



A Coding Method for Efficient Subgraph Querying on Vertex- and Edge-Labeled Graphs

Lei Zhu, Qinbao Song*, Yuchen Guo, Lei Du, Xiaoyan Zhu, Guangtao Wang

Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China

Abstract

Labeled graphs are widely used to model complex data in many domains, so subgraph querying has been attracting more and more attention from researchers around the world. Unfortunately, subgraph querying is very time consuming since it involves subgraph isomorphism testing that is known to be an NP-complete problem. In this paper, we propose a novel coding method for subgraph querying that is based on Laplacian spectrum and the number of walks. Our method follows the filtering-and-verification framework and works well on graph databases with frequent updates. We also propose novel two-step filtering conditions that can filter out most false positives and prove that the two-step filtering conditions satisfy the no-false-negative requirement (no dismissal in answers). Extensive experiments on both real and synthetic graphs show that, compared with six existing counterpart methods, our method can effectively improve the efficiency of subgraph querying.

Citation: Zhu L, Song Q, Guo Y, Du L, Zhu X, et al. (2014) A Coding Method for Efficient Subgraph Querying on Vertex- and Edge-Labeled Graphs. PLoS ONE 9(5): e97178. doi:10.1371/journal.pone.0097178

Editor: Frederique Lisacek, Swiss Institute of Bioinformatics, Switzerland

Received: November 5, 2013; **Accepted:** April 16, 2014; **Published:** May 22, 2014

Copyright: © 2014 Zhu et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Funding: This study is supported by National Natural Science Foundation of China under grant 61373046. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

* E-mail: qbsong@mail.xjtu.edu.cn

Introduction

Labeled graphs, which include both vertex- and edge-labeling, have been widely used to model complicated structures and schemaless data in many domains such as social network [1,2], chemistry [3,4], image analysis [5,6], and XML documents [7,8]. This triggers the needs for effective graph pattern discovery, and the most compelling one is subgraph querying.

The subgraph query problem is to retrieve all the supergraphs of a given graph from a graph database. It can be defined as follows: for a large graph database $D = \{D_1, D_2, \dots, D_n\}$ and a query graph Q , subgraph query is to find all the graphs D_i ($i = 1, 2, \dots, m \leq n$) such that Q is a subgraph of D_i . Fig. 1 shows an example of subgraph query, where the graph database consists of graphs D_1, D_2, D_3 and D_4 , and Q is the query graph. Obviously, only graph D_3 contains Q .

However, it is intractable to find all supergraphs of a query graph from a large graph database, since subgraph query must conduct subgraph isomorphism testing, which is a NP-complete problem [9,10]. In order to address this problem, the filtering-and-verification framework is commonly adopted by most existing methods. These methods first extract some “useful” graph features and build indexes for them; then, in the filtering phase, they traverse the indexes to prune most false positives and generate the candidate graph set; after that, in the verification phase, they validate the candidate graphs with subgraph isomorphism testing and obtain the answer set.

Among the existing subgraph query methods, some of them, such as GraphGrep [11], gIndex [12], FG-Index [13], Treepi [14], Tree+delta [15] and SwiftIndex [16], build the inverted indexes for features that are substructures extracted from graph

databases. The path extracted by GraphGrep is too simple and leads to low filtering efficiency [12]. Other methods have to re-mine frequent substructures and re-build indexes from scratch for the databases with frequent updates, so are quite time consuming [17].

Closure-tree method [18] uses clustering techniques to build indexes. It clusters a set of graphs into several groups, and each group is referred to as a graph closure. The graph closures are then used as nodes to build an index tree. By traversing the index tree, this method finds out a disqualified node via the pseudo subgraph isomorphism testing, and all graphs contained in this node are pruned. As Closure-tree uses the expensive pseudo subgraph isomorphism testing to filter out false positives, it costs too much time in the filtering phase [19,20].

There are subgraph query methods, for example GCoding [17] and LsGCoding [21], which use graph coding methods to build indexes. These methods extract high-quality features from graphs, and map them into numerical space to generate graph codes. For a specific feature, if its corresponding code in a query graph is greater than that of a graph D_i , the query graph is not a subgraph of graph D_i . So, D_i can be filtered as a false positive. According to this logic, these methods build indexes based on codes to filter out false positives. Moreover, these methods individually encode each graph. When the graph database is updated with lots of insertions and deletions, these methods do not need to re-compute graph codes and re-build the indexes from scratch. However, the subtree extracted by GCoding represents partial structure, which degrades its filtering efficiency; and Laplacian matrix used in LsGCoding only represents graphs with unlabeled edges, which makes LsGCoding can only process graphs with unlabeled edges.

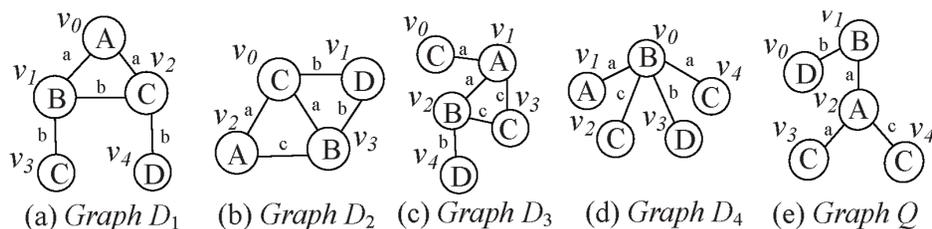


Figure 1. An Example of Subgraph Query. Four labeled graphs (a) *Graph D₁*, (b) *Graph D₂*, (c) *Graph D₃*, and (d) *Graph D₄* compose the database, and (e) *Graph Q* is a query graph.
doi:10.1371/journal.pone.0097178.g001

In order to conduct subgraph query on labeled graphs, we propose a novel Laplacian spectrum and the number of walks based Graph Coding (LnGCoding) method by extending LsGCoding method. The extended method LnGCoding can generate new codes, which include the vertex labels and the labels of adjacent edges consisting of the labels of edges, Laplacian spectrum, and the number of walks. These are new features and not contained in the codes of LsGCoding. Based on the new codes, a novel index tree and a novel two-step filtering conditions are proposed in LnGCoding. Since the codes contain more information, LnGCoding not only conducts subgraph querying on labeled graphs, but also effectively filters out most false positives. Moreover, it works well in the databases with frequent updates. Extensive experiments on both real and synthetic data show that our proposed method LnGCoding can improve the efficiency of subgraph query, especially on dense graphs with labeled edges.

Methods

In this section, we present the novel coding method and its application in subgraph query. At first, we introduce the definitions of vertex and graph codes, the properties of graph features, and the coding method based on these graph features. Then, we state the index building method based on the novel graph codes, and provide the filtering conditions generation method. Finally, based on the indexes and the filtering conditions, we present the filtering-and-verification framework for subgraph query. Note that, a labeled graph is abbreviated to a graph in the rest of this paper.

Definitions of Vertex and Graph Codes

In our method, the vertex and graph codes are based on Laplacian spectrum and the number of walks. Therefore, we first give the definitions of adjacency matrix, Laplacian matrix and spectrum, walk and path. Then, based on these definitions, we define the vertex and graph codes.

Definition 1 (*Adjacency Matrix of Graph*). Given a graph G with n vertices, its adjacency matrix is defined as $M_G = (m_{(i,j)})_{n \times n}$, where

$$m_{(i,j)} = \begin{cases} 1, & \text{if vertex } v_i \text{ is adjacent to vertex } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

Definition 2 (*Laplacian Matrix and Laplacian Spectrum of Graph*). Given a graph G with n vertices, its Laplacian Matrix is defined as $LM_G = (l_{(i,j)})_{n \times n}$, where

$$l_{(i,j)} = \begin{cases} \text{Deg}(v_i), & \text{if } i=j, \\ -1, & \text{if } i \neq j \text{ and vertex } v_i \text{ is adjacent to vertex } v_j, \\ 0, & \text{otherwise,} \end{cases}$$

and $\text{Deg}(v_i)$ is the degree of vertex v_i .

All eigenvalues of LM_G are called graph G 's Laplacian Spectrum.

Definition 3 (*Walk and Path*). A walk in graph G consists of a pair (V, E) of sequences, where V is a vertex sequence: v_0, v_1, \dots, v_k , and E is an edge sequence: e_0, e_1, \dots, e_{k-1} . For $i=0, 1, \dots, k-1$, each successive pair v_i, v_{i+1} of a vertex is adjacent in G , and edge e_i has v_i and v_{i+1} as terminal vertices.

A path is a walk with no repeated edges.

For a path, no edge occurs more than once in the edge sequence. This is different from a walk. The length w of a walk (or path) is the number of edges which occur in the walk (or path).

Definition 4 (*Vertex Code*). Given a graph G and a vertex $v \in G$, the vertex code $vCode$ of v is a quadruple:

$$vCode(v, G) = \langle L(v), A_e(v), Laps(v), N_w(v) \rangle,$$

where $L(v)$ is a length- l_v (l_v is a integer) counter string that denotes the vertex label of v , $A_e(v)$ is a length- l_e (l_e is a integer) counter string that denotes the labels of adjacent edges from v , $Laps(v)$ is the Laplacian spectrum of neighborhood graph of v , and $N_w(v)$ is a length- l_v counter string that denotes the number of walks of length W (W is an integer) from v . Note that, the counter string is an array of multi-digit counters, where each element counts the occurrences of the specified vertices/edges/walks in a graph; And the adjacent edge labels of v are two-tuples, consisting of the labels of edges and the label of the terminal vertex that is on the same edge as v .

Fig. 2 shows $vCode(v_2, Q)$ of vertex $v_2 \in Q$, which occurred in Fig. 1. For the sake of convenience, the first two largest Laplacian eigenvalues are used to denote the Laplacian spectrum of each vertex, and the length W of walks is set to 2.

Definition 5 (*Graph Code*). Given a graph G with n vertices, and that vertex code of vertex v_i is denoted as $vCode(v_i, G) = \langle L(v_i), A_e(v_i), Laps(v_i), N_w(v_i) \rangle$, for $i=0, 1, \dots, n-1$. The graph code $gCode$ of G is defined as a quadruple:

$$gCode(G) = \langle L(G), A_e(G), Lapsseqs(G), N_w(G) \rangle,$$

where $L(G), A_e(G), Lapsseqs(G)$ and $N_w(G)$ are defined as follows:

1. $L(G)[j] = \sum_{i=0}^{n-1} L(v_i)[j], j=0, 1, 2, \dots, l_v - 1;$
2. $A_e(G)[j] = \sum_{i=0}^{n-1} A_e(v_i)[j], j=0, 1, 2, \dots, l_e - 1;$

Vertex	L	A_e	$Laps$		N_w
v_2	000 1	100 000 110	4.17	2.31	1 003

Figure 2. $vCode(v_2, Q)$. The vertex v_2 and graph Q both occurred in Figure 1. doi:10.1371/journal.pone.0097178.g002

- $Lapsseqs(G)_j =$ The ranked Laplacian spectrum $Laps(v_i)[j]$ of all vertices with non-ascending order, $j = 0, 1, i = 0, 1, 2, \dots, n - 1$;
- $N_w(G)[j] = \sum_{i=0}^{n-1} N_w(v_i)[j], j = 0, 1, 2, \dots, l_v - 1$.

Fig. 3 shows the graph code of graph Q . Where $L(Q)$, $A_e(Q)$ and $N_w(Q)$ are generated by combining $L(v_i)$, $A_e(v_i)$ and $N_w(v_i)$ codes of all vertices v_i ($i = 0, 1, 2, 3, 4$) with the element-wise ADD operation. Here, the element-wise ADD operation of counter strings $C_1 = \langle C_1[0], C_1[1], \dots, C_1[k] \rangle$ and $C_2 = \langle C_2[0], C_2[1], \dots, C_2[k] \rangle$ is defined as $\langle C_1[0] + C_2[0], C_1[1] + C_2[1], \dots, C_1[k] + C_2[k] \rangle$, and $C \in \{L, A_e, N_w\}$. For $Lapsseqs(Q)$, we rank all the corresponding eigenvalues $Laps(v_i)[k]$ in the non-ascending order, and the results $Lapsseq1$ and $Lapsseq2$ are its Laplacian spectrum sequences $Lapsseqs$.

The Properties of Graph Features

In our coding method, the codes consist of the following features: i) the labels of vertices and adjacent edges, ii) Laplacian spectrum, and iii) the number of walks. Since these features have the following properties, we can use them to efficiently and effectively filter out false positives.

The labels of vertices and adjacent edges. This is the first graph feature in our proposed method. As we all know, for each vertex (or edge) of a graph, there exists a corresponding vertex (or

edge) in its supergraph. Based on this, we have the lemma as follows.

Lemma 1 Let graph G_1 be a subgraph of graph G_2 , for a specific label l , the number of vertices (or edges) with label l in G_1 is not more than the number of vertices (or edges) with label l in G_2 .

Applying the converse-negative proposition of Lemma 1 to vertices and graphs, we have the following corollaries.

Corollary 1 Given two graphs G_1 and G_2 , and the two vertices $v \in G_1$ and $u \in G_2$ have the same vertex label. If there exists a specific adjacent edge label l , and the number of adjacent edges with label l of vertex v is more than the number of adjacent edges with label l of u , then u is not a corresponding vertex of v .

Corollary 2 Given two graphs G_1 and G_2 , if there exists a specific label l , and the number of vertices (or adjacent edges) with label l in G_1 is more than the number of vertices (or adjacent edges) with label l in G_2 , then G_1 is not a subgraph of G_2 .

Laplacian spectrum. We choose Laplacian spectrum as the second feature, since there exists a relationship between the Laplacian spectrum of a graph and the Laplacian spectra of its subgraphs, and this relationship can be used to efficiently filter out false positives.

In order to prove there does exist the relationship, we first introduce *Min – Max Theorem* [22] as follows.

Theorem 1 (Min – Max Theorem). Given a real symmetric matrix $A_{n \times n}$ (n is an integer), and its eigenvalues are $\lambda_{n-1} \leq \lambda_{n-2} \leq \dots \leq \lambda_0$.

Graph	L	A_e	$Lapsseqs$		N_w
Q	1 211	101 101 112	$Lapsseq1$	$Lapsseq2$	2 644

Vertex	L	A_e	$Laps$		N_w
v_0	1 000	000 001 000	3.0	1.0	1 001
v_1	0 010	000 100 001	4.17	2.31	0 220
v_2	0 001	100 000 110	4.17	2.31	1 003
v_3	0 100	000 000 001	4.0	1.0	0 210
v_4	0 100	001 000 000	4.0	1.0	0 210

$Lapsseq1$	4.17	4.17	4.0	4.0	3.0
$Lapsseq2$	2.31	2.31	1.0	1.0	1.0

Figure 3. $gCode(Q)$. All of $vCode(v_i, Q)$ ($i = 0, 1, 2, 3, 4$) are combined to the $gCode(Q)$. doi:10.1371/journal.pone.0097178.g003

Then the eigenvalues of matrix A are represented as follows:

$$\min_{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \in \mathbb{C}^n} \left\{ \vec{x} \neq 0, \vec{x} \in \mathbb{C}^n, \vec{x} \perp \vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \left(\frac{\vec{x}^* A \vec{x}}{\vec{x}^* \vec{x}} \right) \right\} = \lambda_k,$$

and

$$\min_{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \in \mathbb{C}^n} \left\{ \vec{x} \neq 0, \vec{x} \in \mathbb{C}^n, \vec{x} \perp \vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \left(\frac{\vec{x}^* A \vec{x}}{\vec{x}^* \vec{x}} \right) \right\} = \lambda_k,$$

where \vec{w}_{n-k} ($0 \leq k \leq n-1$) and \vec{x} are n -dimensional vectors, and \vec{x}^* is the transposition of \vec{x} .

In Algebraic Graph Theory [23], according to the properties of Laplacian matrices, the Laplacian matrix of a graph is a real symmetric matrix, and each eigenvalue of a Laplacian matrix is not less than zero. Thus, the Laplacian matrix of a graph is a real symmetric positive semidominant matrix. Applying *Min-Max Theorem* to the positive semidominant matrices, we can have the following corollary.

Corollary 3 Let $A_{n \times n}$ and $B_{n \times n}$ be two real symmetric matrices, and their eigenvalues be $\lambda_{n-1} \leq \lambda_{n-2} \leq \dots \leq \lambda_0$ and $\beta_{n-1} \leq \beta_{n-2} \leq \dots \leq \beta_0$, respectively. If matrix $(B-A)$ is a positive semidominant matrix, then for each $k \in \{0, 1, 2, \dots, n-1\}$, $\lambda_k \leq \beta_k$ holds.

Proof. According to *Min-Max Theorem*, the eigenvalues of B can be represented as follows:

$$\begin{aligned} \beta_k(B) &= \beta_k(B-A+A) \\ &= \min_{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \in \mathbb{C}^n} \left\{ \vec{x} \neq 0, \vec{x} \in \mathbb{C}^n, \vec{x} \perp \vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \left(\frac{\vec{x}^* (B-A+A) \vec{x}}{\vec{x}^* \vec{x}} \right) \right\} \\ &= \min_{\vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \in \mathbb{C}^n} \left\{ \vec{x} \neq 0, \vec{x} \in \mathbb{C}^n, \vec{x} \perp \vec{w}_1, \vec{w}_2, \dots, \vec{w}_{n-k} \left(\frac{\vec{x}^* (B-A) \vec{x}}{\vec{x}^* \vec{x}} + \frac{\vec{x}^* (A) \vec{x}}{\vec{x}^* \vec{x}} \right) \right\} \\ &= \gamma_k(B-A) + \lambda_k(A) \end{aligned}$$

where γ_k is the k -th eigenvalue of matrix $(B-A)$. As matrix $(B-A)$ is a positive semidominant matrix, the γ_k is not less than zero. Thus, we have

$$\beta_k(B) = \gamma_k(B-A) + \lambda_k(A) \geq 0 + \lambda_k(A) = \lambda_k(A).$$

According to Corollary 3, if two real symmetric matrices A and B satisfy that matrix $(B-A)$ is a real symmetric positive

semidominant matrix, the eigenvalues of B is not less than that of A . Since the Laplacian matrix of each graph is a real symmetric positive semidominant matrix, we can apply Corollary 3 to a graph and its subgraphs, and thus have the following theorem.

Theorem 2 For graph G_1 with m vertices and graph G_2 with n ($m \leq n$) vertices, suppose 1) the matrix $A_{m \times m}$ is the Laplacian matrix of G_1 , and $B_{n \times n}$ is the Laplacian matrix of G_2 ; 2) the eigenvalues of matrix A are $\lambda_{m-1} \leq \lambda_{m-2} \leq \dots \leq \lambda_0$, and the eigenvalues of matrix B are $\beta_{n-1} \leq \beta_{n-2} \leq \dots \leq \beta_0$. If G_1 is a subgraph of G_2 , then for each $k=0, 1, \dots, m-1$, Laplacian spectra of G_1 and G_2 satisfy $\lambda_k(G_1) \leq \beta_k(G_2)$.

Proof. (sketch) Since G_1 is a subgraph of G_2 , we can first generate a new graph G_3 by adding $(n-m)$ vertices to graph G_1 , and these vertices occur in G_2 but not in G_1 ; and then achieve the $n \times n$ Laplacian matrix A' of G_3 by adding $(n-m)$ elements "0" to the $m \times m$ matrix A . This ensures that G_3 is also a subgraph of G_2 , and A' have the same non-zero eigenvalues as A . Meanwhile, we generate a new graph G_4 by removing the edges in G_3 from G_2 . And Laplacian matrix of G_4 can be denoted as matrix $(B-A')$. For a given graph, its Laplacian matrix is a real symmetric positive semidominant matrix. Thus, Laplacian matrices A' , B and $(B-A')$ are all real symmetric positive semidominant matrices. According to Corollary 3, for each $k \in \{0, 1, 2, \dots, n-1\}$, we have $\lambda_k(G_3) \leq \beta_k(G_2)$. Furthermore, for each $k \in \{0, 1, 2, \dots, m-1\}$, $\lambda_k(G_1) \leq \beta_k(G_2)$ holds.

Applying the converse-negative proposition of Theorem 2 to Laplacian spectra of graphs, we have a useful corollary as follows.

Corollary 4 Given two graphs G_1 with m vertices and G_2 with n vertices ($m \leq n$), Laplacian spectrum of G_1 is $\lambda_{m-1} \leq \lambda_{m-2} \leq \dots \leq \lambda_0$, and Laplacian spectrum of G_2 is $\beta_{n-1} \leq \beta_{n-2} \leq \dots \leq \beta_0$. If there exists an integer k ($0 \leq k \leq m-1$) such that $\lambda_k(G_1) > \beta_k(G_2)$, then graph G_1 is not a subgraph of graph G_2 .

The number of walks. Paths of a graph are easier to extract and manipulate than trees and subgraphs, so GraphGrep [11] uses paths as index features. The indexes built on this kind of features are usually huge especially when graph databases are large and diverse, thus this method can be inefficient [12]. However, we find that the number of walks of length $k \in \mathbb{N}$ between two terminal vertices can also preserve the basic information of a graph, and the walks of a graph are much more easy to extract and manipulate than paths. Inspired by this, we extract the metrics including the number of walks with specific length as the feature for graph coding and further indexing.

Generally speaking, for each walk from vertex v_i to vertex v_j in a graph, there must exist a corresponding walk from v'_i (corresponding to v_i) to v'_j (corresponding to v_j) in its supergraph. Thus, we have the following lemma.

Lemma 2 Given two graphs G_1 and G_2 , and G_1 is a subgraph of G_2 . For a vertex $v_i \in G_1$, there exists a corresponding vertex $v'_i \in G_2$, and v'_i satisfies that the number of walks of length W from v_i to all vertices with label l in graph G_1 is not more than the number of walks of length W from v'_i to all vertices with label l in graph G_2 .

Applying the converse-negative proposition of Lemma 2 to vertices and graphs, we have two useful corollaries as follows.

Corollary 5 Given two graphs G_1 and G_2 , and the vertices $v \in G_1$ and $u \in G_2$ have the same vertex label. If there exists a specific vertex label l , and the vertex label l satisfies that the number of walks of length W from v to all vertices with label l in graph G_1 is more than the number of walks of length W from u to all vertices with label l in graph G_2 , then u is not a corresponding vertex of v .

Corollary 6 Given two graphs G_1 and G_2 , if there exists a specific vertex label l , and it satisfies that the number of walks of length W from all vertices to all vertices with label l in graph G_1 is more than the number of

walks of length W from all vertices to all vertices with label l in graph G_2 , then G_1 is not a subgraph of G_2 .

According to the above corollaries, we can use these features to filter out false positives. In order to speed up the comparisons between graph features, we map these features into the numerical space to generate vertex and graph codes. In the following subsection, we discuss how to generate vertex and graph codes.

The Proposed Coding Method

In this subsection, we present the novel coding method consisting of three parts: i) L and A_e coding, ii) Laplacian spectrum coding, and iii) N_w coding.

L and A_e coding. For a vertex v , as stated in former section, $L(v)$ is a length- l_v counter string to denote its vertex label, and $A_e(v)$ is a length- l_e counter string to denote its adjacent edge label. For each distinct vertex (or adjacent edge) label, we use hash function to set $K(\geq 1)$ out of l_v (or l_e) elements to 1. Then, $L(v)$ is directly generated from the hash function of vertex label, and the code of each adjacent edge is directly generated from the hash function of adjacent edge label. By adding all adjacent edge codes with the element-wise ADD operation, we can generate $A_e(v)$.

For a graph G , $L(G)$ and $A_e(G)$ are generated by adding $L(v)$ and $A_e(v)$ of all vertices with the element-wise ADD operation.

In Fig. 4, we use vertex v_2 and graph Q as examples to illustrate the generation process of the L and A_e codes.

Figs. 4(a) and 4(b) are the hash functions of vertex label and adjacent edge label, respectively. For convenience sake, we denote distinct vertex (or adjacent edge) label by setting K to be 1.

For vertex v_2 , $L(v_2)$ is the counter string of v_2 in the hash function of vertex label. In order to generate $A_e(v_2)$, we first extract all the adjacent edges of vertex v_2 : $\langle a, B \rangle$, $\langle a, C \rangle$ and $\langle c, C \rangle$. Then, we use hash function of adjacent edge label to encode each adjacent edge. Finally, we add these adjacent edge codes to generate $A_e(v_2)$, as shown in Fig. 4(c).

For graph Q , we combine the $L(v_i)$ and $A_e(v_i)$ of all vertices v_i ($i=0,1,2,3,4$) to generate its $L(Q)$ and $A_e(Q)$ codes by performing the element-wise ADD operation, as shown in Fig. 4(d).

Laplacian spectrum coding. Suppose graph G has n vertices. For each vertex v , we first generate its Level- N Spanning Graph, and then choose some Laplacian eigenvalues of Level- N Spanning Graph to generate its Laplacian spectrum $Lap(v)$. The Level- N Spanning Graph of a vertex is defined as follows.

Definition 6 (Level- N Spanning Graph). Given a graph G and a vertex $v \in G$, Level- N Spanning Graph of v , denoted as $LNSG(G, N, v)$, is a subgraph representing the local structure around v , where v is a center vertex, and the vertices and edges in $LNSG(G, N, v)$ must satisfy the follows:

- for each vertex $v' \in G$, if the length of walk between v and v' is not more than N , vertex v' is in $LNSG(G, N, v)$;
- for each edge $e \in G$, if the two terminal vertices of e are both in $LNSG(G, N, v)$, edge e is in $LNSG(G, N, v)$.

According to the above definition, Level- N Spanning Graph of a vertex is unique. By ranking the $Lap(v)$ of all the vertices in graph G in non-ascending order, we obtain $Lapsseqs(G)$.

In order to better understand the Level- N Spanning Graph, Table 1: Algorithm 1 lists the generation process of $LNSG(G, N, v)$.

In Table 1: Algorithm 1, Lines 1–2 initialize the vertex set and edge set of $LNSG(G, N, v)$, respectively. Line 3 adds vertex v to the vertex set of $LNSG(G, N, v)$. Line 4 uses the function $SEEK(v, G, N)$ to find the other vertices in $LNSG(G, N, v)$. The Function $SEEK(v, G, N)$ uses the *depth-first-search* to traverse graph G and finds out all vertices in $LNSG(G, N, v)$. Lines 5–9 look for all edges of $LNSG(G, N, v)$. For an edge, if its two terminal vertices are both in the vertex set of $LNSG(G, N, v)$, we add this edge to the edge set of $LNSG(G, N, v)$.

Fig. 5 depicts the examples of Level- N Spanning Graph in graphs Q and D_3 , which both shown in Fig. 1. Fig. 5 (a) shows some Level- N Spanning Graphs for vertices v_0, v_1, v_2 and v_4 in graph D_3 . Fig. 5(b) shows some Level- N Spanning Graphs for vertices v_0, v_1, v_2 and v_3 in graph Q . Obviously, Level- N Spanning

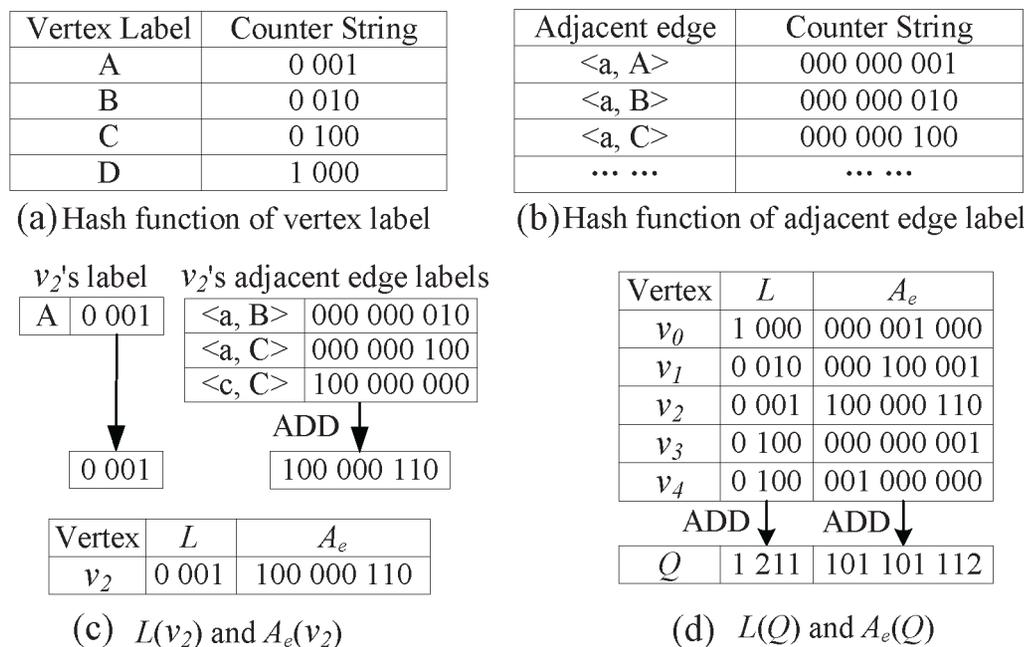


Figure 4. L and A_e coding for graph Q . In this figure, (a) is the hash function of vertex label, (b) is the hash function of adjacent edge label, (c) is the generating process of L and A_e codes for vertex v_2 , and (d) is the generating process of L and A_e codes for graph Q . doi:10.1371/journal.pone.0097178.g004

Table 1. Algorithm 1 Level- N Spanning Graph Generation.

Input: G is a graph, v is a vertex in G , N is the Level number of $LNSG$;
Output: $LNSG(G, N, v)$;
1: $LNSG.V := \emptyset$; //The vertex set
2: $LNSG.E := \emptyset$; //The edge set
3: $LNSG.V := LNSG.V \cup \{v\}$;
4: SEEK (v, G, N);
5: for each edge $e \in G$ do
6: if two terminal vertices of e are in $LNSG.V$ then
7: Insert the edge e into $LNSG.E$;
8: end if
9: end for
10: return $LNSG(G, N, v)$;
Function: SEEK (v, G, N)
1: if $N = 0$ then
2: return ;
3: end if
4: for each neighbor vertex u of vertex v do
5: if u does not exist in $LNSG.V$ then
6: Insert the vertex u into $LNSG.V$;
7: SEEK ($u, G, N - 1$);
8: end if
9: end for

doi:10.1371/journal.pone.0097178.t001

Graph of v_0, v_1, v_2 and v_3 in Q are the subgraphs of that of v_4, v_2, v_1 and v_0 in D_3 , respectively.

From Fig. 5, we also find that there exists the relationship of $LNSG$ between two vertices, which are described by Lemma 3 as follows.

Lemma 3 Let G_1 and G_2 be two graphs, and $v \in G_1$ and $v' \in G_2$ be two vertices which have the same vertex label, if G_1 is a subgraph of G_2 and v' is the corresponding vertex of v in G_2 , then Level- N Spanning Graph of vertex v is a subgraph of the Level- N Spanning Graph of vertex v' .

Proof. According to the subgraph isomorphism relationship, for each vertex u ($u \neq v$) in $LNSG(G_1, N, v)$, there exists a corresponding vertex u' ($u' \neq v'$) in graph G_2 . For each edge e in $LNSG(G_1, N, v)$, there exists a corresponding edge e' in graph G_2 . According to the definition of Level- N Spanning Graph, there exists a walk of length w ($1 \leq w \leq N$) between vertices u and v in $LNSG(G_1, N, v)$. For graph G_2 , there also exists a corresponding walk of length w between vertices u' and v' . Thus, vertex u' is in $LNSG(G_2, N, v')$. That is, the corresponding vertex of each vertex in $LNSG(G_1, N, v)$ is in $LNSG(G_2, N, v')$. Similarly, all the corresponding edges of $LNSG(G_1, N, v)$ are also in $LNSG(G_2, N, v')$. Thus, $LNSG(G_1, N, v)$ is a subgraph of $LNSG(G_2, N, v')$.

In the proposed method, we extract some Laplacian eigenvalues of $LNSG(G, N, v)$ to generate $Laps(v)$, and generate $Lapsseqs(G)$ via ranking the $Laps(v)$ of all the vertices.

In Fig. 6, we use graph Q as example to illustrate the generating process of $Laps(v)$ and $Lapsseqs(G)$. We first compute Laplacian spectrum of each vertex v in graph Q and extract first two largest Laplacian eigenvalues $Eigenvalue1$ and $Eigenvalue2$ to generate $Laps(v)$. According to non-ascending order, we rank the corresponding eigenvalues $Eigenvalue1$ and $Eigenvalue2$ of all vertices to generate $Lapsseqs(Q)$, which contains two Laplacian spectrum sequences $Lapsseq1$ and $Lapsseq2$. For convenience sake, we choose

first two largest eigenvalues to denote $Laps(v)$, and the level N of $LNSG$ is set to 2.

N_w coding. A length- l_v counter string is used to code $N_w(v)$ (or $N_w(G)$), which is the number of walks of length W . It is generated from the W -th power of graph G 's adjacency matrix. In Algebraic Graph Theory [23], there exists a lemma with respect to the number of walks of length W as follows.

Lemma 4 Let M_G be the adjacency matrix of graph G , then the number of walks of length W from the i -th vertex of G to the j -th vertex is $(M_G^W)_{ij}$ that is the entry in row i and column j of the W -th power of M_G .

Given graph G and its adjacency matrix M_G , if the entry in row i and column j of M_G is 1, there exists a walk of length 1 between the i -th vertex and the j -th vertex in G . Similarly, the entry in row i and column j of the W -th power of adjacency matrix (M_G^W) is k and only if there exists k (≥ 0) walks of length W between the i -th vertex and j -th vertex, where the vertices in a walk can be repetitive. Fig. 7 shows the M_Q and M_Q^2 of graph Q respectively.

With the W -th power of adjacency matrix M_G^W of graph G , for each vertex $v_i \in G$, we first extract all its walks of length W , and generate tuple $\langle *, Label(v_j) \rangle$ by recording the label of the terminal vertex v_j in each walk. For the distinct tuple $\langle *, Label(v_j) \rangle$, we use the hash function of walks to set K out of l_v elements to 1. Then, we map each tuple $\langle *, Label(v_j) \rangle$ into the numerical space by using the hash function of walks, and the result is $N_w(\langle *, Label(v_j) \rangle)$. Finally, we add $N_w(\langle *, Label(v_j) \rangle)$ of all walks to generate $N_w(v_i)$ with element-wise ADD operation. Similarly, we add $N_w(v_i)$ of all vertices to generate $N_w(G)$. Note that, tuple $\langle *, Label(v_j) \rangle$ is used to represent all the walks of length W from vertex v_i to vertex v_j , regardless the vertices or edges between them are same or not; And symbol $' * '$ just represents the other vertices and edges appeared in a walk.

In Fig. 8, we use vertex v_2 and graph Q as examples to illustrate the generation process of $N_w(v_2)$ and $N_w(Q)$.

Fig. 8(a) is the hash function of walks, where we represent the distinct walk by setting 1 ($K = 1$) out of l_v elements to 1, and the length W is set to 2. For vertex v_2 , we first extract its four walks of length 2: three walks $\langle *, A \rangle$ and one walk $\langle *, D \rangle$ according to M_Q^2 in Fig. 7, and generate $N_w(\langle *, A \rangle)$ and $N_w(\langle *, D \rangle)$ according to the hash function of walks. By adding $N_w(\langle *, A \rangle)$ and $N_w(\langle *, D \rangle)$ with element-wise ADD operation, we obtain $N_w(v_2)$, as shown in Fig. 8(b). For graph Q , we add $N_w(v_i)$ of all the vertices v_i ($i = 0, 1, 2, 3, 4$) to get $N_w(Q)$, as shown in Fig. 8(c).

With the help of the above methods, we can extract these graph features and generate the corresponding codes. By combining $L(v)$, $A_e(v)$, $Laps(v)$ and $N_w(v)$ of the vertex v in a graph, we can generate $vCode(v, Q)$, as shown in Fig. 2. By combining $L(v_i)$, $A_e(v_i)$, $Laps(v_i)$ and $N_w(v_i)$ of all the vertices v_i in graph Q we can generate graph code $gCode(Q)$, as shown in Fig. 3.

Index Building

Based on the coding method, we build a graph index named LnGCode-Tree, which can improve the filtering efficiency. The construction method of the LnGCode-Tree is presented below.

LnGCode-Tree is based on the GCode-Tree, which is first proposed in GCoding [17]. Similar to S-Tree [24] and GCode-Tree, LnGCode-Tree is also used to handle the signature files, and can be efficient for reducing the number of pairwise comparisons. LnGCode-Tree is a balanced tree as well, and each index node in LnGCode-Tree has at least m ($m \leq 2$) and at most M ($(M + 1) / 2 \geq m$) children. Different from GCode-Tree, we use the labels of vertices and adjacent edges and the number of walks

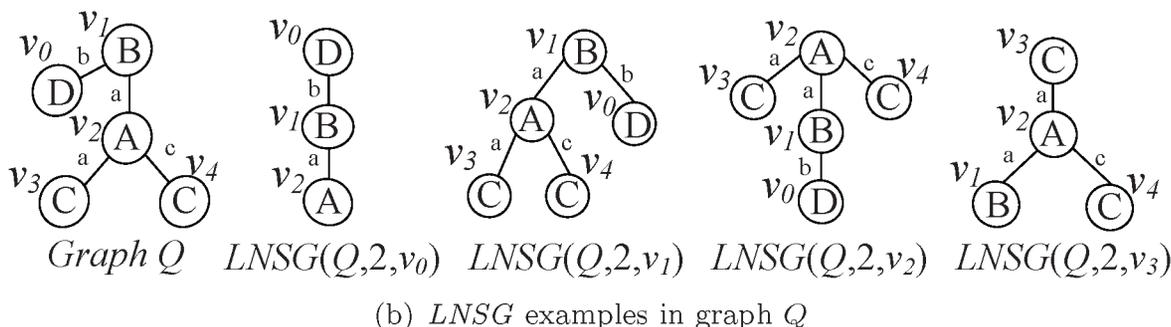
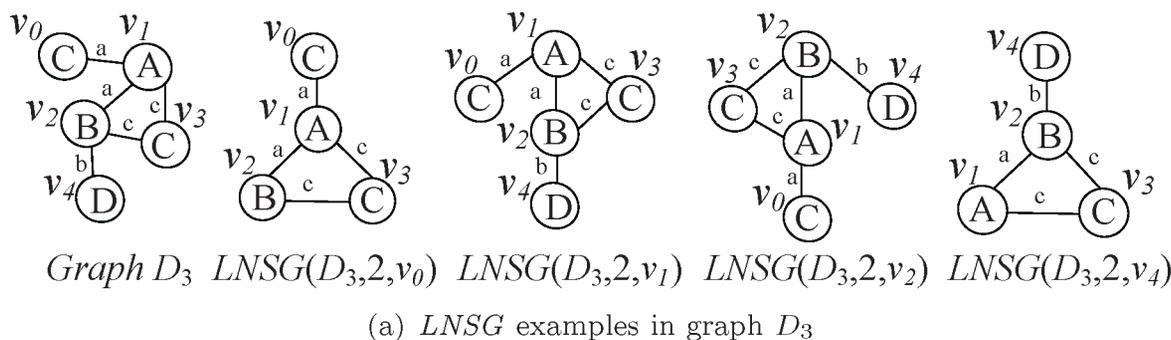


Figure 5. Examples of $LNSG$ in graphs D_3 and Q . In the example, (a) includes the graph D_3 and the $LNSG$ of vertices v_0, v_1, v_2, v_4 ; (b) includes the graph Q and the $LNSG$ of vertices v_0, v_1, v_2, v_3 .
doi:10.1371/journal.pone.0097178.g005

to build LnGCode-Tree, while GCoding just uses the labels of vertices and adjacent edges to build GCode-Tree.

Fig. 9 is a LnGCode-Tree, it is built for the graphs in Fig. 1. The building process can be illustrated as follows.

For each graph D_i , its $L(D_i)$, $A_e(D_i)$ and $N_w(D_i)$ codes of $gCode(D_i)$ are used to build index tree. For graphs with the same L , A_e and N_w codes, a leaf node $LNode$ is built. The code of $LNode$ is consist of the L , A_e and N_w codes of graphs D_i ($i=1,2,3,4$), and $LNode$ also contains the identities of these graphs. An intermediate node $INode$ has m children $CNode$, its code is generated as follows: for each element j in $INode$, $INode.L[j] = \text{Max}(CNode_i.L[j])$, $INode.A_e[j] = \text{Max}(CNode_i.A_e[j])$, and $INode.N_w[j] = \text{Max}(CNode_i.N_w[j])$, where $i=1, \dots, m$.

After the index tree is built, our method generates novel two-step filtering conditions, and follows the filtering-and-verification framework to conduct query processing.

Two-step Filtering Conditions

In this subsection, we present the two-step filtering conditions according to the properties of the graph features, and prove that these conditions satisfy the no-false-negative requirement.

Filtering condition of vertices. Applying Corollary 1, Lemma 3, Theorem 2 and Corollary 5 to vertices, we have a theorem as follows.

Theorem 3 Let G_1 and G_2 be two graphs, $v \in G_1$ and $v' \in G_2$ be two vertices, and $vCode(v, G_1) = \langle L(v), A_e(v), Laps(v), N_w(v) \rangle$ and $vCode(v', G_2) = \langle L(v'), A_e(v'), Laps(v'), N_w(v') \rangle$ be the codes of vertices v and v' respectively. If G_1 is a subgraph of G_2 and v' is the

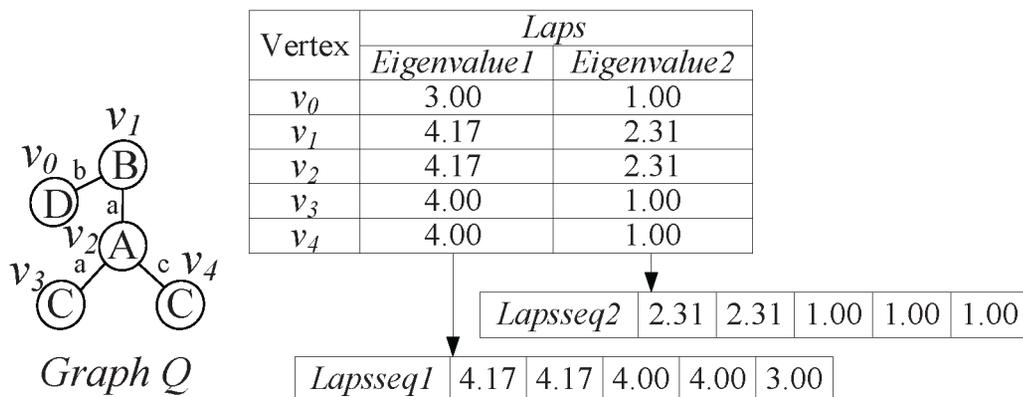


Figure 6. $Laps(v)$ and $Lapsseqs(G)$ coding for graph Q . The $Lapsseqs$ of graph Q is generated by ranking the $Laps$ of all vertices v_i ($i=0,1,2,3,4$).
doi:10.1371/journal.pone.0097178.g006

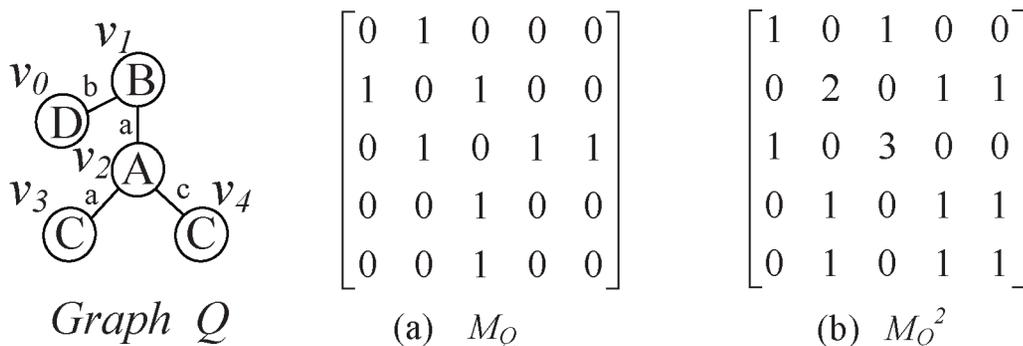


Figure 7. M_Q and M_Q^2 . For the graph Q , (a) is the adjacency matrix of graph Q ; (b) is the square of the adjacency matrix of Q . doi:10.1371/journal.pone.0097178.g007

corresponding vertex of v , then $vCode(v, G_1)$ and $vCode(v', G_2)$ satisfy the following conditions:

1. $L(v')[i] = L(v)[i], i = 0, 1, 2, \dots, l_v - 1;$
2. $A_e(v')[i] \geq A_e(v)[i], i = 0, 1, 2, \dots, l_e - 1;$
3. $Laps(v')[i] \geq Laps(v)[i], i = 0, 1;$
4. $N_w(v')[i] \geq N_w(v)[i], i = 0, 1, 2, \dots, l_v - 1.$

Proof. Since G_1 is a subgraph of G_2 , and v' is the corresponding vertex of v , thus the labels of v' and v are same and their L codes are identical as well. That is, their L codes satisfy condition 1). According to Corollary 1, for each edge label l , the number of adjacent edges with label l of v' is not less than that of v , thus their A_e codes satisfy condition 2). According to Lemma 3 and Theorem 2, $LNSG(G_1, N, v)$ is a subgraph of $LNSG(G_2, N, v')$, and the Laplacian spectra of $LNSG(G_1, N, v)$ and $LNSG(G_2, N, v')$ satisfy condition 3). According to Corollary 5, for each vertex label l , the number of walks of length W from v to all the vertices with label l in G_1 is not more than the number of walks of length W from v' to all the vertices with label l in G_2 , thus their N_w codes satisfy condition 4). Therefore, Theorem 3 is correct.

Theorem 3 shows the relationship between the codes of a vertex and its corresponding vertex. Applying the converse-negative proposition of Theorem 3 to vertices, we have the following first filtering condition.

Filtering condition 1 (*Filtering Condition of Vertices*). Let G_1 and G_2 be two graphs, and $vCode(v, G_1) = \langle L(v), A_e(v), Laps(v), N_w(v) \rangle$ be the code of vertex $v \in G_1$, if there does not exist a vertex $v' \in G_2$, and its code

$vCode(v', G_2) = \langle L(v'), A_e(v'), Laps(v'), N_w(v') \rangle$ satisfies the following conditions:

1. $L(v')[i] = L(v)[i], i = 0, 1, 2, \dots, l_v - 1;$
2. $A_e(v')[i] \geq A_e(v)[i], i = 0, 1, 2, \dots, l_e - 1;$
3. $Laps(v')[i] \geq Laps(v)[i], i = 0, 1;$
4. $N_w(v')[i] \geq N_w(v)[i], i = 0, 1, 2, \dots, l_v - 1.$

then G_1 is not a subgraph of G_2 .

Lemma 5 *Filtering Condition of Vertices satisfies no-false-negative requirement for subgraph query problem.*

Proof. (Proof by contradiction) We assume the *Filtering Condition of Vertices* does not satisfy the no-false-negative requirement. Let G_2 be a graph and G_1 be its subgraph, and *Filtering Condition of Vertices* do not satisfy the no-false-negative requirement if and only if G_2 can be pruned by *Filtering Condition of Vertices*. That is, for a specific vertex $v \in G_1$, there does not exist a vertex $v' \in G_2$, and the $vCodes$ of v and v' satisfy the conditions 1), 2), 3) and 4) in *Filtering Condition of Vertices*. According to Theorem 3, for each vertex $v \in G_1$, there must exist a corresponding vertex $v' \in G_2$, and the $vCodes$ of v' and v satisfy the conditions 1), 2), 3) and 4) in *Filtering Condition of Vertices*. Thus, graph G_2 cannot be pruned by *Filtering Condition of Vertices*. This contradicts the assumption. Therefore, Lemma 5 is correct.

Filtering conditions of graphs. Applying Corollary 2, Lemma 3, Theorem 2 and Corollary 6 to graphs, we have another theorem as follows.

Theorem 4 Let m and n ($m \leq n$) be the numbers of vertices and $gCode(G_1) = \langle L(G_1), A_e(G_1), Lapsseqs(G_1), N_w(G_1) \rangle$ and $gCode(G_2) = \langle L(G_2), A_e(G_2), Lapsseqs(G_2), N_w(G_2) \rangle$ be the codes

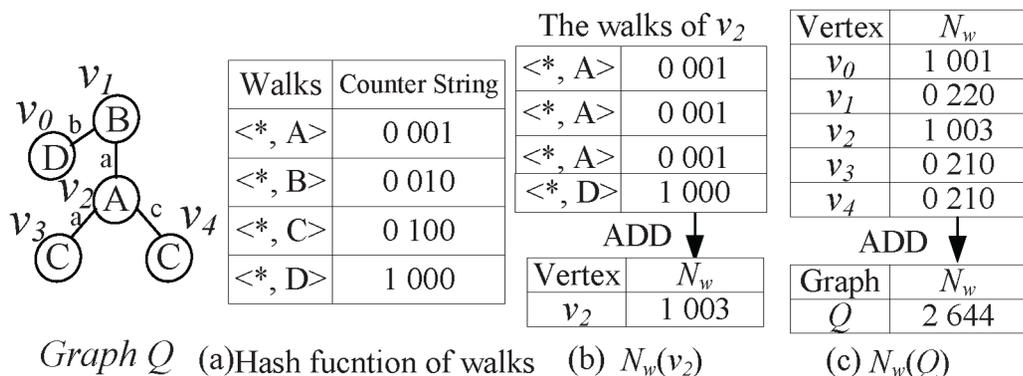


Figure 8. N_w Coding. For the graph Q , (a) is the hash function of walks; (b) is the generating process of N_w code for vertex v_2 ; (c) is the generating process of N_w code for graph Q . doi:10.1371/journal.pone.0097178.g008

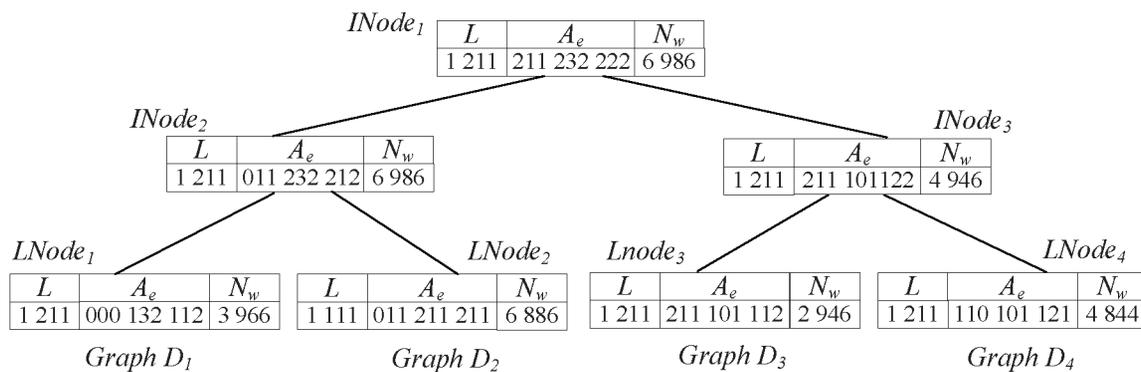


Figure 9. An example of LnGCoding-Tree.
doi:10.1371/journal.pone.0097178.g009

of graphs G_1 and G_2 respectively, if G_1 is a subgraph of G_2 , then their graph codes $gCode(G_1)$ and $gCode(G_2)$ satisfy the following conditions:

1. $L(G_1)[i] \leq L(G_2)[i], i = 0, 1, 2, \dots, l_v - 1;$
2. $A_e(G_1)[i] \leq A_e(G_2)[i], i = 0, 1, 2, \dots, l_e - 1;$
3. $Lapsseqs(G_1)_k[i] \leq Lapsseqs(G_2)_k[i], i = 0, 1 \quad \text{a n d} \quad k = 0, 1, \dots, m - 1;$
4. $N_w(G_1)[i] \leq N_w(G_2)[i], i = 2, \dots, l_v - 1.$

Proof. (sketch) The conditions 1), 2) and 4) can be directly derived from Corollary 2 and Corollary 6. Condition 3) is proved as follows. Since $Lapsseqs(G_1)_k[i]$ is a sorted list in non-ascending order, there exist $k + 1$ vertices $v_j (j = 0, 1, 2, \dots, k)$ in G_1 , and the Laplacian eigenvalue $\lambda_j[i] \geq Lapsseqs(G_1)_k[i]$. According to Theorem 2, for each vertex $v_j (j = 0, 1, 2, \dots, k)$ in G_1 , there exists a corresponding vertex v'_j in G_2 , and the Laplacian eigenvalues $\lambda'_j[i] \geq \lambda_j[i] (j = 0, 1, 2, \dots, k)$. That is, $Lapsseqs(G_1)_k[i] \leq Lapsseqs(G_2)_k[i] (i = 0, 1)$. Thus the condition (3) is correct.

Applying the converse-negative proposition of Theorem 4 to graphs, we have the second filtering condition.

Filtering condition 2 (*Filtering Condition of Graphs*). Let m and $n (m \leq n)$ be the numbers of vertices and $gCode(G_1) = \langle L(G_1), A_e(G_1), Lapsseqs(G_1), N_w(G_1) \rangle$ and $gCode(G_2) = \langle L(G_2), A_e(G_2), Lapsseqs(G_2), N_w(G_2) \rangle$ be the codes of graphs G_1 and G_2 respectively, if $gCode(G_1)$ and $gCode(G_2)$ do not satisfy the following conditions:

1. $L(G_1)[i] \leq L(G_2)[i], i = 0, 1, 2, \dots, l_v - 1;$
2. $A_e(G_1)[i] \leq A_e(G_2)[i], i = 0, 1, 2, \dots, l_e - 1;$
3. $Lapsseqs(G_1)_k[i] \leq Lapsseqs(G_2)_k[i], i = 0, 1 \quad \text{a n d} \quad k = 0, 1, \dots, m - 1;$
4. $N_w(G_1)[i] \leq N_w(G_2)[i], i = 2, \dots, l_v - 1.$

then G_1 is not a subgraph of G_2 .

Lemma 6 *Filtering Condition of Graphs satisfies the no-false-negative requirement for subgraph query problem.*

Proof. Similar to Lemma 5, this lemma can be proved by contradiction according to Theorem 4.

Filtering and Verification

Based on the index and filtering conditions, we follow the filtering-and-verification framework to query subgraphs.

Firstly, we use two-step filtering conditions to filter out false positives. In the first step, we traverse the LnGCoding-Tree of graph database with *Filtering Condition of Graphs*. Specifically, the graph

code $gCode(Q)$ of query graph Q is compared with the intermediate node $INode_k$. If there exists an element i , and it satisfies one of these conditions: i) $gCode(Q).L[i] > INode_k.L[i]$; ii) $gCode(Q).A_e[i] > INode_k.A_e[i]$; or iii) $gCode(Q).N_w[i] > INode_k.N_w[i]$, then the children of $INode_k$ are pruned; otherwise, the graph code $gCode(Q)$ is compared with each child of $INode_k$. For the leaf node $LNode_k$, if there exists an element i , and it satisfies one of these conditions: i) $gCode(Q).L[i] > LNode_k.L[i]$; ii) $gCode(Q).A_e[i] > LNode_k.A_e[i]$; or iii) $gCode(Q).N_w[i] > LNode_k.N_w[i]$, then the graphs contained in $LNode_k$ can be pruned as false positives; otherwise, the graphs contained in $LNode_k$ are added to candidate graphs. After traversing LnGCoding-Tree, LnGCoding filters out some false positives, so the graph database is reduced. Then we compare the $Lapsseqs$ of the query graph with those of the reduced graph database, since LnGCoding-Tree only includes L and A_e, N_w codes. Through this step, we obtain the primary candidate graph set for the query graph.

This step can be illustrated by the graphs in Fig. 1 and the corresponding LnGCoding-Tree in Fig. 9. When traversing $INode_2$, we find that $gCode(Q).A_e[0] = 1 > INode_2.A_e[0] = 0$, thus graphs D_1 and D_2 are pruned. When traversing $LNode_4$, we find that $gCode(Q).A_e[2] = 1 > LNode_4.A_e[2] = 0$, so graph D_4 is pruned. Then, by comparing the $Lapsseqs$ of query graph Q and graph D_3 , we find D_3 is a candidate of Q .

In the second step, we use *Filtering Condition of Vertices* to filter out more false positives. Specifically, we compare each vertex code of the query graph with all the vertex codes of each graph in the primary candidate graph set until all the candidate vertices of this vertex have been found. By now, the candidate graph set and the candidate vertex set are generated.

In Fig. 10, we use graph Q as query graph and D_3 as the primary candidate graph set to illustrate the second step filtering process.

The vertex codes of all vertices in graphs D_3 and Q are shown in Fig. 10(a). After filtering with *Filtering Condition of Vertices*, we generate the candidate vertex set of each vertex in query graph Q as shown in Fig. 10(b). For each vertex in Q , there exist the corresponding candidate vertices in D_3 . Thus, D_3 is a candidate graph of Q .

After the filtering is finished, in the verification phase, we use the state-of-the-art subgraph isomorphism algorithm VF2 [25,26] to validate each candidate graph, and obtain the supergraph set for a query graph.

Vertex	L	A_e	$Lapsseqs$		N_w
v_0	0 100	000 000 001	4.00	3.00	0 210
v_1	0 001	100 000 110	4.30	3.62	1 113
v_2	0 010	100 100 001	4.30	3.62	0 231
v_3	0 100	011 000 000	4.30	3.62	1 311
v_4	1 000	000 001 000	4.00	3.00	1 101

↕

v_0	1 000	000 001 000	3.00	1.00	1 001
v_1	0 010	000 100 001	4.17	2.31	0 220
v_2	0 001	100 000 110	4.17	2.31	1 003
v_3	0 100	000 000 001	4.00	1.00	0 210
v_4	0 100	001 000 000	4.00	1.00	0 210

→

Graph D_3		Vertices in Q	Candidate Vertex Set in D_3
		v_0	v_4
		v_1	v_2
		v_2	v_1
		v_3	v_0
		v_4	v_3

Graph Q

(a) $vCode$ list of each vertex in graphs D_3 and Q (b) Candidate Vertex Set of graph Q

Figure 10. A Filtering Example. For the labeled graphs D_3 and Q , (a) lists the $vCodes$ of all vertices in D_3 and Q ; (b) lists the candidate vertices set for each vertex in Q .
doi:10.1371/journal.pone.0097178.g010

Experimental Results and Discussion

In this section, after introducing the data source, the benchmark methods and parameter setting, and the evaluation criteria, we report the experimental results on efficiency comparison of the different methods, and test the scalability of our method.

Data Source

In this study, both real and synthetic graph databases are used.

Real graph database. The AIDS antiviral screen database contains 43,905 classified chemical molecules, and is publicly available. Many researchers such as Yan et al. [12], Shang et al. [16], Zou et al. [17], and He and Singh [18] used one of its subset to test their methods, we chose it as benchmark data as well.

The subset consists of 10,000 graphs as default database. On average, each graph has 25.4 vertices and 27.3 edges, which means that most of graphs in this real graph database are sparse graphs. Six query graph sets Q_4 , Q_8 , Q_{12} , Q_{16} , Q_{20} and Q_{24} are used to validate the efficiency of subgraph querying methods. Each query graph set Q_i ($i=4,8,12,16,20,24$) consists of 1,000 query graphs with i edges.

Synthetic graph database. GraphGen [27] is a synthetic graph generator. In order to test the performance of existing methods on dense graphs, Han et al. [19,20] used it to generate the synthetic graph database Synthetic.10K.E30.D5.L50. The cardinality of the synthetic database is 10,000, the average size of graphs is 30, the density for each graph is 0.5, and the number of vertex/edge labels is 50.

Benchmark Methods and Parameter Setting

Benchmark methods. The representative methods gIndex [12], FG-Index [13], Tree+delta [15], SwiftIndex [16], GCoding [17], and Closure-tree [18] are selected to be compared with our method. Since LsGCoding [21] aims at coding graphs with unlabeled edge, and optimizes the subgraph isomorphism algorithm according to the properties of graphs with unlabeled edge, thus in our experiments on graph databases with labeled edges, we do not compare LsGCoding with our method.

All these methods are implemented on the iGraph framework [19,20], this enables fair performance comparisons for different methods.

Parameter setting. Our proposed method has three parameters: the level of $LNSG$, the number of first largest Laplacian eigenvalues, and the length of walks. Fig. 11 shows the impact of these parameters on the real graph database.

Fig. 11(a) shows the impact of the level of $LNSG$ on the candidate set size. It indicates that when we choose more levels of $LNSG$, the candidate set size will become smaller. However, the more levels of $LNSG$ we choose, the more time will be consumed in computing Laplacian spectrum. Moreover, choosing 3 or more levels cannot lead to significant reduction in the candidate set size. Therefore, the level N of $LNSG$ is set to 2.

Fig. 11(b) shows the impact of Laplacian eigenvalues on the candidate set size. We observe that choosing more Laplacian eigenvalues can reduce the size of the candidate graph set, but will result in the larger graph code database and more code comparison time. At the same time, choosing 4 or more Laplacian eigenvalues cannot lead to significant reduction in the candidates set size. Therefore, we choose the first three largest eigenvalues in our method.

Fig. 11(c) shows the impact of the length of walks on the candidate set size. From it we know that longer length of walks will result in more computation time of matrix M^W , and choosing 3 or greater length cannot lead to significant reduction in the candidate set size. Thus we set the length W to 2.

As recommended in [17] and [28], the length of L , A_e and N_w codes are set to 30 (i.e. $l_v = l_e = 30$).

For methods gIndex, Tree+delta, SwiftIndex, GCoding and Closure-tree, the recommended parameter values are used. That is, for all substructures based index methods, the support threshold is set to 10%, and the maximum feature size $maxL$ is set to 10. For gIndex and SwiftIndex, γ_{min} is set to 2. For FG-Index, δ is set to 0.1. For gIndex, the same size-increasing function as in [12] is followed. For GCoding, the level N of $LNPT$ is set to 2 and the number of eigenvalues to 2.

Evaluation Criteria

A subgraph query algorithm usually consists of two processes: i) coding and indexing, and ii) subgraph querying. In this section, we briefly introduce some criteria metrics used to evaluate the efficiency of these two parts.

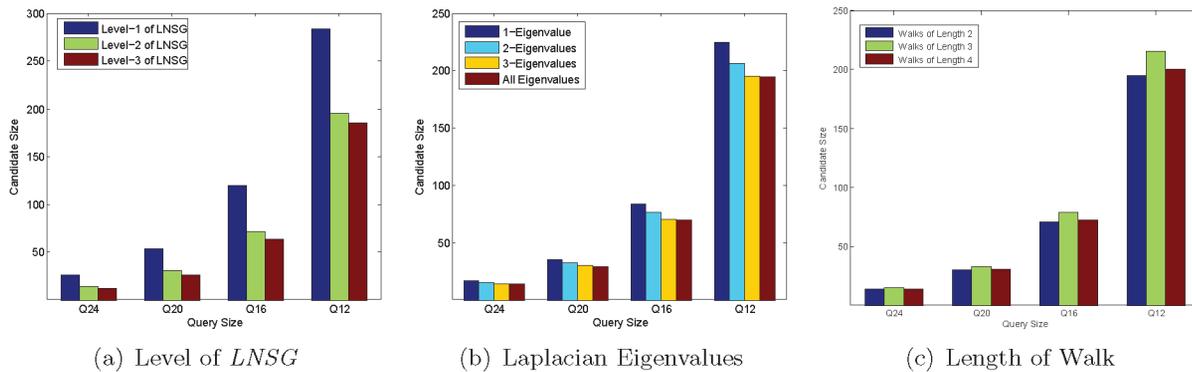


Figure 11. Impacts of Parameters on Candidate Set Size. (a) Level of *LNSG*; (b) Laplacian eigenvalues; (c) Length of Walk. doi:10.1371/journal.pone.0097178.g011

Criteria for coding and indexing. The coding and indexing time and the index size for both graph codes and the index tree are used in this process.

- Coding and Index Time.** The coding and indexing time is the run time used to encode both graphs and their vertices and build the index tree. A less coding and indexing time means higher performance in this process.
- Index Size.** The index size is the size of space used to store both the graph codes and the index tree. In the filtering phase, much time is spent on accessing a larger index, so it partly impacts the filtering efficiency.

Criteria for subgraph querying. The candidate set size, the filtering time, the verification time and the response time are used in this process.

- Candidate Set Size.** The candidate set size is the number of candidate graphs for each query graph. For each subgraph query algorithm, a smaller candidate set size implies higher filtering efficiency.
- Filtering Time.** For each subgraph query method, the Filtering time is the run time to traverse the index to filter out false positives and generate the candidate set. A less filtering time implies higher filtering efficiency.
- Verification Time.** For each subgraph query method, the verification time is the run time to verify each candidate and generate the result set. A less verification time implies higher verification efficiency.
- Response Time.** For each subgraph query method, the response time is defined as the sum of the filtering time and the verification time. A less response time means the higher of querying efficiency.

Our experiments evaluate the efficiency of different subgraph query methods. For each subgraph query method, the run time is the most important criterion in each phase. Thus, in the first phase, the coding and index time is the primary criterion; and in the second phase, the response time is the primary criterion.

Performance on Real Graph Database

Performance of coding and indexing. Fig. 12 shows the performance of the seven methods on the real graphs in the coding and indexing process.

Coding and Indexing Time. Fig. 12(a) shows the coding and indexing time of all the seven methods on the real graph database. From it we observe that, with the increasing of database size from

2 K to 10 K, the coding and indexing time of each methods is increasing.

Compared with Closure-tree, since LnGCoding must compute the expensive Laplacian spectrum, thus the coding and indexing time in LnGCoding is more than that of Closure-tree.

In the coding based index methods, LnGCoding computes not only the Laplacian spectrum but also the number of walks. Thus, the coding and indexing time in LnGCoding is the larger than that of GCoding.

For the substructure based index methods, they extract graph features via expensive frequent subgraph or subtree mining. Thus, their coding and indexing time is greater than that of LnGCoding.

In a word, the coding and indexing time of our method is much less than that of the substructure based index methods, and is comparable with those of GCoding and Closure-Tree.

Index Size. Fig. 12(b) shows the index sizes of the seven methods on the real graph database. From it we know that, when the database size is increasing from 2 K to 10 K, the index size of each method is also increasing.

The index size of Closure-tree is more than that of LnGCoding, since the coding based index methods both map the information of graph features into the numerical spaces, which can save the store space.

The index size of LnGCoding is more than that of GCoding, since the code in LnGCoding consists three parts: the labels of vertices and adjacent edges, the Laplacian spectrum, and the number of walks; while the code in GCoding contains two parts: the labels of vertices and adjacent edges, and the graph spectrum.

Since FG-Index generates all frequent subgraphs and all infrequent edges for completeness, its index size is greater than that of LnGCoding. For the other substructure based index methods, their index sizes are less than that of LnGCoding, because the sizes of mined features or the numbers of mined features are small [19].

Performance of querying. Fig. 13 shows the performance of the seven methods on the real graphs in querying process.

Candidate Set Size. Fig. 13(a) shows that, when query graph set is varying from Q_{24} to Q_4 , the candidate set size of each method is increasing. This is because the answer set is increasing. When query size is larger, such as Q_{24} and Q_{20} , the candidate set sizes of the clustering based and coding based index methods are less than those of the substructure based index methods; while when the query size is smaller, such as Q_8 and Q_4 , the candidate set sizes of the clustering based and coding based index methods are greater than those of the most substructure based index methods. The reason is that for these substructure based index methods, more

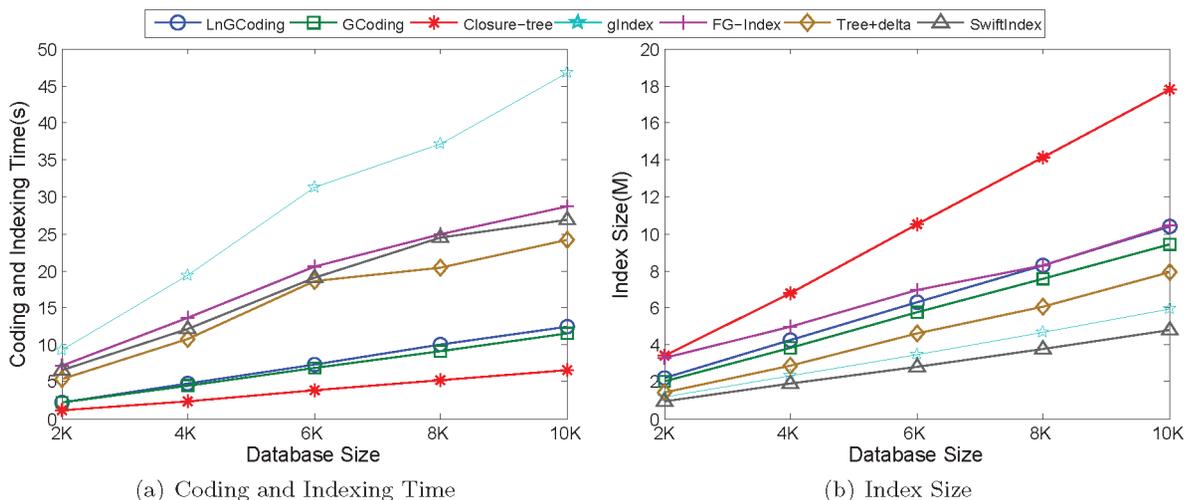


Figure 12. Performance of Coding and Indexing on Real Data. (a) Coding and Indexing Time; (b) Index Size. doi:10.1371/journal.pone.0097178.g012

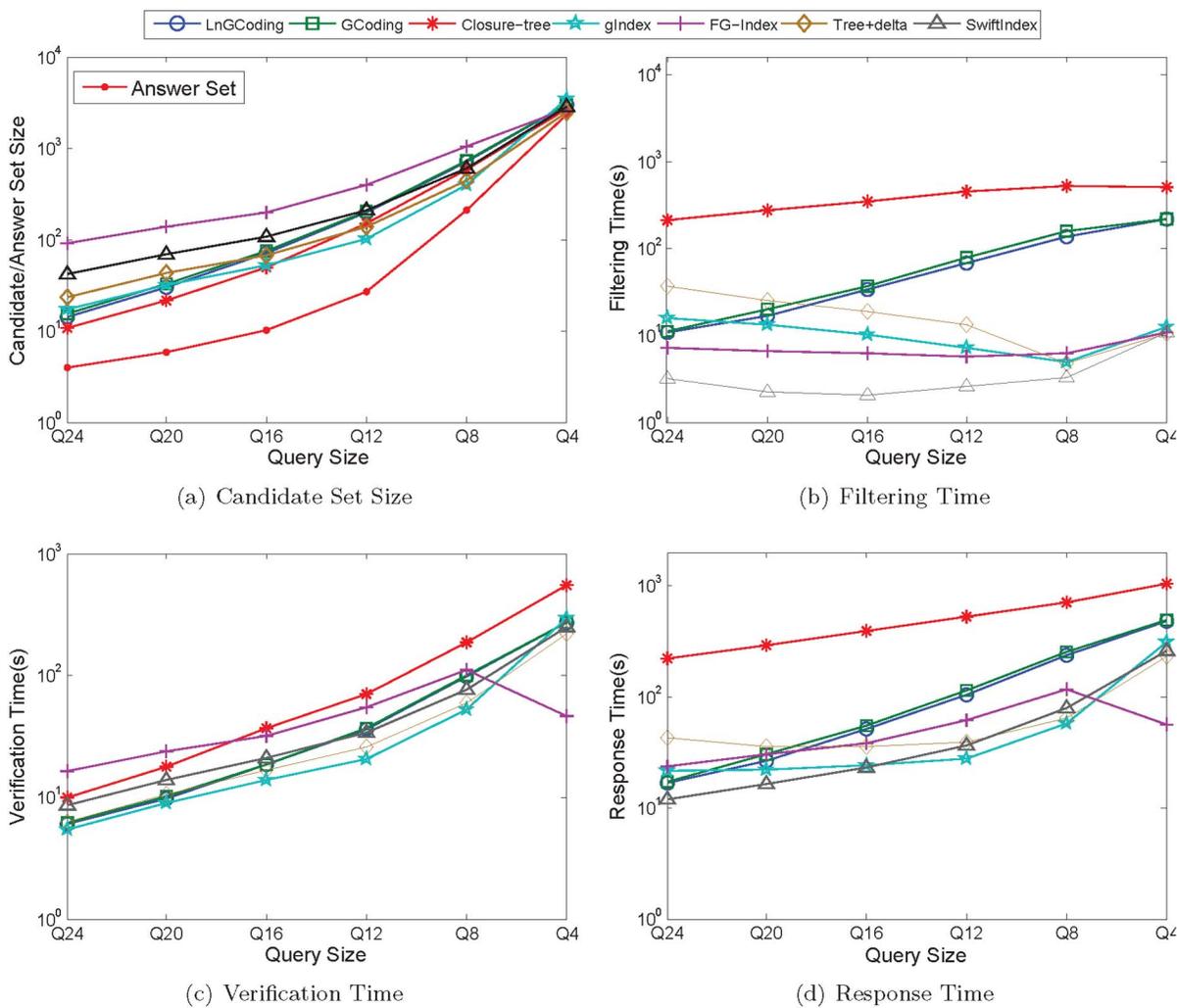


Figure 13. Performance of Querying on Real Data. (a) Candidate Set Size; (b) Filtering Time; (c) Verification Time; (d) Response Time. doi:10.1371/journal.pone.0097178.g013

features are mined on the smaller sized graphs than on the larger sized graphs.

Closure-tree prunes more false positives than that of LnGCoding, since it conducts the pseudo subgraph isomorphism testing, which is similar to the exact subgraph isomorphism algorithm.

Different from the graph spectrum in GCoding, LnGCoding uses Laplacian spectrum and the number of walks as graph features, thus the candidate set size of LnGCoding is less than that of GCoding.

For the substructure based index methods, since their mined index features are less for larger sized query graphs than for smaller sized query graphs, their candidate set sizes are greater than those of LnGCoding when the query graph sets are Q_{24} and Q_{20} . When the size of the query graph is smaller, such as Q_{12} , Q_8 and Q_4 , the candidate set sizes of gIndex, Tree+delta and SwiftIndex are less than those of LnGCoding. FG-Index generates the largest candidate set size, this is because it traverses the index to find a subset of mined features which is a subgraph of the query graph. This means it does not find out all subgraphs of a query graph from its index.

Filtering Time. Fig. 13(b) shows that, when the query graph set is varying from Q_{24} to Q_4 , the filtering time of the clustering based and coding based index methods is increasing, while the filtering time of the substructure based index methods is decreasing. The reason is that for the substructure based index methods, there are less index features in query graph set Q_4 than in Q_{24} , thus there are less comparisons between the query graph and the index features in Q_4 than in Q_{24} .

From Fig. 13(b) we also know that the filtering time of Closure-tree is the largest, as it conducts the pseudo subgraph isomorphism testing that is quite time consuming.

The vertex and graph codes of LnGCoding are more complex than those of GCoding, and the code comparison of the former is more expensive than that of the latter. Thus, the filtering time of LnGCoding is slightly greater than that of GCoding.

For the substructure based index methods, since their index sizes are less than that of LnGCoding, they traverse the index to filter out false positives with less time. Thus, the filtering time of most of them is less than that of LnGCoding.

Verification Time. Fig. 13(c) shows that, when the query graph set is varying from Q_{24} to Q_4 , the verification time of most methods are increasing.

Under the iGraph framework, Closure-tree employs a java bytecode analyzer to verify candidates, while LnGCoding uses the state-of-art subgraph isomorphism algorithm VF2 [25] to verify candidates. Although Closure-tree has the smaller candidate set size than that of LnGCoding, the verification time of Closure-tree is more than that of LnGCoding.

For the graph coding based index methods, the candidate set size of LnGCoding is slightly less than that of GCoding, so the verification time of the former is also slightly less than that of the latter.

For the substructure based index method FG-Index, its verification time is less than that of LnGCoding for query graph set Q_4 , and is more than those of LnGCoding for other query graph sets. The reason is that FG-Index employs a verification free strategy: when the query graph is an indexed feature, it directly reports the answer set without verification. Since Q_4 has most indexed features for all query graph sets, the verification time of FG-Index is less than those of the other methods.

The verification time of gIndex is slightly less than those of LnGCoding for query graph sets Q_{24} and Q_{20} . The reason lies in that, the candidate set sizes of gIndex are slightly more than those of LnGCoding, and the index size of gIndex is much less than that

of LnGCoding, so its cost for finding the candidate graphs is less than that of LnGCoding. For other query graph sets, the verification time of gIndex is less than those of LnGCoding, since the candidate set sizes of gIndex are much less than those of LnGCoding on these query graph sets.

The verification time of LnGCoding is less than those of Tree+delta for query graph sets Q_{24} and Q_{20} , and is greater than those of Tree+delta for query graph sets Q_{16} , Q_{12} , Q_8 and Q_4 . It is because the candidate set sizes of the former are much less than those of the latter for query graph sets Q_{24} and Q_{20} , and the candidate set sizes of the former are greater than those of the latter for query graph sets Q_{16} , Q_{12} , Q_8 and Q_4 .

Due to the sizes of candidate set, the verification time of LnGCoding is less than those of SwiftIndex for query graph sets Q_{24} , Q_{20} and Q_{16} , and is greater than those of SwiftIndex for query graph sets Q_8 and Q_4 . For query graph set Q_{12} , the verification time of LnGCoding is slightly more than that of SwiftIndex, since the candidate set size of SwiftIndex is slightly more than that of LnGCoding for query graph set Q_{12} , and the index size of SwiftIndex is much less than that of LnGCoding.

Response Time. Fig. 13(d) shows that, when the query graph set is varying from Q_{24} to Q_4 , the response times of most methods are increasing.

The filtering time and the verification time of Closure-tree both are the largest, so its response time is the biggest.

Since the filtering time of LnGCoding is much less than that of GCoding, and its verification time is smaller than or comparable to that of the latter, the response time of LnGCoding is less than that of GCoding. This means that LnGCoding performs best on the real graph database among the clustering based and coding based index methods.

For the substructure based index method SwiftIndex, its filtering time is much less than those of LnGCoding on all query graph sets, so the response time is less than those of the latter as well.

For the query graph set Q_{24} , the filtering time of LnGCoding is much less than that of gIndex, thus its response time is less than that of the latter. For other query graph sets, the filtering time and verification time of LnGCoding both are greater than those of gIndex, so its response time is greater than those of the latter.

For the query graph sets Q_{24} and Q_{20} , the filtering time of LnGCoding is much less than those of Tree+delta, and the verification time of LnGCoding is much less than those of FG-Index, thus its response time is less than those of Tree+delta and FG-Index. For other query graph sets, the filtering time of LnGCoding are greater than or much greater than those of Tree+delta and FG-Index, thus its response time is greater than those of Tree+delta and FG-Index.

According to the experimental results on real data, our method works well with larger query size. For the small query size, our method is faster than GCoding and Closure-Tree, but slower than the substructure based index methods.

In a word, for the real data experiment, the response time of LnGCoding is not as good as substructure-based methods like SwiftIndex, but LnGCoding outperforms these substructure-based methods regarding coding and indexing.

Performance on Synthetic Graphs

Performance of coding and indexing. Fig. 14 shows the performance of the seven methods on the synthetic graphs in the coding and indexing process.

Coding and Indexing Time. Fig. 14(a) shows the coding and indexing time of the seven methods on the synthetic graph database. From it we know that, with the increase of the database

size, the coding and indexing time of each method is also increasing.

Since LnGCoding must compute the expensive graph spectrum, thus the coding and indexing time of LnGCoding is greater than that of Closure-tree.

When computing graph spectrum, GCoding generates Level-N Path Tree (*LNPT*) and LnGCoding generates *LNSG*. However, *LNPT* is built by adding reduplicate vertices, and *LNSG* is generated without any reduplicate vertices. Fig. 15 shows the differences between *LNSG* and *LNPT* of vertex v_0 in graph D_2 , which occurred in Fig. 1.

From Fig. 15 we observe that $LNSG(D_2, 2, v_0)$ contains 4 vertices, but $LNPT(D_2, v_0, 2)$ contains 8 vertices. Obviously, $LNPT(D_2, v_0, 2)$ contains four reduplicated red vertices: one vertex v_1 , one vertex v_2 and two vertices v_3 . Since the computational complexity of graph spectrum is $O(N^3)$ (N is the number of vertices), GCoding is much more time consuming than LnGCoding, specially when the graph is dense. In the synthetic graph database, most graphs are dense. Thus, the coding and indexing time of LnGCoding is less than that of GCoding. Meanwhile, we can see that *LNPT* does not contain the cycles occurred in the graph, which degrades the filtering efficiency.

For the substructure based index methods, the coding and indexing time of gIndex is the largest due to it mines much more features, and the coding and indexing time of Tree+delta and SwiftIndex is smaller than that of LnGCoding because the mined features are less.

In a word, the coding and index time of our method is much less than that of gIndex and GCoding, and is comparable with the fastest method Tree+delta.

Index Size. Fig. 14(b) shows the index size of the seven methods on the synthetic graph database. From it we know that, with the increase of database size, the index size of each method is also increasing.

Since most of synthetic graphs are dense, LnGCoding must use more space to store the Laplacian spectrum. Thus, the index size of LnGCoding is greater than that of Closure-tree.

For the coding based index methods, GCoding generates *LNPT* by adding some reduplicate vertices while LnGCoding generates *LNSG* without any reduplicate vertices, thus the index size of LnGCoding is smaller than that of GCoding.

For the substructure based index methods, the mined features of gIndex are much more than those of others, so its index size is greater as well. Moreover, the mined index features of these substructure based index methods are smaller subgraph or substructures, thus the index size of LnGCoding is bigger than those of these methods.

Performance of querying. Fig. 16 shows the performance of the seven methods on the synthetic graphs in querying process.

Candidate Set Size. Fig. 16(a) shows the candidate set sizes of the seven methods on the synthetic graph database. We observe that, when the query graph size is varying from Q_{24} to Q_4 , the candidate set size of each method is increasing, this is because the answer set size of each method is increasing.

Closure-tree conducts the pseudo subgraph isomorphism testing in the filtering phase, thus its candidate set size is less than that of LnGCoding.

For the coding based index methods, GCoding and LnGCoding roughly have the same number of candidates.

For the substructure based index methods, the candidate set sizes of Tree+delta are less than those of LnGCoding on query graph sets Q_{24} , Q_{20} and Q_{16} , since it takes too much time to filter out false positives on these query graphs. For other query graph sets, the candidate set sizes of LnGCoding are smaller than those of Tree+delta. For the other substructure based index methods, as their index features are not effective for dense graphs, their candidate set sizes are greater than those of LnGCoding.

Filtering Time. Fig. 16(b) shows the filtering time of the seven methods on the synthetic graph database.

Since Closure-tree conducts the pseudo subgraph isomorphism testing to filter out false positives, thus its filtering time is much greater than that of LnGCoding.

For the coding based index methods, GCoding filters out more false positives than that of LnGCoding, thus its filtering time is greater than that of LnGCoding.

For the substructure based index methods, gIndex has the most mined features, and the sizes of most index features are small. For the query graph sets Q_{24} , Q_{20} and Q_{16} , gIndex uses ineffective features to minimize the number of candidates, thus its filtering time is greater than those of LnGCoding on these query graph sets. For other query graph sets, its filtering time is less than those of LnGCoding.

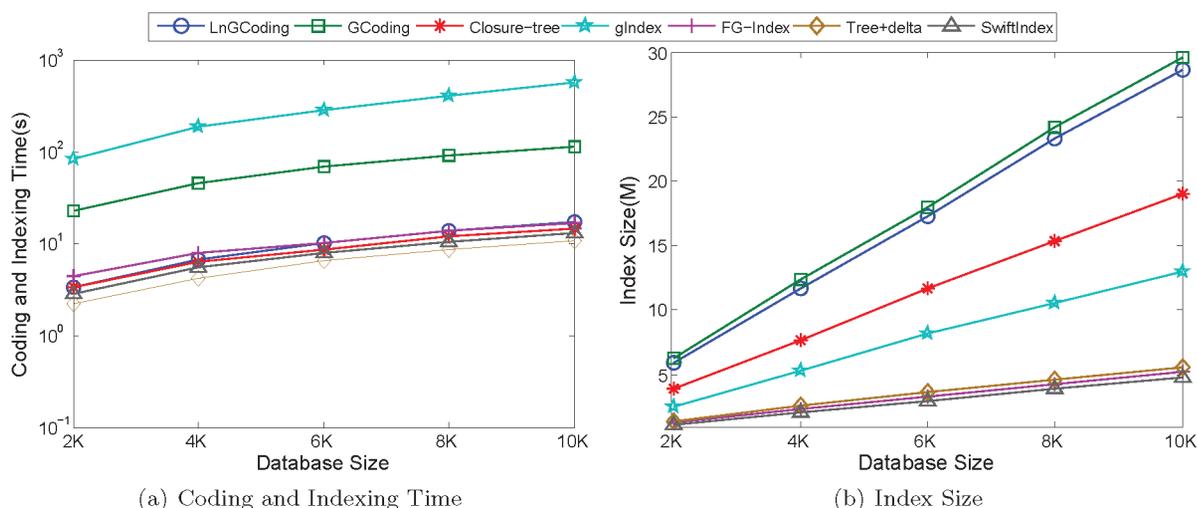


Figure 14. Performance of Coding and Indexing on Synthetic Data. (a) Coding and Indexing Time; (b) Index Size. doi:10.1371/journal.pone.0097178.g014

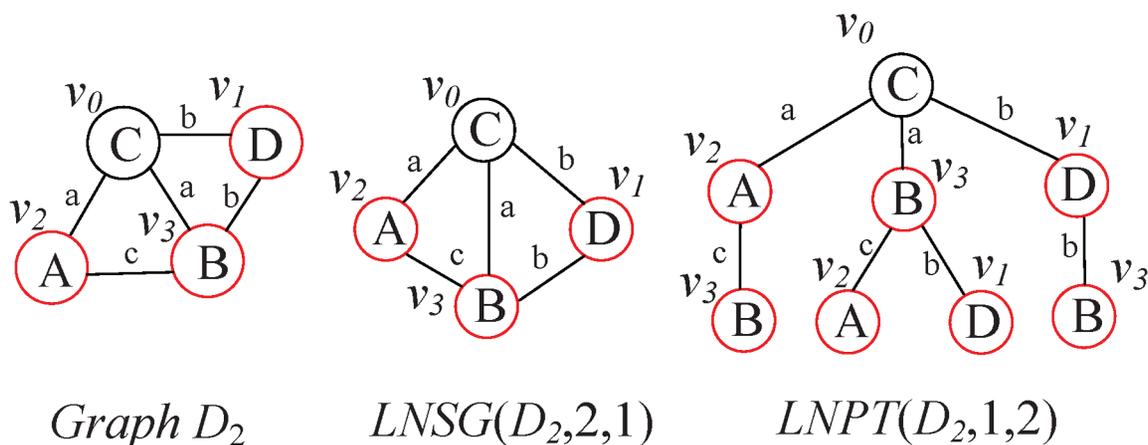


Figure 15. LNSG and LNPT of $v_0 \in D_2$.
doi:10.1371/journal.pone.0097178.g015

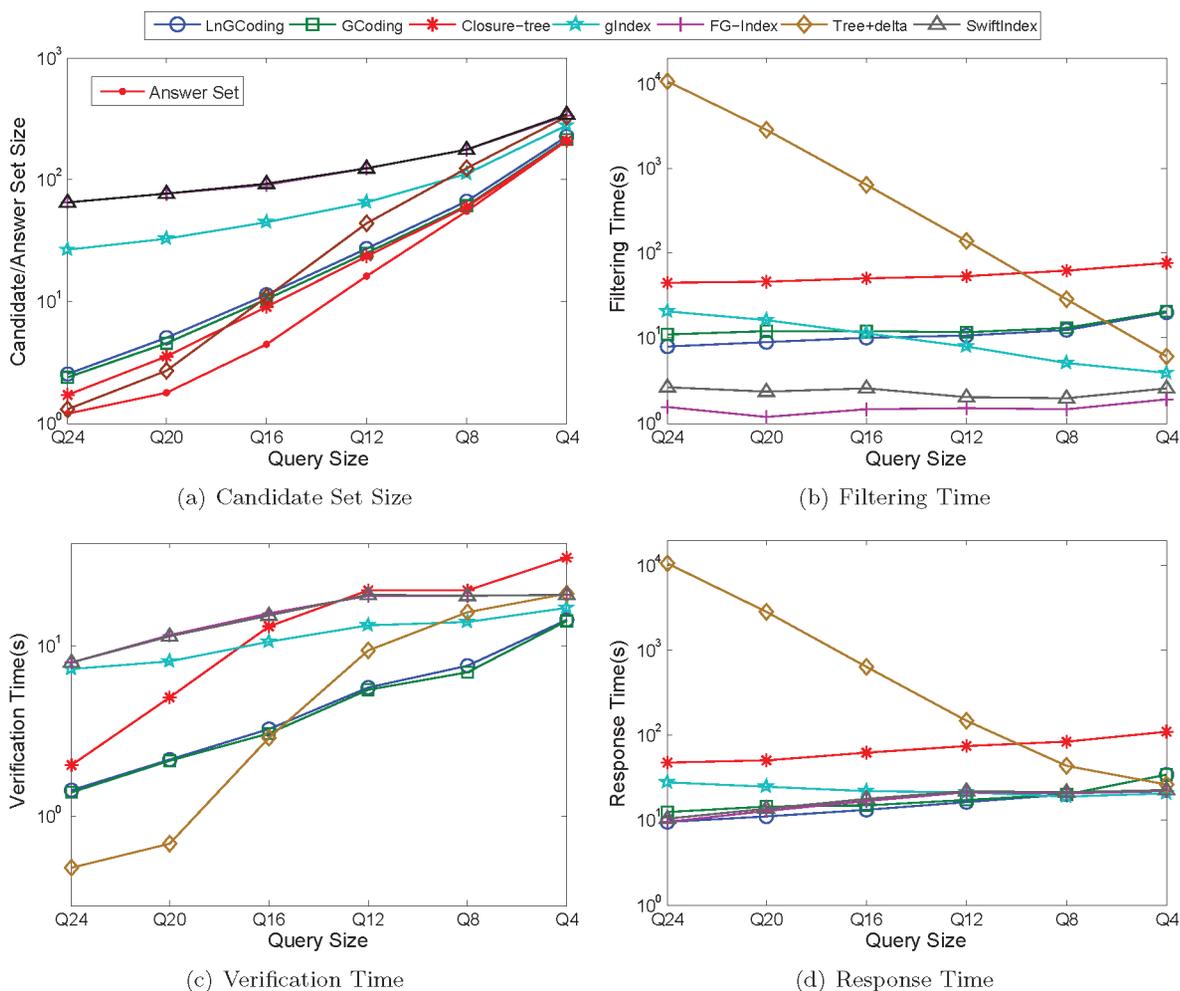


Figure 16. Performance of Query Processing on Synthetic Data. (a) Candidate Set Size; (b) Filtering Time; (c) Verification Time; (d) Response Time.
doi:10.1371/journal.pone.0097178.g016

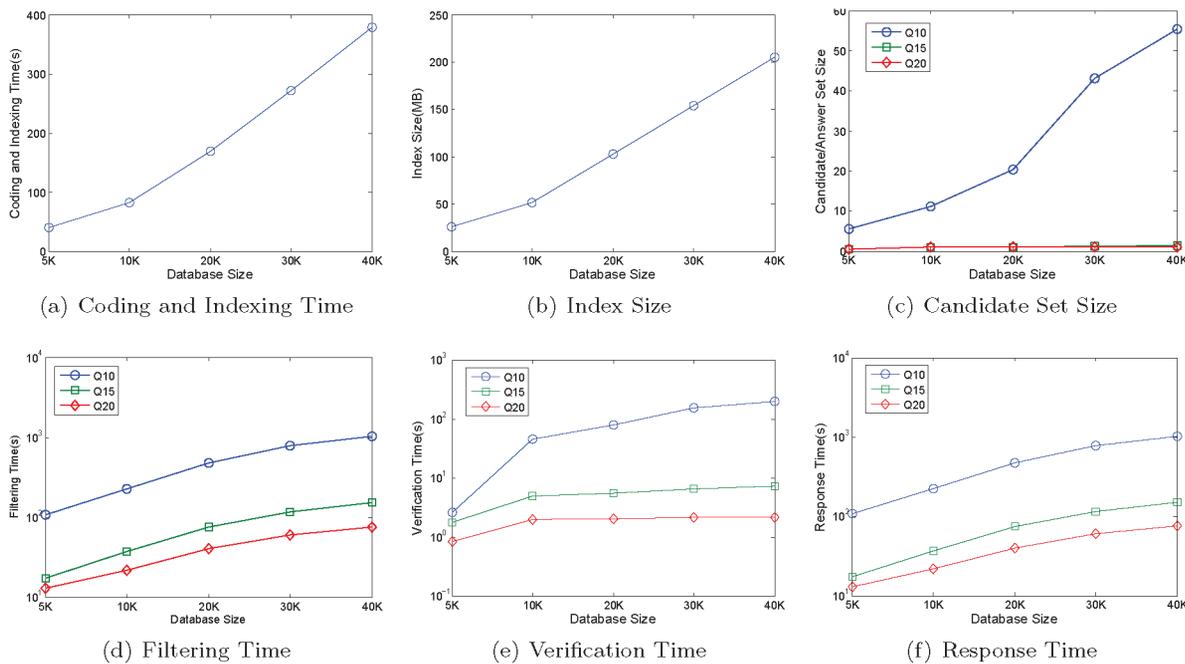


Figure 17. Performance on Graphs with Varying Sizes. (a) Coding and Indexing Time; (b) Index Size; (c) Candidate Set Size; (d) Filtering Time; (e) Verification Time; (f) Response Time. doi:10.1371/journal.pone.0097178.g017

The filtering time of Tree+delta is also greater than that of LnGCoding except for Q_4 . This is because that the query graphs contain many cycles in dense graph database, and Tree+delta mines too many graph features to its “delta”, which is very time consuming.

The mined features of FG-Index and SwiftIndex are not effective for dense graph database, they filter out much less false positives than LnGCoding. Thus, their filtering time are less than that of LnGCoding for all query graph sets.

Verification Time. Fig. 16(c) shows the verification time of the seven methods on the synthetic graph database. From it we know

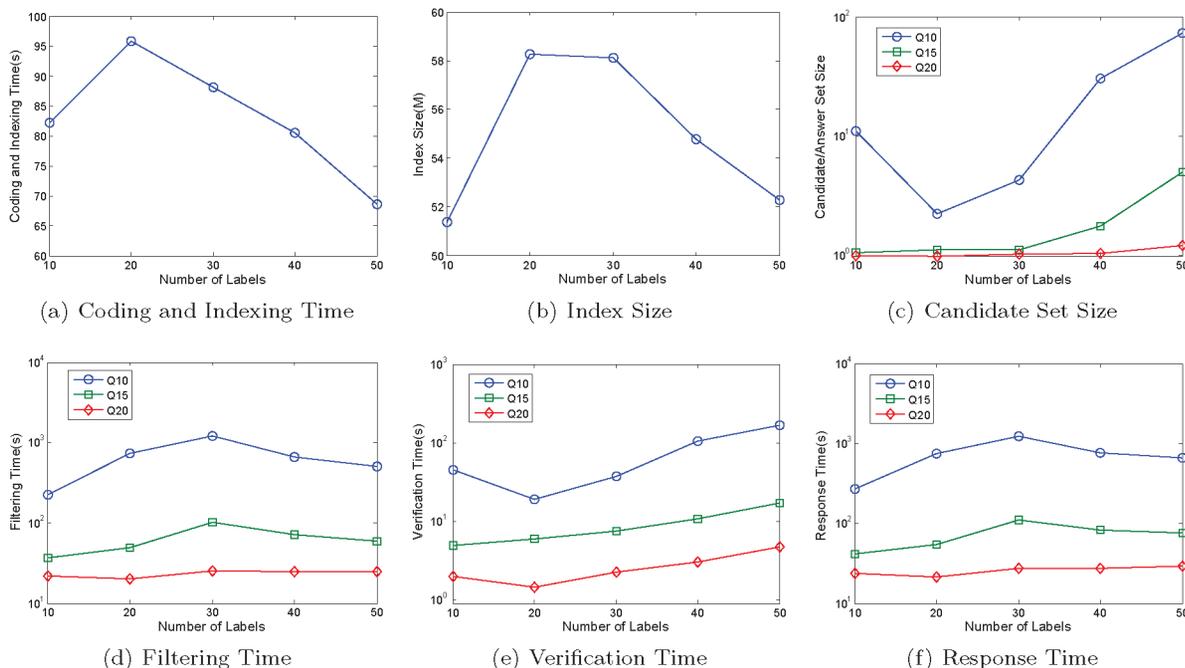


Figure 18. Performance on Graphs with Varying Vertex Labels. (a) Coding and Indexing Time; (b) Index Size; (c) Candidate Set Size; (d) Filtering Time; (e) Verification Time; (f) Response Time. doi:10.1371/journal.pone.0097178.g018

that, with the decrease of the query graph size, the verification time of each method is also increasing. This is because the candidate set size of each method is increasing.

Since Closure-tree follows iGraph's original implementation exactly using a java bytecode analyzer, thus its verification time is greater than that of LnGCoding.

For the coding based index methods, the candidate set size of GCoding is slightly less than that of LnGCoding, so its verification time is slightly smaller than that of LnGCoding.

For the substructure based index method Tree+delta, its candidate set sizes are less than those of LnGCoding for query graph sets Q_{24} , Q_{20} and Q_{16} , so its verification time is smaller than those of LnGCoding on these query graph sets. As for the other query graph sets, since the candidate set sizes of Tree+delta are greater than those of LnGCoding, its verification time is also greater than those of LnGCoding.

For the other substructure based index methods, their candidate set sizes are much more than those of LnGCoding, thus their verification time is also greater than those of LnGCoding. Note that, the verification time of FG-Index is not the least for query graph set Q_4 , since there are not many frequent features on query graph set Q_4 .

Response Time. Fig. 16(d) shows the response time of the seven methods on the synthetic graph database.

Since Closure-tree has the more filtering time and verification time than those of LnGCoding, thus its response time is bigger than that of LnGCoding.

For the coding based index methods, the filtering time of LnGCoding is much less than that of GCoding, thus its response time is less than that of GCoding.

The substructure based index method Tree+delta takes much more time to filter out false positives, thus its response time is greater than that of LnGCoding except for Q_4 .

For the other substructure based index methods, their filtering time is much less than that of LnGCoding for Q_4 , thus their response time is less than that of LnGCoding on query graph set Q_4 . As for the other query graph sets, these methods' verification time is much greater than those of LnGCoding, thus their response time is greater than those of LnGCoding. Thus, the response time of LnGCoding is the least among all methods except for query graph set Q_4 , and our method performs best on dense graph database.

In a word, for the synthetic data with dense graphs, LnGCoding has the best response time and similar coding and indexing time as the fastest methods; FG-Index and SwiftIndex are close competitors to LnGCoding regarding both evaluation measures.

From the experiments over both real and synthetic graph data, we can find that, although none of these methods outperforms others on all the databases, our proposed method does outperform competitors when graphs are dense.

Scalability Test

In order to evaluate the scalability of LnGCoding, we conduct experiments on the synthetic graph data with different sizes and distinct vertex labels.

The synthetic graph data consists of the ten graph databases that are generated with a graph generator, which is developed by

Kuramochi and Karypis [29] and also used in [18] and [17], by varying the cardinality and the vertex labels. Three subsets are selected as the query graph sets to test the scalability of our method.

Performance on graphs with varying sizes. In this experiment, we generated five databases D_{5K} , D_{10K} , D_{20K} , D_{30K} and D_{40K} by varying the database cardinality. For database D_nK ($n = 5, 10, 20, 30, 40$), nK (i.e. $n \times 1000$) graphs are included. The query graph sets are Q_{10} , Q_{15} and Q_{20} , where each query graph set Q_i consists of 1,000 query graphs with i edges.

Fig. 17 shows the performance of our method on graphs with varying sizes. From it we observe that, with the increase of database size, the coding and indexing time and index size are almost linearly increasing. However, increasing rates of the candidate set size, the filtering time, the verification time, and the response time are much smaller except for the query graph set Q_{10} , since its candidate set size grows much faster than those of Q_{15} and Q_{20} . This indicates our method performs well on databases with different sizes.

Performance on graphs with varying vertex labels. In this experiment, we also generated five databases D_{10L} , D_{20L} , D_{30L} , D_{40L} , D_{50L} by varying the vertex label. For database D_nL ($n = 10, 20, 30, 40, 50$), the number of vertex labels is n . The query graph sets are Q_{10} , Q_{15} and Q_{20} , where each query graph set Q_i consists of 1,000 query graphs with i edges.

Fig. 18 shows performance of our method on graphs with varying vertex labels. From it we know that, with the increase of the number of labels, 1) the coding and indexing time and the index size are decreasing except for the graphs with 10 labels, 2) the trends of the candidate set size, the filtering time, the verification time, and the response time are increasing but the growth rates are small or very small. This means our method works well on the graphs with varying vertex labels.

Conclusions

In this paper, we propose a novel graph coding method LnGCoding, which utilizes the combination of Laplacian spectrum and the number of walks for subgraph querying over labeled graphs.

Our method first extracts some new graph features, and then maps these features into the numerical space to generate the vertex and graph codes. A novel index is built to improve the filtering efficiency. We also present novel two-step filtering conditions taking the properties of graph features into account, and the correctness is proved.

In order to evaluate the performance, extensive experiments on both real and synthetic data have been conducted. Experimental results show that, compared with the other six methods, our method works very well, especially when graphs are dense.

In the future, we plan using our graph coding method to explore similarity graph querying and supergraph querying.

Author Contributions

Conceived and designed the experiments: LZ QS. Performed the experiments: LZ YG. Analyzed the data: LD XZ GW. Contributed reagents/materials/analysis tools: YG LD. Wrote the paper: LZ QS GW.

References

- Cai D, Shao Z, He X, Yan X, Han J (2005) Community mining from multi-relational networks. In: Proceedings of the European Conference on Principles and Practice of Knowledge Discovery in Databases. 445–452.
- Cai D, Shao Z, He X, Yan X, Han J (2005) Mining hidden community in heterogeneous social networks. In: Proceedings of the 3rd international workshop on Link discovery. ACM, 58–65.
- Willett P, Barnard J, Downs G (1998) Chemical similarity searching. Journal of Chemical Information and Computer Sciences 38: 983–996.
- James C, Weininger D, Delany J (2003). Daylight theory manual daylight version 4.82.

5. Berretti S, Del Bimbo A, Vicario E (2001) Efficient matching and indexing of graph models in content-based retrieval. *IEEE Trans Pattern Anal Mach Intell* 23: 1089–1105.
6. Petrakis E, Faloutsos A (1997) Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Engineering* 9: 435–447.
7. Chung C, Min J, Shim K (2002) Apex: An adaptive path index for xml data. In: *Proceedings of the ACM SIGMOD international conference on Management of Data*. 121–132.
8. Zhang N, Özsu M, Ilyas I, Aboulmaga A (2006) Fix: Feature-based indexing technique for xml documents. In: *Proceedings of the International Conference on Very Large Data Bases*. 259–270.
9. Cook S (1971) The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 151–158.
10. Garey M, Johnson D (1979). *Computers and intractability: An introduction to the theory of np-completeness*.
11. Giugno R, Shasha D (2002) Graphrep: A fast and universal method for querying graphs. In: *Proceedings of the International Conference on Pattern Recognition*. volume 2, 112–115.
12. Yan X, Yu P, Han J (2004) Graph indexing: A frequent structure-based approach. In: *Proceedings of the ACM SIGMOD international conference on Management of Data*. 2335–2346.
13. Cheng J, Ke Y, NgW, Lu A (2007) Fg-index: towards verification-free query processing on graph databases. In: *Proceedings of the ACM SIGMOD international conference on Management of data*. 857–872.
14. Zhang S, Hu M, Yang J (2007) Treepi: A novel graph indexing method. In: *Proceedings of International Conference on Data Engineering*. 966–975.
15. Zhao P, Yu J, Yu P (2007) Graph indexing: tree+delta>=graph. In: *Proceedings of the International Conference on Very Large Data Bases*. 938–949.
16. Shang H, Zhang Y, Lin X, Yu J (2008) Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In: *Proceedings of the International Conference on Very Large Data Bases*. 364–375.
17. Zou L, Chen L, Yu J, Lu Y (2008) A novel spectral coding in a large graph database. In: *Proceedings of the International Conference on Extending database technology*. 181–192.
18. He H, Singh A (2006) Closure-tree: An index structure for graph queries. In: *Proceedings of the International Conference on Data Engineering*. 38–49.
19. Han W, Lee J, Pham M, Yu J (2010) iGraph: a framework for comparisons of disk-based graph indexing techniques. In: *Proceedings of the International Conference on Very Large Data Bases*. volume 3, 449–459.
20. Han W, Pham M, Lee J, Kasperovics R, Yu J (2011) iGraph in action: performance analysis of disk-based graph indexing techniques. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1241–1242.
21. Zhu L, Song Q (2011) A study of laplaican spectra of graph for subgraph queries. In: *Proceedings of the IEEE International Conference on Data Mining*. 1272–1277.
22. Horn R, Johnson C (2005) *Matrix Analysis*. Cambridge University Press.
23. Godsil C, Royle G (2001) *Algebraic Graph Theory*. New York: Springer.
24. Tousidou E, Bozani P, Manolopoulos Y (2002) Signature-based structures for objects with set-valued attributes. *Information Systems* 27: 93–121.
25. Cordella L, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26: 1367–1372.
26. Vfl2 library. <http://amalfi.dis.unina.it/graph/db/vflib-2.0/>.
27. A synthetic graph data generator. <http://www.cse.ust.hk/graphgen/>.
28. Kitagawa H, Ishikawa Y (1997) False drop analysis of set retrieval with signature files. *IEICE Transactions on Information and Systems*: 653–664.
29. Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: *Proceedings of the IEEE International Conference on Data Mining*. 313–320.