Finding Top-K Similar Graphs in Graph Databases

Yuanyuan Zhu, Lu Qin, Jeffrey Xu Yu, Hong Cheng

The Chinese University of Hong Kong, Hong Kong

{yyzhu,lqin,yu,hcheng}@se.cuhk.edu.hk

ABSTRACT

Querying similar graphs in graph databases has been widely studied in graph query processing in recent years. Existing works mainly focus on subgraph similarity search and supergraph similarity search. In this paper, we study the problem of finding top-k graphs in a graph database that are most similar to a query graph. This problem has many applications, such as image retrieval and chemical compound structure search. Regarding the similarity measure, feature based and kernel based similarity measures have been used in the literature. But such measures are rough and may lose the connectivity information among substructures. In this paper, we introduce a new similarity measure based on the maximum common subgraph (*MCS*) of two graphs. We show that this measure can better capture the common and different structures of two graphs. Since computing the *MCS* of two graphs is NP-hard, we propose an algorithm to answer the top-k graph similarity query using two distance lower bounds with different computational costs, in order to reduce the number of *MCS* computations. We further introduce an indexing technique, which can better make use of the triangle property of similarities among graphs in the database to get tighter lower bounds. Three different indexing methods are proposed with different tradeoffs between pruning power and construction cost. We conducted extensive performance studies on large real datasets to evaluate the performance of our approaches.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithm, Theory, Performance

Keywords

Top-K, Similarity Search, Graph Database

1. INTRODUCTION

Graph is a general tool for modeling structural relationships between data objects. It has been prevalently used in a wide range of application domains, such as chemical compound structures in chemistry, attributed graphs in image processing, food chains in ecology, electrical circuits in electricity, road networks in transport, protein interaction networks in biology, topological networks on the Web, etc. With the increasing popularity of graph databases in various applications, graph query processing has attracted much attention in recent years.

Existing researches on graph query processing mainly fall into two categories: subgraph containment search and supergraph containment search. The former aims to identify a set of graphs that contain a query graph, and the latter aims to find a set of graphs that are contained by a query graph. Many indexing and query processing techniques have been proposed to solve these two problems [24, 7, 27, 16, 25, 18, 9, 28, 22, 8, 5, 26, 6].

Besides exact subgraph/supergraph containment query, some studies allow a small number of edges or nodes missing in the query result. Two kinds of graph similarity queries have been investigated recently, namely, subgraph similarity search [23, 15] and supergraph similarity search [17]. These algorithms take a threshold based approach, i.e., returning graphs whose distances to the query graph q are within a given threshold δ. The similarity (or distance) measure in [15, 17] is defined based on the maximum common subgraph (*MCS*) between a query graph q and a graph g in the graph database, denoted as $\mathsf{mcs}(q,g)$. For subgraph similarity search [15], the similarity of q and g is measured by the distance between the query graph q and $\mathsf{mcs}(q,g)$, i.e., $\mathsf{dist}(q,g) = |E(q)| - |E(\mathsf{mcs}(q,g))|$; while for supergraph similarity search [17], the similarity is measured by the distance between the database graph g and $\mathsf{mcs}(q,g)$, i.e., $\mathsf{dist}(q,g) = |E(g)| - |E(\mathsf{mcs}(q,g))|$.

In many applications, users may want the answer graphs to be similar to the query graph, considering the whole structure matching rather than substructure matching. For example, finding top-k images that are most similar to a query image, or finding top-k chemical compound structures that are most similar to a query chemical compound structure. In such situations, neither subgraph similarity query result nor supergraph similarity query result can be considered as the appropriate answer for users. The reasons are as follows. For subgraph similarity query, since it only considers the distance between the query graph and the *MCS*, the algorithms may return a database graph whose size is far larger than that of the query as long as it contains a subgraph similar to the query. For supergraph similarity query, since it only considers the distance between the database graph and the *MCS*, the algorithms may return a database graph whose size is far smaller than that of the query as long as it is similar to a subgraph of the query. We illustrate such issues using the following example.

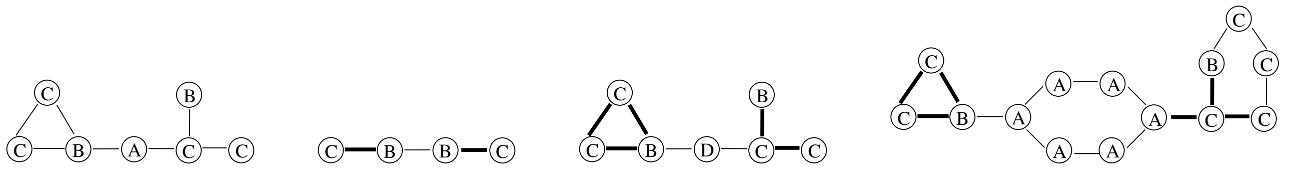

(a) Query Graph q (b) g_1 (c) g_2 (d) g_3

Figure 1: A Motivation Example

Example 1.1: Fig. 1 shows a query graph q and a sample graph database $D = \{g_1, g_2, g_3\}$ (The *MCS* of each database graph and q is shown in bold edges). Suppose a user wants to find the top-1 similar graph in D for q. If we use $\mathsf{dist}(q, g) = |E(q)| - |E(\mathsf{mcs}(q, g))|$ in subgraph similarity query [15] as the distance measure, g_3 will be returned as the answer, since $\mathsf{dist}(q, g_1) = 7 - 2 = 5$, $\mathsf{dist}(q, g_2) = 7 - 5 = 2$, and $\mathsf{dist}(q, g_3) = 7 - 6 = 1$. If we use $\mathsf{dist}(q, g) = |E(g)| - |E(\mathsf{mcs}(q, g))|$ in supergraph similarity query [17] as the distance measure, g_1 will be returned as the answer, since $\mathsf{dist}(q, g_1) = 3 - 2 = 1$, $\mathsf{dist}(q, g_2) = 7 - 5 = 2$, and $\mathsf{dist}(q, g_3) = 16 - 6 = 10$. However, considering the whole structure of q and each graph in the database, neither g_1 nor g_3 will be a good answer from the user's perspective, because g_1 is too small and does not share too much common part with q, while g_3 is too large and has a large part that cannot match q. Obviously, the most similar graph to q should be g_2, because they share the most common substructures with the least difference. □

As the subgraph/supergraph similarity queries have different goals from the graph similarity query, their graph distance measures cannot be applied to solve the latter problem. In addition, the existing techniques to answer subgraph/supergraph similarity queries, e.g., substructure/feature indexing techniques [23, 15, 17], can hardly be used to answer top-k graph similarity query due to the following reason. The existing solutions take a threshold based approach which returns graphs whose distances to the query graph q are within a given threshold δ. So such methods build the index based on different δ values and answer each query with a specific δ from the index. However, as the distribution of different structures in a graph database is not uniform, the number of answer graphs retrieved for the same value of δ can vary significantly for different queries. For example, in a real chemical compound structure database from the National Cancer Institute, for the same value of δ, the answer set size for different queries varies from zero to thousands. Given such a great difference in the answer set size, it is hard for a user to specify a suitable value of δ for each specific query graph. As a result, a user may need to try many times in order to obtain a set of answers that is truly useful. This is why we study to answer top-k graph similarity query instead of using the threshold approach. As far as we know, there are no efficient and effective approaches that can handle such a problem in the literature.

Regarding the similarity measure between a query graph and a database graph, some approximate measures, such as feature-based measure [21] [23] and kernel function based measure [11, 10, 20] have been proposed. The former aims to extract domain-specific elementary structures as features and measure the similarity of two graphs based on the number of common features they have. The latter defines the similarity of two graphs using a graph kernel function, e.g., based on the common random walks, common cyclic substructure patterns, common h-hop neighbors of nodes, etc. Both measures only provide a very rough estimation of structure similarity since they lose the global structural connectivity in the two graphs. An example of feature-based measure is showed in Example 1.2.

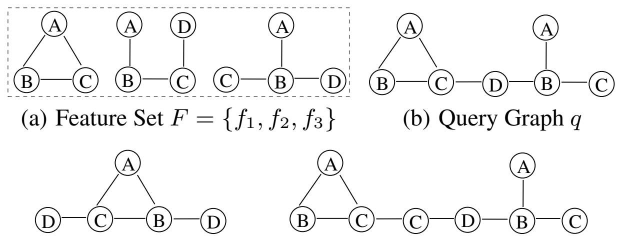

(a) Feature Set $F = \{f_1, f_2, f_3\}$ (b) Query Graph q (c) g_1 (d) g_2

Figure 2: Feature-Based Measure Example

Example 1.2: Fig. 2 shows a feature set $F = \{f_1, f_2, f_3\}$, a query graph q, and a sample graph database $D = \{g_1, g_2\}$. Suppose a user wants to find the top-1 similar graph in D for q. The feature vectors for q, g_1 and g_2 are [1 1 1], [1 1 1] and [1 0 1] respectively. Thus the similarity between q and g_1 is 1, and the similarity between q and g_2 is $2/3 = 0.67$. g_1 will be returned as the answer. However, considering the whole structure of q and each graph in the database, g_2 will be a better answer from the user's perspective. The main reason that the feature-based method fails to return a good answer is it only records whether each feature occurs but cannot capture the connectivity information among features. □

In order to measure the similarity of two graphs more accurately, we propose a new similarity measure based on *MCS* which takes structure connectivity fully into consideration.

To solve the top-k graph similarity query problem, a straightforward solution is to compute the similarity between the query graph q and every graph g in D, and return the top-k similar graphs to q. Since the similarity is based on the *MCS* of two graphs, for each graph g in the graph database, we need to compute the *MCS* of g and q in query processing, which is NP-hard. In this paper, we developed several pruning strategies to prune the graphs, which are not in the top-k answer set, in order to reduce the number of *MCS* computations. Based on simple structure statistics such as edge frequency, we derive two distance lower bounds for pruning. We further exploit the structural similarity between graphs in D and obtain a triangle property of similarities among the query graph and two database graphs. We show that the triangle property can be used to further prune graphs with the help of an index. We design three different indexing techniques: DPIndex, OPIndex, and GSIndex, with different tradeoffs between pruning power and construction cost. We propose efficient algorithms to compute the top-k similar graphs for a given graph query, using the lower bounds and triangle property we derive.

The main contributions of this paper are summarized below. First, we study the top-k graph similarity query based on a new *MCS* based similarity measure, which takes both the common part and different part between the query graph and the answer graph into consideration. Second, we derive two distance lower bounds to reduce the number of *MCS* computations in query processing. We further exploit the structural similarity between graphs in databases to derive a triangle property based bound, which can be used effectively to prune graphs that are not in the top-k answers. Three indexes (DPIndex, OPIndex, and GSIndex) are designed with different construction costs and pruning power to support the triangle inequality based pruning. Based on the lower bounds and indexes, we propose efficient algorithms to solve the problem of top-k graph similarity query in graph databases. Third, we conducted extensive performance studies on a real dataset to test the performance of our algorithms.

The rest of the paper is organized as follows. Section 2 gives the problem statement. Section 3 presents our framework to answer the top-k graph similarity query. Section 4 discusses the pruning method based on two lower bounds without using any index. Section 5 studies the top-k graph similarity query processing based on three different indexing techniques using the triangle property we derived. Our experimental results are shown in Sections 6. Section 7 discusses some related works and Section 8 concludes this paper.

2. PROBLEM STATEMENT

In this paper, we focus on undirected vertex-labeled graphs. Given a set of labels, Σ_V, a graph is denoted by $g = (V, E, l)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and l is a labeling function, $l : V \to \Sigma_V$. For each node $u \in V$, $l(u)$ denotes the label of u. We denote the vertex set and the edge set of graph g by $V(g)$ and $E(g)$, respectively. $|V(g)|$ and $|E(g)|$ represent the number of vertices and edges in graph g, respectively. For simplicity, an undirected vertex-labeled graph is hereafter abbreviated to a graph if the context is obvious.

Definition 2.1: Subgraph Isomorphism. Given two graphs $g_1 = (V_1, E_1, l_1)$ and $g_2 = (V_2, E_2, l_2)$, g_1 is subgraph isomorphic to g_2, if there is an injective function $f : V(g_1) \to V(g_2)$ such that (1) $\forall u \in V_1, v \in V_1$ and $u \neq v$, we have $f(u) \neq f(v)$. (2) $\forall v \in V_1$, we have $f(v) \in V_2$ and $l_1(v) = l_2(f(v))$. (3) $\forall (u, v) \in E(g_1)$, $(f(u), f(v)) \in E(g_2)$. □

Definition 2.2: Common Subgraph. Given two graphs g_1 and g_2, graph g is a common subgraph of g_1 and g_2, if g is subgraph isomorphic to g_1 and g_2, respectively. □

Definition 2.3: Maximum Common Subgraph. Graph g is a maximum common subgraph (*MCS*) of two graphs g_1 and g_2, denoted as $\mathsf{mcs}(g_1, g_2)$, if g is a common subgraph of g_1 and g_2, and there is no other common subgraph g' of g_1 and g_2, such that $|E(g')| > |E(g)|$. □

Definition 2.4: Graph Distance. Given two graphs q and g, the graph distance based on $\mathsf{mcs}(q, g)$ is defined as

$$\mathsf{dist}(q, g) = |E(q)| + |E(g)| - 2 \times |E(\mathsf{mcs}(q, g))| \quad (1)$$

□

In this paper, we allow the *MCS* of two graphs to be disconnected,

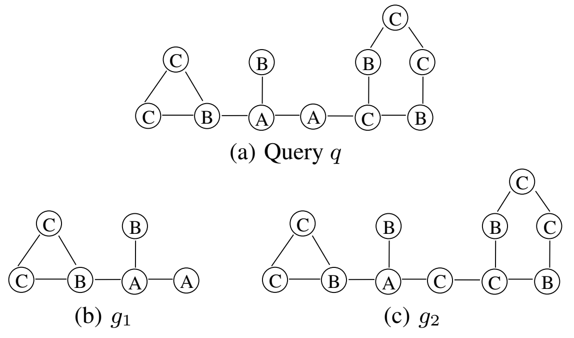


Figure 3: Example for MCS

since it can potentially capture more common substructures of two graphs and evaluate the structure similarity of two graphs more globally. We will show its advantage using the following example.

Example 2.1: Fig. 3 shows a query graph q and a sample graph database $D = \{g_1, g_2\}$. Suppose a user wants to find the top-1 similar graph in D for q. If we require *MCS* to be connected, g_1 will be returned as the answer, because $\mathsf{dist}(q, g_1) = 12 + 6 - 2 \times 6 = 6$ and $\mathsf{dist}(q, g_2) = 12 + 12 - 2 \times 5 = 14$. If we allow the *MCS* to be disconnected, g_2 will be returned as the answer, because $\mathsf{dist}(q, g_1) = 6$ and $\mathsf{dist}(q, g_2) = 12 + 12 - 2 \times 10 = 4$. Clearly, the latter is the desired result for the user, which shows the global description ability of disconnected *MCS*. □

For graph similarity query, two graphs cannot be considered similar if their sizes have a significant disparity, i.e., a database graph can not be considered as a good answer if it is far larger or smaller than the query graph. A good graph distance measure should have the ability to bound the size of the retrieved database graph to make it neither too large nor too small for the query graph. Our graph distance measure in Definition 2.4 has such a property, which is shown as follows.

Theorem 2.1: *Given a query graph q and a small value δ, for a database graph g such that* $\mathsf{dist}(q, g) = |E(q)| + |E(g)| - 2 \times |E(\mathsf{mcs}(q, g))| = \delta$, *the following equation holds:*

$$|E(q)| + \delta \geq |E(g)| \geq |E(q)| - \delta$$

□

Proof Sketch: Since $|E(\mathsf{mcs}(q, g))| \leq |E(g)|$, we have $\delta = |E(q)| + |E(g)| - 2 \times |E(\mathsf{mcs}(q, g))| \geq |E(q)| + |E(g)| - 2 \times |E(g)| = |E(q)| - |E(g)|$, which leads to $|E(g)| \geq |E(q)| - \delta$. Similarly, since $|E(\mathsf{mcs}(q, g))| \leq |E(q)|$, we have $\delta = |E(q)| + |E(g)| - 2 \times |E(\mathsf{mcs}(q, g))| \geq |E(q)| + |E(g)| - 2 \times |E(q)| = |E(g)| - |E(q)|$, which leads to $|E(g)| \leq |E(q)| + \delta$. Thus Theorem 2.1 holds. □

Note that such bounds cannot be obtained for distance measures in [15] and [17]. For distance measure $|E(q)| - |E(\mathsf{mcs}(q, g))|$ in [15], if $|E(q)| - |E(\mathsf{mcs}(q, g))| = \delta$, we only have a lower bound that $|E(g)| \geq |E(q)| - \delta$, which implies that $|E(g)|$ can be arbitrarily large. For distance measure $|E(g)| - |E(\mathsf{mcs}(q, g))|$ in [17], if $|E(g)| - |E(\mathsf{mcs}(q, g))| = \delta$, we only have an upper bound that $|E(g)| \leq |E(q)| + \delta$, which implies that $|E(g)|$ can be arbitrarily small. Our graph distance definition can measure the global structure matching of the query graph and the database graph more accurately, because we take both the distance between q and

Algorithm 1 gs-topk (D, q, k)

```
Input: graph database $D$, a query graph $q$, and an integer $k$;
Output: top-$k$ similar graphs with respect to $q$;
 1: $\mathcal{A} \leftarrow$ max-heap of size $k$ with each value set to be $+\infty$;
 2: $\mathcal{H} \leftarrow$ min-heap initialized to be $\emptyset$;
 3: for all $g \in D$ do
 4:     $\mathcal{H}.push((g, \underline{\mathsf{dist}}(q, g)))$;
 5: while $head(\mathcal{H}).dist < head(\mathcal{A}).dist$ and $\mathcal{H} \neq \emptyset$ do
 6:     $h \leftarrow \mathcal{H}.pop()$;
 7:     $g \leftarrow h.graph$;
 8:     if $\mathsf{dist}(q, g) < head(\mathcal{A}).dist$ then
 9:         $\mathcal{A}.pop()$;
10:         $\mathcal{A}.push((g, \mathsf{dist}(q, g)))$;
11: return the top-$k$ answers in $\mathcal{A}$;
```

$\mathsf{mcs}(q, g)$, i.e., $|E(q)| - |E(\mathsf{mcs}(q, g))|$, and the distance between g and $\mathsf{mcs}(q, g)$, i.e., $|E(g)| - |E(\mathsf{mcs}(q, g))|$ into consideration, which can measure the difference of the query graph and a database graph symmetrically. The following example shows that user desired top-1 similar graph in Example 1.1 can be retrieved by using our new graph distance measure.

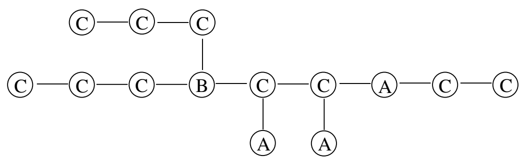
Example 2.2: Suppose a user wants to find the top-1 similar graph for query q from the graph database $D = \{g_1, g_2, g_3\}$ shown in Fig. 1. By using graph distance measure defined in Equation (1), g_2 will be returned as the answer, since $\mathsf{dist}(q, g_1) = 7+3-2\times2 = 6$, $\mathsf{dist}(q, g_2) = 7+7-2\times5 = 4$, and $\mathsf{dist}(q, g_3) = 7+16-2\times6 = 11$. Compared with Example 1.1, our graph distance measure can return the user desired answer. □

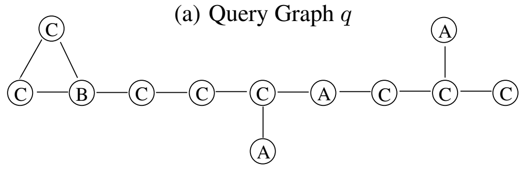
Problem Statement: Given a graph database $D = \{g_1, \ldots, g_n\}$, a query graph q, and a positive integer k, we aim to retrieve the top-k graphs from D with the smallest graph distances with q.

3. THE FRAMEWORK

Since computing *MCS* between two graphs is an NP-hard problem, it is inefficient to compute the exact graph distance between the query graph q and every graph $g \in D$. In this paper, we focus on reducing the number of *MCS* computations as much as possible, rather than making an *MCS* computation faster. To do this, we adopt a graph distance lower bound based pruning strategy: during the query processing, for a graph $g \in D$, if the lower bound of $\mathsf{dist}(q, g)$, denoted as $\underline{\mathsf{dist}}(q, g)$, is no less than the largest distance of the current top-k answers discovered so far, g is not a top-k answer and can be pruned safely.

The framework for the pruning strategy is shown in Algorithm 1. We use a max-heap $\mathcal{A}$ to maintain the current top-k answers, and use a min-heap $\mathcal{H}$ to keep the lower bounds of the graph distance for graphs in D to be processed. Each entry in each heap is with the form $e = (graph, dist)$, where $graph$ is a database graph and $dist$ is the distance value to be sorted in the heap. We use $e.graph$ and $e.dist$ to denote the two fields of e. Initially, each value in $\mathcal{A}$ is set to be $+\infty$ and we insert the lower bounds of graph distances into $\mathcal{H}$ in non-decreasing order for all graphs in D (line 1-4). Next, we iteratively pop the graph g with the minimum value from $\mathcal{H}$ and compute the distance to the query, $\mathsf{dist}(g, q)$ (line 6). If the distance is smaller than the maximum value of current top-k answers, $head(\mathcal{A}).dist$, we will remove the maximum value from $\mathcal{A}$, and push the new distance into $\mathcal{A}$ (line 8-10). We will stop the loop and report the final top-k answer if $\mathcal{H}$ is empty or the minimum value in $\mathcal{H}$ is no less than the largest value in $\mathcal{A}$ (line 5).


(a) Query Graph q


(b) Graph g_1

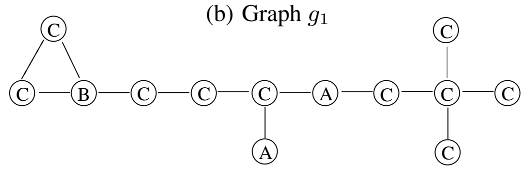

(c) Graph g_2

Figure 4: Example for the Bound of Graph Distance

The early stop condition $head(\mathcal{H}).dist \geq head(\mathcal{A}).dist$ in Algorithm 1 implies the pruning of unqualified candidates. There are two ways to make such pruning strategy more effective. First, to maximize $head(\mathcal{H}).dist$. Since all values in $\mathcal{H}$ are lower bounds of graph distances, a tight lower bound can maximize $head(\mathcal{H}).dist$. Second, to minimize $head(\mathcal{A}).dist$. If we can populate the top-k answer set $\mathcal{A}$ with graphs having very small distances to q as early as possible, then we will have a small $head(\mathcal{A}).dist$ for pruning more unqualified candidates. Thus the order to examine graph candidates is very important. In our framework, graphs are examined in the non-decreasing order of their lower bounds for the actual distance computation and insertion into $\mathcal{A}$. When the distance lower bound is a good estimation of the actual graph distance, $head(\mathcal{A}).dist$ can be minimized. From the above analysis, we conclude that both cases need a tight lower bound for a better pruning power. In the following sections, we propose several techniques to obtain a tight graph distance lower bound in order to reduce the number of *MCS* computations.

4. PRUNING WITHOUT INDEXING

As analyzed above, in order to reduce the number of *MCS* computations, we need a tight lower bound to prune graphs that cannot be a top-k answer. Meanwhile, the distance lower bound computation wrt. the query q should be light-weighted. In this section, we will introduce two lower bounds with different tradeoffs on tightness and computational cost. We discuss how to use them for candidate pruning in top-k graph similarity query.

4.1 Edge Frequency Based Lower Bound

Finding the lower bound of $\mathsf{dist}(q, g)$ for graphs q and g is equivalent to finding the upper bound of $|E(\mathsf{mcs}(q, g))|$. Given q and g, we can compute a rough upper bound of $|E(\mathsf{mcs}(q, g))|$ based on the frequency of distinct edges. A distinct edge in a graph g is defined as $(l(u), l(v))$, where $l(u)$ and $l(v)$ are the labels of vertices u and v. A distinct edge e may appear multiple times in a graph, and we denote the number of occurrences as frequency of e, $f(e, g)$. We denote the set of the distinct edges in g as $E_d(g)$. For two graphs q and g, our first upper bound of $|E(\mathsf{mcs}(q, g))|$, $\overline{\mathsf{emcs}}_1(q, g)$, can be obtained as follows based on their distinct edge frequencies.

$$\overline{\mathsf{emcs}}_1(q,g) = \sum_{e \in E_d(q) \cup E_d(g)} \min\{f(e,q), f(e,g)\} \quad (2)$$

Using the upper bound in Eq. (2), we can obtain our first lower bound of graph distance, $\underline{\mathsf{dist}}_1(q,g)$ as follows.

Theorem 4.1: *Given two graphs q, g, the following equation is a lower bound of* $\mathsf{dist}(q,g)$.

$$\underline{\mathsf{dist}}_1(q,g) = |E(q)| + |E(g)| - 2 \times \overline{\mathsf{emcs}}_1(q,g) \quad (3)$$

□

Proof Sketch: For each distinct edge e, it can contribute at most $\min\{f(e,q), f(e,g)\}$ edges in $\mathsf{mcs}(q,g)$, because only edges with the same label on the two ends can match each other. The sum of $\min\{f(e,q), f(e,g)\}$ for all distinct edges should be an upper bound of $|E(\mathsf{mcs}(q,g))|$. Thus Theorem 4.1 holds. □

Example 4.1: Let's consider graph q and g_1 in Fig. 4. The frequency of edge (A,C), (B,C), and (C,C) are 4, 3, 6 for q and 4, 3, 5 for g_1, respectively. The upper bound of $|E(\mathsf{mcs}(q,g_1))|$ is $\overline{\mathsf{emcs}}_1(q,g_1) = 4+3+5 = 12$ and $\underline{\mathsf{dist}}_1(q,g_1) = 13+12-2\times 12 = 1$. Similarly, we can get the lower bound of $\mathsf{dist}(q,g_2)$ as $\underline{\mathsf{dist}}_1(q,g_2) = 13+13-2\times 12 = 2$. In fact, these two lower bounds for $\mathsf{dist}(q,g_1)$ and $\mathsf{dist}(q,g_2)$ are not tight compared to the actual graph distances which are 9 and 10, respectively. □

Remark 4.1: *The time complexity of the edge frequency based upper bound computation is $O(|E(q)| + |E(g)|)$, since we have to scan all edges in the two graphs and count the frequency of each distinct edge.* □

4.2 Adjacency List Based Lower Bound

A tighter upper bound for $|E(\mathsf{mcs}(q,g))|$ of two graphs q and g is proposed by Raymond et al. [14] based on the sequences of vertices' adjacency list of two graphs. The main idea is as follows.

For each node v in a graph, we use $L(adj(v))$ to denote a multiset consisting of all labels in the adjacent nodes of v. A label may appear multiple times in $L(adj(v))$. Given two graphs q and g, we construct a bipartite graph $B(q,g)$ with $|V(q)|$ nodes on one side and $|V(g)|$ nodes on the other. We use $b(u)$ to denote the corresponding node u in $B(q,g)$. For each pair of nodes $u \in V(q)$ and $v \in V(g)$, there is an edge between $b(u)$ and $b(v)$ in $B(q,g)$ iff $l(u) = l(v)$. For each edge $(b(u), b(v)) \in E(B(q,g))$, the weight of edge $(b(u), b(v))$ is defined as $w(b(u), b(v)) = |L(adj(u)) \bigcap L(adj(v))|$. After constructing the bipartite graph $B(q,g)$, we find the maximum weighted bipartite matching $M(q,g)$ of $B(q,g)$ using Hungarian algorithm. Our second upper bound of $|E(\mathsf{mcs}(q,g))|$ can be derived as follows.

$$\overline{\mathsf{emcs}}_2(q,g) = \frac{\sum_{(b(u),b(v)) \in M(q,g)} w(b(u),b(v))}{2} \quad (4)$$

Based on the upper bound of the *MCS* in Eq. (4), we can obtain the second lower bound for graph distance as follows.

Theorem 4.2: *Given two graphs q, g, the following equation is a lower bound of* $\mathsf{dist}(q,g)$.

$$\underline{\mathsf{dist}}_2(q,g) = |E(q)| + |E(g)| - 2 \times \overline{\mathsf{emcs}}_2(q,g) \quad (5)$$

□

Proof Sketch: For a pair $u \in V(q)$ and $v \in V(g)$, suppose u is matched to v in $\mathsf{mcs}(q,g)$, then it can contribute $w(b(u), b(v))$ edges in $\mathsf{mcs}(q,g)$ at the most. Since $M(q,g)$ is the maximum matching among all matchings based on the weight $w(b(u), b(v))$ for each pair (u,v), $\sum_{(b(u),b(v)) \in M(q,g)} w(b(u), b(v))$ edges can be contributed to $\mathsf{mcs}(q,g)$ in the best case. As each edge is calculated twice at its two ends, the weight sum should be divided by 2. So Eq. (4) is an upper bound of $|E(\mathsf{mcs}(q,g))|$. Thus Theorem 4.2 holds.

Remark 4.2: *The worst case time complexity of the adjacency list based lower bound is $O(n^3)$, where $n = \max\{|V(q)|, |V(g)|\}$, which is the complexity of the Hungarian algorithm for best bipartite graph matching. Obviously, the computation of the adjacency list based lower bound is more costly than the edge frequency based lower bound.* □

Theorem 4.3: *Given two graphs q, g, for the lower bound of* $\mathsf{dist}(q,g)$*, we have $\underline{\mathsf{dist}}_2(q,g) \geq \underline{\mathsf{dist}}_1(q,g)$.* □

Proof Sketch: For a vertex u and a label a, we use $N(u,a)$ to denote the number of neighbors of u with label a, i.e., $N(u,a) = |\{u' | u' \in adj(u), l(u') = a\}|$. For any $(b(u), b(v)) \in E(B(q,g))$, $w(b(u), b(v))$ can be calculated as:

$$w(b(u), b(v)) = \sum_{b \in L(adj(u)) \bigcap L(adj(v))} \min\{N(u,b), N(v,b)\}$$

Let L be the set of all labels, we have:

$$\begin{aligned} 2 \times \overline{\mathsf{emcs}}_2(q,g) &= \sum_{(b(u),b(v)) \in M(q,g)} w(b(u), b(v)) \\ &= \sum_{a \in L} \sum_{(b(u),b(v)) \in M(q,g), l(u)=a} \\ &\quad \sum_{b \in L(adj(u)) \bigcap L(adj(v))} \min\{N(u,b), N(v,b)\} \\ &\leq \sum_{a \in L, b \in L} \min\{ \sum_{u \in V(q), l(u)=a} N(u,b), \\ &\quad \sum_{v \in V(g), l(v)=a} N(v,b)\} \end{aligned}$$

For $\overline{\mathsf{emcs}}_1(q,g)$, we can derive the following equation:

$$\begin{aligned} 2 \times \overline{\mathsf{emcs}}_1(q,g) = \sum_{a \in L, b \in L} \min\{ &\sum_{u \in V(g), l(u)=a} N(u,b), \\ &\sum_{v \in V(q), l(v)=a} N(v,b)\} \end{aligned}$$

Thus, we have $2 \times \overline{\mathsf{emcs}}_2(q,g) \leq 2 \times \overline{\mathsf{emcs}}_1(q,g)$. As a result, $\underline{\mathsf{dist}}_2(q,g) \geq \underline{\mathsf{dist}}_1(q,g)$ holds.

Example 4.2: For graphs q and g_1 in Fig. 4, after constructing the bipartite graph $B(q,g_1)$, we can find the upper bound of $|E(\mathsf{mcs}(q, g_1))|$ as $\overline{\mathsf{emcs}}_2(q,g_1) = 11$, and we have $\underline{\mathsf{dist}}_2(q,g_1) = 13+12-2\times 11 = 3$, which is tighter than $\underline{\mathsf{dist}}_1(q,g_1) = 1$ in Example 4.1. Similarly, we can get the lower bound of $\mathsf{dist}(q,g_2)$ for graphs q and g_2 as $\underline{\mathsf{dist}}_2(q,g_2) = 13+13-2\times 12 = 2$, which is the same as $\underline{\mathsf{dist}}_1(q,g_2) = 2$ in Example 4.1. □

Algorithm 2 gs-topk-noidx (D, q, k)

```
Input: graph database $D$, a query $q$, and an integer $k$;
Output: top-$k$ similar graphs with respect to $q$;
1: $\mathcal{A} \leftarrow$ max-heap of size $k$ with each value set to be $+\infty$;
2: $\mathcal{H} \leftarrow$ min-heap initialized to be $\emptyset$;
3: for all $g \in D$ do
4:     $\mathcal{H}.push((g, \underline{\mathsf{dist}}_1(q, g)))$;
5: while $head(\mathcal{H}).dist < head(\mathcal{A}).dist$ and $\mathcal{H} \neq \emptyset$ do
6:     $h \leftarrow \mathcal{H}.pop()$;
7:     $g \leftarrow h.graph$;
8:     if $h.dist = \underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g) < \underline{\mathsf{dist}}_1(q, g)$ then
9:         $\mathcal{H}.push((g, \underline{\mathsf{dist}}_2(q, g)))$;
10:        continue;
11:    if $\mathsf{dist}(q, g) < head(\mathcal{A}.dist)$ then
12:        $\mathcal{A}.pop()$;
13:        $\mathcal{A}.push((g, \mathsf{dist}(q, g)))$;
14: return the top-$k$ answers in $\mathcal{A}$;
```

4.3 Query Processing

As stated in the above two subsections, for two graphs q and g, computing $\underline{\mathsf{dist}}_2(q, g)$ is much more expensive than computing $\underline{\mathsf{dist}}_1(q, g)$. Thus we only compute $\underline{\mathsf{dist}}_2(q, g)$ for graphs that cannot be pruned by $\underline{\mathsf{dist}}_1(q, g)$. Furthermore, computing $\mathsf{dist}(q, g)$ is more expensive than computing $\underline{\mathsf{dist}}_2(q, g)$, thus we will not compute $\mathsf{dist}(q, g)$ unless g cannot be pruned by both $\underline{\mathsf{dist}}_2(q, g)$ and $\underline{\mathsf{dist}}_1(q, g)$.

Based on the above analysis on the tradeoffs between the computational cost and the bound tightness, we develop an algorithm to compute top-k graph similarity query, that follows the framework in Algorithm 1 with more details on updating the bounds. Our algorithm is shown in Algorithm 2. Initially, we compute $\underline{\mathsf{dist}}_1(q, g)$ for all database graphs wrt. q, and push them into $\mathcal{H}$ (line 3-4). In line 5, we iteratively update $\mathcal{A}$ and $\mathcal{H}$ with the same stop condition as in Algorithm 1. In each iteration, after popping the top entry h from $\mathcal{H}$, we first test whether h has been updated with $\underline{\mathsf{dist}}_2(q, g)$. If not and $\underline{\mathsf{dist}}_2(q, g) < \underline{\mathsf{dist}}_1(q, g)$, we update h with $\underline{\mathsf{dist}}_2(q, g)$ (line 8-9). If both $\underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g)$ have been used to update h, we know that the current lower bound in h is the best one we can get currently. Then we update the graph g in h using the real graph distance $\mathsf{dist}(q, g)$ and insert it into $\mathcal{A}$ if $\mathsf{dist}(q, g)$ is smaller than the largest value in $\mathcal{A}$ (line 11-13).

Remark 4.3: *With Algorithm 2, we can make sure that the number of* $\underline{\mathsf{dist}}_2$ *calculations is no larger than the number of* $\underline{\mathsf{dist}}_1$ *calculations and the number of* dist *calculations is no larger than the number of* $\underline{\mathsf{dist}}_2$ *calculations. This is because only the entries in* $\mathcal{H}$ *that have been updated with* $\underline{\mathsf{dist}}_1$ *can be updated using* $\underline{\mathsf{dist}}_2$*, and only those entries in* $\mathcal{H}$ *that have been updated using both* $\underline{\mathsf{dist}}_1$ *and* $\underline{\mathsf{dist}}_2$ *can be updated with* dist *and pushed into* $\mathcal{A}$. □

5. PRUNING WITH INDEXING

Up to now, we have derived two lower bounds $\underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g)$ on $\mathsf{dist}(q, g)$ for graphs q and g. Both $\underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g)$ only consider the relationship between q and g. From Example 4.1 and 4.2, we see that the two lower bounds can sometimes be very loose. In order to find tighter bounds, we study to derive the lower bound of $\mathsf{dist}(q, g)$ from a new angle, with the help of another database graph g'. The main observation is as follows. If g and g' are very similar, then $\mathsf{dist}(q, g)$ and $\mathsf{dist}(q, g')$ will not differ too much. Suppose we have computed $\mathsf{dist}(q, g')$, then we can use $\mathsf{dist}(q, g')$ to derive a new lower bound of $\mathsf{dist}(q, g)$. In the extreme case, when g and g' are isomorphic, $\mathsf{dist}(q, g)$ is the same as $\mathsf{dist}(q, g')$.

In this section, we first study how to derive a new lower bound using a triangle property of graph distance. In order to make use of such a property, we need to index the distances between database graphs. We design an algorithm on query processing using the index and introduce three different implementations of the index structures.

5.1 The Triangle Property of Graph Distance

First of all, we introduce the triangle inequality property of the graph distances for three graphs g_1, g_2 and g_3.

Lemma 5.1: *Given three graphs g_1, g_2, and g_3, the following triangle property holds.*

$$\mathsf{dist}(g_1, g_3) \leq \mathsf{dist}(g_1, g_2) + \mathsf{dist}(g_2, g_3) \tag{6}$$
□

Proof Sketch: Let α be the number of edges in g_2 not included in $E(\mathsf{mcs}(g_1, g_2))$ and $E(\mathsf{mcs}(g_2, g_3))$. we can express $|E(g_2)|$ as follows.

$$\begin{aligned} |E(g_2)| =& \alpha + |E(\mathsf{mcs}(g_1, g_2))| + |E(\mathsf{mcs}(g_2, g_3))| \\ & - |E(\mathsf{mcs}(g_1, g_2)) \cap E(\mathsf{mcs}(g_2, g_3))|. \end{aligned} \tag{7}$$

Also, for graphs g_1, g_2, and g_3, we have:

$$|E(\mathsf{mcs}(g_1, g_3))| \geq |E(\mathsf{mcs}(g_1, g_2)) \cap E(\mathsf{mcs}(g_2, g_3))| \tag{8}$$

Based on Eq. (7) and Eq. (8), we obtain:

$$\begin{aligned} &|E(\mathsf{mcs}(g_1, g_3)| \\ &\quad \geq \alpha + |E(\mathsf{mcs}(g_1, g_2))| + |E(\mathsf{mcs}(g_2, g_3))| - |E(g_2)| \\ &\quad \geq |E(\mathsf{mcs}(g_1, g_2))| + |E(\mathsf{mcs}((g_2, g_3))| - |E(g_2)| \end{aligned}$$

After multiplying the above equation by -2 and adding $|E(g_1)|$ and $|E(g_3)|$ to both sides, we have:

$$\begin{aligned} &|E(g_1)| + |E(g_3)| - 2 \times |E(\mathsf{mcs}(g_1, g_3)| \\ &\quad \leq |E(g_1)| + |E(g_2)| - 2 \times |E(\mathsf{mcs}((g_1, g_2))| \\ &\quad\quad + |E(g_2)| + |E(g_3)| - 2 \times |E(\mathsf{mcs}((g_2, g_3))| \end{aligned}$$

In other words, $\mathsf{dist}(g_1, g_3) \leq \mathsf{dist}(g_1, g_2) + \mathsf{dist}(g_2, g_3)$, which completes the proof. □

Theorem 5.1: *Given a query graph q and two graphs g and g' in graph database D, suppose we have computed* $\mathsf{dist}(q, g')$ *and* $\mathsf{dist}(g, g')$*, then a lower bound of* $\mathsf{dist}(q, g)$ *can be computed as follows.*

$$\underline{\mathsf{dist}}_3(q, g)[g'] = \mathsf{dist}(q, g') - \mathsf{dist}(g, g') \tag{9}$$
□

Theorem 5.1 can be derived directly from Lemma 5.1. Note that the lower bound $\underline{\mathsf{dist}}_3(q, g)[g']$ is determined by $\mathsf{dist}(q, g')$ and $\mathsf{dist}(g, g')$. If $\mathsf{dist}(q, g')$ is small and $\mathsf{dist}(g, g')$ is large, the lower bound might be quite loose, or even looser than $\underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g)$, which will be useless for pruning. If $\mathsf{dist}(q, g')$ is large and $\mathsf{dist}(g, g')$ is small, we might get a large lower bound of $\mathsf{dist}(q, g)$, which can be tighter than $\underline{\mathsf{dist}}_1(q, g)$ and $\underline{\mathsf{dist}}_2(q, g)$, and thus helpful for pruning.

Example 5.1: Let's consider the query graph q and database graphs g_1 and g_2 shown in Fig. 4. Suppose we have computed $|E(mcs(q,$

Algorithm 3 gs-topk-withidx (D, q, k, I)

```
Input: graph database $D$, a query $q$, an integer $k$, and index $I$;
Output: top-$k$ similar graphs with respect to $q$;
 1: $\mathcal{A} \leftarrow$ max-heap of size $k$ with each value set to be $+\infty$;
 2: $\mathcal{H} \leftarrow$ min-heap initialized to be $\emptyset$;
 3: for all $g \in D$ do
 4:     $\mathcal{H}.push((g, \underline{\mathsf{dist}}_1(q, g)))$;
 5: while $head(\mathcal{H}).dist < head(\mathcal{A}).dist$ and $\mathcal{H} \neq \emptyset$ do
 6:     $h \leftarrow \mathcal{H}.pop()$;
 7:     $g \leftarrow h.graph$;
 8:     if $\underline{\mathsf{dist}}_2(q, g) < h.dist$ then
 9:         $\mathcal{H}.push((g, \underline{\mathsf{dist}}_2(q, g)))$;
10:         continue;
11:     if $\mathsf{dist}(q, g) < head(\mathcal{A}).dist$ then
12:         $\mathcal{A}.pop()$;
13:         $\mathcal{A}.push((g, \mathsf{dist}(q, g)))$;
14:     update-using-index $(\mathcal{H}, q, g, I)$;
15: return the top-$k$ answers in $\mathcal{A}$;

16: Procedure update-using-index $(\mathcal{H}, q, g, I)$
17: for all $G_i \in I$ such that $g \in G_i$ do
18:     if $g = c_i$ then
19:         for all $g' \in G_i$ and $g' \neq g$ do
20:             $\mathcal{H}.update((g', \underline{\mathsf{dist}}_3(q, g')[g]))$;
21:     else
22:         if $\mathcal{H}.update((c_i, \underline{\mathsf{dist}}_3(q, c_i)[g]))$ then
23:             for all $g' \in G_i$ and $g' \neq g$ and $g' \neq c_i$ do
24:                 $\mathcal{H}.update((g', \underline{\mathsf{dist}}_4(q, g')[c_i]))$;
```

$g_1))| = 8$, then we can get $\mathsf{dist}(q, g_1) = 12 + 13 - 2 \times 8 = 9$. In addition, suppose we have precomputed $\mathsf{dist}(g_1, g_2) = 12 + 13 - 2 \times 11 = 3$. According to Theorem 5.1, we can obtain a lower bound of $\mathsf{dist}(q, g_2)$ to be $\underline{\mathsf{dist}}_3(q, g_2)[g_1] = 9 - 3 = 6$, which is much tighter than $\underline{\mathsf{dist}}_2(q, g_2) = 2$ in Example 4.2. Similarly, if we have obtained $\mathsf{dist}(q, g_2) = 13 + 13 - 2 \times 8 = 10$ and $\mathsf{dist}(g_1, g_2) = 3$, we can derive that $\underline{\mathsf{dist}}_3(q, g_1)[g_2] = 10 - 3 = 7$, which is also much tighter than $\underline{\mathsf{dist}}_2(q, g_1) = 3$ in Example 4.2. □

Computing $\underline{\mathsf{dist}}_3(q, g)$ requires two actual graph distances $\mathsf{dist}(q, g')$ and $\mathsf{dist}(g, g')$. In case we have $\underline{\mathsf{dist}}(q, g')$ but not $\mathsf{dist}(q, g')$, we can still derive a relaxed lower bound of $\mathsf{dist}(q, g)$ based on the triangle property as follows. We first compute a lower bound of $\mathsf{dist}(q, g')$, denoted $\underline{\mathsf{dist}}(q, g')$, using any one of the three lower bounds introduced above. The relaxed triangle based bound is presented in Theorem 5.2.

Theorem 5.2: *Given a query graph q and two graphs g and g′ in graph database D, suppose we have computed* $\mathsf{dist}(g, g')$ *and a lower bound of* $\mathsf{dist}(q, g')$ *denoted as* $\underline{\mathsf{dist}}(q, g')$*, then a lower bound of* $\mathsf{dist}(q, g)$ *can be computed as follows.*

$$\underline{\mathsf{dist}}_4(q, g)[g'] = \underline{\mathsf{dist}}(q, g') - \mathsf{dist}(g, g') \tag{10}$$

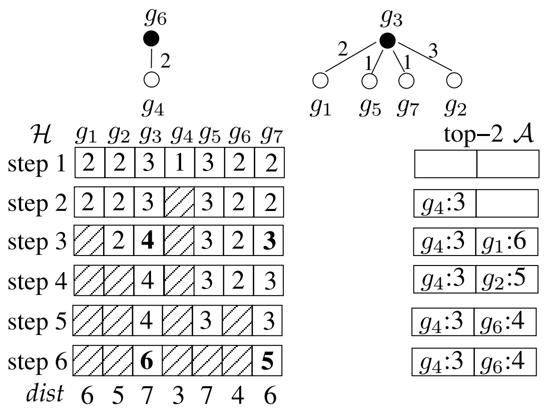
□

Proof Sketch: Since $\underline{\mathsf{dist}}(q, g') \leq \mathsf{dist}(q, g')$, we have:

$$\begin{aligned}\underline{\mathsf{dist}}_4(q, g)[g'] &= \underline{\mathsf{dist}}(q, g') - \mathsf{dist}(g, g')\\ &\leq \mathsf{dist}(q, g') - \mathsf{dist}(g, g')\\ &= \underline{\mathsf{dist}}_3(q, g)[g']\end{aligned}$$

In fact, $\underline{\mathsf{dist}}_4(q, g)[g']$ is a relaxation of the lower bound $\underline{\mathsf{dist}}_3(q, g)[g']$. □

Remark 5.1: *If we know* $\mathsf{dist}(g, g')$*, and one of* $\mathsf{dist}(q, g')$ *and its lower bound* $\underline{\mathsf{dist}}(q, g')$*, the time complexity to update the lower bound of* $\mathsf{dist}(q, g)$ *using* $\underline{\mathsf{dist}}_3(q, g)[g']$ *or* $\underline{\mathsf{dist}}_4(q, g)[g']$ *is* $O(1)$*. Since calculating* $\underline{\mathsf{dist}}_3(q, g)[g']$ *or* $\underline{\mathsf{dist}}_4(q, g)[g']$ *is very cheap, we should use them as many times as needed.* □



$\mathcal{H}$	g_1	g_2	g_3	g_4	g_5	g_6	g_7	top–2 $\mathcal{A}$	
step 1	2	2	3	1	3	2	2		
step 2	2	2	3	▨	3	2	2	g_4:3	
step 3	▨	2	**4**	▨	3	2	**3**	g_4:3	g_1:6
step 4	▨	▨	4	▨	3	2	3	g_4:3	g_2:5
step 5	▨	▨	4	▨	3	▨	3	g_4:3	g_6:4
step 6	▨	▨	**6**	▨	▨	▨	**5**	g_4:3	g_6:4
dist	6	5	7	3	7	4	6		

Figure 5: An Example for Top-k Query

5.2 Query Processing

Based on the above discussion, we know that the key point to get a tight bound for $\mathsf{dist}(q, g)$ is to choose a graph g' with a small distance $\mathsf{dist}(g, g')$. Since g and g' are database graphs, the graph distance $\mathsf{dist}(g, g')$ can be precomputed offline. A straightforward way is to precompute the graph distances for all pairs of graphs in D. Then we need to do $O(|D|^2)$ *MCS* computations, which is too expensive to be practical, as $|D|$ can range from thousands to millions. In addition, we know that a large value of $\mathsf{dist}(g, g')$ might not help us get a tight lower bound of $\mathsf{dist}(q, g)$. So it will be sufficient to index the pairs of graphs with small graph distances for pruning. In order to do this, we define a graph index I as follows.

Definition 5.1: Top-k Graph Similarity Index: Given a graph database D, a top-k graph similarity index I consists of a set of groups, $I = \{G_1, \ldots, G_{|I|}\}$, where $G_i \subseteq D$ for $i = 1, \ldots, |I|$, and $G_1 \cup \ldots \cup G_{|I|} = D$. For each group G_i, there is a center graph $c_i \in G_i$, denoted as $c(G_i)$, and the graph distance $\mathsf{dist}(g, c_i)$ for each $g \in G_i$ and $g \neq c_i$ is precomputed and stored in I. □

Ideally, graphs having small distances with each other should be assigned to the same group of I to support effective pruning. In the following, we focus on how to process queries using I. We will discuss the construction of I in details in the next subsection.

Our main algorithm is shown in Algorithm 3. The process is similar to Algorithm 2. The only difference is that, after calculating the real graph distance $\mathsf{dist}(q, g)$ in line 11-13, we need to use $\mathsf{dist}(q, g)$ and the index I to update the lower bounds of other graphs, by invoking a procedure update-using-index $(\mathcal{H}, q, g, I)$. Next we introduce the procedure update-using-index in details. In order to update the lower bounds of other graphs using g and I, we should find all groups $G_i \in I$ that contain g. For each group G_i that contains g, there are two cases. First, g is the center c_i of G_i. In this case, we can update the lower bound of every non-center graph $g' \in G_i$ using $\underline{\mathsf{dist}}_3(q, g')[g]$ by invoking a function $\mathcal{H}.update((g', \underline{\mathsf{dist}}_3(q, g')[g]))$ (line 19-20). The function $\mathcal{H}.update((g, dist))$ works as follows. We find the entry h in $\mathcal{H}$ where $h.graph = g$. If $dist > h.dist$, then the original entry h is updated to be $(g, dist)$, and the function returns true. Otherwise, nothing is done on $\mathcal{H}$ and the function returns false. Second, g is not the center c_i of G_i. In this situation, we first update the lower bound of the center c_i, using $(c_i, \underline{\mathsf{dist}}_3(q, c_i)[g])$. If $\underline{\mathsf{dist}}_3(q, c_i)[g]$ is larger than the original one in $\mathcal{H}$, we can use $\mathsf{dist}(c_i, g')$ and the lower bound $\underline{\mathsf{dist}}_3(q, c_i)[g]$ to update the lower bound of each other graph g' using $\underline{\mathsf{dist}}_4(q, g')[c_i]$ (line 22-24).

Example 5.2: Fig. 5 shows an example for Algorithm 3. Suppose we have a sample database $D = \{g_1, \ldots, g_7\}$, and the index I contains 2 groups, which are shown in the upper part of Fig. 5. The black node in each group is the center of the group. For ease of reference, we label the real graph distance between each non-center graph and the center graph in the figure. Suppose we have a query q and we want to find its top-2 similar graphs. We show $\mathcal{H}$ which maintains the lower bound for each graph on the left side and the answer set $\mathcal{A}$ on the right side. The steps are listed as follows.

- Step 1: $\mathcal{H}$ contains the best lower bound we have obtained for each graph using $\underline{\mathsf{dist}}_1$ and $\underline{\mathsf{dist}}_2$.
- Step 2: Since the lower bound of g_4 is the minimum, we compute $\mathsf{dist}(q, g_4) = 3$ and push it into $\mathcal{A}$. We also compute $\underline{\mathsf{dist}}_3(q, g_6)[g_4] = \mathsf{dist}(q, g_4) - \mathsf{dist}(g_6, g_4) = 1$. As $\underline{\mathsf{dist}}_3(q, g_6)[g_4]$ is smaller than the current lower bound of g_6 in $\mathcal{H}$, it is not updated.
- Step 3: We compute $\mathsf{dist}(q, g_1) = 6$ and push it into $\mathcal{A}$. We also have $\underline{\mathsf{dist}}_3(q, g_3)[g_1] = 6{-}2 = 4$ and $\underline{\mathsf{dist}}_4(q, g_7)[g_3] = 4 - 1 = 3$. So the lower bound of g_3 is updated to 4 and the lower bound of g_7 is updated to 3 in $\mathcal{H}$.
- Step 4: We compute $\mathsf{dist}(q, g_2) = 5$ and it replaces g_1 in $\mathcal{A}$ as $\mathsf{dist}(q, g_2) < \mathsf{dist}(q, g_1)$. We compute $\underline{\mathsf{dist}}_3(q, g_3)[g_2] = 5 - 3 = 2$, which is smaller than the current lower bound of g_2 in $\mathcal{H}$. So the lower bound of g_2 is not updated.
- Step 5: We compute $\mathsf{dist}(q, g_6) = 4$ and it replaces g_2 in $\mathcal{A}$.
- Step 6: We compute $\mathsf{dist}(q, g_5) = 7$. We can update the lower bounds of g_3 and g_7 in $\mathcal{H}$ as $\underline{\mathsf{dist}}_3(q, g_3)[g_5] = 7{-}1 = 6$ and $\underline{\mathsf{dist}}_4(q, g_7)[g_3]{=}\ 6 - 1 = 5$. As the min value 5 in $\mathcal{H}$ is no less than the max value in $\mathcal{A}$ for the current top-2 answers, g_3 and g_7 can be pruned and the two graphs in $\mathcal{A}$ are returned as the answers.

In the above process, we totally need 5 ***MCS*** computations to get the top-2 answers. If we do not use the lower bounds $\underline{\mathsf{dist}}_3$ and $\underline{\mathsf{dist}}_4$, no graph can be pruned, because the lower bounds from $\underline{\mathsf{dist}}_1$ and $\underline{\mathsf{dist}}_2$ are loose and smaller than the max value in $\mathcal{A}$. □

5.3 Indexing

In this subsection, we discuss how to build index I for effective pruning based on lower bounds $\underline{\mathsf{dist}}_3$ and $\underline{\mathsf{dist}}_4$. We design three indexing schemes, namely disjoint partition index, overlapping partition index and general similarity index, with different tradeoffs between construction cost and pruning power.

Disjoint Partition Index (DPIndex)**:** As analyzed above, in each group of the index, graphs should have small distances with each other. It is natural to partition the graphs in D into non-overlapping clusters. Based on this idea, we build the Disjoint Partition Index (DPIndex) as follows. Given the number of clusters m, we randomly pick m graphs as m center nodes for the clusters. Then for each non-center graph $g \in D$, we assign it to the nearest center. We consider each cluster as a group in the index. Suppose $D_c \subset D$ is the set of center nodes we select. Finding the nearest center for a graph g can be considered as answering a top-1 graph similarity query in D_c, and can be solved using gs-topk-noidx $(D_c, g, 1)$ shown in Algorithm 2.

Lemma 5.2: *Constructing* DPIndex *needs* $|D| - m$ *times of* gs-topk-noidx $(D_c, g, 1)$ *computations. The size of the* DPIndex *is* $O(|D|)$. □

Overlapping Partition Index (OPIndex)**:** In DPIndex, each graph only belongs to one group. There are two drawbacks for DPIndex. First, if the center nodes are not selected properly, it is possible to derive useless lower bounds. Second, since computing lower bounds $\underline{\mathsf{dist}}_3$ and $\underline{\mathsf{dist}}_4$ using index is very fast, assigning each graph to only one group does not make full use of the triangle based bounds. In order to increase the chance for graphs to get a tighter lower bound, we develop another index, called Overlapping Partition Index (OPIndex), that allows each graph to belong to multiple groups. Suppose that each graph can belong to at most l groups. The construction of OPIndex is similar to DPIndex, the only difference is that, after we randomly select m graphs in D as centers, for each non-center graph $g \in D$, we assign it to the l nearest centers instead of only one nearest center. Finding l nearest centers can be considered as answering a top-l graph similarity query in D_c, and can be solved using gs-topk-noidx (D_c, g, l) shown in Algorithm 2.

Lemma 5.3: *Constructing* OPIndex *needs* $|D| - m$ *times of* gs-topk-noidx (D_c, g, l) *computations. The size of the* OPIndex *is* $O((|D| - m) \cdot l)$. □

General Similarity Index (GSIndex)**:** For DPIndex and OPIndex, there is no overlapping between the set of center graphs and non-center graphs. As a result, a graph can either be a center graph or a non-center graph but not both. For a non-center graph g, it is with large possibility that the l nearest graphs in D are not in the center set D_c, but in the non-center set $D - D_c$. To solve this problem, we develop a General Similarity Index(GSIndex) as follows. We treat each graph in D as the center. Then for each center, we find its nearest l graphs in D, and putting the $l + 1$ graphs together as a group. Finding l nearest graphs in D can be considered as answering a top-l graph similarity query in D, and can be solved using gs-topk-noidx (D, g, l) shown in Algorithm 2.

Lemma 5.4: *Constructing* GSIndex *needs* $|D|$ *times of* gs-topk-noidx (D, g, l) *computations. The size of the* GSIndex *is* $O(|D| \cdot l)$. □

Remark 5.2: *Among the three indexes,* DPIndex, OPIndex *and* GSIndex, GSIndex *has the largest pruning power while* DPIndex *is the weakest. Regarding the construction cost, computing* gs-topk-noidx (D, g, l) *is more expensive than computing* gs-topk-noidx (D_c, g, l)*, because* $D_c \subset D$*, and computing* gs-topk-noidx (D_c, g, l) *is more expensive than* gs-topk-noidx $(D_c, g, 1)$*, because* $l > 1$*. Thus* GSIndex *has the highest construction cost while* DPIndex *has the lowest construction cost. Regarding the index size,* GSIndex *has the largest size while* DPIndex *has the smallest size. But even for* GSIndex*, the index is very compact and can be memory resident, as there are totally* $|D| \cdot l$ *index entries in the form of* $(gid, dist)$*. For a graph database with millions of graphs and a typical setting of* $l = 10$*, we only need to store* $\sim 10^7$ *entries for less than 100 megabytes.* □

6. PERFORMANCE STUDIES

In this section, we conducted extensive performance studies to evaluate the effectiveness and efficiency of our algorithms. To the best

Table 1: Parameters

parameter	range	default
k	10, 20, 30, 40, 50	30
$\|V(q)\|$	1∼7, 8∼9, 10∼11, 12∼13, 14∼15, 1∼15	1∼15
$\|D\|$	10K, 20K, 40K, 60K, 80K, 100K	10K
m	1K, 2K, 3K, 4K, 5K	5K
l	2, 6, 10, 14, 18	10

of our knowledge, there is no existing algorithm on top-k graph similarity query using the *MCS* based similarity measure as defined in this paper. The most related and recent work is subgraph similarity query [15] and supergraph similarity query [17]. Both of them are threshold based approaches, which aim to answer the subgraph/supergraph similarity query with respect to a user-specified threshold. We first compare our algorithms with these two works to demonstrate the superiority of our algorithms in finding the most similar graphs to a query. Next we validate the efficiency of our algorithms by comparing them with a naive sequential scan method as a performance benchmark, followed by an extensive evaluation of our indexing techniques. All the algorithms were implemented using Visual C++ 2005 and tested on a PC with 2.66GHz CPU and 3.43GB memory running Windows XP.

We denote the subgraph/supergraph similarity query in [15] and [17] as SubSim and SuperSim respectively. The benchmark algorithm is denoted as SeqScan, which answers a top-k similarity query by performing a sequential scan on the graph database D for a query q. For each $g_i \in D$, SeqScan will first check whether $||E(q)| - |E(g_i)||$ (a simple lower bound of $\mathsf{dist}(q, g_i)$) is less than the largest distance of the current top-k answers. If yes, it performs an *MCS* computation to get the graph distance and decides whether to put g_i into the answer set or not. We denote our four approaches as noIndex, DPIndex, OPIndex and GSIndex respectively. Among them, noIndex is the algorithm to prune graphs using two lower bounds $\underline{\mathsf{dist}}_1$ and $\underline{\mathsf{dist}}_2$ without using any index (refer to Algorithm 2), DPIndex, OPIndex and GSIndex denote the approaches that follow Algorithm 3, using the indexes DPIndex, OPIndex and GSIndex respectively.

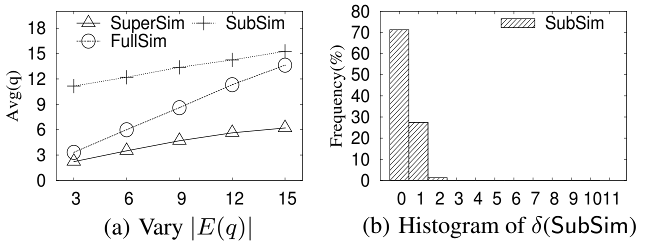
Datasets: We evaluate the performance of our algorithms on a real NCI dataset downloaded from the National Cancer Institute Open Database website[1]. The NCI dataset contains a list of compound structures of cancers and AIDS, where each structure can be modeled as an undirected and labeled graph. We extract six datasets that consist of 10K, 20K, 40K, 60K, 80K and 100K graphs, respectively. We also randomly extract another 1,000 graphs as the query set. The size of the query graphs ranges from 1 to 15.

Parameters: We evaluate our approaches by varying five parameters, namely, the top-k value in the query, the size of the query graph $|V(q)|$, the number of graphs in the graph database $|D|$, the number of groups m used in DPIndex and OPIndex, and the maximum number of groups l that each graph can belong to in OPIndex and GSIndex. The ranges for the five parameters and their default values are shown in Table 1. If not otherwise specified, we will use the default value for each parameter. For the query graph size $|V(q)|$, except for cases when varying $|V(q)|$, we test all 1,000 query graphs and report the average performance.

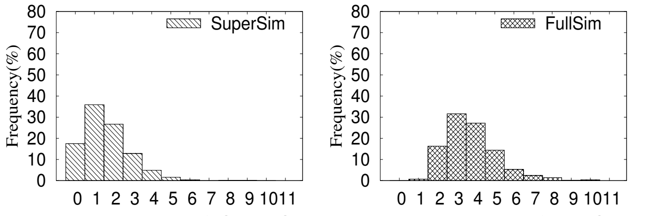
6.1 Similarity Measures Evaluation

~~We first introduce the similarity q~~ueries to be evaluated below.

[1]http://cactus.nci.nih.gov/download/nci/

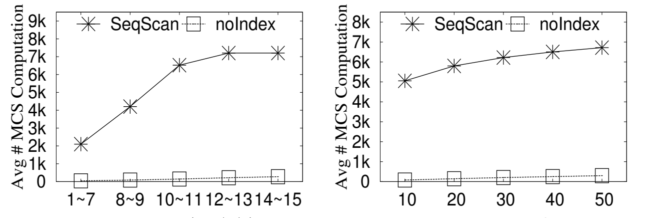


(a) Vary $|E(q)|$ (b) Histogram of δ(SubSim)



(c) Histogram of δ(SuperSim) (d) Histogram of δ(FullSim)

Figure 6: Similarity Measures Comparison



(a) Vary $|V(q)|$ (b) Vary k

Figure 7: Power of Pruning Strategy

- SubSim: Subgraph similarity query in [15], where graph distance is defined as $\mathsf{dist}(q, g) = |E(q)| - |E(\mathsf{mcs}(q, g))|$.
- SuperSim: Supergraph similarity query in [17], where graph distance is defined as $\mathsf{dist}(q, g) = |E(g)| - |E(\mathsf{mcs}(q, g))|$.
- FullSim: Full graph similarity query defined in this paper, where graph distance is defined as $\mathsf{dist}(q, g) = |E(q)| + |E(g)| - 2 \times |E(\mathsf{mcs}(q, g))|$ in Equation (1).

First of all, we show why the query results of subgraph/supergraph similarity query are not good answers of graph similarity query by empirical testings. Both SubSim and SuperSim are designed for threshold based query instead of top-k query. To get top-k results for these two graph distance measures, we embed them into our algorithm noIndex by replacing our graph distance by graph distances in SubSim and SuperSim respectively. The two lower bounds $\underline{\mathsf{dist}}_1$ and $\underline{\mathsf{dist}}_2$ are modified accordingly. The testing is conducted on dataset D that consists of 10K graphs for the query set of 1,000 graphs.

As we stated before, two graphs cannot be considered similar if their sizes have a significant disparity. To evaluate these three graph distance measures, we report the average size of the top-k answer graphs, $Avg(q)$, for a query q, defined as $Avg(q) = \frac{1}{k}\sum_{g \in \mathcal{A}} |E(g)|$ where $\mathcal{A}$ is the top-k answer set of q. Fig. 6(a) shows $Avg(q)$ for query q with size $|E(q)|$ in the range $3 \sim 15$. As we can see, no matter how $|E(q)|$ changes, SubSim always returns the answer graphs of very large sizes, e.g., $Avg(q) = 12$ even for the small query of size 3. In contrast, SuperSim always returns the answer graphs of small sizes no matter how large the query size is, e.g., $Avg(q) = 6$ even for the large query of size 15. Our measure FullSim can report different average sizes of answer graphs for different query sizes, and the average size of the answer graphs is increasing linearly as the query size increases.

Next, we empirically show that threshold based query techniques in [15] [17] can hardly be used to answer top-k queries. We tested 1,000 queries on dataset D of 10K graphs with $k = 30$ for these three measures respectively. For each query, we denote the largest distance between the top-k answers and the query q as δ. We report the frequency of δ for the 1,000 queries in Fig. 6(b) $\sim$ Fig. 6(d). Fig. 6(b) shows that for most queries in SubSim, δ is less than 2, where $\delta = 0$ occurs with frequency around 70% and $\delta = 1$ occurs with frequency around 30%. For SuperSim, Fig. 6(c) shows that δ falls into [0, 5] for most queries, where $\delta = 1$ and $\delta = 2$ occur with high frequencies. Fig. 6(d) shows that for most queries in FullSim, δ falls into interval [2, 7] where $\delta = 3$ and $\delta = 4$ occur with high frequencies. From Fig. 6(b) $\sim$ Fig. 6(d), we can see that, given a specific value of k, δ varies a lot for all these three measures, especially FullSim. It is impossible to determine a fixed δ to answer the exact top-k answers for different query graphs.

6.2 Query Performance Evaluation

We evaluate query performances of our four algorithms noIndex, DPIndex, OPIndex and GSIndex against the benchmark algorithm SeqScan. First of all, we test the power of our lower bound based punning strategy with $\underline{\text{dist}}_1$ and $\underline{\text{dist}}_2$. We compare noIndex with SeqScan with respect to two parameters $|V(q)|$ and k. In order to vary the number of nodes in the query graph, we divide the 1,000 query graphs into 5 groups, with size $|V(q)|$ in intervals 1 $\sim$ 7, 8 $\sim$ 9, 10 $\sim$ 11, 12 $\sim$ 13, and 14 $\sim$ 15 respectively. The result on the average number of *MCS* computations is shown in Fig. 7(a). When the query size increases, the number of *MCS* computations for noIndex and SeqScan increases. SeqScan needs around 7000 *MCS* computations for graph with size larger than 10, while noIndex needs no more than 500 *MCS* computations for all testing cases, which improves the performance by about two orders of magnitude. We vary the top-k value from 10 to 50, and report the average number of *MCS* computations in Fig. 7(b). When k increases, the average number of *MCS* computations of both noIndex and SeqScan increase. This is because when k is large, the distance of the k-th graph in the answer set will be large, thus the filtering condition for both noIndex and SeqScan will be hard to be satisfied for candidate pruning. Compared with SeqScan, noIndex also improves the performance by about two orders of magnitude.

Next, we evaluate our three indexing techniques with noIndex as the baseline. We vary five parameters $|V(q)|$, m, l, k, and $|D|$. For each testing case, we report the average number of *MCS* computations and the average query processing time for the corresponding algorithm.

Vary Query Size $|V(q)|$: The result on the average number of *MCS* computations is shown in Fig. 8(a). When the query size increases, the number of *MCS* computations for all four algorithms increases. noIndex needs nearly 300 *MCS* computations using the database with 10K graphs. GSIndex only needs around 150 *MCS* computations, which save 50% cost on *MCS* computations. The result on the average query processing time is shown Fig. 8(b). When the query size is large, the query processing time for all four algorithms has a sharp increase. This is because the cost of *MCS* computation increases exponentially with respect to the graph size. The query time is in the scale of seconds, as the major computation cost is from *MCS* computation which is expensive. In all cases, GSIndex performs best and noIndex performs worst.

Vary Number of Groups m: Fig. 8(c) and Fig. 8(d) show the curves when varying the number of groups m in DPIndex and OPIndex. We also show the performance of noIndex, which is constant wrt. m, as a baseline performance. In Fig. 8(c), when m increases, the number of *MCS* computations for both DPIndex and OPIndex decreases. This is because when the number of groups is larger, more graphs are selected as the center graphs, which have larger pruning power than non-center graphs. The gap between DPIndex and OPIndex becomes larger when m increases. It is because when the number of groups becomes larger, each non-center graph will have a larger chance to find a good center graph that is similar to the non-center one, and thus the pruning power will increase. The curves for the processing time in Fig. 8(d) are similar to the curves for the number of *MCS* computations. OPIndex has larger pruning power than DPIndex in all cases.

Vary l: We vary the maximum number of groups l each graph can belong to in OPIndex and GSIndex. Fig. 8(e) shows that, when l increases from 2 to 18, the pruning power for both OPIndex and GSIndex increases. This is because when l is larger, the lower bound for each graph is more likely to be updated by the center graphs. GSIndex performs better than OPIndex, and the gap between OPIndex and GSIndex increases when l increases. This is because, in GSIndex, each graph finds its top-l similar graphs in the database D for the lower bound update; while in OPIndex, each non-center graph finds its top-l similar graphs in the center graph set D_c. As $D_c \subset D$, the top-l distances indexed for each graph in OPIndex is no less than those in GSIndex, which will cause the triangle based lower bounds looser. Fig. 8(f) shows the processing time when varying l. The curves are similar to those in Fig. 8(e).

Vary k: We vary the top-k value from 10 to 50, and report the average processing time in Fig. 8(g). When k increases, the processing time for all four algorithms increases. This is because when k is large, the distance of the k-th graph in the answer set will be large, thus the early stop condition will be hard to be satisfied for candidate pruning. GSIndex performs best in all cases. The curves for the average number of *MCS* computations are similar to those for the average processing time, and not shown here due to the lack of space.

Vary Database Size $|D|$: Fig. 8(h) shows the average processing time when varying the number of graphs in the database $|D|$. The processing time increases sub-linearly with respect to $|D|$, because k is fixed and when the number of graphs in the database is large, the distance of the k-th graph in the answer set will be small, thus the early stop condition will be effective for candidate pruning. When $|D| = 100$K, GSIndex can save nearly 65% computational cost compared to noIndex. The curves for the average number of *MCS* computations are similar to those for the average processing time, and not shown here due to the lack of space.

6.3 Indexing Cost Evaluation

We test the three indexes DPIndex, OPIndex and GSIndex for different m, l and $|D|$ values. For each testing case, we report the corresponding index construction time and the size of the index. The results are shown in Fig. 9.

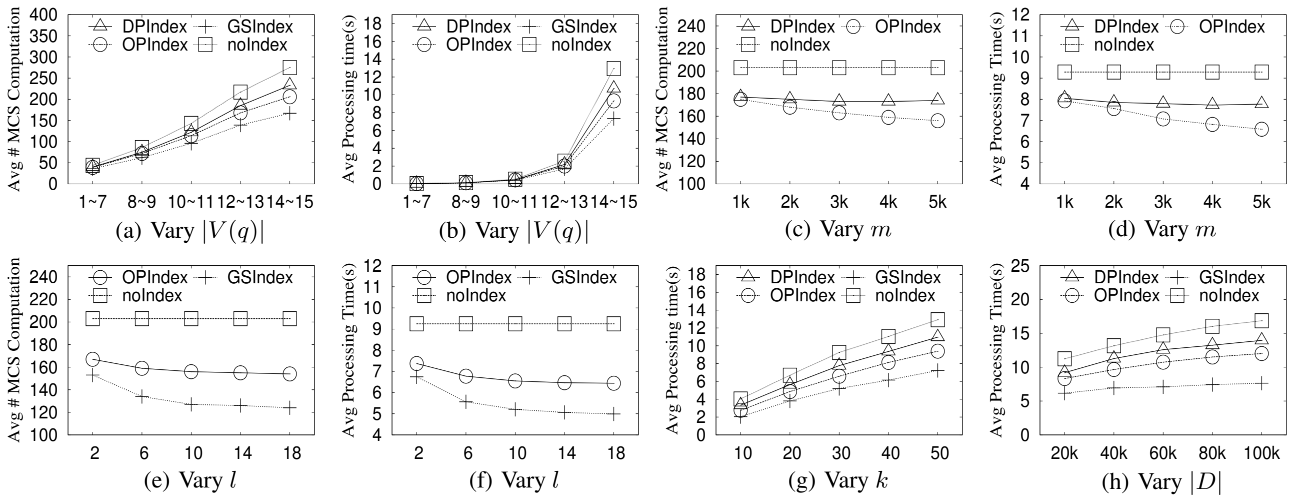


Figure 8: Scalability Testing

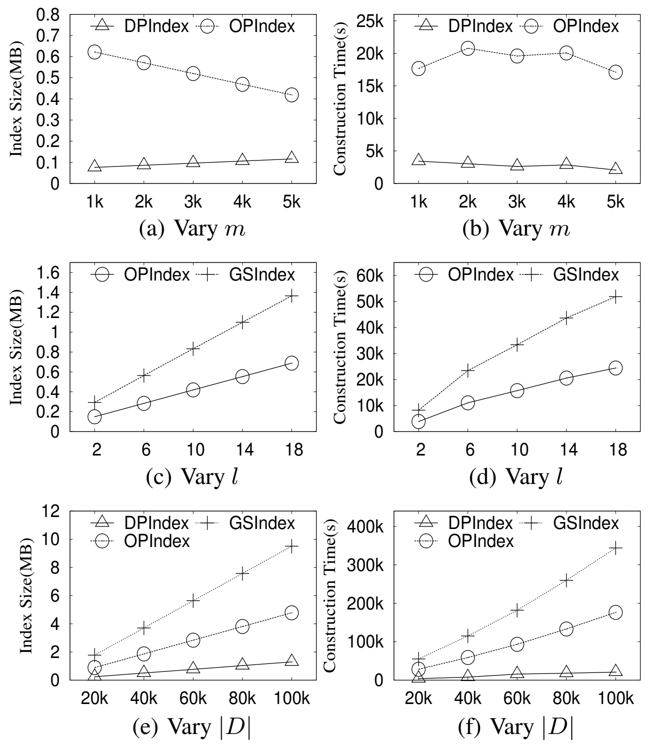


Figure 9: Index Testing

Vary Number of Groups m: Fig. 9(a) shows the index size for DPIndex and OPIndex when varying the number of groups m. The size of OPIndex is several times larger than that of DPIndex on average since OPIndex allows graphs to belong to multiple groups. As the number of groups increases, the size of DPIndex remains almost constant while the size of OPIndex decreases linearly, which confirm the space analysis for DPIndex ($O(|D|)$) and OPIndex ($O((|D| - m) \cdot l)$). Fig. 9(b) shows that the construction time of OPIndex is about 10 times larger than that of DPIndex on average. This is because the computation cost of gs-topk-noidx $(D_c, g, 10)$ is much larger than that of gs-topk-noidx $(D_c, g, 1)$. In addition, it shows that the construction time of OPIndex is smaller when m is either too large or too small. The underlying reason is as follows: when m is too small, D_c will be small, and gs-topk-noidx (D_c, g, l) will be fast; when m is too large, $|D| - m$ will be small, and we only need to compute gs-topk-noidx (D_c, g, l) for $|D| - m$ graphs, which also makes the index construction fast.

Vary l: We test the index size and the index construction time for OPIndex and GSIndex when varying the maximum number of groups l that each graph can belong to. The results are shown in Fig. 9(c) and Fig. 9(d) respectively. The results on both index size and construction time are consistent with the theoretical results in Lemma 5.3 and Lemma 5.4. GSIndex is larger and needs more construction time in all cases.

Vary Database Size $|D|$: Fig. 9(e) and Fig. 9(f) show the index size and index construction time for all three indexes DPIndex, OPIndex and GSIndex when varying the number of graphs $|D|$ in the database. The index sizes for all indexes increase linearly with respect to $|D|$. The processing time for all indexes also increases when $|D|$ increases. GSIndex is the most costly one among all indexes on both index size and construction time. Overall all indexes are very compact. The largest size is 10 megabytes for GSIndex when $|D| = 100$K.

7. RELATED WORK

Graph query processing has been studied a lot in recent years. Existing works on graph query processing can mainly be divided into three categories, namely, subgraph/supergraph containment query, subgraph/supergraph similarity query and full graph similarity query.

Subgraph/supergraph containment query aims to find graphs that contain the query graph or are contained by the query graph in a graph database. There are a lot of algorithms and indexing techniques proposed for subgraph query. Most of them use a mining based approach, e.g., gIndex [24], FG-Index [7], Tree+Δ [27], SwiftIndex [16], TreePi [25], etc. Others use non-mining based approach, e.g., GraphGrep [18], C-Tree [9], gCode [28], and GDIndex [22]. An extensive comparison and evaluation of these algorithms can be found in iGraph [8]. There have been several recent works that focus on answering supergraph query, e.g., CIndex [5], GPTree [26], and IG [6].

Subgraph/supergraph similarity query is a variant of the subgraph/ supergraph containment query that allows the missing of edges or nodes in the query result. Several approaches, such as C-Tree [9], GDIndex [22], Grafil [23], and Grafil+ [15] are proposed for subgraph similarity search. A recent work [17] has been proposed to solve the supergraph similarity search problem. Among them, C-Tree [9] measures the similarity using a heuristic graph mapping method and does not support exact *MCS* based similarity query. [22] aims to find subgraphs with the minimum number of vertex and edge mismatches bounded by a given threshold. The similarity measures of [15], [17], and [23] are based on the *MCS* of the query and the database graph. They focus on a threshold based subgraph/supergraph similarity query which can not be used to answer the top-k graph similarity query studied in this paper.

Full graph similarity query aims to find graphs that are similar to the query graph. There are two categories based on different similarity measures. The first category is to use the feature-based measure, in which domain-specific elementary structures are first extracted as features and the similarity of two graphs is measured by the number of common features they have. A lot of feature extraction methods have been developed in [24] [7] [27] [16] [25] [23] and techniques of feature based similarity search are summarized in [21]. The second category is to use the kernel-based measure where similarity is defined using a graph kernel function based on walks [11], cyclic patterns [10] or h-hop neighbors [20]. Both the feature-based approach and kernel-based approach only provide a very rough measure on structure similarity since they lose the global structural connectivity of two graphs.

Algorithms for computing the maximum common subgraph (*MCS*) of two graphs have been extensively studied. A lot of related work on maximum common subgraph can be found in [13] [2] [3] [12] [19] [1] [4] and [14]. Our work is different from these approaches, because we aim at reducing the number of *MCS* computations rather than making an *MCS* computation faster.

8. CONCLUSION

In this paper, we study to find top-k similar graphs for a query graph in graph databases. Existing solutions are all threshold based subgraph or supergraph similarity search, which cannot be used to solve our problem properly. We introduce a new graph distance measure using the maximum common subgraph (*MCS*), which are more accurate than the feature based measures by considering the global structures. In order to reduce the number of *MCS* computations, which is NP-hard, we propose two distance lower bounds with different tightness and computational costs to be used in pruning unqualified candidates. We further introduce a triangle property to lower bound the graph distance. We show that the triangle property can be efficiently and effectively used in pruning with the help of an index. We design three variants of the index with different trade-offs between construction cost and pruning power. We conducted extensive performance studies using a real dataset to test our approaches.

Acknowledgments The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China No. 419008, 419109 and 411211.

9. REFERENCES

[1] F. N. Abu-Khzam, N. F. Samatova, M. A. Rizk, and M. A. Langston. The maximum common subgraph problem: Faster solutions via vertex cover. In *AICCSA*, pages 367–373, 2007.

[2] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. *SIAM J. Comput.*, 15(4):1054–1068, 1986.

[3] I. M. Bomze, M. Budinich, M. Pelillo, and C. Rossi. Annealed replication: a new heuristic for the maximum clique problem. *Discrete Applied Mathematics*, 121(1-3):27–49, 2002.

[4] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3-4):255–259, 1998.

[5] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *VLDB*, pages 926–937, 2007.

[6] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *The VLDB Journal*, pages 1–19, 2010.

[7] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[8] W. Han, J. Lee, M. Pham, and J. Yu. igraph: a framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

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

[10] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In *KDD*, pages 158–167, 2004.

[11] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *ICML*, pages 321–328, 2003.

[12] E. B. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Softw., Pract. Exper.*, 34(6):591–607, 2004.

[13] J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Softw., Pract. Exper.*, 12(1):23–34, 1982.

[14] J. Raymond, E. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631, 2002.

[15] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, pages 903–914, 2010.

[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[17] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 637–648, 2010.

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

[19] W. Suters, F. Abu-Khzam, Y. Zhang, C. Symons, N. Samatova, and M. Langston. A new approach and faster exact methods for the maximum common subgraph problem. *Computing and combinatorics*, pages 717–727, 2005.

[20] X. Wang, A. M. Smalter, J. Huan, and G. H. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, pages 472–480, 2009.

[21] P. Willett, J. Barnard, and G. Downs. Chemical similarity searching. *Journal of Chemical Information and Computer Sciences*, 38(6):983–996, 1998.

[22] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985, 2007.

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

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

[25] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975, 2007.

[26] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

[27] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta $>=$ graph. In *VLDB*, pages 938–949, 2007.

[28] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.