World Wide Web
DOI 10.1007/s11280-014-0274-4

Efficient processing of graph similarity search

Ryan Choi · Chin-Wan Chung

Received: 17 January 2013 / Revised: 22 December 2013 / Accepted: 3 January 2014

Abstract A graph similarity search is to find a set of graphs from a graph database that are similar to a given query graph. Existing works solve this problem by first defining a similarity measure between two graphs, and then presenting a filtering mechanism that reduces the number of candidate graphs. The candidate graphs are then verified by performing expensive graph search operations such as finding maximum common subgraphs. Existing works, however, do not report some similar graphs from a graph database while dissimilar graphs are not discarded during the filtering phase. To overcome this problem, in this paper, we first present a graph distance measure that can identify hidden but similar graphs that could not be discovered by previous graph distance measures. We then devise a series of filtering and validation rules to discard and identify non-matching and definitely-matching graphs, respectively, by calculating lower and upper bounds of the distance between a query and a data graph. To execute these filtering and validation rules efficiently during runtime, an index structure is also proposed. Lastly, a verification algorithm that verifies candidate graphs according to our graph distance measure is presented. Experiments on real datasets show that our approach can efficiently and effectively perform graph similarity search by significantly reducing the number of candidate graphs that must be verified, and by returning similar graphs.

Keywords Graph · Graph similarity search · Graph database · Algorithms

1 Introduction

Graphs are widely used to model complex structured data in many advanced applications such as bioinformatics [12], image processing [17], social networks [28], etc. In these applications, graph queries are performed to discover new domain knowledge, i.e., new graph

R. Choi · C.-W. Chung (✉)
Department of Computer Science, KAIST, Daejeon, Korea
e-mail: chungcw@kaist.edu

R. Choi
e-mail: rchoi@islab.kaist.ac.kr

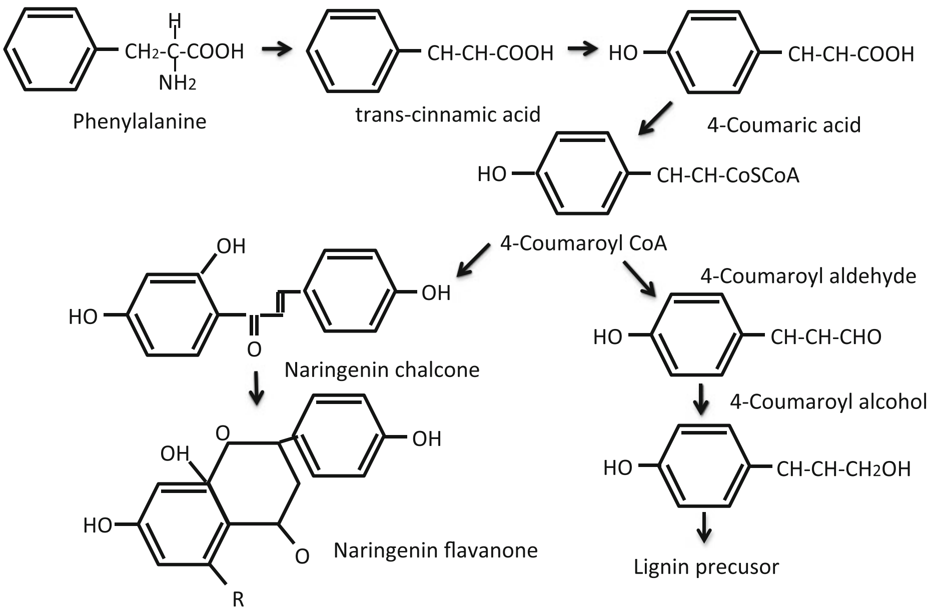


Figure 1 Oxidation of lignans

patterns are retrieved and analyzed further to discover new relationships between primitive components, or given a set of desired features, graph patterns that contain those features can be retrieved. For example, in an image processing system, an image can be represented by a graph constructed from a set of primitive objects found in the image, and relationships between two objects are represented by edges/paths [17]. Then, the images that have the same features as a given image can be retrieved by comparing the graph structure of each image in the database against the graph structure of the given image. Another example is, in social networks, a community can be modeled by a graph, and the communities that share the same characteristics can be found by checking whether desired characteristics represented by graphs exist in the graph representations of the communities [28].

We have a close look at how graph similarity search is used in biochemistry. Chemical compounds can be well modeled by graphs. In biochemistry and metabolic engineering, graph similarity search is frequently used to find a family of a known/unknown biochemical compound. Figure 1 shows an oxidation pathway of a chemical compound called lignans. By oxidation, a chemical compound X is transformed to X' by adding/removing organic atoms. In addition, there are certain places in a chemical compound where these addition and removal of atoms can take place. For example, in Figure 1, two edges with O can be added to the middle of compound B to be transformed to compound C. Furthermore, all chemical compounds in Figure 1 are in the same family, because there is a main chemical structure that is commonly shared by all these compounds. In this example, it is the two hexagonal carbon shapes that are found at the end of each compound. Moreover, by a metabolic process, a chemical compound can be transformed to a compound that is in a different family. Figure 2 shows a metabolic pathway from phenylalanine to two chemical compounds, flavanone and lignans, each of which is in a different family. Finding a metabolic pathway is important, because not all chemical compounds are natural products,

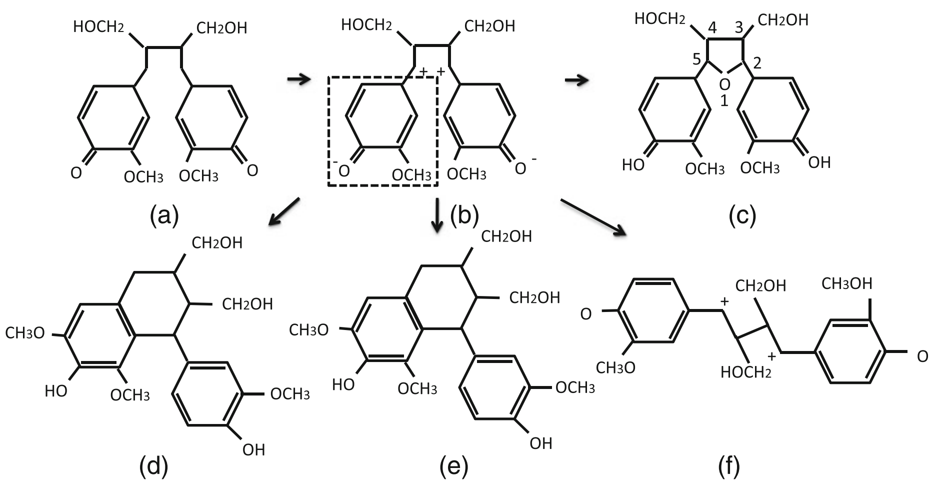


Figure 2 Metabolic pathway

i.e., some compounds must be obtained by sequentially biosynthesizing a natural product with atoms and enzymes. A metabolic pathway documents how next biosynthesized products are derived from a natural product. Conversely, metabolic pathway is also used to find a precursor of a known/unknown compound. In this metabolic engineering, the following are frequent queries.

- Q1: Given an unknown chemical compound X, find a list of possible biosynthesized chemical compounds from: (1) a database of known chemical compounds; or (2) generate a list of possible chemical compounds not found in the database.
- Q2: Given an unknown chemical compound X, find a list of possible precursors from: (1) a database of known chemical compounds; or (2) generate a list of possible chemical compounds not found in the database.

Biochemistry researchers benefit from a system if it automatically searches/generates a list of possible biosynthesized/precursors of unknown chemical compounds, as it reduces the number of candidate compounds to validate.

In the above applications, one of the most important operations that these applications must support is an efficient and effective graph search operation. A traditional graph search operation that looks for graphs that are structurally the same as a given query graph does not work well in practice. This is because the connectivity constraints between nodes are too restrictive in a sense that too few matches are returned. For example, in an image processing application, it is easy to see that, there may not be many images whose graph structures are exactly the same as the given image. Another problem is that, graph data in real-world contain noises that prevent us from performing exact-match graph search operations. To overcome these problems, the applications need to relax a query graph such that, the graphs that are similar to the given query graph are retrieved. This is called a graph similarity search.

There are many previous works in the area of graph similarity search, and they can be classified into two broad categories. The first category of a graph similarity search is called a subgraph containment search. That is, given a query graph q, and a graph database $D = \{g_1, ..., g_n\}$, we return the set of graphs that contain q as a subgraph, i.e., $D' =$

$\{g \mid q \subseteq g, g \in D\}$. Checking $q \subseteq g$ for each $g \in D$ is difficult since a graph isomorphism test between q and g must be performed, which is known to be NP-complete [8]. To address this problem, previous works [3, 4, 22, 24, 31, 34, 36] adopt a filtering-and-verification process, where definitely non-matching graphs are discarded, and graph isomorphism tests are performed on a smaller number of candidate graphs to verify whether q is a subgraph in g. The second category of a graph similarity search is called a subgraph isomorphism similarity search. Unlike the subgraph containment search, given q, we return the set of graphs, each of which contains a (maximum) subgraph of q. The similarity between a subgraph of q and a subgraph of g is measured by a distance function, $dist(q, g)$, and a subgraph isomorphism similarity search is defined as $D' = \{g \mid dist(q, g) \leq \sigma, g \in D\}$. Previous works [21, 32, 33] define various graph distance measures. In these measures, q and g are considered similar if the distance between q and g is within a certain threshold, i.e., $dist(q, g) \leq \sigma$. Our work belongs to this category, and we propose a meaningful graph distance measure, and efficient techniques on finding similar graphs.

To measure the distance between q and g, two types of distance measures are proposed in the literature. The first one is based on a graph edit distance [10]. This measure calculates the number of edges required to add/remove to transform q to g. However, this measure often cannot consider how a node is connected to other nodes, i.e., connectivity between nodes. The second one is based on a maximum common subgraph (MCS) [19, 32] between q and g, or MCS's variations. Grafil [32] is based on the MCS-based distance measure, and it is designed to address the problems occurred from not capturing the global graph structure. However, there are cases where Grafil still returns less meaningful answers. This is because some edges in a graph are aggressively relaxed, it looses global structural information. To address this problem, GrafD-index [21] proposes a more restrictive form of the MCS-based distance measure based on the maximum connected common subgraph (MCCS). The intuition is to find a largest group of connected nodes in q which is isomorphic to a subgraph in g, and then compare the size of such a group with the size of q. That way, GrafD-index returns a list of graphs, each of which contains the biggest isomorphic subgraph between q and g.

While GrafD-index provides the state-of-the-art graph similarity distance measure, it may miss some graphs that are similar to q. GrafD-index finds a maximal common connected subgraph (MCCS) of X and Y, and calculates the number of edges that are not included in X. If the number of missing edges is greater than σ, X is reported as not similar. For example, consider the chemical compound B and C shown in Figure 1. MCCS of B and C is denoted by the subgraph in dotted line in Figure 1. Assume that the number of edges that can be relaxed is $\sigma = 2$. GrafD-index reports B and C are not similar, because the distance between B and C is $|E(B)| - |E(mccs(B, C))| = 23 - 6 = 17 > \sigma$, where $|E(X)|$ is the number of edges in X and $|E(mccs(X, Y))|$ is the number of edges in the MCCS of X and Y. It is, however, clear that compound B and C share the same main chemical structure, and hence they are related, i.e., C is an oxidation product of B. In fact, the only difference is, C contains two additional edges, (5, 1) and (1, 2). It is interesting to note that, Grafil reports g_1 and g_2 as matching graphs, while GrafD-index reports non of the graphs as matching graphs. Another problem of GrafD-index is that, simply retrieving all graphs $g \in D$ with $dist(q, g) \leq \sigma$ can return graphs whose $dist(g, q) > \sigma$. This problem arises from the asymmetric characteristic of the distance measure, i.e., $dist(q, g) \neq dist(g, q)$, unless $|E(q)| = |E(g)|$. For example, consider the graphs in Figure 3. An MCCS of q_2 and g_3 is marked by the dotted line. Given $\sigma = 3$, GrafD-index reports that q_2 is similar to g_3, as $dist(q_2, g_3) = 5 - 2 = 3 \leq \sigma$. However, g_3 is not similar to q_2 as $dist(g_3, q_2) = 8 - 2 = 6 > \sigma$.

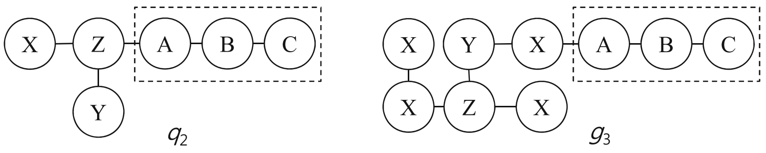


Figure 3 A query and a data graph, q_2 and g_3

Although these previous works [21, 32] proposed how to compare two graphs by meaningfully measuring the distance between two graphs, as we discussed above, there are still problems with these approaches. Motivated by the problems above, in this paper, we propose a graph similarity distance measure based on the concept of multiple maximal common subgraphs (MMCS). The motivation behind our approach is that, two graphs are similar if they share as many structurally similar nodes as possible. A group of nodes in a graph g_1 is structurally similar to the nodes in a graph g_2 if this group of nodes in g_1 are isomorphic to some nodes in g_2. When measuring commonality between two graphs, MMCS-based approach considers multiple groups of structurally similar nodes that are commonly found between two graphs. In addition, we take into account how these groups of nodes are connected, how many edges are miss-matched in q, as well as by how many edges, in total, these groups of nodes are separated. For example, given q_1, g_1, $\sigma = 2$, and a distance relaxation parameter $k = 2$, our similarity distance measure reports g_1 as q_1's similar graph. This is because there exists two groups of structurally similar nodes, namely B' and C' (note: these are the carbon hexagons at the two ends), and although these two groups of nodes are separated, they are not separated by too far. the path between these two groups of nodes can be relaxed in a way that k edges can be added/removed to connect these two groups of nodes. In this example, the path between the two nodes are the same except two edges (5,1) and (1,2) are added. The maximum number of additional edges that connect B' and C' are specified by k. Note that, while in this example, the two additional edges do not change the path between the two groups of nodes, the path could also be changed by addition/removal of edges up to k. This way, our distance measure allows us to report two graphs as similar if two graphs contain as many similarly structured nodes as possible, and the groups of these nodes are separated by not too many edges, i.e., the edge connectivities between these groups of similarly structured nodes are also similar. This distance measure also means that two graphs will contain a small number of non-matching edges, but not all graphs that are reported as similar by previous distance measures, such as graph edit distance, are included, as we also consider the overall graph structures. Furthermore, our approach allows us to discard ambiguous matching graphs such as g_3 for q_2 by checking $dist(q, g) \leq \sigma$ and $dist(g, q) \leq \sigma$. Hence, if the number of non-matching edges between g_1 and g_2 is $\leq \sigma$, then the number of non-matching edges between g_2 and g_1 must also be $\leq \sigma$ to report g_1 as similar to g_2.

In this paper, we devise a technique that can efficiently perform MMCS-based graph similarity search. We investigate relationships between a query q, a data graph g, and a set of subgraphs F_g that are commonly contained by q and g, and formally establish the lower and upper bounds on the distance between q and g by measuring the distances between q and F_g, and F_g and g. Based on these lower and upper bounds, we develop filtering and validation rules that allow us to discard and retrieve non-matching and definitely-matching data graphs, respectively, without running verification algorithm. We then develop a verification algorithm that efficiently verify similarity conditions by utilizing node connectivity and avoiding duplicate computation while g is traversed. Finally, an index structure, called feature-index, is proposed to index groups of data graphs according to the distances from sets of frequent subgraphs discovered over a graph database. This feature-index is used to efficiently perform filtering and validation rules.

The contributions of this paper are summarized as follows.

- We propose a graph similarity search based on the concept of MMCS. By considering multiple common subgraphs between q and g, we can effectively retrieve previously undiscovered similar graphs, and yet discard dissimilar graphs reported by many previous works.
- We formally establish relationships between q, g, and F_g, and establish lower and upper bounds on the distance between q and g. Based on these bounds, we propose filtering and validation rules to identify non-matching and definitely-matching data graphs. Then, we propose an efficient MMCS-based verification algorithm that verifies whether q and g satisfy the similarity condition. Lastly, we propose a feature-index to efficiently perform our filtering and validation rules.
- Extensive experimental studies are conducted to evaluate the efficiency and effectiveness of our filtering and validation rules, and verification algorithm.

Organization Section 2 presents related works in the area of graph similarity search. Section 3 formally defines the problem. A series of filtering and validation rules are proposed in Section 4. Section 5 devises a verification algorithm. Section 6 proposes a feature-index to process graph similarity search efficiently. Section 7 presents the overall graph similarity search framework. Section 8 presents experimental results, and we conclude the paper in Section 9.

2 Related work

We classify existing graph (similarity) search techniques into several categories according to their characteristics.

The first group of techniques calculate a common subgraph that exists between two graphs by calculating a maximum clique. Raymond et al. [19] calculate the graph similarity distance by finding a maximum common edge subgraph (MCES), and an MCES is found by finding a maximum clique on a modular product of the line graphs created from two graphs to compare. Raymond & Willett [20] reviewed MCS/MCES algorithms that are based on finding maximum cliques. Koch [15] adapts Bron & Kerbosch [1] algorithm to find all maximal cliques for various types of datasets. Cuissart & Hebrard [5] improve Koch's [15] algorithm by avoiding creating a compatibility graph from two graphs. Their algorithm creates a tree structure for a graph, and performs a matching process while it traverses the tree. As finding cliques involves much more computation than required by graph search, these works are not suitable.

The second group of techniques perform a graph similarity search by finding all graphs in a graph database that contains a query graph as a subgraph, i.e., subgraph containment query, $q \subseteq g$. GraphGrep [24] is a path-based technique in a way that, it enumerates all existing paths in the graphs in a graph database up to a maximum length, and each graph is indexed with the paths found in the graph. The index size, however, increases dramatically as the size of indexed path set increases. gIndex [31] introduces the concept of discriminative features, and creates an index based on frequent and discriminative subgraphs. The graphs that contain all the frequent features found in a query graph are first found, and then subgraph isomorphism tests are run to verify these graphs. TreePi [34] uses frequent and discriminative subtrees as indexing features to simplify index processing compared to when processing graph-based features. Tree+Δ [36] uses frequent tree-features and discriminative subgraphs as indexing features, and FG-index [3, 4] uses frequent subgraphs and edges as indexing features. QuickSI [22] proposes a fast subgraph isomorphism verification algorithm by improving Ullmann's [26] subgraph isomorphism algorithm to preserve the node connectivity constraints during the candidate node matching phase, and presents a feature-based index to support subgraph containment queries. Since all techniques in this group process subgraph containment queries, they are not suitable for answering subgraph similarity search queries.

The third group of techniques support graph similarity search by using graph edit distance. Similarity search is supported by finding a subgraph or a set of subgraphs that commonly exist between two graphs. Closure-Tree [10] creates a generalized graph to represent a set of graphs by combining (i.e., union) graphs that share many common vertices and edges. These graphs are then organized in a tree similar to the tree structures used to process spatial queries (e.g., R-trees). When a similarity search is performed, approximate edit distance is calculated. GDIndex [29] decomposes a graph into a set of subgraphs, and these subgraphs are represented by a DAG structure. Similarity search is performed by finding the common subgraph that gives the minimum distance between a query and the subgraph. SIGMA [18] checks whether a graph g can be covered by all the features that a graph q contains, and by examining missing features, it calculates the number of edges to delete. SEGOS [27] split graphs into smaller units and indexes the graph with a two-level inverted index, and compare these sets of units to calculate the distance between two graphs. Since the above works use an (approximate) graph edit distance to measure the distance between two graphs, less meaningful graphs are also returned when similarity search is performed.

The fourth group of techniques support a graph similarity search by relaxing edges in a query graph, and they are the most related to our work. Grafil [32] builds a feature graph matrix, which stores the number of subgraph isomorphic mappings for each frequent feature found in the data graphs, and an edge feature matrix is built on the query graph to calculate the number of edges that can be mapped to each frequent feature. Using this edge feature matrix, the maximum number of matching features when σ edges from the query are removed, is calculated. Then, the graphs that misses up to d number of features become candidate graphs. GrafD-index [21] proposes a more meaningful graph similarity search distance measure based on the concept of a maximum common connected subgraph (MCCS), and presents effective pruning and validation rules based on the triangular inequality relationship between a query, feature, and a data graph. While GrafD-index is the state-of-the-art technique, as explained earlier, there are undiscovered graphs that are similar to query graphs, and dissimilar graphs are also returned. Fan et al. [7] propose approximate graph similarity search algorithms based on the idea of matching an edge to a path. Unlike Grafil and GrafD-index, two graphs are reported as similar if the root-to-leaf paths in one graph is similar to the root-to-leaf paths in another graph. However, since this approach is path-based, it is difficult to capture global structural information. PRAGUE [13] is a recent work that uses a MCCS-based distance measure. It focuses on how to interactively query similar graphs using graphical user interface.

Lastly, there are some works that do not fall into the above categories. cIndex [2] and Shang et al. [23] perform supergraph containment queries, i.e., $q \supseteq g$. GString [12] is designed for finding similar chemical compounds. Zeng et al. [33] propose an approximate graph similarity search. Tong et al. [25] find subgraphs in a large graph that match a user query pattern. Gaddi [35] indexes a large graph with frequent substructures, and finds all subgraphs that are isomorphic to q. Khan et al. [14] find top-k approximate matches in a large graph. Fan et al. [6] provide incremental graph pattern matching algorithm for dynamic graphs. Turbo [9] proposes a subgraph isomorphism algorithm for graph containment queries. It identifies candidate regions in g where the regions may contain q, and performs subgraph isomorphism search in the order obtained from a BFS tree from q. Zhu et al. [37] proposes an approximate graph matching algorithm for finding a common subgraph between two large graphs. The quality of the common subgraph is evaluated by a scoring function, which counts the number of edges of the common subgraph. Finding the optimal common subgraph is the same as finding a maximum common subgraph between the two graphs. Finally, Lee et al. [16] empirically compared the performance of recent subgraph isomorphism algorithms, and reports that graph searching order hugely affects the number of candidate sets, which impacts the overall performance. Since all the above works are designed to answer other types of graph search problems (i.e., graph containment queries, approximate graph queries, etc), it is difficult to extend them for graph similarity search.

3 Problem statement

In this paper, a graph similarity search is performed on undirected graphs with labeled vertices. Let us represent an undirected graph $G = (V, E, \lambda)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $\lambda : V \rightarrow W$ is a vertex labeling function, where W is a finite set of labels. The label of a vertex v is denoted as $\lambda(v)$. Let us denote the set of vertices and edges of a graph g by $V(g)$ and $E(g)$, respectively. Also, the number of vertices and edges in g are denoted as $|V(g)|$ and $|E(g)|$, respectively. Furthermore, a path, $\langle v_1, ..., v_n \rangle$ where $(v_i, v_{i+1}) \in E(g)$ for $1 \leq i < n$, in a graph is denoted as $v_1 \rightsquigarrow v_n$, and the length of a path, $|v_1 \rightsquigarrow v_n|$, is defined as the number of edges in the path.

Definition 1 (Subgraph isomorphism) Given two graphs g' and g, g' is a subgraph of g if there exists an injective function $f : V(g') \rightarrow V(g)$ that satisfies the following conditions: (1) $\forall v \in V(g'), f(v) \in V(g)$ such that $\lambda'(v) = \lambda(f(v))$; and (2) $\forall (u, v) \in E(g'), (f(u), f(v)) \in E(g)$.

There may exists multiple injective functions between two graphs. Given two graphs g' and g, if there exists an injective function f, let us represent the relationship between g' and g as $g' \subseteq_f g$, and denote g' as a subgraph of g, and g as a supergraph of g'. Moreover, let us omit the injective function f to simplify $g' \subseteq_f g$ to $g' \subseteq g$ when the context is clear.

Definition 2 (Maximal common subgraph) Given two graphs g_1 and g_2, a subgraph $g_1' \subseteq g_1$ is a maximal common subgraph (MCS$'$) of g_1 and g_2, if g_1' is subgraph isomorphic to g_2, and $g_1' \not\subset g_1''$, where g_1'' is a supergraph of g_1' that is subgraph isomorphic to g_2. MCS$'$ of g_1 and g_2 is denoted as $mcs'(g_1, g_2)$.

Definition 3 (Spath) Given g_1, g_2, $(u, v) \in E(g_1)$, and subgraph isomorphism functions $f_1 : V(g_1) \rightarrow V(g_2)$ and $f_2 : V(g_1) \rightarrow V(g_2)$, a path $f_1(u) \rightsquigarrow f_2(v)$ in g_2 is a spath of the edge $(u, v) \in E(g_1)$, if $\lambda_1(u) = \lambda_2(f_1(u))$ and $\lambda_1(v) = \lambda_2(f_2(v))$. The length of an spath is the number of edges between $f_1(u)$ and $f_2(v)$.

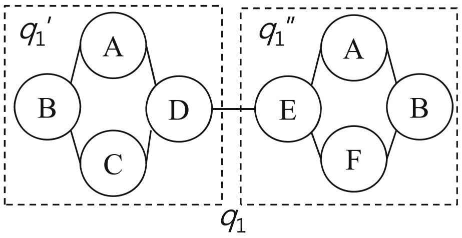


Figure 4 A query graph, q_1

Definition 4 (Multiple maximal common subgraphs) Given g_1, g_2, a set of subgraph isomorphism functions, $f_i : V(g_1) \rightarrow V(g_2)$, and a distance relaxation r, multiple maximal common subgraphs (MMCS) of g_1 and g_2, denoted as $mmcs(g_1, g_2)$, is a set of MCS′s of g_1 and g_2, i.e., $mmcs(g_1, g_2) = \bigcup_{i=1}^{n}\{mcs_i(g_1, g_2)\}$, that satisfies the following conditions: (1) $\bigcap_{i=1}^{n} E(mcs_i'(g_1, g_2)) = \emptyset$; and (2) the maximum length of each spath that connects $mcs_i(g_1, g_2)$ and $mcs_j(g_1, g_2)$ is $\leq r$, where $1 \leq i, j \leq n$ and $i \neq j$.

Example 1 Consider q_1 and g_1 shown in Figures 4 and 5, respectively. Given $r = 2$, q_1' and q_1'' are $mmcs(q_1, g_1)$, since the matching MCS′ of q_1' and q_1'', namely g_1' and g_1'', are connected by a path $\langle D, G, E\rangle$, which is $(D, E) \in E(q_1)$'s spath, and the length of this spath is 2. Let us consider q_1 and g_2 in Figures 4 and 5, respectively. Given $r = 2$, $mmcs(q_1, g_1)$ is either g_2' or g_2'', as the length of (D, E)'s spath, $\langle D, G, E\rangle$, in g_2 is > 2.

Definition 3 and 4 establish the properties of common subgraphs between two graphs. Having defined MMCS, our graph similarity distance measure is defined as follows.

Definition 5 (Graph similarity distance measure) Given two graphs g_1 and g_2, the distance between g_1 and g_2 is defined as:

$$dist(g_1, g_2) = |E(g_1)| - |E(mmcs(g_1, g_2))| \tag{1}$$

$$dist(g_2, g_1) = |E(g_2)| - |E(g_1)| + dist(g_1, g_2) \tag{2}$$

where $E(mmcs(g_1, g_2)) = \bigcup_{i=1}^{n} E(mcs_i'(g_1, g_2))$.

The distance measure is not symmetric in a sense that, $dist(g_1, g_2)$ is not always the same as $dist(g_2, g_1)$. Given $dist(g_1, g_2)$, $dist(g_2, g_1)$ is defined as $dist(g_2, g_1) = |E(g_2)| - |E(mmcs(g_1, g_2))|$, and $dist(g_2, g_1)$ can be rewritten in terms of $dist(g_1, g_2)$, and this is shown in (2).

Lastly, MMCS-based graph similarity search is defined as follows.

Definition 6 (MMCS-based graph similarity search) Given a query graph q, a set of graphs $D = \{g_1, ..., g_n\}$, an edge relaxation σ and a distance relaxation k, a MMCS-based

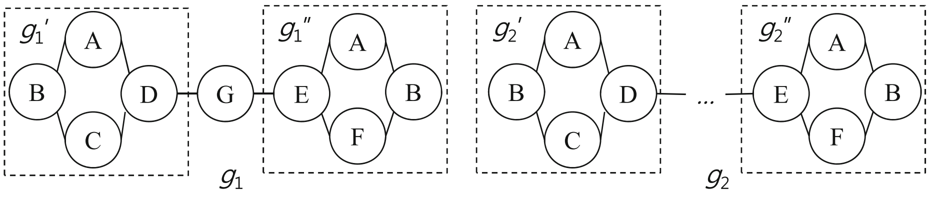


Figure 5 Data graphs, g_1 and g_2

graph similarity search is to find the subset of graphs $D' = \{g \mid g \in D, dist(q, g) \leq \sigma, dist(g, q) \leq \sigma,$ and the total lengths of all spaths are $\leq k\}$.

An edge relaxation σ specifies the number of edges that may be missed in q when MMCS-based graph similarity search is performed. A distance relaxation k specifies the total number of edges in g that are used to separate MCSs in g. The value of k is determined by $k = \alpha|E(q)| - |E(q)|$, where $\alpha \geq 1$, to decide the number of edges in g to use to relax non-matching edges in q, e.g., if the size of a matching graph in g can be up to 40 % larger than the size of q by including the spaths in g, then $k = 1.4 \times 10 - 10 = 4$. Definition 6 shows that two graphs are similar if the size of two graphs to compare are not too much different. The difference in size is controlled by σ and k.

Example 2 Consider q_1 and g_1 shown in Figures 4 and 5, respectively, and let $\sigma = 2$ and $k = 2$. q_1 and g_1 are similar, as $dist(q_1, g_1) = 9 - 8 = 1 \leq \sigma$, $dist(g_1, q_1) = 10 - 8 = 2 \leq \sigma$, and $|D \rightsquigarrow E| \leq k$. In addition, q_1 and g_2 are not similar, as $dist(q_1, g_2) = 9 - 4 = 5 > \sigma$ and $|D \rightsquigarrow E| > k$. Lastly, given $\sigma = 3$, q_2 and g_3 are not similar, as $dist(q_2, g_3) = 5 - 2 \leq \sigma$, but $dist(g_3, q_2) = 8 - 2 > \sigma$.

In biochemistry and metabolic engineering, the distance between two components is important, as two compounds, each with different lengths between two components could represent two very different chemical compounds. Furthermore, we also need to consider the overall structure of chemical compound by considering how components in a chemical compound are nearby. To restrict how far each component can be separated overall, summation of spaths are used. However, there are certain applications that measuring the minimum distance between MCS is useful. Further investigation on considering the use of minimal distance is left as future work.

Problem statement In this paper, given a query graph and a set of graphs, we propose a technique that can efficiently perform MMCS-based subgraph similarity search.

4 Measuring graph similarity

In this section, we present how maximum and minimum distances between a query q and a data graph $g \in D$ can be defined by calculating the distances between q and feature graphs, and g and feature graphs. Then, filtering and validation rules are established to remove non-matching and identify definitely-matching graphs, respectively.

4.1 Estimating graph similarity distance

Definition 7 (Feature graph) Given a graph database $D = \{g_1, ..., g_n\}$, let $D_f = \{g \mid g \in D, f \subseteq g\}$. A subgraph f is a feature graph in D if $\frac{|D_f|}{|D|} \geq k$, where k is a feature selection factor whose range is $0 \leq k \leq 1$.

Definition 8 (Maximum common subgraph) A maximum common subgraph (MCS) of g_1 and g_2 is a largest $mcs'(g_1, g_2)$, and is denoted as $mcs(g_1, g_2)$.

Let $F_g = \{f_1, ..., f_n\}$ be a set of feature graphs contained by g, i.e., $\forall f \in F_g$, $f \subseteq g$, and $E(F_g)$ be a set of edges from all feature graphs in F_g. The maximum common subgraphs

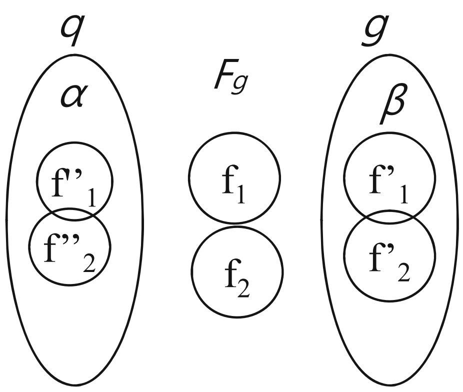


Figure 6 A relationship between q, g and F_g

between g and F_g is defined as $mcs(g, F_g) = \bigcup_{i=1}^{|F_g|} mcs(g, f_i)$. Theorem 1 establishes the bounds on the maximum and minimum distances between q and g.

Theorem 1 *Given a query graph q, a data graph $g \in D$, and a set of feature graphs F_g, the distance between q and g is bounded by $|E(\alpha)| - |E(\beta)| \leq dist(q, g) \leq |E(\alpha)|$, where $\alpha = E(q) \setminus E(mcs(q, F_g))$, $\beta = E(g) \setminus E(mcs(g, F_g))$.*

Proof We establish a relationship between q and g as follows. There may exist subgraph isomorphism mappings between α and β. $dist(q, g)$ is minimum, if all edges in α are subgraph isomorphic to some edges in β, and $dist(q, g)$ is maximum, if non of the edges in α are subgraph isomorphic to the edges in β. Hence, the minimum set of non-matching edges in q are bounded by $\alpha \setminus \beta$, and the maximum set of non-matching edges in q are bounded by α. Therefore, $min(dist(q, g)) = |E(\alpha \setminus \beta)| = |E(\alpha)| - |E(\beta)|$, and $max(dist(q, g)) = |E(\alpha)|$, and they can be rewritten as $|E(\alpha)| - |E(\beta)| \leq dist(q, g) \leq |E(\alpha)|$. □

Example 3 Figure 6 shows a relationship between q, g and F_g. F_g contains two feature graphs, $\{f_1, f_2\}$. $E(mcs(g, F_g))$ is represented by $E(f_1') \cup E(f_2')$ in g, and β represents the edges in $E(g) \setminus E(mcs(g, F_g))$. $E(mcs(q, F_g))$ is represented by $E(f_1'') \cup E(f_2'')$ in q, and α represents the edges in $E(q) \setminus E(mcs(q, F_g))$. $dist(q, g)$ is minimum when all edges in β has matching edges in α, i.e., there are $|E(\alpha)| - |E(\beta)|$ non-matching edges in q, and $dist(q, g)$ is maximum when non of the edges in β has matching edges in α, i.e., there are $|E(\alpha)|$ non-matching edges in q.

The size of α and β are also affected by the number of common edges between the feature graphs in F_g. Since $mcs(g, F_g) = \bigcup_{i=1}^{|F_g|} mcs(g, f_i)$, the size of β becomes minimum when there are no common edges between feature graphs, i.e., $|E(\bigcup_{i=1}^{|F_g|} f_i)| = \sum_{i=1}^{|F_g|} |E(f_i)|$. Similarly, the size of β becomes maximum when the feature graphs contain the edges that are also found in other feature graphs, i.e., $f_k = \bigcup_{i=1}^{|F_g|} f_i$, where f_k is a biggest feature graph in F_g. Consequently, the size of α is increased (resp. decreased) when the size of β increases (resp. decreases), as the common edges between q and g are decreased (resp. increased).

4.2 Filtering and validation rules

Having established the upper and lower bounds on the distance between q and g, we now derive filtering and validation rules (FR and VR, respectively) based on Theorem 1. Given D, FR and VR are used to discard non-matching and retrieve definitely-matching graphs in D, respectively. Given an edge relaxation σ, FR and VR are defined as follows.

Filtering Rule 1: $dist(q, g) > \sigma$, if $\exists f \in F_g$ such that, $|E(f)| - |E(mcs(f, q))| > \sigma$.

Proof The number of non-matching edges in a feature graph is already greater than σ. □

Filtering Rule 2: $dist(q, g) > \sigma$ if $\gamma > \sigma$, where $\gamma = E(\bigcup_{i=1}^{|F_g|} f_i) \setminus E(\bigcup_{i=1}^{|F_g|} mcs(f_i, q))$.

Proof Note that, $\forall f \in F_g$, $f \subseteq g$, but $mcs(f, q) \subseteq f$. This is because there may exist edges in f that are missing in $mcs(f, q)$. Since all feature graphs in F_g are isomorphic to g, and the total number of non-matching edges in q are already greater than σ, $dist(q, g) > \sigma$. □

Filtering Rule 3: $dist(q, g) > \sigma$ if $|E(\alpha)| - |E(\beta)| > \sigma$.

Proof $dist(q, g)$ is lower bounded by $|E(\alpha)| - |E(\beta)|$. Hence, if $|E(\alpha)| - |E(\beta)| > \sigma$, $dist(q, g) > \sigma$. □

Validation Rule: $dist(q, g) \leq \sigma$ if $|E(\alpha)| \leq \sigma$, and the length of spaths are $\leq k$.

Proof $dist(q, g)$ is upper bounded by $|E(\alpha)|$. Hence, if $|E(\alpha)| \leq \sigma$ and length of spaths are $\leq k$, $dist(q, g) \leq \sigma$. □

Applying similar ideas from FR 3 and VR, additional rules can be derived for checking $dist(g, q) \leq \sigma$. In the following rules, $E(mcs(g, F_g)) \setminus E(mcs(q, F_g))$ represents the edges in g that do not have matching edges in q.

Filtering Rule 3′: $dist(g, q) > \sigma$ if $|E(mcs(g, F_g))| - |E(mcs(q, F_g))| + |E(\beta)| - |E(\alpha)| > \sigma$

Validation Rule′: $dist(g, q) \leq \sigma$ if $|E(mcs(g, F_g))| - |E(mcs(q, F_g))| + |E(\beta)| \leq \sigma$

Proof $|E(\beta)| - |E(\alpha)|$ represents the minimum number of non-matching edges in g. Hence, if $|E(mcs(g, F_g))| - |E(mcs(q, F_g))| + |E(\beta)| - |E(\alpha)| > \sigma$, $dist(g, q) > \sigma$. Also, $E(\beta)$ are the edges with unknown matching status. But if $|E(mcs(g, F_g))| - |E(mcs(q, F_g))| + |E(\beta)| \leq \sigma$, $dist(g, q) \leq \sigma$, regardless of the matching status. □

Note that, additional rules for FR 1 and 2 are not needed for checking $dist(g, q) \leq \sigma$, because all feature graphs in F_g are already isomorphic to g.

5 Verification algorithm

In this section, we present a subgraph isomorphism verification algorithm that verifies whether subgraphs in g_1 are subgraph-isomorphic to subgraphs in g_2. Our algorithm extends

QuickSI [22] and Shang et al. [21] to find non-intersecting multiple maximal common subgraphs of g_1 and g_2 in order to calculate $dist(g_1, g_2) \leq \sigma$. The difference between our algorithm and previous works is that, we continue to explore g_2 to find a set of MCS′s of g_1 and g_2 that gives us $dist(q, g) \leq \sigma$. To reduce the number of node probings, we utilize the node connectivity when exploring nearby nodes.

Enumerating all possible common subgraphs that exist between q and g in order to find MCS′s is inefficient. To solve this problem, let us first represent a graph by a set of smaller subgraphs of g. Given a graph g, let $g = \{g'_1, ..., g'_n\} \cup \delta$, where $g'_i \subseteq g$, and $V(g'_i) \cap V(g'_j) = \emptyset$ and $E(g'_i) \cap E(g'_j) = \emptyset$ for all $i, j \leq n, i \neq j$. In this case, δ represent the set of edges that connect two subgraphs. Let $t(g')$ be a spanning tree of g' in g, and $T(g) = \{t(g'_1), ..., t(g'_n)\}$ be a set of spanning trees of the subgraphs in g. In $t(g')$, there are two types of edges, namely a forward and a backward edge. Let $E_f(t(g'))$ and $E_b(t(g'))$ be a set of forward and backward edges in $t(g')$, respectively. We can then express the set of edges in g in terms of $t(g')$.

$$E(g) = \bigcup_{i=1}^{|T(g)|} (E_f(t(g'_i)) \cup E_b(t(g'_i))) \cup E(\delta) \tag{3}$$

The problem of finding a set of subgraphs in g is converted to finding a set of spanning trees with backward edges in g. The advantage of having a set of subtrees is that, it gives us a systematic graph traversal order in g, and also allows us to reduce the node search space, as only the nodes that satisfy node connectivity constraints are considered.

Similar to the previous works [21, 22], we first generate a spanning tree of q, denoted as $t(q)$, and then we generate a node traversal order, called a tree-sequence, denoted as $ts(q)$, which tells us the node traversal order in g. An initial tree-sequence is obtained by traversing $t(q)$ in pre-order, and this tree-sequence is subsequently modified as node matching process progresses. Figure 7 shows examples of tree-sequences of q, and $ts_1(q)$ is an example of the initial tree-sequence of q. The dotted line in $ts_1(q)$ represents a backward edge. After a tree-sequence is generated, g is traversed by following $ts(q)$. While g is traversed, $ts(q)$ is modified such that, when a non-matching node in $ts(q)$ is found, the edge that connects the non-matching node is removed from $ts(q)$. After the edge is removed, one of the backward edges that connects the non-matching node becomes a forward edge, and a part of $ts(q)$ is restructured such that, the node that has the new incoming forward edge becomes the next node to process in $ts(q)$, and $ts(q)$ is updated accordingly by performing a preorder traversal on the nodes in $ts(q)$ that have not yet visited. For example, in Figure 7, $ts_2(q)$ is an example of a newly generated sequence from $ts_1(q)$, when (B, C) is found to be non-matching edge. To generate $ts_2(q)$, (B, C) in $ts_1(q)$ is removed, (B, D) in $ts_1(q)$ becomes a forward edge, and the order of C, D, E, and F is modified such that, D becomes the next node to process, as D has the newly generated forward edge. At some point, removing edges eventually split $ts(q)$, creating two disconnected sub tree-sequences, $ts'(q)$ and $ts''(q)$. For example, in Figure 7, removing (B, D) in $ts_2(q)$ makes $ts_2(q)$ split, and $ts_3(q)$ is the split tree-sequence of $ts_2(q)$. In this case, not-yet-traversed sub tree-sequence, $ts''(q)$, is traversed, if the length of a spath between the last node in $ts'_1(q)$ and the first node in $ts''_2(q)$ is $\leq k$. Continuing to traverse g according to the order specified in $ts(q)$ generates a set of MCS′s of q and g.

Algorithm 1 and 2 describe our graph verification procedure. The idea of Algorithm 1 is to find a starting node in g that will lead to the set of subgraphs that satisfies σ and k.

Lines 1–5: Given q, q’s spanning trees with different roots are generated, and a tree-sequence for each spanning tree is generated by traversing $t(q)$ in preorder.

Then, each node $v \in g$ that has the same label as the first node in $ts(q)$ becomes the first candidate matching node. Algorithm 2 is executed on g to match the remaining edges in $ts(q)$.

Lines 6–9: If a set of matching subgraphs cannot be found, q is relaxed by removing an edge from q, and the above procedure is recursively invoked to find matching MCS′s.

Algorithm 1: Verification(q,g,σ,k)

```
input  : $q$: query graph; $g$: graph; $\sigma$: edge relaxation;
         $k$: maximum lengths of spaths
output : boolean
1  foreach $ts(q) \in genSequences(genSpanningTrees(q))$ do
2      foreach $v \in V(g)$ do
3          if $\lambda(ts(q)[0]) = \lambda(v)$ then
4              if $VerificationSub(ts(q), q, g, \sigma, 0, 1)$ then
5                  return true;
6  if $\sigma \geq 0$ then
7      foreach $q' \in q.removeEdge()$ do
8          if $Verification(q', g, \sigma - 1, k)$ then
9              return true;
10 return false;
```

Algorithm 2 is recursively executed until the number of non-matching edges is $\leq \sigma$, or the current tree-sequence cannot be matched anymore.

Lines 1–4: We return true, if the number of non-matching edges is $\leq \sigma$, and false if the current tree-sequence does not lead to matching nodes that satisfy σ.

Lines 5–10: This part checks whether there exists a node that matches the current node in the tree-sequence, denoted as $ts(q)[i]$. Given $ts(q)[i]$, $candidates()$ generates a set of candidate nodes in g whose labels are the same as $ts(q)[i]$. To reduce the number of candidate nodes for $ts(q)[i]$, we first find $ts(q)[i-1]$'s matching node u in g, and get u's neighbouring nodes whose labels are the same as $ts(q)[i]$ and not-yet-previously matched. Furthermore, these candidate nodes are further reduced by testing them against a set of rules that define relationship between nodes. For example, in biochemistry, there exists a set of chemical reaction rules that show what type of atoms can participate in a chemical reaction. Additional filtering takes place in present of these rules. Then, we calculate the number of forward and backward edges that can be matched when u is matched, and calculate the number of missing edges so far. If the number of missing edges is $\leq \sigma$, we update the set of matching edges, and Algorithm 2 is called.

Lines 11–15: This part of code is executed when lines 5–10 cannot find $ts(q)[i]$'s matching node in g. In this case, removeEdge() modifies $ts(q)$ such that, the edge between $ts(q)[i-1]$ and $ts(q)[i]$ is removed, and checks whether there exists a path between $ts(q)[i-1]$ and $ts(q)[i]$. If there exists such a path, one of the nodes on the path contains a backward edge. The backward edge on the path is converted to a forward edge, and the sub-spanning tree starting from $ts(q)[i]$ is restructured such that, the node that had the backward edge is positioned to $ts(q)[i]$. Then, the tree-sequence is updated according

Algorithm 2: VerificationSub($ts(q), q, g, \sigma, k, i$)

```
input   : $ts(q)$: tree-sequence of $q$; $q$: query; $g$: graph; $\sigma$: edge relaxation; $i$: index number
          of the node in $ts(q)$ to process; $R$: rules to define valid node-to-node relationships
output  : boolean
require : $M$: a set of matching edges; $u$: previously matched vertex in $g$
1  if $|E(q)| - |M| \leq \sigma$ then
2      return true;
3  else if $pos \geq |ts(q)|$ then
4      return false;
5  foreach $v \in candidates(ts(q)[i], g, R)$ do
6      if $(\mu = |E(q)| - ckMatchingEdges(M, v, k)) \leq \sigma$ then
7          $updateMatchingEdges(M \cup \{v\})$;
8          if $VerificationSub(ts(q), q, g, \sigma - \mu, k, i + 1)$ then
9              return true;
10         $updateMatchingEdges(M - \{v\})$;
   /* remove an edge */
11 if $1 > \sigma$ and $i \geq 2$ then
12     $(ts(q)', ts(q)^d) = ts(q).removeEdge()$;
13     if $|ts(q)'| > i$ then
14         if $VerificationSub(ts(q)', q, g, \sigma - 1, k, i)$ then
15             return true;
       /* traverse the disconnected subtree */
16     if $seq^d \neq \emptyset$ then
17         foreach $c \in candidatesK(ts(q)[i], g, k, R)$ do
18             if $VerificationSub(ts(q)^d, q, g, \sigma, k, 0)$ then
19                 return true;
20 return false;
```

to the preorder traversal of the restructured sub-spanning tree. The updated tree-sequence is then traversed by calling Algorithm 2.

Lines 16–19: This part of code is executed if lines 11–15 generate a disconnected sub tree-sequence when an edge is removed. In this case, $candidatesK()$ generates a set of not-yet-matched nodes in g that are at most k distance away from $ts(q)[i-1]$'s matching node in g, and have the same label as $ts(q)[i]$. This set of candidate nodes are then sorted by distance, and each node is traversed by calling Algorithm 2.

Example 4 Given $\sigma = 2$ and $k = 2$, Figure 7 shows how q can be verified against g. A tree-sequence of q, $ts_1(q)$, is generated by traversing q in preorder. The dotted line in $ts_1(q)$ represents a backward edge. Then, g is traversed by following the node order represented by $ts_1(q)$. The first two nodes, A and B, in $ts_1(q)$ are matched, as $(A, B) \in E(g)$, but C cannot be matched, as $(B, C) \notin E(g)$. In this case, the edge (B, C) in $ts_1(q)$ is first removed, then (B, D) in $ts_1(q)$ is converted to a forward edge, and finally, the sub tree-sequence having C,D,E and F are restructured to give a new sub-sequence. This restructured sub-sequence starts with D, as the edge (B, D) in $ts_1(q)$, i.e., the backward edge that connects D, is converted to a forward edge. The newly generated sequence is shown as $ts_2(q)$. We now check whether D in $ts_2(q)$ can be matched, but it fails as $(B, D) \notin E(g)$. Therefore, the edge (B, D) in $ts_2(q)$ is removed. This time, the removal of the edge causes $ts_2(q)$ split

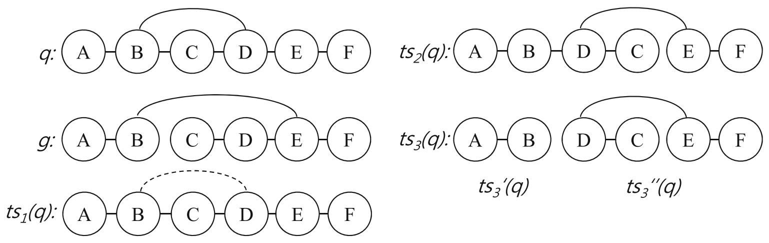


Figure 7 q, g, and tree-sequences of q

into two sequences. A newly generated sequence is shown as $ts_3(q)$, and $ts_3(q)$ has two sub sequences, $ts_3'(q)$ and $ts_3''(q)$. B in $ts_3'(q)$ is the previously matching node, and it is matched with $B \in V(g)$. Also, note that $ts_3''(q)$ starts with D. We now check whether there exists a node $D \in V(g)$ that is at most $k = 2$ edges away from $B \in V(g)$. $D \in V(g)$ is reachable from $B \in V(g)$, and it is 2 edges away. Now, $D \in V(g)$ becomes a candidate node, and it is matched with D in $ts_3''(q)$. We now check whether $ts_3''(q)$ can be matched. C in $ts_3''(q)$ can be matched with $C \in V(g)$, as $(C, D) \in E(g)$. E in $ts_3''(q)$ can also be matched as $(D, E) \in E(g)$. Similarly, F in $ts_3''(q)$ is matched, as $(E, F) \in E(g)$. At this stage, (B, C) and (B, D) in q are the only non-matching edges, i.e., $\sigma = 2$, and (B, D) in q is matched with a path $\langle B, E, D\rangle$ in g, i.e., $k = 2$. Hence, g is reported as a matching graph of q

Cost Analysis Given q and g, we analyze the total number of node comparisons made while verifying isomorphism mappings between q and g. Since we traverse g in a depth-first-search manner, the degree of a node in g reflects the number of possible paths to choose from to go 1 level deeper. At each node in a tree-sequence $ts(q)$, we check for matching nodes in g by following forward and backward edges. The total cost c is given as follows.

$$c \leq |V(g)| \cdot d_{max}(g)^{|V(g)|} \cdot d_{max}(ts(q))$$

where $d_{max}(g)$ is the maximum degree of a node in g, $d_{max}(ts(q))$ is the maximum number of forward and backward edges in a node in $ts(q)$. The space usage for our verification algorithm is $O(|ts(q)| + |g|)$, where $|g|$ is the space required to store g.

6 Feature-index

In this section, we describe a feature-index that is used to process MMCS-based graph similarity search queries efficiently.

Definition 9 (Feature-index) Let $F_D = \{f_1, ..., f_n\}$ be a set of feature graphs generated from the graphs in a graph database D, and let $F_G \subseteq F_D$. Given F_G, let $G = \{g \mid g \in D \text{ and } \forall f \in F_G, f \subseteq g\}$. A feature-index $\mathcal{F}$ is $\mathcal{F} = \bigcup_{i=1}^{n}\{(F_{G_i}, (d, G_i))\}$, where $d = dist(G_i, F_{G_i})$ and $D = \bigcup_{i=1}^{n}\{G_i\}$.

Figure 8 shows an example of a feature-index. A feature-index $\mathcal{F}$ is a set of tuples of the form (F_G, G), where F_G and G are a set of feature and data graphs, respectively, such that, each data graph in G contains all the feature graphs in F_G. A feature-index provides a way of grouping data graphs in D that contain the same feature graphs. The main idea

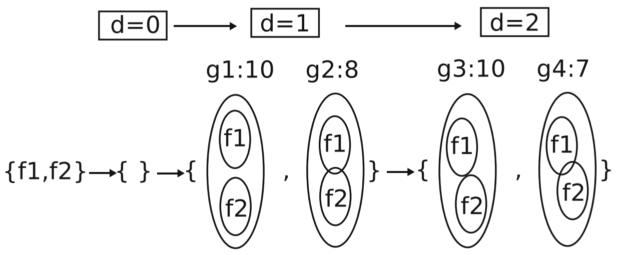


Figure 8 A feature-index

behind using a feature-index is that, we evaluate groups of data graphs to share the costs of subgraph isomorphism tests amongst the graphs that contain the same set of feature graphs. Similarity between data graphs are measured by feature-based closeness distance measure.

Definition 10 (Feature closeness) Given two graphs, $g_1, g_2 \in D$, and F_D, the closeness between g_1 and g_2 is calculated as:

$$closeness(g_1, g_2) = f(g_1) \cdot f(g_2) = \sum_{i=1}^{|F_D|} f(g_1)[i] \times f(g_2)[i] \qquad (4)$$

where $f(g)$ is a vector of feature graphs found in g, and $f(g)[i]$ is the value of ith feature in F_D, which is 1 if $f(g)[i] \subseteq g$, and 0 otherwise.

The closeness score is proportional to the number of feature graphs that are commonly found between g_1 and g_2. It is, however, difficult to make feature and data graph tuples in a way that, the number of feature graphs that each group of data graphs contains are maximized while reducing the total number of feature and data graph tuples in $\mathcal{F}$. A similar problem is shown to be NP-complete [32]. To solve this problem, we create groups of feature and data graph tuples empirically. Algorithm 3 describes the procedure.

Graphs that have similar set of features are grouped by merging the graphs iteratively. Graphs are initially stored in a list. We choose the first graph, g_1 (Line 4), and for each graph g_i in the list, we calculate the closeness score between g_1 and g_i (Line 6). Then, g_1 and g_i that have the highest closeness score are merged to form a group (Line 7). Once two graphs are merged, the vector of feature graphs are updated by calculating the dot product

Algorithm 3: GenerateGraphGroups(D, F_D)

```
input  : $D$: data graphs; $F_D$: feature graphs
output : $D'$: a set of $(F_g, G)$
1 $D'$ = generate a $(F_G, G)$ tuple for each $g \in D$, where $F = D_f$ and $G = \{g\}$;
2 $D''$ = insert $D'$ to a list;
3 while $|D''| \geq 0$ do
4     remove the first tuple, $(F_{g1}, G_1)$, from $D''$;
5     foreach $(F_G, G) \in D'$ do
6         if $(F_G, G)$ is the closest to $(F_1, G_1)$ then
7             update $G$ by adding graphs in $G_1$;
8             update the vector of features $F_g$ by intersecting with $F_1$;
9 return $D'$;
```

Algorithm 4: GenerateFeatureIndex(D, F_D, n)

```
input  : $D$: data graphs; $F_D$: Feature graphs; $n$: number of graph groups to generate
output : $\mathcal{F}$: a feature-index
1  $\mathcal{F} = \{\}$; $D' = GenerateGraphGroups(D, n)$;
2  foreach $(F_G, G) \in D'$ do
3      $G' = \{\}$;
4      foreach $g \in G$ do
5          $M = \{\}$;
6          foreach $f \in F$ do
7              perform an isomorphism test between $f$ and $g$ to find matching edges in $g$;
8              $M = M \cup$ {the matching edges in $g$};
9          $d = |E(g)| - |M|$;
10         $G' = G' \cup \{(d, g)\}$;
11     sort $G'$ according to distance $\mathcal{F} = \mathcal{F} \cup \{(F_G, G')\}$;
12 return $\mathcal{F}$;
```

of two vectors of feature graphs (Line 8). This vector of feature graphs are then used to calculate the closeness score in the following iterations. Lastly, the above merging process is repeated until all groups of graphs are merged.

Next, a feature-index is generated using these groups. Algorithm 4 describes the procedure. For each tuple (F_G, G), we extract the set of common feature graphs, F_G (Line 2). This set of feature graphs represent the group of graphs, G. Next, for each graph g in G, we calculate the distance between g and each feature graph in F_G, and the union of all matching edges are calculated (Lines 4–8). Then, the number of edges in g that are not in the union of matching edges are calculated, and we set this number as the distance between g and F_G (Lines 9–11). Lastly, the graphs in the group are sorted according to the distance from F_G (Line 11).

Example 5 Figure 8 shows an example of a feature-index. In this example, two feature graphs, $\{f_1, f_2\}$, are isomorphic to each data graph in $G = \{g_1, g_2, g_3, g_4\}$. The distance between $\{f_1, f_2\}$, and each graph $g \in G$ is shown on the top of each graph, e.g., g_1 and g_2 are 1 distance away from $\{f_1, f_2\}$. Some edges in g are common to both $\{f_1, f_2\}$, and these are represented by the intersection of ellipses. Hence, the total number of edges in g that are isomorphic to $\{f_1, f_2\}$ may be different for each g, e.g., g_1 has 10 isomorphic edges, while g_2 has 8 isomorphic edges.

6.1 Feature selection

In this section, we discuss the characteristics of feature graphs used when a feature-index is generated. Given D, gSpan [30] is used to extract frequent feature graphs. Feature frequency is defined as follows.

Definition 11 (Feature frequency) The feature frequency of a feature graph f is $|D_f|$, where $D_f = \{g \mid g \in D, f \subseteq g\}$, and it is denoted as $freq(f)$.

Given a feature graph f, $freq(f)$ denotes the number of graphs $g \in D$ that contains f as a subgraph.

By analyzing sets of feature graphs for groups of graphs, interesting observations were made. First, selectivity of feature graphs is strongly related to filtering power. Selecting feature graphs that have lower feature frequency increases the number of candidate graphs. To address this problem, feature graphs whose feature frequency is greater than a certain threshold t are selected.

Second, the number of feature graphs in the set may be too large, and many of them are not necessary to describe the graphs in the same group. This is because there are feature graphs, f' and f, where f' is a subgraph of f. Expressing the same set of graphs using f' and f do not improve filtering power, as it leads to perform unnecessary isomorphism tests. To fix this problem, we select the feature graphs that no other feature graphs in the set are supergraphs of those selected feature graphs, i.e., $\{f' \mid f', f \in F_G, f' \nsubseteq f\}$.

Third, even if the feature graphs are selected as above, some feature graphs are overlapped in a way that, two feature graph edges, one from each feature graph, may be mapped to the same edge in a data graph g. According to our filtering rules, it is better if the number of overlapped edges between these feature graphs are reduced, as that allows us to index more number of distinct edges found in data graphs. Hence, we try to select feature graphs that do not have overlapping edges. It is, however, often not possible to select such feature graphs, because those feature graphs may not exist or even if they exist, finding them is expensive. To solve this problem, we select feature graphs that have as little overlapping edges as possible. To do so, we select a set of feature graphs such that, the number of distinct edges in the union of all edges in the graphs is bigger than a certain threshold. The threshold is determined by the maximum number of edges that may be overlapped between two feature graphs. This establishes an inequality with the number of overlapping edges in g as shown below.

$$|E(F_g)| \geq |E(f_1)| + ... + |E(f_n)| - mn \tag{5}$$

where m is the maximum number of overlapping edges allowed between two feature graphs. After these set of feature graphs are selected, exact number of matching edges for each graph is calculated by performing subgraph isomorphism tests while a feature-index is constructed.

7 Similarity search framework

In this section, given a query graph q, a graph database D, an edge relaxation σ, a distance relaxation k, and a feature-index $\mathcal{F}$, we describe how MMCS-based graph similarity search is performed. Algorithm 5 describes the overall procedure.

Lines 2–7: When a query q is issued, q is processed for each feature index-entry $(F_G, (d, G)) \in \mathcal{F}$. For each feature graph $f \in F_G$, f is checked whether f is subgraph-isomorphic to q. If there exists $f \in F_G$ that is not subgraph-isomorphic to q, or if the number of non-matching edges in f is $> \sigma$, we stop processing the current $(F_G, (d, G))$, and process the next $(F_G, (d, G)) \in \mathcal{F}$, as the number of missing edges between q and G already exceeds σ, i.e., applying FR 1.

Lines 8–10: Next, we calculate the total number of the edges that are not subgraph-isomorphic to q. If this value, denoted as γ, is $> \sigma$, we stop processing the current $(F_G, (d, G))$ and process the next $(F_G, (d, G)) \in \mathcal{F}$, i.e., applying FR 2.

Lines 12–22: Next, for each $g \in G$, we check whether the minimum number of edges that can be missed is $> \sigma$, and if so, g is discarded, i.e., applying FR 3 and 3′. Otherwise, we check whether the maximum number of non-matching

Algorithm 5: GraphSimilaritySearch(q, D, $\mathcal{F}$, σ, k)

```
input  : $q$: a query graph; $D$: graph database; $F$: feature-index; $\sigma$: edge relaxation factor;
         $k$: edge connectivity factor
output : $R$: matching graphs
R = {}; C = {}; /* graph result and candidate set */
foreach $(F_G, (d, G)) \in \mathcal{F}$ do
    foreach $f \in F$ do
        find $q' \subseteq q$ that is subgraph-isomorphic to $f$;
        if number of non-matching edges in $f > \sigma$ then
            Go to the next $(F_G, (d, G))$; /* Filtering Rule 1 */
        $M = M \cup \{\text{matching edges in } q\}$
    $\gamma = E(F_G) \setminus M$; /* cal non-matching edges */
    if $|E(\gamma)| > \sigma$ then
        Go to the next $(F_G, (d, G))$; /* Filtering Rule 2 */
    else
        foreach $g \in G$ do
            $|E(\beta)| = d$;
            $\alpha = E(q) \setminus E(F_G)$;
            if $|E(\alpha)| - |E(\beta)| > \sigma$ then
                $g$ is unmatched. /* Filtering Rule 3 */
            else if $|E(mcs(F_G, g))| - |E(M)| + |E(\beta)| - |E(\alpha)| > \sigma$ then
                $g$ is unmatched. /* Filtering Rule 3' */
            else if $|E(\alpha)| \leq \sigma$ and $|E(mcs(F_G, g))| - |E(M)| + |E(\beta)| \leq$
            $\sigma$ and all lengths of spaths are $\leq k$ then
                $R = R \cup \{g\}$ /* Validation Rules */
            else
                $C = C \cup \{g\}$ /* Candidates */
foreach $g \in C$ do
    if $dist(q, g) \leq \sigma$ and $dist(g, q) \leq \sigma$ then
        $R = R \cup \{g\}$
return $R$;
```

edges is $\leq \sigma$, and if so, g is a potential matching graph. We then check if the lengths of all spaths are $\leq k$, and if so, g is added to the graph results set, R, i.e., applying VR. All other graphs are added to the candidate graph set, C. The above procedure is repeated for all $(F_G, (d, G)) \in \mathcal{F}$.

Lines 23–25: Lastly, for each $g \in C$, we check $dist(q, g) \leq \sigma$ and $dist(g, q) \leq \sigma$, and if so, g is added to C. Note that, checking $dist(g, q)$ does not involve any graph isomorphism tests. Finally, the graph result set is returned to the user.

8 Experiment

This section presents experimental results in detail. All experiments were executed on an Intel i7 2.8 GHz desktop with 4 GB ram running Linux with 2.6 kernel. Our approach was implemented in Java 1.6.

The AIDS antiviral dataset is a collection of chemical compounds that are made publicly available by Developmental Therapeutics Program.[1] The dataset contains chemical

[1]http://dtp.nci.nih.gov/docs/aids/aids_data.html

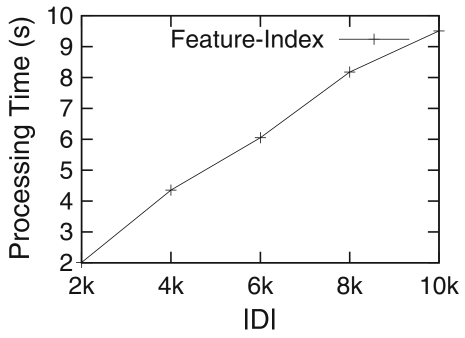


Figure 9 Building a feature-index

compounds for evidence of anti-HIV activity. A real-world usecase includes, given a known/unknown chemical compound q, possibly related to HIV/anti-HIV activity, find a list of known chemical compounds that could be related to q, such as results of oxidation/reduction/other chemical reactions of q. Then, researchers can further conduct experiments to verify whether there is a connection between q and each compound, and how addition/removal of organic atoms/compounds affect q. In this case, the system helps researchers to reduce the number of chemical compounds to verify.

A subset of dataset, which was prepared by Yan et al [31],[2] was used by many recent related works [4, 21, 22, 31, 32] for performance studies. The same dataset was used to evaluate our technique. The dataset contains 10,000 graphs with 51 distinct labels, and each graph contains 25.4 nodes and 27.4 edges on average. As for the query sets, following the previous works [4, 21, 22, 31], Q8, Q12, Q16, Q20, and Q24 query sets are used. Each query set, Qi, contains 1,000 query graphs with i edges. To generate large datasets, we used a synthetic graph data generator from He & Singh [11], and Q_i query sets are generated by extracting random subgraphs with size i and modified by adding/removing x random edges. Lastly, the processing time is the average processing time per query.

In biochemistry and metabolic engineering, the typical size of chemical compounds is around 10. This is relatively small compared to the graphs used in other domains, such as social network graph, which can be greater than millions. However, real-world examples in Section 1 and AIDS dataset show the typical size of chemical compounds that biochemistry researchers analyze. Unlike social network, in biochemistry, supporting a large number of small graphs is more important in this domain.

Figure 9 shows the feature-index construction time. As the size of dataset increases, the construction time almost linearly increases. This is because, the number of isomorphism tests that performed between feature graphs and data graphs are linearly increased. Also, the size of feature graphs remains almost constant, and the size is not affected by the number of data graphs.

Figure 10 shows the processing time when queries are evaluated with and without using the feature-index. The experiment shows that sequentially checking for $dist(q, g) \leq \sigma$ individually is several orders of magnitude slower than using the feature-index.

Figure 10 shows that our approach performs and scales better than the sequential approach. The rapid increase of processing time at the beginning only indicates that the filtering power is reduced for the dataset. This is more related to the characteristics of AIDS

[2]http://www.cs.ucsb.edu/~xyan/software.htm

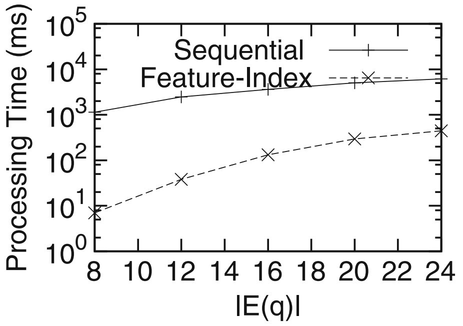


Figure 10 Comparing seq and feature-index

dataset that we used. However, as the number of queries increases, the increase of processing time cannot be higher than that of sequential approach, due to our filtering rules while searching for candidate graphs. For the following experiments, we use the feature-index generated in the pre-processing phase.

Figure 11 shows the processing time for evaluating the queries with various values of σ and k. The processing time increases as σ increases, and this is related to the number of verification algorithms that must be executed. As the value of k increases, the processing time also increases, and this indicates the additional overhead on executing the verification algorithm. We set $k = 1$ and compared the performance with DistVP. The performance difference for smaller queries is related to smaller number of candidate graphs to verify. When the sizes of candidate graphs to verify are almost the same, our approach slows down due to extra filtering/verification steps that need to be done. This is related to the semantics of

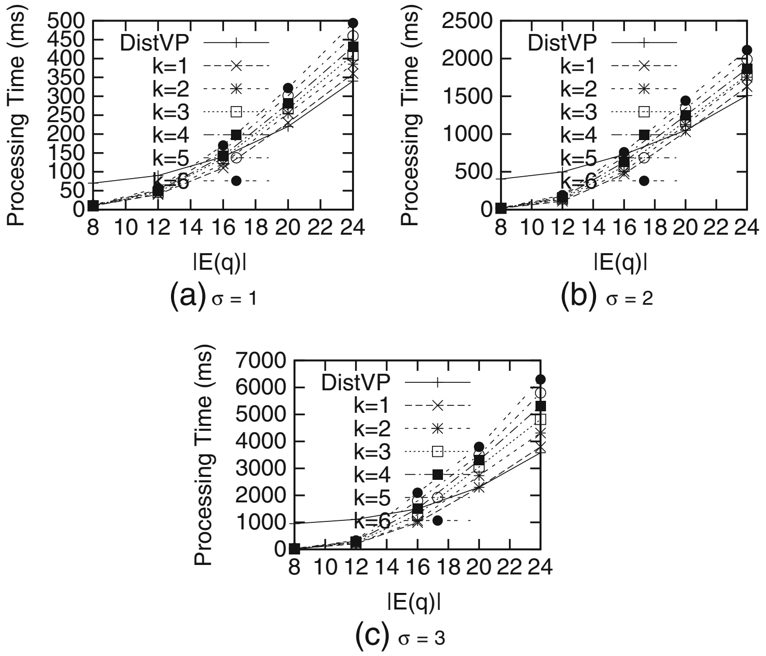


Figure 11 Processing queries with different σ

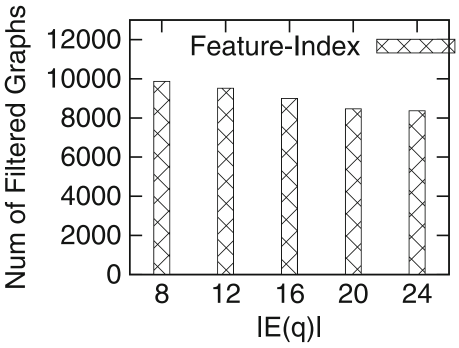


Figure 12 Comparing filtering power

our distance measure, which is not supported by previous work. However, our approach performs well for the queries whose size is less than or equal to 16, which is the typical chemical compound size in biochemistry and metabolic engineering as the example in Section 1 shows.

Figure 12 shows the number of filtered graphs when our filtering rules are applied. Our filtering rules can remove more non-matching graphs, especially for the queries with smaller number of edges, because in the filtering phase, the graphs that do not satisfy both $dist(q, g) \leq \sigma$ and $dist(g, q) \leq \sigma$ are also removed. In this experiment, the filtering power of our approach is related to the number of matching graphs in the AIDS dataset. For smaller graphs, there are more graphs that are considered as matching graphs by MMCS, but rejected by our approach, due to the distance constraints between common components.

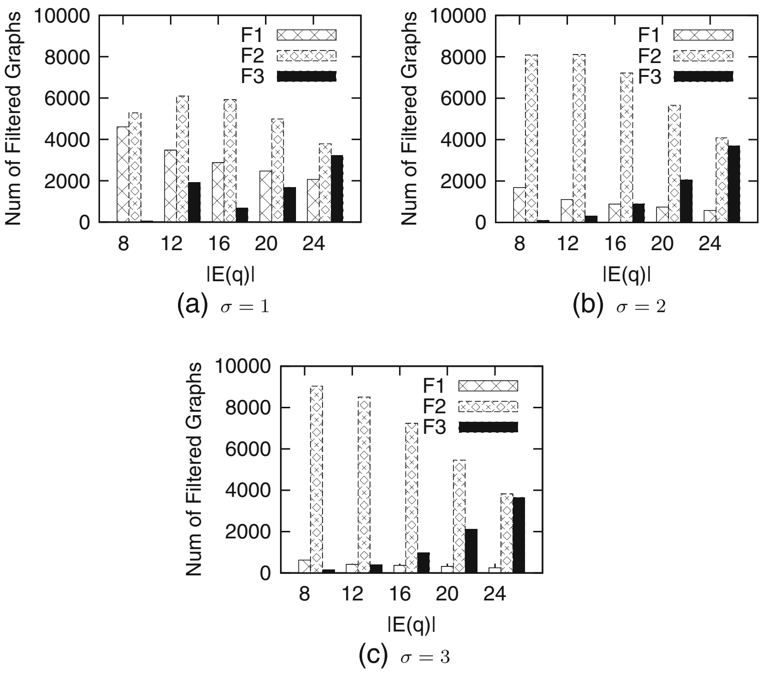


Figure 13 Number of filtered graphs with different σ

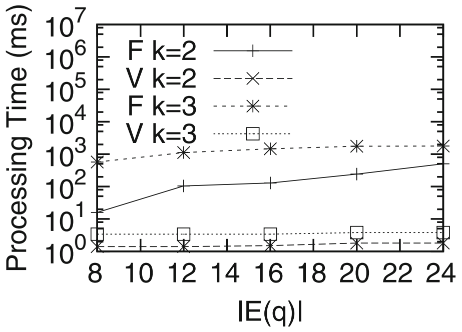


Figure 14 Filtering and verification, $\sigma = 2$

But for bigger graphs, there are fewer graphs that are rejected by our approach, because most bigger graphs in the dataset that satisfy MMCS constraints also satisfy our distance measure. This is related to the size of common components. As the size of graph increases, the size of common component also increases, and that reflects to reducing the distance between these components.

Figure 13 shows the number of graphs filtered by our filtering rules. Each filtering rule has different filtering power. Filtering Rule (FR) 1 and 2 are generally more effective on smaller query sets while FR 3 is more effective on larger query sets. This is related to the size of feature graphs and the size of data graphs. Smaller queries can only be filtered by smaller feature graphs. If the size of feature graphs in the index are bigger than the feature graphs contained by query graphs, the data graphs that are indexed with larger feature graphs can be discarded. When the size of query graphs are small, there are relatively many such data graphs, and these data graphs are all discarded by FR 1 and 2. FR 3 discards the data graphs that contain similar set of feature graphs as a query graph, but much larger than the query graph.

Figure 14 shows the time spent when filtering and verification algorithm with different k values are performed. This figure shows that the verification algorithm dominates the overall processing time. Also, similar patterns as Figure 11 are observed, and the explanation is similar.

Figure 15 shows the processing time when query graphs are evaluated on various sizes of datasets. Figure 15a shows the overhead when Q12 with different k values are evaluated. The difference on processing time reflects the extra number of verification algorithms that performed. Figure 15b shows the overhead when Q12 and Q16 are evaluated. Similarly, the

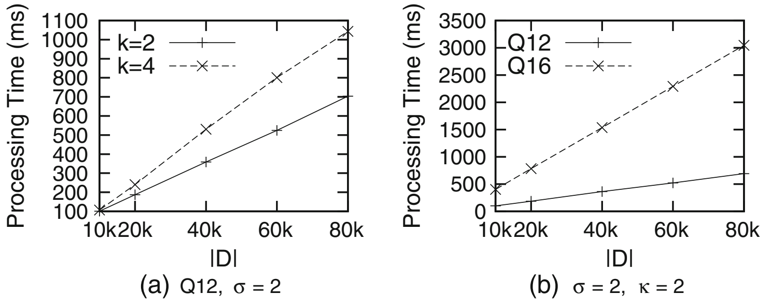
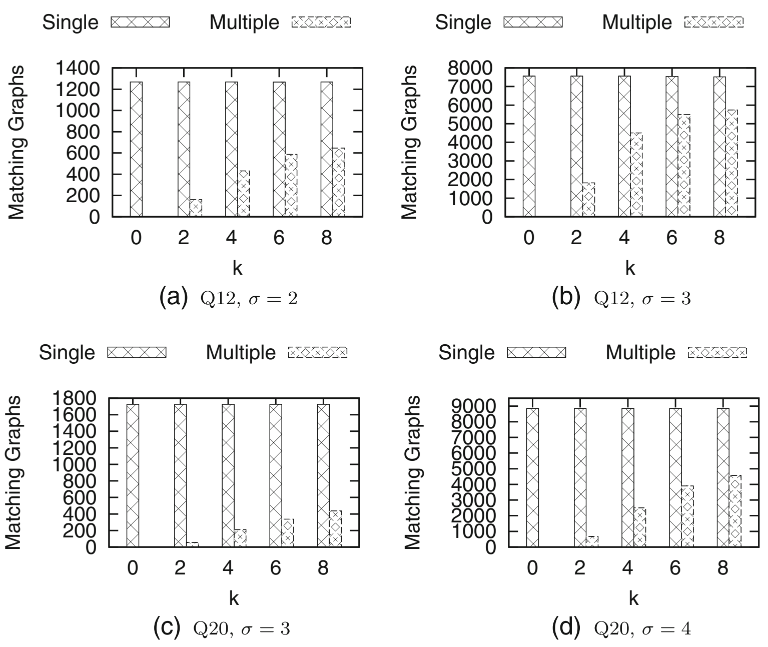


Figure 15 Testing scalability



(a) Q12, $\sigma = 2$ (b) Q12, $\sigma = 3$ (c) Q20, $\sigma = 3$ (d) Q20, $\sigma = 4$

Figure 16 Effectiveness of using MMCS

processing time difference reflects the extra number of steps required by the verification algorithm.

To test effectiveness of MMCS-based graph similarity search, we count the number of graphs that contain single MCSs and multiple MCS′s. Figure 16 shows the number of matching graphs. In this experiment, 100 queries are randomly selected from Q12 and Q20. Single represents the number of matching graphs, each of which contains a single MCS, and multiple represents the number of newly discovered matching graphs that contain multiple MCS′s. As the value of k increases, the number of matching graphs that contain multiple MCS′s increases. This is because, as the length of spath increases, the search space for finding MCS′s increases, and this leads to returning more matching graphs. Compared to DistVP and Grafil, the set of matching graphs returned by our approach generally contains what DistVP returns, and is contained by the graph set returned by Grafil. This is because, compared to DistVP, we relaxes MCS semantics to include the graphs that are rejected by DistVP. But, by restricting the distance between subgraphs in MMCS, we only return the graphs that are more meaningful. By increasing (or decreasing) the distance between subgraphs in MMCS, the overlap between the sets of matching graphs returned by our approach and Grafil increases (or decreases).

9 Conclusion

In this paper, we study the problem of graph similarity search. We proposed the MMCS-based graph similarity distance measure to retrieve previously undiscovered similar graphs, and yet discard dissimilar graphs reported by many previous works. We then proposed a series of filtering and validation rules to effectively remove non-matching graphs by calculating the upper and lower bounds on the distances between a query and feature graphs, and data and feature graphs. Then, an MMCS-based graph verification algorithm was proposed to effectively verify candidate graphs. Lastly, a feature-index was proposed to group similar data graphs and to efficiently support our filtering and validation rules. Experimental studies on real datasets showed that our filtering techniques can significantly reduce the number of candidate graphs, while our verification algorithm can efficiently verify candidate graphs.

A possible future work includes extending our technique to find a set of similar subgraphs that exist in a large data graph with tens of thousands of vertices.

Acknowledgments We would like to thank the editor and anonymous reviewers. This work was supported by the National Research Foundation of Korea grant funded by the Korean government (MSIP) (No. NRF-2009-0081365).

References

1. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
2. Chen, C., Yan, X., Yu, P.S., Han, J., Zhang, D.-Q., Gu, X.: Towards graph containment search and indexing. In: Proceedings of the 33rd International Conference on very Large Data Bases, pp. 926–937. ACM, Vienna (2007)
3. Cheng, J., Ke, Y., Ng, W.: Efficient query processing on graph databases. ACM Trans. Database Syst. **34**(1) (2009). Article No. 2
4. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 857–872. ACM, Beijing (2007)
5. Cuissart, B., Hébrard, J.-J.: A direct algorithm to find a largest common connected induced subgraph of two graphs. In: GbRPR, pp. 162–171. Springer, Poitiers (2005)
6. Fan, W., Li, J., Luo, J., Tan, Z., Wang, X., Wu, Y.: Incremental graph pattern matching. In: SIGMOD, pp. 925–936 (2011)
7. Fan, W., Li, J., Ma, S., Wang, H., Wu, Y.: Graph homomorphism revisited for graph matching. PVLDB **3**(1), 1161–1172 (2010)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
9. Han, W.-S., Lee, J., Lee, J.-H.: Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 337–348. ACM, New York (2013)
10. He, H., Singh, A.K.: Closure-tree: an index structure for graph queries. In: Proceedings of the 22nd International Conference on Data Engineering, p. 38. IEEE Computer Society, Atlanta (2006)
11. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 405–418. ACM, Vancouver (2008)
12. Jiang, H., Wang, H., Yu, P.S., Zhou, S.: Gstring: a novel approach for efficient search in graph databases. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 566–575. IEEE, Istanbul (2007)
13. Jin, C., Bhowmick, S.S., Choi, B., Zhou, S.: Prague: towards blending practical visual subgraph query formulation and query processing. In: IEEE 28th International Conference on Data Engineering, pp. 222–233. Washington (2012)
14. Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., Tao, S.: Neighborhood based fast graph search in large networks. In: SIGMOD, pp. 901–912 (2011)
15. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. Theor. Comput. Sci. **250**(1–2), 1–30 (2001)
16. Lee, J., Han, W.-S., Kasperovics, R., Lee, J.-H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2), 133–144 (2012)

17. Lee, J., Oh, J.-H., Hwang, S.: Strg-index: Spatio-temporal region graph indexing for large video databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 718–729. ACM, Baltimore (2005)
18. Mongiovì, M., Natale, R.D., Giugno, R., Pulvirenti, A., Ferro, A., Sharan, R.: Sigma: a set-cover-based inexact graph matching algorithm. J. Bioinforma. Comput. Biol. **8**(2), 199–218 (2010)
19. Raymond, J.W., Gardiner, E.J., Willett, P.: Rascal: calculation of graph similarity using maximum common edge subgraphs. Comput. J. **45**(6), 631–644 (2002)
20. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. **16**(7), 521–533 (2002)
21. Shang, H., Lin, X., Zhang, Y., Yu, J.X., Wang, W.: Connected substructure similarity search. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 903–914. ACM, Indianapolis (2010)
22. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1), 364–375 (2008)
23. Shang, H., Zhu, K., Lin, X., Zhang, Y., Ichise, R.: Similarity search on supergraph containment. In: Proceedings of the 26th International Conference on Data Engineering, pp. 637–648. IEEE, Long Beach (2010)
24. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS, pp. 39–52. ACM, Madison (2002)
25. Tong, H., Faloutsos, C., Gallagher, B., Eliassi-Rad, T.: Fast best-effort pattern matching in large attributed graphs. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 737–746. ACM, San Jose (2007)
26. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976)
27. Wang, X., Ding, X., Tung, A.K.H., Ying, S., Jin, H.: An efficient graph indexing method. In: IEEE 28th International Conference on Data Engineering, pp. 210–221. IEEE Computer Society, Washington (2012)
28. White, S., Smyth, P.: Algorithms for estimating relative importance in networks. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 266–275. ACM, Washington (2003)
29. Williams, D.W., Huan, J., Wang, W.: Graph database indexing using structured graph decomposition. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 976–985. IEEE, Istanbul (2007)
30. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining, pp. 721–724. IEEE Computer Society, Maebashi City (2002)
31. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 335–346. ACM, Paris (2004)
32. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 766–777. ACM, Baltimore (2005)
33. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. PVLDB **2**(1), 25–36 (2009)
34. Zhang, S., Hu, M., Yang, J.: Treepi: a novel graph indexing method. In: Proceedings of the 23rd International Conference on Data Engineering, pp. 966–975. IEEE, Istanbul (2007)
35. Zhang, S., Yang, J.: Gaddi: distance index based subgraph matching in biological networks. In: Proceedings of the 12th International Conference on Extending Database Technology, pp. 192–203. ACM, Saint Petersburg (2009)
36. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: tree + delta >= graph. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 938–949. ACM, Vienna (2007)
37. Zhu, Y., Qin, L., Yu, J.X., Ke, Y., Lin, X.: High efficiency and quality: large graphs matching. VLDB J. **22**(3), 345–368 (2013)