | 41.81 | 6.3 | 48.64 | 12.0 | 106.80 | 18.1 | 234 | 157.9 | 496.10 | 2254 | 823 |

Note : Q24[1] : The edge number of Q24 is 24.
Cand.[2] : Candidate Size ;
Time [3] : All running time reported in this section are the total running time in 1000 queries in each query set.

tive algorithm for computing the eigenvalues. According to our experiments, on average, we need to about $0.008 \sim 0.010$ second to compute the vertex signatures and graph code for each graph. gIndex is much slower than Closure-tree and our method, since it needs expensive mining process to find some discriminative fragments. Actually, other data mining based methods have the similar limitations, since they all need to perform expensive frequent sub-graphs (or subtrees) mining algorithms, such as FGIndex and Tree+$\delta$.

### *5.3.2 Efficiency of Two Step Filter*

As we know, according to the online process framework in Figure 10, in order to answer a sub-graph query, we need to compute GCode for each query graph, perform two step filtering phases and do sub-graph isomorphism test in the verification process. In this experiment, we test the efficiency of two step filtering. Table 1 shows the running time of each step and pruning power of each filtering step. Note that all running time (including Filtering Time and Total Response Time) reported in this section are the running time of 1000 queries in each query set. For the left candidate graphs, we implement ULLMANN algorithm [17] to perform sub-graph isomorphism test.

In the first filtering step, we use Filtering Rule 2 in Section 3.2 by comparing GCodes to filter out the unqualified graphs. Observed from Table 1, the first step is the fastest, which prunes most false positives. After that, we compare the vertex signatures in query graph and data graphs by Filtering Rule 1. The filtering time in the second step is slower than that in the first step. However, it is clear to know the filtering time in the 2nd-Step pruning phase is faster than sub-graph isomorphism test in verification process. Furthermore, the candidate size after the 2nd-step filtering is less than $\frac{1}{4}$ of that after the 1st-step filtering. Therefore, in order to obtain fast total response time, it is worth performing 2-step filtering phase in GCoding method.

### *5.3.3 Performance Comparison with Related Work*

We compare the pruning power, the filtering time and total response time with the existing counterpart methods on both AIDS and synthetic datasets.

In the software of FGIndex provided by authors in [2], the filtering process and verification process are implemented together. It does not report the candidate size and filtering time. In order to obtain candidate size in FGIndex, we work as follows. According to [2], frequent sub-graphs and edges are chosen as indexed features to build the inverted index. Some of them (that are $\delta$-TCFG) are maintained in memory to build core-FGIndex (First Level), and the others are used to build disk-resident FGIndex (Second Level). It is clear to know that, the pruning power of FGIndex are only derived from all indexed frequent sub-graphs and edges, including the first level and the second level. Therefore, in our experiments, according to [2], we mine all frequent sub-graphs by the software gSpan [20], and set minimum support to be 0.1 (suggested value in [2]). Then, we build the simple inverted index for all frequent sub-graphs and edges. When a query $Q$ is given, we can fix the candidates by scanning the simple inverted index. The candidate size in the straightforward method must be equal to that in FGIndex, since they use the same indexed features. However, the filtering time in FGIndex is much quicker than that in the straightforward method, since it optimizes the cost of scanning the inverted index by two levels of FGIndex. To be fair, we omit the comparison with filtering time of FGIndex.

As we all know, sub-graph search contains of two processes: *filtering* and *verification*. The total response time includes filtering time and verification time. In our experiments, we implement ULLMANN algorithm to implement sub-graph isomorphism algorithm in verification process. In fact, different sub-graph isomorphism algorithms affect the total response time greatly. Since all sub-graph search methods employ the *filter-and-verification* framework [3], therefore, to evaluate the pruning power of each method, the candidate size should make more sense than total response time.

Observed from Figure 15(a), we can find that GCoding has the largest pruning power in all query sets. In fact, it is really hard to prove that GCoding can get the largest pruning power from theoretical aspect. Nonetheless, we can still provide some theoretical analysis. First, in practice, data graphs in the database are always sparse, which is also mentioned in [25]. In a sparse graph, the local structure around each vertex is tree-like structure, whose structural information can be captured by $LNPT$ around the vertex. Second, in GCoding, we consider all local structural information around each vertex. However, in gIndex, FGIndex and Tree+$\delta$, for a graph $G$, its structural information is only captured by some features that are included in $G$. In other words, some local structure in $G$ that does not includes features are not considered. For example, in Figure 14, in GCoding, we consider all local structure around each vertex in $G$. However, in feature-based methods, we only consider the shaded part of $G$, since the non-shaded part does not include "features". Therefore, compared with feature-based methods, GCoding consider more structural information.

Due to expensive structure comparison and maximal match-

[3]In FGIndex, when query $Q$ is a frequent sub-graph, it does not need verification process, which will be discussed later.

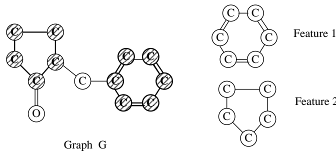



**Figure 14: Local Structure in Feature-Based Approaches**

ing algorithm in the filtering phase, the filtering time of Closure-tree is the slowest, which is about $5 \sim 10$ times than that in gIndex and GCoding method in Figures 15(c). In GCoding method, the filtering time in $Q24$ is faster than $Q4$, since there are more false positives in $Q24$ that are pruned in the first filtering step than that in $Q4$. As we know, the filtering time in the first filtering step is less than that of the second filtering step, which is also confirmed in Table 1. In gIndex, there are less "indexed features" in $Q4$ than that in $Q24$. It means that scanning inverted index consumes less time in $Q4$ than that in $Q24$. Therefore, the filtering time in $Q4$ is faster than that in $Q24$ in gIndex method. Due to the same reason, in Tree+$\delta$, the filtering time in $Q4$ is faster than that in $Q24$.

In FGIndex, if the query $Q$ is a frequent sub-graph, namely, $Q$ is isomorphism to an indexed features, it can report the answer set without verification process. Therefore, it can obtain the fast response time when $Q$ is a frequent sub-graph. However, there are few queries that are frequent sub-graphs, especially when queries are large graphs. Thus, only in $Q4$, the response time in FGIndex is smaller than that in GCoding. In other query sets, since the candidate size in FGIndex is larger than that in GCoding, the response time in FGIndex is larger than that in GCoding, as shown in Figure 15(e). GCoding is also faster than gIndex, Closure-tree and Tree+$\delta$ in total response time, since GCoding has the least candidate size in each query set.

We also test GCoding in synthetic datasets together with other methods. The performance of GCoding is similar with that in AIDS dataset, which outperforms other methods in most cases.

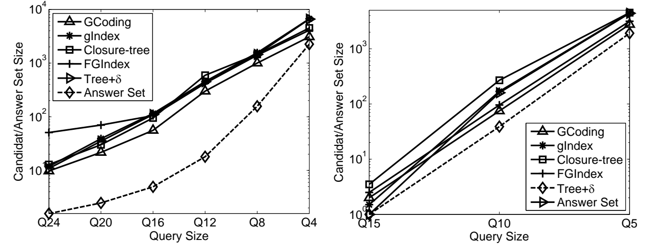
### 5.3.4 Performance on Incremental Maintenance

In Section 1, we discussed that feature-based graph indexing methods (state art of graph indexing methods) may degrade their pruning power over time in dynamic setting of graph database. In this experiment, we test the performance of GCoding over dynamic datasets, and compare it with gIndex (the classical feature-based graph indexing method).

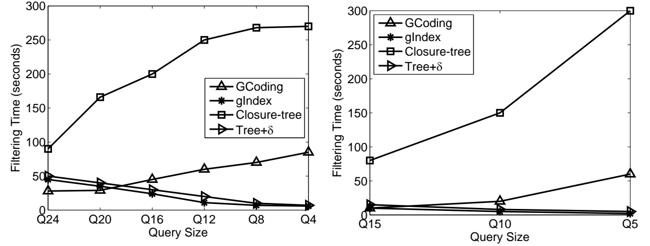
As discussed in Section 4, to handle insertions, we only need to compute GCode and vertex signatures for the new inserted graphs. Then we insert them into the graph code database. According to [22], there are two kinds of methods to handle insertions in gIndex: 1)update the id lists of involved fragments; or 2) re-build the index in batch from scratch.

In experiments, we vary the database size from 2K to 10K, and we choose $Q16$ as query sets. Figure 16(a) shows the pruning power of each method, where pruning power is defined as $\frac{DBSize - CandidateSize}{DBSize - \text{ResultSize}}$. Observed from Figure 16(a), the pruning power of GCoding is stable irrespective to database updates.

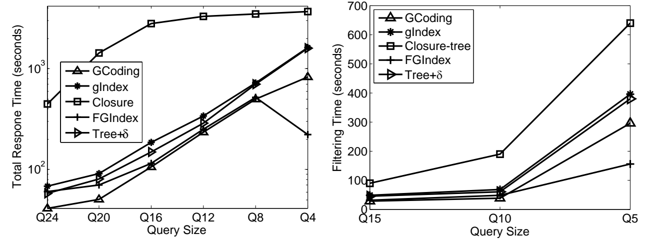
For gIndex, we use two methods to handle insertion. First,




(a) Candidate Size in AIDS Dataset (b) Candidate Size in Synthetic Dataset



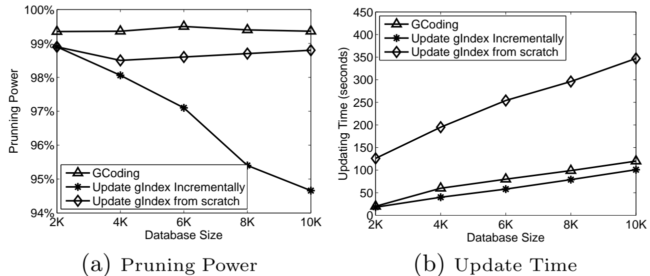

(c) Filtering Time in AIDS Dataset * (d) Filtering Time in Synthetic Dataset *




(e) Total Response Time in AIDS Dataset (f) Total Response Time in Synthetic Dataset

* We omit the comparison with FGIndex in Filtering time. The reason is explained in Section 5.3.3

**Figure 15: Evaluating Online Performance**




(a) Pruning Power (b) Update Time

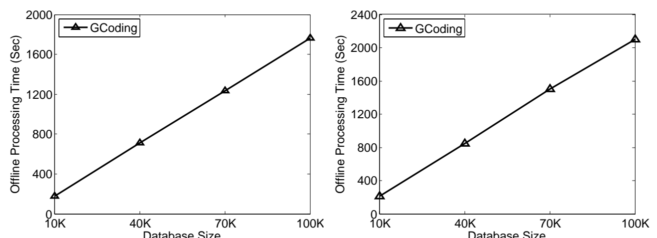
**Figure 16: Evaluating Dynamic Performance**

we build the inverted index on 2K dataset and then update the id lists of involved fragments on other datasets. We refer the method as "update gIndex Incrementally" in Figure 16. Second, we build the inverted index for each dataset from scratch. We refer the method as "update gIndex from scratch" in Figure 16.

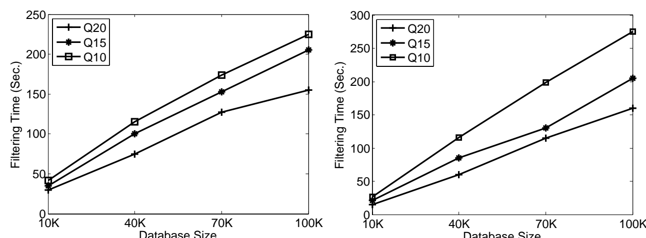
Observed from Figure 16, the pruning power in "update gIndex from scratch" is more stable than "update gIndex Incrementally". However, it is more expensive for "update gIndex from scratch" to handle insertions, which is about 3∼5 times than GCoding update time in Figure 16(b). The pruning power of 'update gIndex Incrementally" fall down

about 5% during insertion in Figure 16(a). In fact, other feature-based methods have the same problems in dynamic setting. Therefore, we can say that, in dynamic setting, feature-based methods is either ineffective ('update index Incrementally") or inefficient ('update index from scratch").

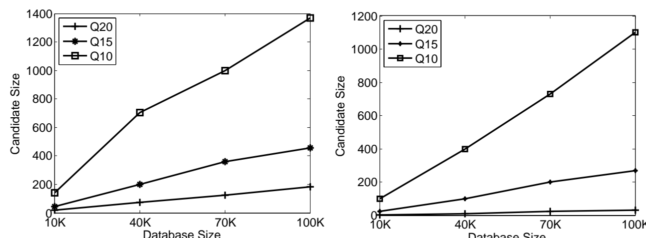
We can also find from Figure 16 that GCoding method is more suitable for dynamic graph database. First, its pruning power is stable for dynamic setting; Second, the update time is fast.




(a) Offline Processing Time in NCI dataset (b) Offline Processing Time in Large Synthetic dataset




(c) Filtering Time in NCI dataset (d) Filtering Time in Large Synthetic dataset




(e) Candidate Size in NCI dataset (f) Candidate Size in Large Synthetic dataset

**Figure 17: Scalability of GCoding**

### 5.3.5 *Scalability Test*

In order to test the scalability of the GCoding, we use two large graph databases, i.e. the NCI dataset and a large synthetic dataset, and run $Q10$, $Q15$ and $Q20$ on these two datasets, respectively. The cardinalities in both datasets are 100K respectively. From Figure 17, we find the GCoding scale well on both offline and online processing. We find that compared to $Q20$ and $Q15$, the candidate sizes of $Q10$ change faster. This is because the number of qualified results for $Q10$ is much larger than those for $Q20$ or $Q15$.

## 6. RELATED WORK

As a popular data structure, graphs have been used to model many complex data objects in real world, such as, chemical compounds [19] [9], entities in images [13], XML documents[23] and social networks [1]. Due to the wide usage of graphs, it is quite important to provide users to organize, access and analyze graph data efficiently. Therefore, graph database has attracted a lot of attentions from database and data mining community, such as sub-graph search [14, 22, 7, 24, 6, 3, 2, 25], frequent sub-graph mining [8, 12, 21] and correlation sub-graph query [10].

Among many graph-based applications, it is quite important to retrieve related graphs containing a query graph from a large graph database efficiently, which is called *sub-graph search* problem. As we all know, sub-graph isomorphism is NP-complete problem [4]. Usually, to speed up the sub-graph search, we use the filter-and-verification framework. First, we remove false positives (graphs that are not possible in the results) by *pruning strategy*; Then, we perform sub-graph isomorphism algorithm on each candidate to obtain the final results. Obviously, less candidates mean better search performance. So far, there are many pruning strategies have been proposed [14, 22, 7, 24, 6, 3, 2, 25]. Basically, they can divided into two categories:

1) *Data-mining based filtering*. The approaches in this category apply data mining techniques to extract some discriminate sub-structures as indexed features, then, build inverted index for each feature. Query graph $Q$ is denoted as a set of features, and the pruning power always depends on selected features. By the inverted indexes, we can fix the complete set of candidates. Many algorithms have been proposed to improve the effectiveness of selected features, such as gIndex [22], TreePi [24], FG-Index [2] and Tree+$\delta$[25]. In gIndex, Yan et al. propose a "discriminative ratio" for features. Only frequent and discriminative subgraphs are chosen as indexed features. In TreePi, due to manipulation efficiency of trees, Zhang et al. propose to use frequent and discriminate subtrees as feature set. In FGIndex, Cheng et al. use frequent sub-graphs and edges as indexed features. In Tree+$\delta$, Zhao et al. use frequent free trees and a small number of discriminative sub-graphs as indexed features. Usually, on static graph databases, data mining methods always find "good" features, which leads to good pruning power [22, 24, 2, 25]. However, they have some intrinsic limitations. For example, they assume that the database is static, or at least statistics of the graph database do not change. In practice, the frequent updates to the graph database do exist. For example, approximate 4,000 new compound structures are added every data, according to the report by the SCI Finder. In pattern recognition, the new recognized graph patterns are always inserted into the graph pattern database. In order to handle updates, data mining-based filtering has two methods, as suggested in [22]: 1) update the id lists of involved features; 2) re-do feature selection and re-build the index from scratch. The former method cannot guarantee the effectiveness, namely, pruning power will degrade. We have evaluated it in Section 5. The latter method is time consuming. In practice, it is difficult for us to re-do feature selection and then re-build index from scratch.

2) *Non-data mining based filtering*. Different from the first category, no data mining-based feature selections are needed in this category. There are two representative algorithms, GraphGrep [14] and Closure-tree [7]. In GraphGrep, Shasha et al. propose to use all pathes up to $maxL$ length as index features. Similarly, GraphGrep also builds inverted index for each path. Different from the first category, we enumerate all pathes up to $maxL$ length, and no feature selection are needed. Since paths are less discriminative than sub-

structures, GraphGrep has less pruning power. In Closure-tree, He and Singh propose pseudo subgraph isomorphism by checking the existence of a semi-perfect matching from vertices in query graph to vertices a data graph (or graph closure). The major limitation of Closure-tree is the high cost in filtering phase, since Closure-tree needs to perform expensive structure comparison and maximal matching algorithm in filtering phase. From our performance study, even though we set pseudo compatibility level to 1 (default value in Closure-tree software), the filtering time of Closure-tree is 5∼10 times higher than that in GCoding. In GCoding, we only need to perform "cheap" numerical operations.

There are some other interesting recent work in graph search problem, such as [3] [6]. In [3], Williams et al. propose to enumerate all connected induced subgraphs in the graph database, and organize them in a Graph Decomposition Index (GDI). It is difficult for the method to work in a graph database with large-size graphs, due to combination explosion of enumerating all connected induced subgraphs. Jiang et al. propose gString [6] for compound database. It is not straightforward to extent gString to graph database in other applications. However, the framework of GCoding can be extended to many other graph-based applications, such as pattern recognition and protein structure analysis.

In XML database, Zhang et al. in [23] propose a pruning strategy based on Interlacing Theorem. In [15], based on spectral coding method, Shokoufandeh et al. discuss similarity search in a tree database. More difficult than tree database, Interlacing Theorem cannot always hold between a graph and its sub-graph. The main contribution of GCoding is that, we use Interlacing Theorem to conduct sub-graph search with no positive dismissal.

# 7. CONCLUSIONS

In this paper, in order to answer sub-graph search, we propose a novel spectral graph coding technique, i.e GCoding. Benefiting from GCoding, we map the structure information of graphs into the numerical space by vertex signatures and GCodes. Then, we design two-step filtering process. Compared with existing approaches, GCoding has significant advantages. First, it has fast offline processing time and is easy for maintenance. Therefore, it can handle static and dynamic graph databases well. Second, GCode has both good pruning power and fast filtering time. Third, good scalability of GCoding ensures its good performance in very large graph databases.

**Acknowledgments** The authors would like to thank Xifeng Yan and Jiawei Han for providing gIndex, and Huahai He and Ambuj K. Singh for providing Closure-tree, and Michihiro Kuramochi and George Karypis for providing the synthetic graph data generator, and Peixiang Zhao for providing Tree+$\delta$. The authors also acknowledge the valuable comments of Dennis Shasha.

# 8. REFERENCES

[1] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Community mining from multi-relational networks. In *PKDD*, 2005.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu. $fg$-index: Towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[3] J. H. D.W. Williams and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.

[4] S. Fortin. The graph isomorphism problem. *Department of Computing Science, University of Alberta*, 1996.

[5] H. H. Goldstine, F. J. Murray, and J. von Neumann. The jacobi method for real symmetric matrices. *J. ACM*, 6(1):59–96, 1959.

[6] P. Y. H. Jiang, H. Wang and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

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

[8] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, 2000.

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

[10] Y. Ke, J. Cheng, and W. Ng. Correlation search in graph databases. In *SIGKDD*, 2007.

[11] H. Kitagawa and Y. Ishikawa. False drop analysis of set retrieval with signature files. *Inf. Syst.*, 27(2), 2002.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, 2001.

[13] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Enginnering*, 9(3), 1997.

[14] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[15] A. Shokoufandeh, S. J. Dickinson, K. Siddiqi, and S. W. Zucker. Indexing using a spectral encoding of topological structure. In *CVPR*, 1999.

[16] E. Tousidou, P. Bozanis, and Y. Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, 27(2), 2002.

[17] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.

[18] D. B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, 2000.

[19] P. Willett. Chemical similarity searching. *J. Chem. Inf. Comput. Sci*, 38(6), 1998.

[20] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.

[21] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *Proc. of Int. Conf. on Data Mining*, 2002.

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

[23] N. Zhang, M. T. Özsu, I. F. Ilyas, and A. Aboulnaga. Fix: Feature-based indexing technique for xml documents. In *VLDB*, 2006.

[24] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, 2007.

[25] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, 2007.