A Novel Spectral Coding in a Large Graph Database

Lei Zou*
Huazhong University of Science and Technology
Wuhan, China
zoulei@mail.hust.edu.cn

Lei Chen
Hong Kong University of Science and Technology
Hong Kong, China
leichen@cse.ust.hk

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

Yansheng Lu
Huazhong University of Science and Technology
Wuhan, China
lys@mail.hust.edu.cn

ABSTRACT

Retrieving related graphs containing a query graph from a large graph database is a popular problem in many graph-based applications, such as drug discovery and structural pattern recognition. Because sub-graph isomorphism is a NP-complete problem [5], we need an effective and efficient filtering method to prune false alarms as many as possible before expensive verification. In this paper, we address the sub-graph search problem by proposing a novel spectral encoding method, i.e. GCoding. We assign a signature to each vertex based on its local structures. After that, we generate a spectral graph code by combining all vertex signatures in a graph. Due to spectral encoding technique, graph structure information is mapped into the numerical space. Then, based on graph codes, we propose an effective and efficient pruning strategy. Extensive experiments show that GCoding outperforms existing methods in both pruning power and index size.

1. INTRODUCTION

Graph is an important data structure in computer science, which has been used to model many objects and complex relationships in the real world, such as chemical structures [17], entities in images [13] and social networks [2]. A general graph information system is very useful in many applications. For example, given a large molecule database, a chemist wants to find all molecules having a particular sub-structure, which is a very popular operation in science experiments and analysis. Given another example, in pattern recognition, it is often necessary to match an unknown sample against a database of candidate graph patterns. Formally speaking, one of the key problems in a graph information system is how to efficiently process sub-graph search, which is defined as follows: *Given a query graph Q, we need to find all data graphs G_i in the graph database, where query graph Q is sub-graph isomorphism to data graph G_i.* A running example is shown as follows:

Example 1 (Running Example). A graph database having 4 graphs and a query graph Q are given in Figure 1. The number beside the vertex is vertex ID and the letter in the vertex is vertex label.

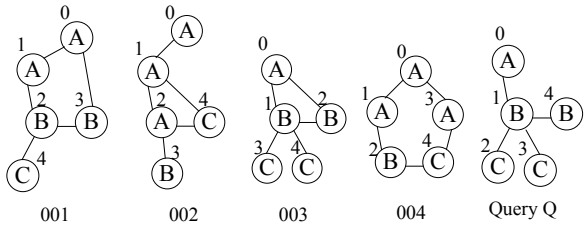


Figure 1: A Graph Database

Usually, there are two phases in the sub-graph search problem, i.e. *filtering* and *verifying*. In the filtering phase, we prune false alarms in the graph database as many as possible to obtain candidates. And then, in the verifying phase, for each candidate graph G_i, we perform sub-graph isomorphism algorithm to check whether G_i is a correct answer. Since sub-graph isomorphism is a classical NP-complete problem [5], an effective and efficient *pruning strategy* in a large graph database will speed up the total response time.

1.1 Motivation

The most popular pruning strategy is "feature-based" [14] [18] [19] [10]. Basic idea in feature-based pruning strategy is that some sub-structure of graph Q does not exist in a data graph G, Q cannot be a subgraph of G. Usually, an inverted index is build: all selected features form "indexed elements", and all data graphs containing these features are linked to them. The relationship between indexed elements and data graphs is similar to the one between indexed words and documents in Information Retrieve(IR). In GraphGrep,

*The work was done when the first author visited Hong Kong University of Science and Technology as a visiting scholar.

all pathes up to *maxL* length are extracted as indexed element. Different from GraphGrep, frequent and discriminative fragments with no more than c vertexes are mined from graph database to be indexed elements in gIndex approach. In order to answer sub-graph query Q, Q are denoted as a set of "indexed fragments" in gIndex. By examining the inverted index, we can retrieve all data graphs that may contain query Q as candidate results. In feature-based pruning strategy, different feature selections provide different pruning power in the filtering phase. Based on effective feature selection strategy, feature-based pruning approach can provide good pruning power [18, 19]. However, also because of feature selection, this kind of pruning approach always assume that the graph database is static, or statistics of the graph database do not change. The quality of feature-based pruning power may degrade over time after lots of insertions and deletions [18]. We have to re-select features and rebuild the index, which is very time consuming. In practice, take compound structure database for example. According to the report by the SCI Finder [1], approximate 4,000 new compound structures are added each day. Therefore, an efficient pruning strategy should also provide good pruning power in the dynamic situation.
Different from feature-based pruning strategy, Closure-tree proposed another pruning strategy [8], that is pseudo sub-graph isomorphism. Its basic idea is that if we cannot find an injective function from vertexes in graph Q to ones in graph G, Q cannot be sub-graph isomorphism to G . Superior to feature-based methods, the pruning power will not degrade in the dynamic situation. However, in the filtering phase of Closure-tree, we need to perform expensive structure comparison and maximal matching algorithm. The time complexity of the pruning algorithm in Closure-tree is $O(ln_1n_2(d_1d_2 + M(d_1, d_2)) + M(n_1, n_2))$, where l is the pseudo compatibility level, n_1 and n_2 are numbers of vertexes in G_1 and G_2, d_1 and d_2 are the maximum degrees of G_1 and G_2, M() is the time complexity of maximal cardinality matching for bipartite graphs, which is $O(n^{2.5})$ in Hopcroft and Karp's algorithm [9].

In this paper, we address sub-graph search problem in a large graph database. Different from the previous methods, we encode structure information based on graph spectral theory, which maps graph's structure information into the numerical space. To our best knowledge, there is little work on spectral encoding technique in sub-graph search problem. To be an effective spectral encoding technique in sub-graph search problem, it should: 1) map a graph structure to the numerical space; 2) based on the encoding technique, pruning strategy would promise no positive dismissal; 3) the pruning strategy based on the graph coding technique should provide great pruning power. In this paper, we present a novel encoding technique satisfying the above criteria.

1.2 Our Contributions

In summary, our contributions are as follows:
1) For each graph, we assign a signature to each vertex, which is derived from the eigenvalues of the matrix about the local structure around the vertex. By combining all vertex signatures in a graph, we can obtain the graph code. After encoding a graph, the structure information of a graph is mapped into the numerical space. Based on graph code comparison, we propose the efficient and effective pruning strategy.
2) We propose an efficient storage schema for the graph code database and gCode-tree index.
3) Through extensive experiments, we show that our method is superior to existing methods in pruning power and online response time.

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. Section 5 discuss the parameter setting problem. We evaluate our method in the extensive experiments in Section 6. The related work is discussed in details in Section 7. Section 8 concludes the paper.

2. BACKGROUND

Definition 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 vertexes; (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: $V \rightarrow L_v$ that assigns labels to vertexes; (6) F_e is a function: $E \rightarrow L_e$ that assigns labels to edges.

Definition 2. **Sub-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 sub-graph isomorphism to G_2, if and only if there exists at least one injective function $f : V_1 \rightarrow V_2$ 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 3. **Induced Sub-graph.** An induced sub-graph is a subset of the vertices of a graph together with any edges whose endpoints are both in this subset.

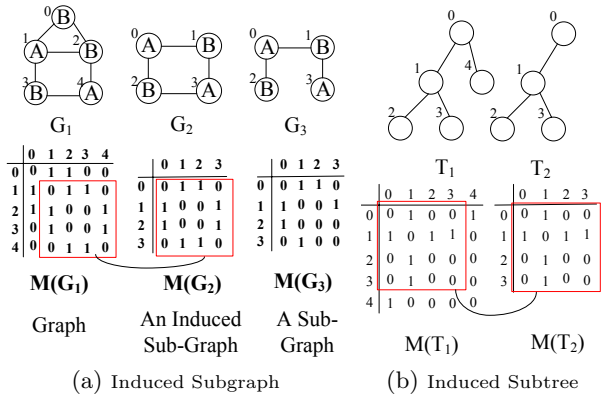
Notice that, in Figure 2(a), only G_2 is an induced sub-graph of G_1, and G_3 is a general sub-graph of G_1. The adjacency matrix of a graph G is denoted as $M(G)$. If a graph Q is an induced sub-graph of G, $M(Q)$ is a principle sub-matrix [2] of $M(G)$. In Figure 2(a), $M(G_2)$ is a principle sub-matrix of $M(G_1)$, but $M(G_3)$ is not, since G_3 is not an induced sub-graph of G_1.

Definition 4. **Trees.** A tree is an acyclic connected directed graph. In this paper, our trees are always unordered, unlabeled, and rooted, which is 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 our trees are always unordered, so they have no predefined ordering among each set of siblings.

Definition 5. **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 $f(u)f(v) \in E$;

[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/

[2]A principal sub-matrix $M(Q)$ of a matrix $M(G)$ formed by selecting certain rows and columns from the matrix $M(G)$



(a) Induced Subgraph (b) Induced Subtree

Figure 2: Induced Subgraph and Induced Subtree

In Figure 2(b), we also use $M(T_1)$ to denote the adjacency matrix of the tree T_1. In tree T_1, if the node i is a parent of node j, we set $M(T_1)_{ij}$ =1 and $M(T_1)_{ji}$ =1. Therefore, $M(T_1)$ is always a symmetric matrix. If the tree T_2 is an induced sub-tree of T_1, $M(T_2)$ is a principle sub-matrix of $M(T_1)$. In mathematics, the Interlacing Theorem illustrates the relationship between the eigenvalues of a matrix and its principle sub-matrix.

Theorem 1. (*Interlacing Theorem*) 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).

Given two graphs Q and G, Q is an *induced* sub-graph of G. $M(Q)$ and $M(G)$ are their corresponding adjacency matrixes. $M(Q)$ is a principal sub-matrix of $M(G)$. Therefore, the eigenvalues of $M(Q)$ and $M(G)$ satisfy the relationships in Theorem 1. Notice that if Q is a *general* sub-graph of G (not induced sub-graph), we cannot promise that the eigenvalues of $M(Q)$ and $M(G)$ satisfy the relationship in Theorem 1. Therefore, Theorem 1 cannot be used as the pruning rule directly in sub-graph search problem.

3. GRAPH ENCODING

As we know that when the query graph Q is sub-graph isomorphism to a data graph G under some injective function g, for each vertex v in Q, there must exist a corresponding vertex $g(v)$ in G. Furthermore, the local structure around v in Q should be preserved in that around $g(v)$ in G. Intuitionally, we can assign a signature to any vertex v based on its local structure. Benefiting from the vertex signature, for any vertex v in the query Q, we can easily determine whether there exists a corresponding vertex $g(v)$ in the data graph G. Obviously, if we cannot find the corresponding vertex $g(v)$ in G, Q cannot be sub-graph isomorphism to G. Notice that the above method may yield a *false positive*, where it suggests that there exists a corresponding vertex $g(v)$ in graph G even though there exists nothing. Fortunately, *false positive* does not affect the correctness of candidate results. However, we should guarantee that the above method satisfy *no false negative* requirement. We will discuss the following two problems in Section 3.1 and 3.2 respectively: 1) We propose *Vertex Signature* based on spectral graph theory. According to vertex signature comparison, the Filtering Rule 1 is proposed; 2) By combining all vertex signatures, we develop a *Graph code* for each graph. Based on the graph code comparison, Filtering Rule 2 is prosed, which is superior to the former one due to its efficiency.

3.1 Vertex Signature

In order to assign a signature to each vertex v in a graph G, we map the local structure around v into the numerical space. The local structure is defined as follows: we propose an algorithm in Figure 3 to extract all n-step simple pathes[3] from the vertex v in G, which are collected to form the **Level-n Path Sub-tree**, denoted as $LNST(G, v, n)$. Different from level-n adjacent sub-tree in [8], only simply pathes are considered in $LNST$.

Algorithm: Extracting Level-n Path Sub-tree
Input: a graph G and a vertex v in G.
Output: level-n path sub-tree around the vertex v, denoted as $LNST(G, v, n)$.

1: set the vertex v as the root of $LNST(v)$
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 $LNST(v)$.
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 $LNST(v)$.
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**

Figure 3: Extract Level-n Path Subtree

Given a graph G in Figure 4a, $LNST(G, 0, 2)$ is shown in Figure 4b. According to the definition in [8], the level-2 adjacent subtree from vertex 0 is given in Figure 4. The path $(A^0D^1A^0)$ is not a simple path, which does not exist in Figure 4b, but exists in Figure 4c.

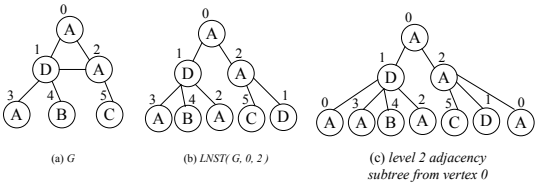


Figure 4: Level-n Path Sub-tree

Lemma 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 $LNST(Q, v, n)$. We have a vertex

[3]A path in a graph with no repeated vertices is called a simple path

v' in graph G, where $v' = g(v)$. Then, $LNST(Q, v, n)$ is a sub-tree of $LNST(G, v', n)$.

PROOF. (Sketch) For each node u in $LNST(Q, v, n)$, according to the definition of $LNST$, 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 $LNST(Q, v, n)$ to node u' in $LNST(G, v', n)$: $u' = f(u)$, where the path vu is corresponding to the path $v'u'$. Under the injective function f, it is straightforward to prove that $LNST(Q, v, n)$ is a sub-tree of $LNST(G, v', n)$, according to the Definition 5. □

Definition 6. **Vertex Topology Signature**. Given a graph G and a vertex $v \in G$, the adjacency matrix of $LNST(G, v, n)$ is denoted as M. The eigenvalues of M are $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_m$. The *Vertex Topology Signature* of the vertex v is defined as the sorted list $< \lambda_1, \lambda_2 \ ... \ \lambda_m >$.

According to Definition 6, the local structure around a vertex is mapped into the numerical space. Since spectral graph theory suggests that some largest eigenvalues determine greatly the graph's topological structure, it is not necessary to consider all eigenvalues of $LNST(G, v, n)$ to define the vertex's topology signature. In experiment section, we will evaluate different performances under different eigenvalues. Without loss of generality, we choose the two largest eigenvalues in the following discussion.

Lemma 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 $< \lambda_1, \lambda_2 >$. There exits a vertex v' in graph G, where $v' = g(v)$. For v', its topology signature is $< \beta_1, \beta_2 >$. We can say that $\lambda_1 \leqslant \beta_1 AND \lambda_2 \leqslant \beta_2$.

PROOF. According to Lemma 1, $LNST(Q, v, n)$ is a sub-tree of $LNST(G, v', n)$. It also means that the adjacency matrix associated with $LNST(Q, v, n)$ is a principal sub-matrix of that associated with $LNST(G, v', n)$. Based on Theorem 1, it is straightforward to know that $\lambda_1 \leqslant \beta_1$ AND $\lambda_2 \leqslant \beta_2$. □

As we know that the limitation of spectral methods is that they are purely structural, in the sense that they are not able to exploit vertex or edge labels. In order to consider label information to improve pruning power, for each vertex v, we encode its vertex label, its neighbor vertex labels and adjacent edge labels. According to the method in signature file [4], we propose the following approach.

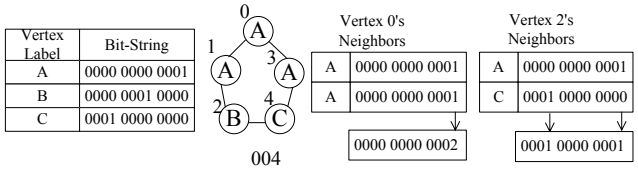


Figure 5: Vertex Label and Neighbor Vertex Labels

Vertex Label

We use a length-X bit string to denote the vertex label. Initially, all bits are set "0". Using the hash function, for each distinct vertex label, we set m out of X bits to 1. For example, we use the length-12 bit strings on the left of Figure 5 to denote the vertex labels "A","B" and "C" in the running example.

Neighbor Vertex labels

Figure 5 shows an example of encoding the neighbor labels. Given a graph 004 in Figure 5, the vertex 0 has two neighbor vertexes labeled with "A" and "A". The bit string for the "A" is "0000 0000 0001". We also use a length-X "counter" string to denote the neighbors of vertex 0. We get "0000 0000 0002" by bitwise "ADD"-operation. Similarly, for the neighbor labels of vertex 2, we get "0001 0000 0001", as shown in Figure 5.

Adjacent Edge Labels

In Figure 6, the vertex 0 has two neighbor vertexes and two adjacent edges. We denote the adjacent edge label and neighbor vertex label as a pair $< eL, vL >$, where eL is adjacent edge label and vL is the corresponding neighbor vertex label. The vertex 0 has two pairs, such as $< a, B >$ and $< c, B >$. For each distinct pair, we also use a distinct bit string to denote it, such as "0000 0010 0000" for $< a, B >$ and "0000 0100 0000" for $< c, B >$. By bitwise "ADD"-operation, we get the "counter"-string, which not only encodes neighbor vertex labels but also the adjacent edge labels. In fact, the method in encoding adjacent edge labels is the same with that in encoding neighbor vertex labels. For illustration convenience, we do not consider the adjacent edge labels in the following discussion.

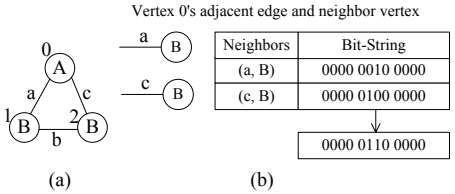


Figure 6: Encoding Edge Labels

Definition 7. **Vertex Signature**. Given a graph G and a vertex $v \in G$, the vertex signature of v is $< L, N, \lambda_1, \lambda_2 >$, where L is a length-X bit string to denote the vertex label, and N is a length-X counter string to denote the neighbor labels, $< \lambda_1, \lambda_2 >$ is its *vertex topology signature*, defined in Definition 6. We always use *sig(v)* to refer to the signature of a vertex v.

The vertex signatures of graph 002 ,003 and query Q in the running example are shown in Figure 13.

Lemma 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 $< L_1, N_1, \lambda_1, \lambda_2 >$. In graph G, there is a vertex v', where v'=$g(v)$. Its signature is $< L_2, N_2, \beta_1, \beta_2 >$. The two vertex signatures satisfy the following conditions:

1) $L_1[i] = L_2[i]$, $i = 0...X\text{-}1$;
2) $N_1[i] \leqslant N_2[i]$, $i = 0...X\text{-}1$;
3) $\lambda_1 \leqslant \beta_1$ AND $\lambda_2 \leqslant \beta_2$,

PROOF. (Sketch)

1) According to the sub-graph isomorphism definition, the labels of v and v' are the same with each other, i.e. $L_1[i] = L_2[i]$.

VertexID	L	N	λ_1	λ_2
0	000000000001	000000000001	1.73	0.00
1	000000000001	000100000002	2.05	1.20
2	000000000001	000100010001	2.05	1.20
3	000000010000	000000000001	1.73	0.00
4	000100000000	000000000002	2.00	1.41
Gcode(002)	000100010003	000200010007	Seq_1	Seq_2

Seq_1	2.05, 2.05, 2.00, 1.73, 1.73
Seq_2	1.41, 1.20, 1.20, 0.00, 0.00

(a) 002

VertexID	L	N	λ_1	λ_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	Seq_1	Seq_2

Seq_1	2.14, 2.10, 2.10, 2.00, 200
Seq_2	1.26,1.26, 1.00, 0.00, 0.00

(b) 003

VertexID	L	N	λ_1	λ_2
0	000000000001	000000010000	2.00	0.00
1	000000010000	000200010001	2.00	0.00
2	000100000000	000000010000	2.00	0.00
3	000100000000	000000010000	2.00	0.00
4	000000010000	000000010000	2.00	0.00
GCode(Q)	000200020001	000200050001	Seq_1	Seq_2

Seq_1	2.00, 2.00, 2.00, 2.00, 2.00
Seq_2	0.00, 0.00, 0.00, 0.00, 0.00

(c) Query Q

Figure 7: Vertex Signatures and Graph Codes

2) 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.
3) Condition 3) has been proved in Lemma 2. □

Given two vertex v in graph Q and v' in G, if $sig(v)$ and $sig(v')$ can satisfy three conditions in Lemma 3, we can say $sig(v)$ is **compatible** to $sig(v')$.

Pruning Rule 1. Given two graphs Q and G, for some 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. □

Though the above pruning rule can provide great pruning power, they need "vertex to vertex" comparison. Therefore, for a large graph database, we should have the more efficient pruning strategy.

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 vertex signatures, which maps the global structure of a graph G into the numerical space. In the new pruning strategy, we only need to compare two graph codes, which avoids the "vertex to vertex" comparison.

Definition 8. **Spectral Graph Code.** The code of a graph G is a denoted as $GCode(G) = < L, N, Seq_1, Seq_2 >$, where L and N are length-X counter strings. Assume that graph G has n vertexes, and each vertex v_j is denoted as $sig(v_j) = < L_j, N_j, \lambda_{j1}, \lambda_{j2} >$.
1) $L[i] = \sum_{j=0}^{j=n-1} L_j[i]$, where i=0... X-1 and $L[i]$ is the i-th counter of L.
2) $N[i] = \sum_{j=0}^{j=n-1} N_j[i]$, where i=0... X-1 and $N[i]$ is the i-th counter of N.
3) For all vertex signatures $sig(v_j)$, all λ_{j1} are ranked according to non-ascending order to form the sorted list Seq_1. Similarly, all λ_{j2} are ranked to form the sorted list Seq_2.

The $GCode(001)$, $GCode(002)$ and $GCode(Q)$ are shown in Figure 13.

Lemma 4. Given two graphs Q with n_1 vertexes and G with n_2 vertexes, where $n_1 \leq n_2$, their graph codes are denoted as $GCode(Q) = < QL, QN, QSeq_1, QSeq_2 >$ and $GCode(G) = < GL, GN, GSeq_1, GSeq_2 >$. 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 bit of QL;
2) $QN[i] \leq GN[i]$, where $QN[i]$ is i-th bit of QN;
3) $QSeq_1[j] \leq GSeq_1[j]$, j = 0...$n_1 - 1$.
4) $QSeq_2[j] \leq GSeq_2[j]$, 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 $sig(v) = < vL, vN, \lambda_1, \lambda_2 >$ and $sig(u) = < uL, uN, \beta_1, \beta_2 >$.
1) According to Lemma 3, we know that $vL[i] = uL[i]$. Since the function g is an injective function from vertexes in Q to vertexes in G, it means that $\sum vL_{j1}[i] \leq \sum uL_{j2}[i]$, where j_1=1...n_1 and j_2=1...n_2. Therefore, $QL[i] \leq GL[i]$.
2) According to Lemma 3, we know that $vN[i] \leq uN[i]$.Since the function f is an injective function from vertexes in Q to vertexes in G, it means that $\sum vN_{j1}[i] \leq \sum uN_{j2}[i]$, where j_1=1...n_1 and j_2=1...n_2. Therefore, $QN[i] \leq GN[i]$.
3) We prove the condition 3) by *contradiction*.
Assume that Condition 3) is not correct, that is $QSeq_1[j] > GSeq_1[j]$.
$QSeq_1$ is a sorted list according to non-decreasing order, therefore, $QSeq_1[i] \geq GSeq_1[j]$, where $i = 0...j$. It means that there exist *j+1* vertexes v_i in graph Q (i=0...j), whose λ_1 are larger than $GSeq_1[j]$. 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, the λ_1 of vertex v_i is no larger than that of vertex u_i. It means that there must exist *j+1* vertexes u_i in graph G, whose λ_1 are larger than $GSeq_1[j]$.
As we know, $GSeq_1$ is a non-decreasing sorted list and $GSeq_1[j]$ is the j-th largest one. It means that there exist at most j vertexes, whose λ_1 are larger than $GSeq_1[j]$, which is contradicted to the above analysis. Therefore, condition 3) is correct.
4) We can prove the condition 4) by the similar method in 3). □

Based on the Lemma 4, we have the following pruning strategy.
Pruning Rule 2. Given two graphs Q, their graph codes are denoted as $GCode(Q) = < QL, QN, QSeq_1, QSeq_2 >$ and $GCode(G) = < GL, GN, GSeq_1, GSeq_2 >$. If $GCode(Q)$ and $GCode(G)$ cannot satisfy at least one follow-

ing condition, 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) $QN[i] \leq GN[i]$;
3) $QSeq_1[j] \leq GSeq_1[j]$, j = 0...$n_1 - 1$.
4) $QSeq_2[j] \leq GSeq_2[j]$, j = 0...$n_1 - 1$.

For example, we compare $GCode(Q)$ and $GCode(002)$. Since $GCode(Q).Seq_1[3]$=2.00 > $GCode(002).Seq_1[3]$=1.73, therefore graph 002 is pruned safely.

4. SUB-GRAPH SEARCH

In the offline phase, for each graph G_i in the graph database, we compute all vertex signatures in G_i and $GCode(G_i)$. 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. All $GCode(G_i)$ are collected to form the graph code database. In the running example, the corresponding graph code database is shown in Figure 8.

	L	N	Seq_1	Seq_2
Gcode(001)	000200020001	000200050001	1.73, 1.73, 1.73, 1.73, 1.73	0.00, 0.00, 0.00, 0.00, 0.00
Gcode(002)	000100010003	000200010007	2.05, 2.05, 2.00, 1.73, 1.73	1.41, 1.20, 1.20, 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)	000100020002	000100050004	1.93 1.90 1.90 1.73 1.73	1.18 1.18 1.00 1.00 0.00
Gcode(Q)	000200020001	000200050001	2.00, 2.00, 2.00, 2.00, 2.00	0.00, 0.00, 0.00, 0.00, 0.00

Figure 8: Graph Code Database

According to the discussion in Section 3, we have two pruning rules. In pruning power, there exist some false alarms that can pass Pruning Rule 2 will be pruned by Pruning Rule 1. Figure 11 discussed in the next section is an example. In fact, the pruning power of Rule 1 is not always superior to the Rule 2. For another example, in query graph Q, more than one vertex signatures are only compatible to one vertex signature in graph G. It is obvious that graph Q cannot be sub-graph isomorphism to graph G, since the sub-graph isomorphism is always an injective function. However, this kind of false alarms cannot be filtered out by Pruning Rule 1. Due to "count" information in graph code, we can prune them in Rule 2. Therefore, we can say that the pruning power of Rule 1 and 2 are not "parallel" to each other. We need to combine these two pruning rules in a special order. In Pruning Rule 2, we only need to compare two graph codes. However, we have to perform "vertex to vertex" comparison in Pruning Rule 1. It is straightforward to know that the former (Rule 2) is much more efficient than the later (Rule 1). Therefore, we have the online framework shown in Figure 9. We first use Rule 2 to prune most false alarms in the 1-st filter process, and then use Rule 1 to filter out more false positives in the 2-nd filtering process. After that, we perform expensive sub-graph isomorphism checking in the verification phase. Notice that, the cost of both filtering rules are much less than sub-graph isomorphism algorithm, which is NP-complete. We will evaluate the pruning power, filtering time in each step and sub-graph isomorphism checking time in experiment section.

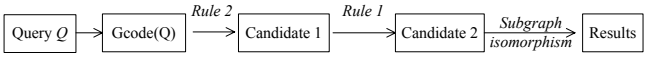


Figure 9: Online Processing

Given a query graph Q, we compute Q's graph code, i.e. $GCode(Q)$. Using the Pruning Rule 2, we can do 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, since we only need to perform numerical operations. However, as we know that many existing chemical databases contain more than 500,000 or more compounds, such as "Combined Chemical Dictionary" [4]. Therefore, the efficient index structure in the large graph code database will avoid the pairwise comparison, and it leads to less filtering time.

Inspired by S-tree [15] for the signature files, we develop our GCode-Tree. For the running example, the corresponding GCode-Tree is shown in Figure 10. Since each $GCode(G_i)$= $< L_i, N_i, Seq_1, Seq_2 >$, we extract the first two parts, i.e. L and N, to build 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 dictionary node I in GCode-Tree has children nodes C_i, the I and C_i are all denoted as $< L, N >$. We set $I.L[j]$ = Max($C_i.L[j]$), where $I.L[j]$ is the j-th counter of $I.L$. Similarly, we set $I.N[j]$ = Max($C_i.N[j]$). Since the dynamic operations in GCode-Tree, such as insertion and operation, are analogous to that in S-tree. Due to space limited, we omit the detail discussion about dynamic operations in this paper.
Given a query graph Q, we can get the $GCode(Q)$ =< $L, N, Seq_1, Seq_2 >$. For the dictionary node I =< L, N > in GCode-Tree, 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 will be pruned safely. For example, given the query graph Q, $GCode(Q)$ is shown in Figure 8. For the node I_1 in GCode-Tree, $GCode(Q).L[3] = 2 > I_1.L[3] = 1$, all descendants of I will not be considered. It means that graph 001 and 002 can be pruned safely. Therefore, scanning the GCode-Tree, we can prune all graph codes that cannot satisfy the condition 1) or 2) in Pruning Rule 2. For each each remaining graph code $GCode(G_i)$, we check whether $GCode(Q)$ and $GCode(G_i)$ satisfy the condition 3) and 4) in Pruning Rule 2. Then, in 2-nd filtering process, we perform Pruning Rule 1 for each candidate after 1-st filtering process.

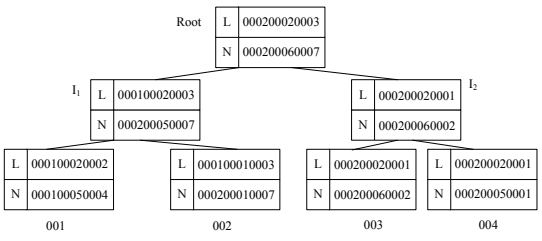


Figure 10: GCode-Tree

[4] http://ccd.chemnetbase.com/

5. THE LENGTH OF L AND N

In this section, we discuss the problem about the length of L and N in vertex signature and graph code. Similar with signature file, there may exist "false drop" problem [11] when comparing graph codes by Rule 2. Figure 11 shows a false drop example. Here, we discuss the length of L, therefore we omit the N and eigenvalues in vertex signatures and graph codes in Figure 11. Given a query graph Q and a data graph G, we have the hash function in Figure 11. The length of L is 4. In the hash function, for each distinct label, we set 2 out of 4 bits to "1". Unfortunately, the $GCode(Q).L$ is equal to $GCode(G).L$, though we have different vertex labels in Q and G. It means that the L part of graph code does not provide pruning power in this example. In fact, it is similar with the "False Drop" problem in signature file [11]. Though, in GCoding, the above false drop problem can be overcome in the 2-nd filtering step, that is Filtering Rule 1 (vertex to vertex comparison). However, in order to improve the pruning power in the 1-st filtering step, we should decrease the possibility of false drop. For example, we can set the length of L to be 5. For each distinct label, we set one of 5 bits to "1". In this way, we do not get any false drop. However, this straightforward approach leads to another problem, that is the space cost. For example, if we have 100 distinct vertex labels, the length of L is 100. Obviously, it is not space efficiency in a large graph database. To tradeoff the space cost and pruning power in 1-st filtering process, considering the property of real graph database, we propose the following approach to determine the length of L and N.

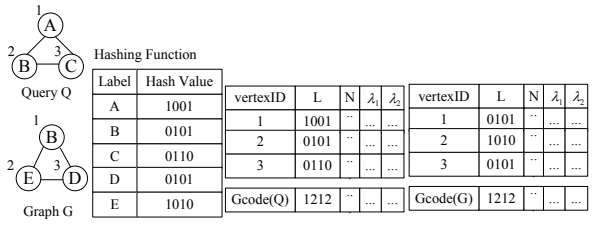


Figure 11: False Drop

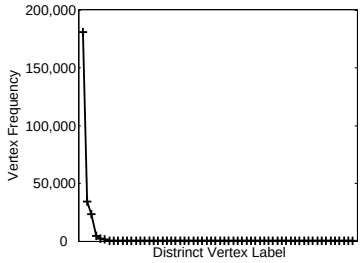


Figure 12: Vertex Label Frequencies

We have a real graph database (AIDS dataset in Section 6) that has 10,000 compounds. There are 248,075 atoms (vertexes) in all compounds (graphs). The major portion of the atoms are "C", "O", "N". For example, there are 180879 "C" atoms in all compounds. Figure 12 shows the vertex label frequency. Obviously, it likes Zipf distribution[5]. It means that few vertex labels have very large frequencies. Actually, it is a popular phenomenon in compound structure database. Therefore, we define a threshold α. Then we get the two vertex label sets $S_1 = \{L_i | fre(L_i) \geq \alpha\}$, and $S_2 = \{L_i | fre(L_i) < \alpha\}$, where $fre(L_i)$ is the frequency of vertex label L_i. Due to Zipf distribution, $|S_1| \ll |S_2|$. We use length-$|S_1|$ bit string to denote the vertex labels in S_1. For each distinct vertex label in S_1, we set one of $|S_1|$ bits to "1", which leads to no false drop. To denote the vertex labels in S_2, according to analysis in [11][6], we can determine the length of bit string. The false drop possibility is given in the following Formula 1, where X: the length of bit string; m: for each vertex label, we set m out of X bits to "1"; D_q and D_t: the vertex number in query graph Q and data graph G. For each given maximal false drop possibility, we can use the following Formula 1 to guide setting the length X. Then we use length-X bit string to denote the vertex label in S_2. Therefore, the length of L is equal to $|S_1|+X$. Using the similar method, we can also get the length of $|N|$.

$$P_{false\_drop} = (1 - e^{-\frac{mD_t}{X}})^{mD_q} [11] \quad (1)$$

6. EXPERIMENTS

In this section, we evaluate the performance of our method, i.e. GCoding. For sub-graph search, gIndex and Closure-Tree 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.

6.1 Datasets and Setting

In experiments, we consider vertex-labeled and edge-labeled graphs. In GCoding, we use the method mentioned in Figure 6 to encode edge labels. 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 to be made in the case of sub-graph search.

1) **AIDS Antiviral Screen Dataset** The real dataset we test is AIDS dataset containing chemical compounds. This dataset is available publicly on the website of the Developmental Therapeutics Program [7]. As of March 2004, the dataset contains 43,850 classified chemical molecules. 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 by AIDS dataset. Each query set Qm has 1000 connected query sub-graphs and each query sub-graph in Qm is a connected size-m sub-graph, which is extracted from each data graph

[5]A distribution characterized by Zipf's law is called Zipf distribution, and Zipf's law states that the size of the r'th largest occurrence of the event is inversely proportional to it's rank r

[6]Different from [11], we may have duplicate label in a graph, such as label "A" in graph 001 in Figure 1. However, for any vertex label l in S_2, it is a little possibility that a graph may contain duplicate l. Therefore, for the set S_2, we have the similar false drop possibility with [11].

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

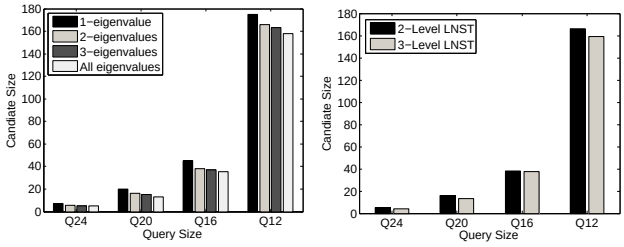
randomly. We use six query sets, $Q4$, $Q8$, $Q12$, $Q16$, $Q20$ and $Q24$.

2) **Synthetic Dataset** The synthetic dataset is generated by a synthetic graph generator provided by authors of [12]. More details about the synthetic data generator are available in [12]. We generate the graph database by the following parameters: D=10,000, L=5, I= 6, T=20, V=5, E=5.

In gIndex and Closure-tree algorithms, we choose the default or the suggested values for parameters according to [18, 8]. We discuss the parameter setting problems for our GCoding in Section 6.2. Except for experiments in Section 6.2, we always choose the level-2 Path Sub-tree and use the 2 largest eigenvalues to define vertex signatures and graph codes. In experiments, the default length of L and N is 26.

6.2 Parameter Setting

In this subsection, we discuss some parameter setting problems in GCoding method. We first discuss the length of L and N in vertex signature and graph codes. In Section 5, we have discussed how to set the length of L and N. In AIDS dataset, we have 248,075 vertexes in all graphs and 62 distinct vertex labels. Only 6 vertex labels have more than 1000 occurrences in these 248,075 vertexes. There are average 24.80 vertexes in each data graph. Therefore, we use 6 bits to denote the frequent 6 vertex labels. For each frequent vertex label, we set one of 6 bits to "1". In fact, there is a little possibility (< 0.001) that one data graph contains duplicate unfrequent vertex labels. We use Formula 1 in Section 5 to guide setting the length of L and N. We set $F = 20$ and $m = 2$, that is to set two out of 20 bits to "1" for each unfrequent vertex label. According to Formula 1, the false drop possibility is less than 0.3%. Thus, the total length of L is $20 + 6$. Similarly, the length of N is also set to 26.



(a) (b)

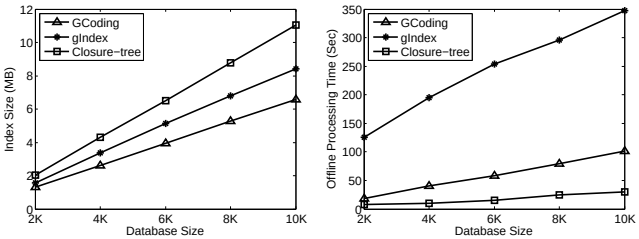
Figure 13: The Pruning Power at (a) Different Eigenvalues and (b) Different Level LNST in AIDS Dataset

Figure 13(a) shows the performance in different eigenvalues. Observed from the Figure 13(a), we find that choosing three or more eigenvalues cannot lead to the great improvement in pruning power. Furthermore, choosing more eigenvalues means the larger graph code database. Therefore, we always choose the two largest eigenvalues, i.e. λ_1 and λ_2 in our experiments.

As we know, $LNST(G, v, 1)$ is the level-1 path sub-tree around vertex v. Actually, all nodes in $LNST(G, v, 1)$ are all v's neighbor vertexes. It means that 1-st level $LNST$ information has been coded into the "N" part of vertex signatures and graph codes. Therefore, we should use 2 or more levels $LNST$. However, choosing more levels means that the corresponding matrix is larger, and it also means that we need more time to compute eigenvalues for vertex signatures and graph codes. In fact, $|LNST(G, v, n)| = O(d^n)$, where $|LNST(G, v, n)|$ is the number of nodes in $LNST(G, v, n)$ and d is the average vertex degree. Observed from Figure 13(b), choosing 3-levels does not leads to significant improvement in pruning power. Therefore, to obtain less offline processing time and online filtering time, we always use 2-level $LNST$ in GCoding.

6.3 Performance Study

We first evaluate the performance of GCoding in the offline process, such as the size of "graph code database together with Indexes + vertex signature dictionary" and offline processing time. In order to compute the eigenvalues of a symmetric matrix in our method, we implement the Jacobi method [6].



(a) (b)

Figure 14: (a) Index Size and (b) Offline Processing Time in AIDS dataset

In Figure 14(a), we shows the index size in different methods. To facilitate the illustration, in GCoding method, we refer to "graph code database together with Indexes + vertex signature dictionary" as "index". As the increment of dataset, from 2K to 10K, the index size in our method is always the smallest. Take 10K dataset for example, there are 248,075 vertexes in all 10,000 data graphs. However, there are only 1,898 distinct vertex signatures. Therefore, the size of the vertex signature dictionary is only 0.315M bytes. As we know, the graph code $GCode(G_i) = < L, N, Seq_1, Seq_2 >$. In experiments, the lengths of L and N are set to 26 respectively. On average, there are 24.80 vertexes in a data graph G_i. It is straightforward to know that the average lengthes of Seq_1 and Seq_2 are also 24.80 respectively. Since $GCode(G_i).Seq_1$ is a list of λ_1 for all vertexes in graph G_i, the length of $GCode(G_i).Seq_1$ is always equal to the number of vertexes in graph G_i. Therefore, we only need to store (26*2+24.80*2) numerical values for each $GCode(G_i)$ in graph code database. The size of the graph code database together with GCode-tree indexes is 5.074M bytes in ADIS dataset. We need extra 1.334M byes to store the link information between graph ID and vertex signatures in the dictionary. Therefore, the total size in GCoding method in ADIS dataset is 6.723M bytes, which is about one half of the index in Closure-tree method.

Figure 14(b) shows the offline processing time in different methods. Closure-tree is the fastest. Our methods is slower than Closure-tree. According to our testing, on average, we need to about $0.008 \sim 0.010$ second to compute the vertex

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

	Q24		Q20		Q16		Q12		Q8		Q4	
	Cand.[1]	Time[2]	Cand.	Time	Cand.	Time	Cand.	Time	Cand.	Time	Cand.	Time
Compute Graph Codes		10.03		8.56		7.52		7.03		6.30		5.89
1-st filtering	46	1.08	96	2.15	293	3.15	1033	4.42	2511	6.47	5527	9.09
2-nd filtering	10	3.56	27	5.85	75	14.56	380	58.08	1300	101.27	4088	150.01
Total filtering time		14.67		16.56		25.23		69.53		114.04		164.99
Refining phase	3	250.80	6	841.48	12	1532.23	230	1850.50	615	2011.20	2150	2300.15

Note : Cand.[1] : Candidate Size ;
Time [2] : All running time reported in this section are the running time in 1000 queries in each query set.

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, all feature-based methods have the similar limitations.

As we know, according to the online process framework in Figure 9, in order to answer a sub-graph query, we need to compute graph code, two step filtering phase and sub-graph isomorphism in the verification process. Table 1 shows the running time in each step and pruning power in each filtering step. Notice that all running time (including Filtering Time and Total Response Time) reported in this section are the total running time in 1000 queries in each query set. We implement ULLMANN algorithm [16] to check sub-graph isomorphism in the verification process. In the first filtering step, we compare the graph codes between query graph and data graph by Filtering Rule 2. Observed from Table 1, the first step is the fastest, which prunes most false alarms. The candidate size after 1-step Filtering is about $0.5\% \sim 3\%$ of the original size of graph database. 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, compared with expensive refining phase, the total filtering time is about $\frac{1}{10} \sim \frac{1}{100}$ of sub-graph isomorphism checking time. Furthermore, the candidate size after 2-step Filtering is less than $\frac{1}{4}$ of that after 1-step Filtering. Therefore, in order to obtain fast total response time, it is worth performing 2-step filtering phase in GCoding method.

We also compare the pruning power, the filtering time and total response time with some existing methods on both AIDS dataset and synthetic dataset in Figure 15. Observed from Figure 15(a) and 15(d), GCoding has the largest pruning power. Due to expensive structure comparison and maximal matching 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 Figure 15(a) and 15(e). In GCoding method, the filtering time in Q24 is faster than Q4, since there are more false alarms 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 faster than the second filtering step, which is analyzed in Table 1. In gIndex, there are less "indexed fragments" 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. Generally speaking, the filtering time in gIndex and GCoding can be ignored in the total response time, since it is less than $\frac{1}{10} \sim \frac{1}{100}$ of sub-graph isomorphism checking time in verification phase. Because GCoding has the least candidate size and the filtering time is little, the total response time is also the least, which is shown in Figure 15(c) and 15(f).

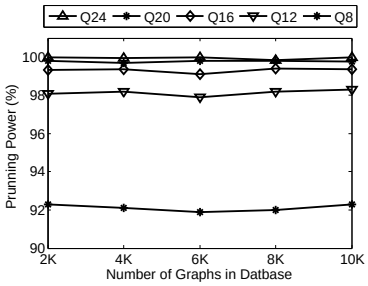


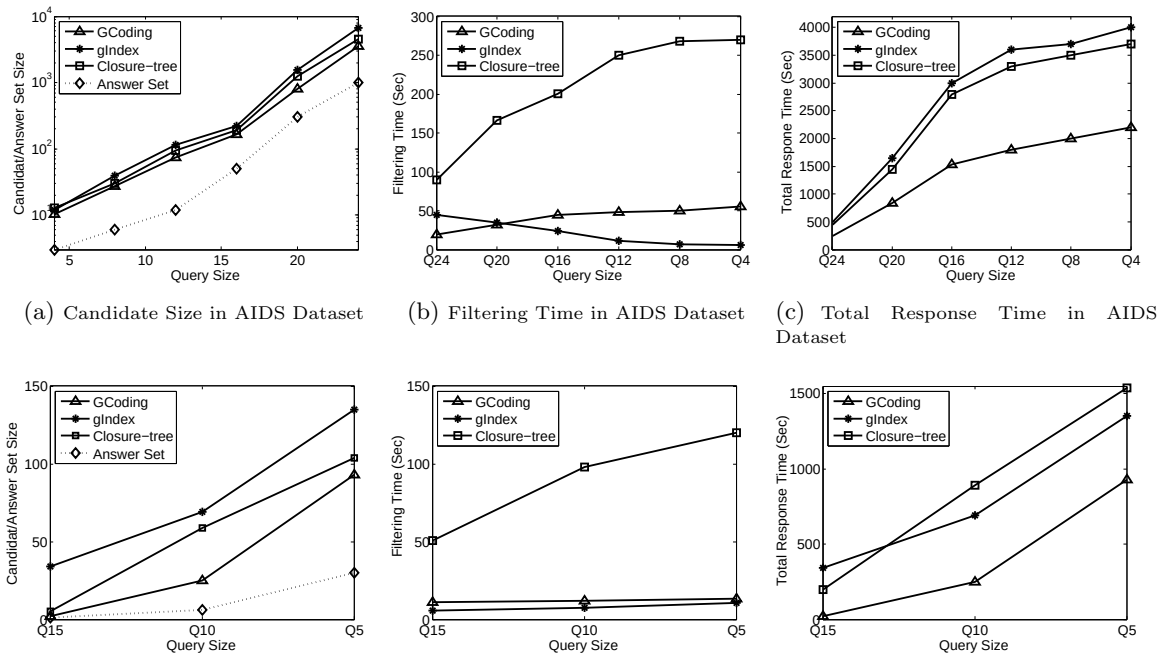
Figure 16: Pruning Power in Dynamic Situation of AIDS Dataset

Figure 16 shows the pruning power of GCoding in dynamic situation of AIDS Dataset, where pruning power is defined as $\frac{DBSize - CandidateSize}{DBSize - \text{ResultSize}}$. We insert graphs in to the database from 2K to 10K. Observed from Figure 16, the pruning power under different query sets always keeps around some value. In experiments, the pruning power of Q4 are always around 75.3%.

7. RELATED WORK

Sub-graph search, which is defined as given a graph Q, retrieving all graphs containing Q as a sub-graph in the graph database. There are two process, that are filtering and verification processes. In order to improve the search performance, most existing methods try to select some features, such as paths, frequent sub-graphs and frequent sub-trees, and build an invert index on these features. Benefiting from these features, we can reduce the candidate answer set size without expensive sub-graph isomorphism checking, which is NP-complete [5]. For example, in GraphGrep [14], all paths up to $maxL$ length are chosen as index features. Extended from path-based indexes, gIndex [18] uses frequent sub-graphs as index features. In gIndex, frequent sub-graphs with size up to $maxL$ are first generated by frequent sub-graph algorithm. Then, a gIndex Tree is built on these frequent sub-graphs. Finally, a candidate set is generated by searching the query graph in the gIndex tree. In [19], authors suggest using frequent and discriminative subtrees to index all graphs. However, because of feature selection in [18] and [19], this kind of pruning approach always assume that the graph database is static, or statistics of the graph database do not change. The quality of feature-based pruning power may degrade over time after lots of insertions and deletions [18]. Furthermore, due to expensive mining process in offline processing, this kind of methods cannot be straightforward to be extended in a very large graph database. In fact, in practice, the graph database is always very large and dynamic (i.e. frequent insertion and deletion). In our method, the offline processing time is always linear with the graph database size.

Furthermore, authors propose another kind of pruning strategy in filtering phase in Closure-tree [8], that is pseudo sub-



(a) Candidate Size in AIDS Dataset (b) Filtering Time in AIDS Dataset (c) Total Response Time in AIDS Dataset

(d) Candidate Size in Synthetic Dataset (e) Filtering Time in Synthetic Dataset (f) Total Response Time in Synthetic Dataset

Figure 15: Evaluating Online Performance

graph isomorphism, which tries to find a semi-perfect matching from vertexes in query graph to vertexes a data graph (or graph closure). If we cannot find the matching, graph (or graph closure) is pruned out safely. Because we need to do expensive structure comparison and maximal matching algorithm in filtering phase, whose time complexity is $O(ln_1n_2(d_1d_2 + M(d_1, d_2)) + M(n_1, n_2))$, where l is the pseudo compatibility level, n_1 and n_2 are numbers of vertexes in G_1 and G_2, d_1 and d_2 are the maximum degrees of G_1 and G_2, M() is the time complexity of maximal cardinality matching for bipartite graphs, which is $O(n^{2.5})$ in Hopcroft and Karp's algorithm [9]. From our performance study, even though we set pseudo compatibility level to 1 (default value), the filtering time is large, 5∼10 times larger than that in GCoding. In GCoding, we only need to perform "cheap" numerical operations.

There are some interesting recent work in graph search problem, such as [3] [19] [7]. In [3], authors enumerate all connected induced subgraphs in the graph database, and organize them in a Graph Decomposition Index (GDI). This method cannot work well in a graph database with large-size graphs, due to combination explosion of enumerating all connected induced subgraphs. Authors propose a graph string method [7] for compound database.

Interlacing Theorem (i.e. Theorem 1 in Section 2) in spectral graph theory tells us that there is a relationship between eigenvalues of adjacency matrixes of Q and G, where Q is an induced subgraph of G. An important limitation of Interlacing Theorem is that it can only hold when Q is an **induced** sub-graph of G, not a general sub-graph. In tree databases, Interlacing Theorem always hold [1]. Another important limitation of spectral methods is that they are purely structural, in the sense that they are not able to exploit node or edge labels. GCoding, proposed in this paper, overcome the above two limitations, i.e. it can work on general graph database problem and it can also handle vertex and edge labels. To our best knowledge, there is little work on spectral encoding technique in sub-graph search problem.

8. CONCLUSION

In this paper, we propose a novel spectral graph coding technique, i.e GCoding. Benefiting from GCoding, we transform the original graph database into the graph code database. In the filtering phase, we prune most false alarms based on graph code comparisons. Extensive experiments show that GCoding has a significant improvement to existing methods on both offline and online processes.

9. ACKNOWLEDGMENTS

We 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.

10. REFERENCES

[1] N. Z. 0002, M. T. Özsu, I. F. Ilyas, and A. Aboulnaga. Fix: Feature-based indexing technique for xml documents. In *VLDB*, pages 259–270, 2006.

[2] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Community mining from multi-relational networks. In *PKDD*, pages 445–452, 2005.

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

[4] C. Faloutsos and S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.

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

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

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

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

[9] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

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

[11] H. Kitagawa and Y. Ishikawa. False drop analysis of set retrieval with signature files. *Inf. Syst.*, 27(2):93–121, 2002.

[12] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

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

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

[15] E. Tousidou, P. Bozanis, and Y. Manolopoulos. Signature-based structures for objects with set-valued attributes. *Inf. Syst.*, 27(2):93–121, 2002.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Assocmtlon for Computing Machinery*, 23:31–42.

[17] P. Willett. Chemical similarity searching. *J. Chem. Inf. Comput. Sci*, 1998.

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

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