IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 June 2022, accepted 25 July 2022, date of publication 28 July 2022, date of current version 2 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3194559

RESEARCH ARTICLE

Efficient Top-*k* Graph Similarity Search With GED Constraints

JONGIK KIM
Department of Artificial Intelligence, Chungnam National University, Daejeon 34134, South Korea
e-mail: jongik@cnu.ac.kr

This work was supported by the Research Fund of Chungnam National University.

ABSTRACT It is essential to identify similarity between graphs for various tasks in data mining, machine learning and pattern recognition. Graph edit distance (GED) is the most popular graph similarity measure thanks to its flexibility and versatility. In this paper, we study the problem of top-*k* graph similarity search, which finds *k* graphs most similar to a given query graph under the GED measure. We propose incremental GED computation algorithms that compute desired GED lower and upper bounds. Based on the algorithms, we develop novel search frameworks to address the top-*k* search problem. Our frameworks are also designed to use a state-of-the art indexing technique to speed up top-*k* search. By conducting extensive experiments on real datasets, we show that the proposed frameworks significantly improve the performance of top-*k* graph similarity search.

INDEX TERMS Top-*k* similarity search, graph edit distance, incremental GED computation.

I. INTRODUCTION

Because of the representational flexibility, graphs are widely used to model complex and interconnected data in many application scenarios. For example, images such as envelops, lunar surfaces, and topological building structures have been represented by graph models [1]–[3]; fingerprints and cancerous tissue of biological cells have been modeled by graphs for identification and classification [4], [5]; Control flow graphs have been used in business process management [6]; and malware detection by comparing call graphs of programs have been studied in [7], [8]. In these applications, error-tolerant graph search, *i.e.*, graph similarity search, is essential due to inconsistency, noises, and representational differences of graph data [9]–[15].

To quantify graph similarities, various similarity measures have been proposed such as missing edges and features [16], [17], maximum common subgraphs (MCS) [10], [18], graph alignment [19], and graph edit distance (GED) [20]–[26]. Among them, GED is the most commonly used measure, because GED precisely captures structural difference between graphs and it is applicable to all types of graphs [13], [27]. The GED between two graphs is the minimum number of graph edit operation to make the graphs isomorphic, where a graph edit operation is an insertion, deletion, or relabeling of a single vertex or edge.

Many techniques have been developed to solve the problem of threshold-based similarity search, which is to find all data graphs whose GED to the query is within a given threshold. However, the number of results for the same threshold can vary significantly for different queries because the distribution of different structures in a graph database is not uniform [28]. This makes it hard for a user to choose a suitable threshold for each query. In this paper, therefore, we study the problem of top-*k* graph similarity search, which finds *k* graphs most similar to the query under the graph edit distance measure.

The major difficulty in graph similarity search with GED constraints is that GED computation is NP-hard. Existing solutions adapt a filtering-and-verification framework, which generates candidate graphs in the filtering phase and verifies each candidate through GED computation in the verification phase. To reduce the overhead of GED computation, it is essential to generate a small set of candidate graphs. Therefore, the focus of existing solutions has been on developing effective filtering techniques for candidate generation.

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

Commonly used techniques in the filtering phase are to utilize features of graphs, which are substructures of graphs [9], [11], [13], [14], [27], [30], [31]. These techniques have been mainly developed for solving the problem of threshold-based similarity search, and filtering conditions are established based on a GED threshold. They typically build an inverted index on features of data graphs to generate candidate graphs.

In top-k similarity search, however, a user specified GED threshold is not available, and thus existing feature-based filtering techniques cannot be directly used. We are aware of a top-k graph similarity search technique ParsK [31], which are based on a partition-based filtering technique. ParsK builds a hierarchical partition-based index motivated by a top-k string similarity search technique [41]. However, recent work has revealed that the filtering power of feature-based indexing techniques is inherently limited [29], [33], [37]. For example, Inves [29] and AStar$^+$-LSa [33] have empirically showed that efficient verification techniques invalidate the filtering effect of existing feature-based indexing techniques.

To address the problems, in this paper, we develop efficient top-k graph similarity search frameworks (Section III). A naive approach is to compute the GED between the query and each data graph and return top k best results. However, this naive approach obviously incurs prohibitive GED computation overheads. As a baseline of top-k search, therefore, we adapt existing threshold-based approach by issuing a series of queries by increasing the threshold one by one until k result graph are found. This baseline approach requires multiple verifications of the same candidates in different thresholds, which are computationally expensive. To improve the performance of top-k similarity search, we develop a few search frameworks. The first is to derive different thresholds from intermediate results of top-k search and utilize these thresholds for efficient verification of data graphs (Section III-A). We further optimize top-k search by utilizing precise GED lower and upper bounds. To obtain precise GED bounds, we propose novel GED computation algorithms that stop and resume GED computation. Based on the algorithms, we explore two search frameworks that reduce GED computation overheads of unpromising graphs (Section III-B and III-C). To utilize the latest pre-computed GED-based index proposed in our previous work [37], we also propose a hybrid search framework that integrates threshold-based approach with our top-k search techniques (Section III-D). We conduct extensive experiments on widely used real datasets and show that the proposed frameworks significantly improve the performance of top-k graph similarity search (Section IV).

The rest of the paper is organized as follows. Section II formulates the problem of top-k similarity search with GED constraints and reviews GED computation algorithm. Section III proposes incremental GED computation algorithms for obtaining desired GED lower and upper bounds and develops novel top-k search frameworks. Experimental results are reported in Section IV. Section V lists related work and Section VI concludes the paper.

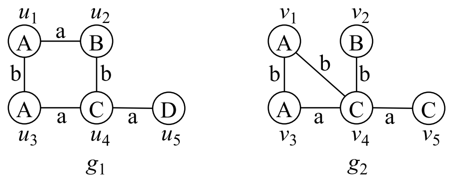


FIGURE 1. **Example graphs g_1 and g_2.**

II. PRELIMINARIES

A. PROBLEM FORMULATION

A simple graph is a labeled undirected graph having neither self-loops nor multiple edges. In this paper, we use simple graphs for the ease of exposition. A labeled simple graph g is represented in a triple of a set of vertices $V(g)$, a set of edges $E(g) \subseteq V(g) \times V(g)$, and a labeling function $l : V(g) \cup (V(g) \times V(g)) \rightarrow \Sigma$, where Σ is the label set of vertices and edges. If there is no edge between two vertices u and v, $l(u, v)$ returns a unique value λ distinguished from all other labels. We also define a blank vertex ε such that $\forall v \in V(g)\ l(\varepsilon) = l(\varepsilon, v) = l(v, \varepsilon) = \lambda$. We remark that λ is not a label and used only for identifying the absence of a vertex or edge.

To measure the similarity between a pair of graphs, we use graph edit distance defined in Definition 1.

Definition 1 (Graph Edit Distance): The graph edit distance (GED) between two graphs g_1 and g_2, which is denoted by $\mathsf{ged}(g_1, g_2)$, is the minimum number of edit operations that transform g_1 into g_2, where an edit operation is one of the following:

1) insertion of an isolated labeled vertex.
2) deletion of an isolated labeled vertex.
3) relabeling of a labeled vertex.
4) insertion of a labeled edge.
5) deletion of a labeled edge.
6) relabeling of a labeled edge.

Example 1: Figure 1 shows example graphs g_1 and g_2. The graph g_1 is transformed into the graph g_2 by performing the following edit operations: deleting the edge between u_1 and u_2, inserting an edge labeled with b between u_1 and u_4, and relabeling the vertex u_5 with C. Therefore, the GED between g_1 and g_2 is 3.

We formulate the problem of top-k graph similarity search as follows.

Definition 2 (Top-k Graph Similarity Search Problem): Given a graph database $\mathcal{D}$, a query graph q, an integer k, and a maximum GED threshold τ_{max}, top-k graph similarity search finds a result set, denoted by $\mathcal{K}(q, k, \tau_{max})$, that contains every data graph $g \in \mathcal{D}$ such that

$$|\mathcal{K}(q, k, \tau_{max})| = \begin{cases} k, & \text{if } |\mathcal{R}(q, \tau_{max})| \geq k \\ |\mathcal{R}(q, \tau_{max})|, & \text{otherwise} \end{cases}$$

and $\forall g \in \mathcal{K}(q, k, \tau_{max})$, $\nexists g' \in G \backslash R$, $\mathsf{ged}(q, g') < \mathsf{ged}(q, g)$.

Similar to existing work for threshold-based similarity search [31], [37], in Definition 2, we introduce a maximum

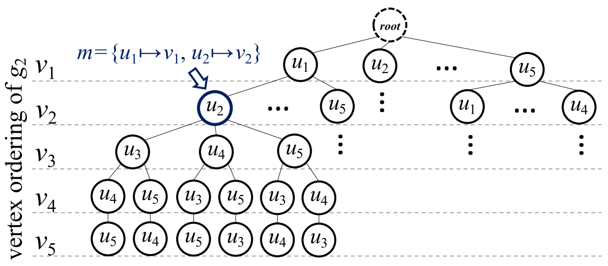


FIGURE 2. Search tree for graphs in Figure 1.

threshold τ_{max} for top-k graph similarity search to prevent excessive GED computation for a large k.

B. GED COMPUTATION

In this subsection, we review existing GED computation algorithms. Given two graphs g_1 and g_2, if $|V(g_1)| \neq |V(g_2)|$, we add $||V(g_1)| - |V(g_2)||$ copies of ε into either $V(g_1)$ or $V(g_2)$ to make the sizes of g_1 and g_2 the same. So, we assume that $|V(g_1)| = |V(g_2)|$ throughout the paper. The GED between two graphs g_1 and g_2 is basically computed by enumerating all possible vertex mappings between the graphs, where a vertex mapping is a bijection of $V(g_1)$ onto $V(g_2)$. By imposing a pre-defined ordering of $V(g_2)$, which is called the *vertex ordering* of g_2, we represent a vertex mapping as an ordered set of mapped vertex pairs. Given a vertex mapping m, the edit cost of m is the number of edit operations to make each mapped vertex pair identical in terms of the vertex label and adjacent edges.

Definition 3 (Edit Cost): The edit cost for a vertex mapping $m = \{u_1 \mapsto v_1, \ldots, u_n \mapsto v_n\}$ is defined as:

$$\mathsf{ec}(m) = \sum_{i=1}^{n}\left(\mathsf{D}(l(u_i), l(v_i)) + \sum_{j<i}\mathsf{D}(l(u_i, u_j), l(v_i, v_j))\right),$$

where $\mathsf{D}(x, y)$ is 0 if $x = y$, 1 otherwise.

Example 2: For the two graphs in Figure 1, consider a vertex mapping $m = \{u_1 \mapsto v_1, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5\}$. The edit cost of m is computed as follows. we omit $\mathsf{D}(\varepsilon, \varepsilon)$ in the following.

- $u_1 \mapsto v_1 : \mathsf{D}(A, A) = 0$
- $u_2 \mapsto v_2 : \mathsf{D}(B, B) + \mathsf{D}(a, \varepsilon) = 1$
- $u_3 \mapsto v_3 : \mathsf{D}(A, A) + \mathsf{D}(b, b) = 0$
- $u_4 \mapsto v_4 : \mathsf{D}(C, C) + \mathsf{D}(a, a) + \mathsf{D}(b, b) + \mathsf{D}(\varepsilon, b) = 1$
- $u_5 \mapsto v_5 : \mathsf{D}(D, C) + \mathsf{D}(a, a) = 1$

Therefore, $\mathsf{ec}(m) = 0 + 1 + 0 + 1 + 1 = 3$.

GED computation is to find a vertex mapping among all possible vertex mappings whose edit cost is the minimum. By merging the shared prefix of different vertex mappings, all possible vertex mappings form a state-space tree, which is called *search tree*. For example, Figure 2 shows the search tree for the graphs in Figure 1. In the figure, we assume the vertex ordering of g_2 is $(v_1, \ldots, v_5)$. Each leaf node of the tree represents a vertex mapping, and an intermediate node is a shared prefix of vertex mappings, which is called a *partial mapping*. In the figure, a tree node at level j, denoted by u_i, represents a partial mapping $m_p \cup \{u_i \mapsto v_j\}$, where m_p is the partial mapping of the parent node and the root node represents an empty mapping. For example, the node indicated by an arrow represents a partial mapping $m = \{u_1 \mapsto v_1, u_2 \mapsto v_2\}$. In the remainder of the paper, we abuse a mapping to refer to a partial mapping if clear from the context.

Before we present a GED computation algorithm, we introduce a lower bound of a partial mapping, which is used for computing GED.

Definition 4 (Lower Bound of a Partial Mapping): Given a partial mapping m, let S_m be the set of the vertex mappings in the leaf nodes rooted by m. A lower bound of m, denoted by $\mathsf{lb_M}(m)$, is a number lb such that $0 \leq lb \leq \min_{m' \in S_m} \mathsf{ec}(m')$.

Given a partial mapping m, $\mathsf{lb_M}(m)$ can be computed by the sum of the edit cost of the mapped part and that of the unmapped part. It is clear that the number of edit operations required in mapped vertices and edges is $\mathsf{ec}(m)$. To obtain a lower bound of m, therefore, we need to estimate a lower bound of the edit cost required in unmapped vertices and edges. The label set-based lower bound in Definition 5 is the most widely used lower bound function to estimate the edit cost of the unmapped part.

Definition 5 (Label Set-Based Lower Bound [30], [37]): The label set-based lower bound between two graphs g_1 and g_2 is defined as:

$$\mathsf{lb_L}(g_1, g_2) = \Gamma(L_V(g_1), L_V(g_2)) + \Gamma(L_E(g_1), L_E(g_2)),$$

where $L_V(g)$ and $L_E(g)$ respectively denote the label multisets of vertices and edges of g, and $\Gamma(S_1, S_2) = \mathbf{max}(|S_1 - S_2|, |S_2 - S_1|)$.

Example 3: Consider we have a partial mapping $m = \emptyset$, which corresponds to the root node of the search tree. Since there is no mapped vertices and edges in m, the unmapped part of the mapping is g_1 and g_2. A lower bound of m can be computed as:

$$\mathsf{lb_M}(m) = \mathsf{ec}(m) + \mathsf{lb_L}(g_1, g_2) = 0 + 2 = 2,$$

since $\mathsf{ec}(m) = 0$, $\Gamma(L_V(g_1), L_V(g_2)) = \Gamma(\{A, A, B, C, D\}, \{A, A, B, C, C\}) = 1$, and $\Gamma(L_E(g_1), L_E(g_2)) = \Gamma(\{a, a, a, b, b\}, \{a, a, b, b, b\}) = 1$.

The tightness of a lower bound of a mapping is crucial in reducing the search space of GED computation. Various techniques have been proposed to compute a tightened lower bound of a mapping. For example, Inves [29] captures edit errors in bridges, which are edges connecting mapped vertices to unmapped vertices. Nass [37] applies different lower bound functions to the unmapped part to obtain a tighter lower bound.

Now, we will review a threshold-based GED computation algorithm.[1] Algorithm 1 outlines a GED computation algorithm that uses the A* search strategy. It first determines the order of vertices in g_2 (Line 1). A common intuition behind

[1] Although a user-specified GED threshold is not available in top-k graph similarity search, we will derive different thresholds while processing a top-k query in Section III.

Algorithm 1 GED(g_1, g_2, τ)

```
input  : $g_1$ and $g_2$ are graphs, and $\tau$ is a GED threshold
output: ged($g_1, g_2$) if ged($g_1, g_2$) $\leq \tau$, $\tau + 1$ otherwise
1  $\mathcal{O} \leftarrow$ vertex ordering of $V(g_2)$;
2  initialize a priority queue $Q$ with an empty mapping;
3  while $Q \neq \emptyset$ do
4      $m \leftarrow Q$.pop();
5      if $|m| = |V(g_1)|$ then return ec($m$);
6      if lb_M($m$) $\leq \tau$ then
7          $v \leftarrow$ next unmapped vertex in $V(g_2)$ as per $\mathcal{O}$;
8          foreach $u \in V(g_1)$ s.t. $v \notin m$ do
9              $m_c \leftarrow m \cup \{u \mapsto v\}$;
10             if lb_M($m_c$) $\leq \tau$ then $Q$.push($m_c$);
11 return $\tau + 1$;
```

the vertex ordering is that infrequent vertices are matched first while preserving the connectivity [29], [33]. After determining the vertex ordering of g_2, the algorithm pushes the initial state, i.e., an empty mapping, which corresponds to the root node of the search tree, into the queue (Lines 2). In the main loop, it pops a mapping m from the queue (Line 4). If the mapping m popped from the queue is a *full mapping* (i.e., m contains all vertices of g_1 and g_2), it returns the edit cost of the mapping as $\mathsf{ged}(g_1, g_2)$ (Line 5). If m is a partial mapping and its lower bound is less than τ (Line 6), it expands the search tree by mapping the next unmapped vertex v in g_2 (Line 7) to each unmapped vertex u in g_1 (Line 8). It pushes each expanded tree node m_c into the queue if the lower bound of m_c is less than τ (Lines 9–10). If the queue is empty, the algorithm returns $\tau + 1$ (Line 11).

III. TOP-*k* GRAPH SIMILARITY SEARCH

In this section, we develop techniques for top-*k* graph similarity search. To prevent excessive computation for some queries, we assume a maximum threshold τ_{max} for top-*k* search, similar to [37] and [31]. Because of the maximum threshold, the number of results for some queries can be less than k. For the ease of presentation, however, we assume that we have at least k results within τ_{max} throughout this section. The maximum threshold τ_{max} enables us to generate candidate graphs using an existing feature-based index such as Pars [13] and MLIndex [9]. For a large threshold such as τ_{max}, however, the filtering effects of these feature based indexing techniques are almost the same as that of a basic filter that uses the label set-based lower bound [33], [37]. Therefore, we use the label set-based lower bound (Definition 5) to generate an initial candidate set with τ_{max}.

As we described in Section I, we use a threshold-based top-*k* similarity search as a baseline algorithm. The threshold-based top-*k* similarity search is to evaluate a series of similarity queries by increasing the threshold τ (initially, $\tau = 0$) one by one until k result graphs are found. However, this baseline algorithm can be computationally expensive because

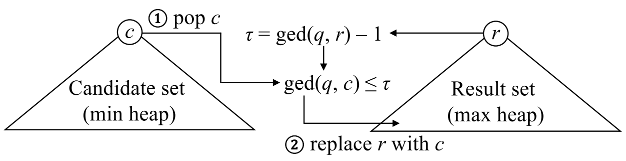


FIGURE 3. An overview of basic top-*k* search.

it requires multiple verifications of the same candidates in different thresholds. To improve the baseline algorithm, we propose three top-*k* search algorithms. Each algorithm is also designed to use an index structure based on the pre-computed GEDs, which has been proposed in our previous work [37].

A. BASIC TOP-K SEARCH ALGORITHM

To avoid redundant verifications of the same candidates, the first technique we propose is to use a min heap $\mathcal{C}$ to keep candidate graphs with their GED lower bounds and a max heap to keep the current best k results. When $\mathcal{R}$ is filled up, we can utilize the GED of the head of $\mathcal{R}$ as a GED threshold for searching remaining data graphs. This threshold plays an important role in reducing the overhead of GED computation. We call this technique a basic top-*k* search.

Figure 3 shows an overview of the basic top-*k* search. Assume that we have a candidate set $\mathcal{C}$ organized into a min heap using GED lower bounds of candidates. We also organize the result set $\mathcal{R}$ into a max heap using GEDs of current results. After $\mathcal{R}$ is filled up, the head of $\mathcal{R}$ has the largest GED among other graphs in $\mathcal{R}$. Hence, we can filter out remaining graphs whose GED to the query is not less than $\mathsf{ged}(q, r)$, where r is the head of $\mathcal{R}$. That is, we can set a threshold $\tau = \mathsf{ged}(q, r) - 1$. Given the threshold τ, ① we pop a candidate graph c from $\mathcal{C}$, and ② if $\mathsf{ged}(q, c) \leq \tau$, we pop r from $\mathcal{R}$ and push c into $\mathcal{R}$. This approach overcomes the disadvantage of the baseline approach since it verifies each candidate graph only once.

Algorithm 2 outlines the basic top-*k* algorithm. The algorithm initializes a min-heap $\mathcal{C}$ to keep pairs of (candidate graph, GED lower bound *lb*) (Line 1). Given a maximum threshold τ_{max}, the algorithm generates a candidate set using the label set-based lower bounds $\mathsf{lb_L}$ of each data graph (Lines 2–3). After generating a set of candidates, it initializes the threshold τ to τ_{max} and a max-heap $\mathcal{R}$ to keep pairs of (result graph, GED *dist*) (Lines 4–5). From the candidate set $\mathcal{C}$, it pops out a candidate graph having the minimum lower bound (Line 7), and computes GED δ between the candidate and the query (Line 8). If the current candidate g is identified a result, *i.e.*, one of the current best k results (Line 9), the algorithm remove the head element from the result set $\mathcal{R}$ and push g with the GED δ into $\mathcal{R}$ (Lines 10–11). If it has k results in $\mathcal{R}$ (Line 12), it updates the threshold τ to $\mathcal{R}$.head().distance $-$ 1 (Lines 13) because $\mathcal{R}$.head().distance is the largest GED among currently identified results. By tightening the threshold τ, it accelerates the GED computation in Line 8 as well as reduces the number of candidates to verify in Line 6. To efficiently pruning unpromising graphs, our algorithm

Algorithm 2 TopK-Basic($\mathcal{D}, q, k$)

```
input : $\mathcal{D}$ is a graph database,
        $q$ is a query graph, and $k$ is an integer
output: $\mathcal{K}(q, k, \tau_{max})$
1  $\mathcal{C} \leftarrow$ min-heap initialized to be $\emptyset$;
2  foreach graph $g \in \mathcal{D}$ do
3      if $\mathsf{lb}_{\mathsf{L}}(q, g) \leq \tau_{max}$ then $\mathcal{C}$.push($g$, $\mathsf{lb}_{\mathsf{L}}(q, g)$);
4  $\tau \leftarrow \tau_{max}$;
5  $\mathcal{R} \leftarrow$ max-heap of size $k$ with each value set to (nil, $\infty$);
6  while $\mathcal{C} \neq \emptyset$ and $\mathcal{C}$.head().$lb \leq \tau$ do
7      $(g, -) \leftarrow \mathcal{C}$.pop();
8      $\delta \leftarrow$ GED($q, g, \tau$);
9      if $\delta \leq \tau$ then
10         $\mathcal{R}$.pop();
11         $\mathcal{R}$.push($g, \delta$);
12         if $\mathcal{R}$.head().$dist \neq \infty$ then
13             $\tau \leftarrow \mathcal{R}$.head().$dist - 1$;
14         if index is available then
15             $\mathcal{C} \leftarrow$ regenCandidates($\mathcal{C}, \delta, \tau$);
16 return $\mathcal{R}$;
```

uses an index based on the pre-computed GEDs of data graphs, which is proposed in our previous work [37]. If the index is available, we can regenerate a candidate set whenever a result is found. Given a graph g, the index requires a GED threshold τ and $\delta = \mathsf{ged}(q, g)$ to regenerate a candidate set. regenCandidates regenerates candidates using the index if the index is available (Line 15, refer to [37] for the details of the index and candidate regeneration). The algorithm finally returns the result set $\mathcal{R}$ (Line 16).

Theorem 1: Algorithm 2 correctly returns top-k result graphs.

Proof: It is obvious that the initial candidate set contains all top-k results (by Lines 2–3). Let's assume that current $\mathcal{R}$ correctly contains top-k results from candidates verified so far. Let the next candidate graph be g and the maximum distance in $\mathcal{R}$ be d_m.

1) If $\delta = \mathsf{ged}(q, g) > \tau$, $\delta \geq d_m$ since $\delta > \tau = d_m - 1$. Hence, excluding g from the result set (in Line 9) does not affect the correctness of the result set $\mathcal{R}$. Here, we do not consider the case where $\tau = \tau_{max}$ because it is trivial to exclude g in this case.
2) If $\delta = \mathsf{ged}(q, g) \leq \tau$, the head element in $\mathcal{R}$ is removed (in Line 10) and g is inserted into $\mathcal{R}$ (in Line 11). Since the GED of the removed head element has the largest GED in $\mathcal{R}$ and $\delta \leq \tau < d_m$, replacing the largest element with g guarantees the correctness of $\mathcal{R}$.

Therefore, the algorithm correctly returns top-k results. □

The following lemma states that the algorithm keeps decreasing the overhead of GED computation while it finds result graphs.

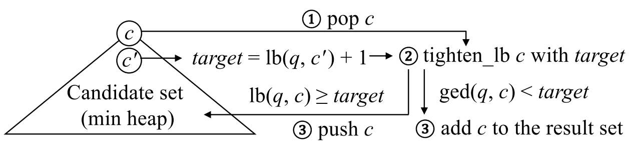


FIGURE 4. An overview of lower bound based top-k search.

Lemma 1: The GED threshold τ in Algorithm 2 decreases monotonically.

Proof: As shown in Theorem 1, the algorithm always replaces the largest GED from $\mathcal{R}$ with a smaller one. Hence, it guarantees that the largest GED d_m in $\mathcal{R}$ decreases monotonically. Since τ is either τ_{max} or $d_m - 1$ and $d_m \leq \tau_{max}$, τ is also decreases monotonically. □

B. GED LOWER BOUND-BASED TOP-k SEARCH ALGORITHM

The problem with the TopK-Basic algorithm proposed in the previous subsection is that it uses loose GED lower bounds in $\mathcal{C}$. Hence, it may push some graphs having high GED values into $\mathcal{R}$ and pop them from $\mathcal{R}$ later, which causes unnecessary GED computation. We may use a different feature-based lower bound function for the candidate set. However, the problem with existing feature-based lower bound functions is that they generally produce a loose lower bound. To address the problem, in this subsection, we develop an incremental GED computation algorithm to precisely obtain a desired GED lower bound and propose another top-k search algorithm based on incremental GED computation.

Figure 4 shows an overview of the lower bound-based top-k search. We keep a candidate set $\mathcal{C}$ which is organized into a min heap using GED lower bounds of candidates. Initially, the GED lower bound of each candidate is computed using the label set-based lower bound function. We generate result graphs by incrementally tightening the lower bounds of candidates as follows. ① We first pop the head element c from $\mathcal{C}$. Then, we look up the head c' of $\mathcal{C}$ and set a target lower bound to $\mathsf{lb}(q, c') + 1$, where $\mathsf{lb}(q, c')$ is the current lower bound of c'. ② We incrementally tighten the lower bound of c up to the target. ③ If we successfully tighten the lower bound of c, we push c into $\mathcal{C}$. We can fail to tighten the lower bound of c if and only if we obtain $\mathsf{ged}(q, c)$ which is less than the target (see Algorithm 3). In this case, c is added into the result set because it is a top-k result (we will formally state this observation in Lemma 2). The intuition behind this approach is that we do not always fully compute the GED of the head element of the candidate set.

Definition 6 (GED Computation Context): A GED computation context γ is a triple (Q, lb, ub), where Q is a priority queue containing mappings, lb is a GED lower bound, and ub is a GED upper bound.

We use the GED computation context in Definition 6 to stop and resume GED computation. For each candidate graph, the main idea is to compute GED until a desired lower bound is found. The A* search strategy finds an optimal path in the search tree by increasing the lower bound of each mapping.

If the minimum lower bound of the A* search is greater than or equal to a given target lower bound, therefore, we can immediately stop GED computation and keep the current context of GED computation for later use.

Algorithm 3 TightenLB($g_1, g_2, \gamma, target, \tau$)

```
input : $g_1$ and $g_2$ are input graphs,
        $\gamma$ is current GED computation context,
        $target$ is a target lower bound,
        and $\tau$ is a GED threshold
output: updated context $\gamma$
1  while $\gamma.Q \neq \emptyset$ do
2      $m \leftarrow \gamma.Q.\mathbf{head}()$;
3      $\gamma.lb \leftarrow \mathsf{lb}_\mathsf{M}(m)$;
4      if $|m| = |V(g_2)|$ then
5          $\gamma.ub \leftarrow \gamma.lb$;
6          return $\gamma$;
7      if $\gamma.lb \geq target$ then return $\gamma$ ;
8      $\gamma.Q.\mathbf{pop}()$;
9      foreach child node $m_c$ of $m$ do
10         if $\mathsf{lb}_\mathsf{M}(m_c) \leq \tau$ then $\gamma.Q.\text{push}(m_c)$;
11 $\gamma.lb \leftarrow \gamma.ub \leftarrow \tau + 1$;
12 return $\gamma$;
```

Algorithm 4 TopK-LB($\mathcal{D}, q, k$)

```
input : $\mathcal{D}$ is a graph database,
        $q$ is a query graph, and $k$ is an integer
output: $\mathcal{K}(q, k, \tau_{max})$
1  $\mathcal{C} \leftarrow$ min-heap initialized to be $\emptyset$;
2  foreach graph $g \in \mathcal{D}$ do
3      if $\mathsf{lb}_\mathsf{L}(q, g) \leq \tau_{max}$ then
4          $\mathcal{C}.\mathsf{push}(g, (\{\emptyset\}, \mathsf{lb}_\mathsf{L}(q, g), \tau_{max} + 1))$
5  $\mathcal{R} \leftarrow \emptyset$;
6  while $|\mathcal{R}| < k$ and $\mathcal{C} \neq \emptyset$ do
7      $(g, \gamma) \leftarrow \mathcal{C}.\mathsf{pop}()$;
8      if $\gamma.lb = \gamma.ub$ then $\mathcal{R} \leftarrow \mathcal{R} \cup \{g\}$;
9      else
10         $(-, \gamma') \leftarrow \mathcal{C}.\mathsf{head}()$;
11         $\gamma \leftarrow \mathbf{TightenLB}(q, g, \gamma, \gamma'.lb + 1, \tau_{max})$;
12         if $\gamma.lb \leq \gamma'.lb$ then
13             $\mathcal{R} \leftarrow \mathcal{R} \cup \{g\}$;
14             if index is available then
15                 $\mathcal{C} \leftarrow \mathsf{regenCandidates}(\mathcal{C}, \gamma.lb, \tau_{max})$;
16         else $\mathcal{C}.\mathsf{push}(g, \gamma)$;
17 return $\mathcal{R}$;
```

Algorithm 3 shows incremental GED computation for finding a lower bound greater than or equal to a given target bound. The algorithm looks up the first mapping from the priority queue (Line 2) and save the lower bound of the mapping in the current context γ (Line 3). If the current mapping is a full mapping, the algorithm makes the upper bound of the current context equivalent to the lower bound and return the context (Lines 4–6). By making both bounds the same, we can see that the exact GED is found. If the lower bound of the current mapping is not less than the target bound, the algorithm stops GED computation and returns the current context (Line 7). Otherwise, it removes the current mapping from the queue and expands child of current mappings (Lines 8–10). If the queue is empty, the algorithm saves $\tau + 1$ as the GED into the context and return the context (Lines 11–12).

Lemma 2: Given a top-k query q and a data graph g,

$$\mathsf{ged}(q, g) \leq \min_{g' \in \mathcal{C}} \mathsf{lb}(q, g') \implies g \in \mathcal{R},$$

where $\mathcal{C}$ is a candidate set containing currently remaining candidates and $\mathcal{R}$ is a result set.

Proof: Since $\mathsf{ged}(q, g) \leq \mathsf{lb}(q, g') \leq \mathsf{ged}(q, g')$ for any remaining candidate graph g', g has the lowest GED, which implies that g is an answer of the top-k query. □

Given the algorithm TightenLB that incrementally tightens the lower bound of a candidate graph, we develop a top-k search algorithm based on Lemma 2. Algorithm 4 encapsulates the top-k search algorithm. The algorithm first generates an initial candidate set using the label set-based lower bound function, where the candidate set contains pairs of (candidate graph g, GED context γ of g). The GED computation context of each candidate g consists of a priority queue having an empty mapping, which denotes the root node of the search tree, a label set-based lower bound $\mathsf{lb}_\mathsf{L}(q, g)$, and an upper bound $\tau_{max} + 1$ (Lines 2–4). It retrieves a candidate having the lowest lower bound from the candidate set (Line 7). If the lower bound is the exact GED of the candidate, the algorithm adds the candidate into the result set based on Lemma 2 (Line 8). Otherwise, it tightens the lower bound of the current candidate using Algorithm 3, where a target bound is set to the smallest lower bound + 1 of the remaining candidates (Lines 10–11). If the tightened lower bound is not greater than the target bound, we can assure that the exact GED of the current candidate is found while tightening its lower bound. Hence, the algorithm includes the current candidate into the results by Lemma 2, and regenerates the candidate set using our index (Lines 12–15). Otherwise, it pushes the candidate with its GED computation context into the candidate set (Line 16). The algorithm repeats the same procedure until either k results are found or the candidate set empties (Line 6), and finally it returns the result set (Line 17).

Theorem 2: Algorithm 4 correctly returns top-k result graphs.

Proof: It is trivial that the initial candidate set contains all results. Since the algorithm pushes a candidate into the result set only when the candidate has a GED less than or equal to the minimum lower bound of remaining candidates,

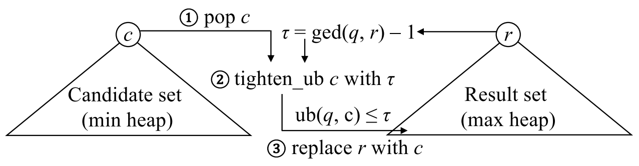


FIGURE 5. An overview of upper bound based top-*k* search.

by Lemma 2, the algorithm correctly collects top-*k* results. □

C. GED UPPER BOUND-BASED TOP-*k* SEARCH ALGORITHM

The problem with the lower bound approach is that it does not utilize GED thresholds. Since it collects exactly k results only, it relies only on τ_{max} in pruning the search space of GED. Although it can avoid full GED computation for many candidate graphs, it suffers from the large search space due to the absence of practical thresholds. To overcome the problem, in this subsection, we develop another top-*k* search algorithm that utilizes GED upper bounds. The basic and lower bound approaches require full GED computation for all top-*k* results. The main idea of this subsection is that some results can avoid full GED computation if their GED upper bounds are less than the maximum GED of the graphs the result set.

Figure 5 shows an overview of upper bound based top-*k* search. Again, we keep a min heap $\mathcal{C}$ for candidates and a max heap $\mathcal{R}$ for results and produce top-*k* result graphs as follows. ① We pop the head element c of $\mathcal{C}$. We set the threshold $\tau = \mathsf{ged}(q, r) - 1$, where r is the head element of $\mathcal{R}$. ② we tighten the upper bound of c down to τ. ③ If we successfully tighten the upper bound of c, we pop r from $\mathcal{R}$ and push c into $\mathcal{R}$. We can fail to tighten the upper bound of c if and only if we obtain $\mathsf{ged}(q, c)$ which is greater than τ (see Algorithm 5). In this case, c cannot be a result and we simply remove c. To correctly set the threshold τ, we compute the GED of the head element of $\mathcal{R}$ whenever necessary (see Algorithm 6).

To obtain a desired GED upper bound, we basically traverse the search tree in a depth-first manner and incrementally tighten an upper bound. To this end, we adapt the hybrid depth-first search approach, which we have proposed in our previous study [35]. Before we present our GED upper bound algorithm, we briefly summarize the hybrid search technique. We first find a mapping having the smallest GED at the *global* level, *i.e.*, from all mappings in the queue. Then, we expand child mappings of the mapping and keep going down to the next level of the current mapping. At each level, we select a mapping having the smallest GED at the level. We stop the downward traversal if we encounter a leaf node of the search tree or there is no mapping at the current level. Then, we select another mapping at the global level to start with.

Algorithm 5 shows an incremental GED computation algorithm to obtain an upper bound that is not greater than a target bound. The algorithm uses a special priority queue,[2] which

[2]Please refer to [35] for the details of the implementation of the queue.

Algorithm 5 TightenUB($g_1, g_2, \gamma, target$)

```
input : $g_1$ and $g_2$ are input graphs,
        $\gamma$ is current GED computation context,
        and $target$ is a target upper bound.
output: updated context $\gamma$
1  $lv \leftarrow global$;
2  while $\gamma.Q \neq \emptyset$ do
3      $m \leftarrow \gamma.Q.\mathsf{pop}(lv)$;
4      if $m = \mathsf{nil}$ or $\mathsf{lb_M}(m) > target$ then
5          $lv \leftarrow global$
6      else if $|m| = |V(g_2)|$ then
7          if $\mathsf{lb_M}(m) < \gamma.ub$ then
8              $\gamma.ub \leftarrow \mathsf{lb_M}(m)$
9          if $lv = global$ then
10             $\gamma.lb \leftarrow \gamma.ub$;
11             return $\gamma$;
12         else if $\gamma.ub \leq target$ then return $\gamma$;
13         else $lv \leftarrow global$;
14     else
15         foreach child node $m_c$ of $m$ do
16             if $\mathsf{lb_M}(m_c) \leq target$ then
17                 $\gamma.Q.\text{push}(m_c)$
18         $lv \leftarrow |m| + 1$;
19 $\gamma.lb \leftarrow \gamma.ub \leftarrow target + 1$;
20 return $\gamma$;
```

can return a mapping having the lowest GED lower bound at a specific level of the search tree. To this end, it receives a level in its pop method. When the level is it global, it returns a mapping having the lowest GED among all mappings in the queue. The algorithm first finds a mapping having the smallest GED from the global level (Line 1). In every iteration in the while loop (Lines 2–18), it basically expands the child mappings of the current mapping (Lines 15–17) and goes down to the next level of the current mapping (Line 18). If there is no mapping at the current level or the minimum GED lower bound of the current level is greater than the target bound, the algorithm stops searching downwards and selects a mapping at the global level in the next iteration (Lines 4–5). When it meets a leaf node (Line 6), it updates the current upper bound if necessary (Lines 7–8). Then, it returns if either the exact GED is found (Lines 9–11) or the current upper bound is not greater than the target bound (Line 12). Otherwise, it sets the level to the global level for the next iteration (Line 13). We remark that we use the target upper bound as a threshold for computing the lower bound of a mapping in the algorithm. Hence, the algorithm does not take a threshold as a parameter.

Algorithm 6 finds top-*k* results by incrementally tightening GED upper bounds. Like other algorithms, it generates an initial set of candidates using the label set-based lower bound function (Lines 1–3). The initial GED computation context of

Algorithm 6 TopK-UB($\mathcal{D}$, q, k)

```
input  : $\mathcal{D}$ is a graph database,
         $q$ is a query graph, and $k$ is an integer
output: $\mathcal{K}(q, k, \tau_{max})$
1  $\mathcal{C} \leftarrow$ min-heap initialized to be $\emptyset$;
2  foreach graph $g \in \mathcal{D}$ do
3      if $\mathsf{lb_L}(q, g) \leq \tau_{max}$ then
           $\mathcal{C}$.push($g$, ($\{\emptyset\}$, $\mathsf{lb_L}(q, g)$, $\tau_{max} + 1$));
4  $\tau \leftarrow \tau_{max}$;
5  $\mathcal{R} \leftarrow$ a max-heap of size $k$ with each value set to (nil, nil);
6  while $\mathcal{C} \neq \emptyset$ and $\mathcal{C}$.head().$\gamma.lb \leq \tau$ do
7      $(g, \gamma) \leftarrow \mathcal{C}$.pop();
8      if $|\mathcal{R}| < k$ then $\mathcal{R}$.push($g, \gamma$) ;
9      else
10         $\gamma \leftarrow$ TightenUB($q, g, \gamma, \tau$);
11         if $\gamma.ub \leq \tau$ then
12             $\mathcal{R}$.pop();
13             $\mathcal{R}$.push($g, \gamma$);
14     while $|\mathcal{R}| = k$ and $\mathcal{R}$.head().$lb \neq \mathcal{R}$.head().$ub$ do
15         $(g, \gamma) \leftarrow \mathcal{R}$.pop();
16         TightenUB($q, g, \gamma, 0$);
17         if $\gamma.ub \leq \tau_{max}$ then
18             $\mathcal{R}$.push($g, \gamma$);
19             if index is available then
20                 $\mathcal{C} \leftarrow$ regenCandidates($\mathcal{C}, \gamma.lb, \tau_{max}$);
21     if $|\mathcal{R}| = k$ then $\tau \leftarrow \mathcal{R}$.head().$ub - 1$;
22 return $\mathcal{R}$;
```

each candidate is the same as Algorithm 4. After generating the initial candidate set, it initializes the threshold τ to τ_{max} (Line 4) and make a result set $\mathcal{R}$ which is a max-heap of pairs of (result graph g, GED computation context γ), where the ordering of the pairs in the heap is determined by the GED upper bound in γ (Line 5). For each candidate graph and its GED computation context, the algorithm directly pushes the candidate to $\mathcal{R}$ if $\mathcal{R}$ is not filled (Line 8). Otherwise, it tightens the upper bound of the candidate (Line 10). If the tightened upper bound is less than the GED of the head of $\mathcal{R}$ (*i.e.*, the maximum GED value in $\mathcal{R}$), it removes the head of $\mathcal{R}$ and push the current candidate to the result set (Lines 11–13). If $\gamma.ub > \tau$, TightenUB computes the exact GED between q and g, and since the GED, which is $\gamma.ub$, is greater than τ, we can safely remove g. So, the algorithm do nothing in this case. In the remaining part of the algorithm (Lines 14–21), it basically updates the threshold using the maximum GED in the result set (Line 21). Since the GEDs of some graphs in $\mathcal{R}$ are not fully computed, the algorithm finds the maximum GED by incrementally computing the GED of the head graph of $\mathcal{R}$ (in the while loop of Line 14). To this end, it calls TightenUB by passing the target threshold 0, which forces TightenUB to compute an exact GED (Line 16). After tightening the head, it could not be the graph having the largest GED. In this case, it repeats the same procedure. In the meanwhile, it regenerates candidate graphs using our index if an index is available (Lines 19–20). Remark that the algorithm tightens the head only when the result set is filled. In case that the result set is just filled by the current candidate in Line 8, the GED of the head graph may exceed τ_{max} because we simply push the candidate into the result set without any computation. In this case, we simply remove the head graph from $\mathcal{R}$ (see the if statement in Line 17).

Lemma 3: The algorithm does not always compute exact GEDs of all graphs in the final result set.

Proof: Let an upper bound of a graph g in $\mathcal{R}$ be u, which is not the exact GED of g. Whenever the algorithm computes the exact GED of another graph in Line 17, consider the GED is not less than u. In this case, the algorithm does not need to compute the exact GED of g since $\mathsf{ged}(g, q) \leq u \leq m$, where m is the maximum GED of the result graphs in $\mathcal{R}$. □

Lemma 3 states that the algorithm can save some GED computation for finding top-k result graphs.

Lemma 4: The GED threshold τ in Algorithm 4 monotonically decreases.

Proof: The threshold τ is updated either (1) $\mathcal{R}$ is filled up for the first time or (2) the head graph in $\mathcal{R}$ is removed from $\mathcal{R}$. In the first case, τ does not increase since the algorithm collects graphs whose GED is not greater than τ_{max} and the initial value of τ is τ_{max}. In the second case, let the removed graph g and the new head graph g'. In this case, τ is updated from $\mathsf{ged}(q, g) - 1$ to $\mathsf{ged}(q, g') - 1$ (by Line 22). Since $\mathsf{ged}(q, g) \geq \mathsf{ged}(q, g')$, τ does not increase in this case. □

Theorem 3: Algorithm 6 correctly returns the top-k results.

Proof: For simplicity, we assume that we have more than k graphs whose GED is not greater than τ_{max}. Again, it is of certain that $\mathcal{C}$ contains all results. Let the graph having the maximum GED in $\mathcal{R}$ be g_{max}. Consider there is a graph $g \in \mathcal{C}$ such that $g \notin \mathcal{R}$ but $\mathsf{ged}(q, g) < \mathsf{ged}(q, g_{max})$. For the graph g, we have the following two cases.

1) The first case is that g is pushed into $\mathcal{R}$ and removed from $\mathcal{R}$ later. At the time g is removed from $\mathcal{R}$, g should be in the head of $\mathcal{R}$ and its GED is already computed (by Line 17). The algorithm assures that either the GED or an upper bound of every graph in $\mathcal{R}$ is not greater than $\mathsf{ged}(q, g)$. Hence, $g_{max} \notin \mathcal{R}$. g_{max} cannot be pushed into $\mathcal{R}$ after g is popped because τ is set to a value less than $\mathsf{ged}(q, g)$ (by Line 22) when g is the head of $\mathcal{R}$, and it is monotonically decreases by Lemma 4 (*i.e.*, $\mathsf{ged}(q, g_{max}) > \mathsf{ged}(q, g) > \tau$). This contradicts the hypothesis $g_{max} \in \mathcal{R}$.
2) The second case is that g is never pushed into $\mathcal{R}$. At the time the algorithm tests the graph g, it is obvious that $\mathsf{ged}(q, g) > \tau$. Since $\mathsf{ub}(q, g_{max}) \geq \mathsf{ged}(q, g_{max}) > \mathsf{ged}(q, g) > \tau$, $g_{max} \notin \mathcal{R}$ at the moment. Afterwards,

g_{max} cannot be inserted into $\mathcal{R}$ as τ decreases monotonically by Lemma 4. This contradicts the hypothesis again.

Therefore, both cases cannot be possible. □

D. HYBRID TOP-K SEARCH WITH AN INDEX

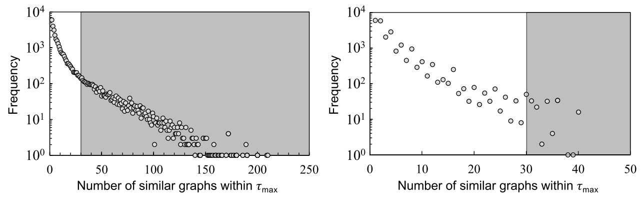
The upper bound approach remedies the problem with the basic and lower bound approaches. But there is a problem with the upper bound approach when it is used with an index based on pre-computed GEDs [37]. The upper bound approach tends to avoid GED computation for graphs having low GED values. Since the upper bound approach computes GEDs of graphs appeared in the head of the max-heap for the result set, GEDs produced by the approach are usually high. Due to the overhead of pre-computation of pairwise GEDs, however, the index omits those pairs having high GED values (see [37] for the details of the index). Therefore, the index is hardly beneficial to the upper bound approach. To remedy the problem, in this subsection, we propose a simple but efficient hybrid approach for top-*k* graph similarity search.

The main observation is that existing threshold-based algorithms are extremely fast for low GED thresholds (*e.g.* $\tau \leq 3$). Therefore, we perform a threshold-based similarity search for a low threshold τ_c, which is a tunable parameter. We then remove unpromising graphs from $\mathcal{D}$ by utilizing the result graphs of the threshold-based search. We finally perform top-*k* graph search. We remark that this hybrid approach can be used for all top-*k* algorithms proposed in this paper.

Algorithm 7 TopK-Hybrid($\mathcal{D}$, q, k)

```
input : $\mathcal{D}$ is a graph database,
        $q$ is a query graph, and $k$ is an integer
output: $\mathcal{K}(q, k, \tau_{max})$
1  $\mathcal{R}_\tau \leftarrow$ simsearch($\mathcal{D}, q, \tau_c$);
2  $\mathcal{C} \leftarrow \emptyset$;
3  foreach graph $g \in \mathcal{D}$ do
4      if lb_L($q, g$) $\leq \tau_{max}$ then $\mathcal{C} \leftarrow \mathcal{C} \cup \{g\}$;
5  foreach $g \in \mathcal{R}_\tau$ do
6      remove $g$ from $\mathcal{C}$;
7      $\mathcal{C} \leftarrow$ regenCandidates($\mathcal{C}$, ged($g, q$), $\tau_{max}$);
8  if $|\mathcal{R}_\tau| > k$ then $\mathcal{R}_\tau \leftarrow$ top $k$ results in $\mathcal{R}_\tau$;
9  $\mathcal{R} \leftarrow$ TopKsearch($\mathcal{C}, q, k - |\mathcal{R}_\tau|$);
10 return $\mathcal{R} \cup \mathcal{R}_\tau$;
```

Algorithm 7 shows the hybrid top-*k* search framework. It first performs a threshold-based similarity search with a pre-defined low threshold τ_c and saves the results in $\mathcal{R}_\tau$ (Line 1). Then, the algorithm generates a candidate set $\mathcal{C}$ using the label set-based lower bound function (Lines 2–4). it removes unpromising graphs from $\mathcal{C}$ using the index with the result graphs in $\mathcal{R}_\tau$ (Lines 5–7). Since it has all graphs whose GEDs are within τ_c in $\mathcal{R}_\tau$, it needs to find remaining $k - |\mathcal{R}_\tau|$ best graphs in the modified candidate set



(a) Frequency (AIDS) (b) Frequency (Chem)

FIGURE 6. Frequencies of numbers of similar graphs within τ_{max}.

(Lines 8–9). It finally returns the union of $\mathcal{R}_\tau$ and remaining $k - |\mathcal{R}_\tau|$ best graphs (Line 10).

IV. EXPERIMENTS

From the experimental study in this section, we shows that the proposed approaches improve the performance of top-*k* similarity search.

A. EXPERIMENTAL SETTING

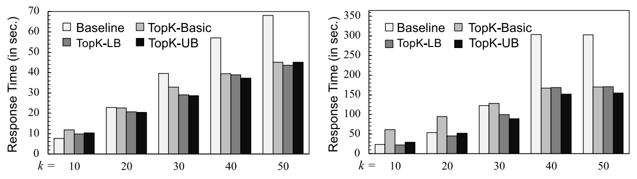
In our experiments, we used the following two real datasets, which were frequently used in related work [13], [14], [29]–[31], [33], [34].

- AIDS: The AIDS dataset consists of 42,689 graphs representing antiviral screen compound data. The average numbers of vertices and edges are 25.6 and 27.6, respectively. The numbers of distinct labels of vertices and edges are 62 and 3, respectively.
- PubChem: The PubChem dataset has 22,794 graphs representing chemical compounds. The average numbers of vertices and edges are 48.11 and 50.56, respectively. The numbers of distinct labels of vertices and edges are 10 and 3, respectively.

Similar to [13], [37], we set $\tau_{max} = 6$ as users are typically more inclined to search for similar graphs from graph databases [9]. One problem with top-*k* GED search is that many graphs have small number of similar graphs within τ_{max}. If we randomly select queries from the dataset, therefore, most queries return less than *k* results within τ_{max} for a moderate size of *k* (*e.g.*, $k = 30$). Figure 6 shows the frequencies of similar graphs in AIDS and PubChem datasets. Given a value *n* of the x-axis, the y-axis shows the number of graphs that have exactly *n* similar graphs within τ_{max}. Notice that the y-axis of the figure is log-scaled. Based on the observation, we reasonably assume that top-*k* queries are issued for those graphs having many similar graphs. For evaluating top-*k* search, by the assumption, we randomly selected 100 query graphs in the gray areas in Figure 6, which have at least 30 similar graphs. For each dataset, we evaluated the selected 100 queries with $k \in \{10, 20, 30, 40, 50\}$ and obtained the total numbers of results for different *k* values as shown in Table 1.

We reported aggregated results of 100 queries in the experiments.

Our GED computation algorithms were implemented in C++ based on the latest algorithm NassGED [37]. They

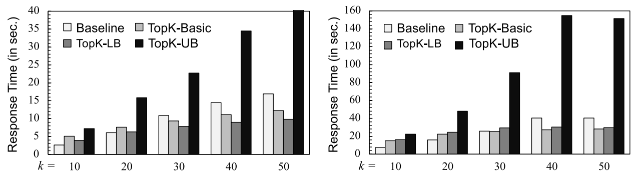


(a) Top-k Query Time (AIDS) (b) Top-k Query Time (Chem)

FIGURE 7. Response times of top-*k* queries.

TABLE 1. Total numbers of top-*k* results for 100 queries.

Dataset	k				
	10	20	30	40	50
AIDS	1000	2000	3000	3835	4400
Chem	1000	2000	3000	3411	3411



(a) Top-k Query Time (AIDS) (b) Top-k Query Time (Chem)

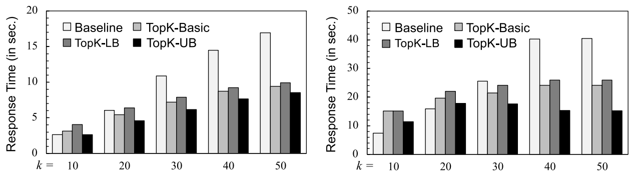
FIGURE 8. Response times of Top-*k* query with NassIndex.

were complied using GCC with the -O3 flag. All experiments were conducted in a single machine with an Intel core i7 and 32GB RAM running a 64-bit Ubuntu OS. Data graphs and indices were kept in memory.

B. EVALUATION OF TOP-K SEARCH

In this subsection, we evaluate top-k graph similarity search algorithms proposed in Section III. The purpose of evaluation is to compare the baseline algorithm with TopK-Basic, TopK-LB, and TopK-UB. We remark that we used the state-of-the art GED computation algorithm NassGED [37] for the baseline algorithm. We omitted the comparison with ParsK [31] as ParsK relies on an outdated indexing technique Pars.

Top-k evaluation results of alternative algorithms are reported in Figures 7 and 8. On the both datasets, the baseline algorithm exhibited good performance for small k values (*e.g.*, $k \in \{10, 20\}$). This was because most results had small GEDs w.r.t the query. For larger k (*e.g.*, $k \in \{30, 40, 50\}$), however, the baseline algorithm showed poor performance because it had to repeat GED computation of many graphs for each distinct threshold $\tau \in [0, \tau_{max}]$. The proposed algorithms were significantly faster than the baseline algorithm for $k \in \{30, 40, 50\}$. On the PubChem dataset, for example, TopK-UB was two times faster than the baseline algorithm when $k = 40$. TopK-Basic exhibited poor performance for small k values on the both datasets because it needed to push unnecessary graphs having large GEDs into the queue and pop them from the queue later. TopK-LB and TopK-UB reduced such unnecessary GED computation and outperformed TopK-Basic for small k values. On the PubChem dataset, for example, TopK-LB and TopK-UB was about 2 times faster than TopK-Basic for $k \leq 20$. For larger k values, TopK-Basic ran as fast as TopK-LB and TopK-UB as shown in the figures. For $k \geq 30$, TopK-UB was the best performer among the algorithms. This was because TopK-UB did not need to perform full GED computation for some result graphs.



(a) Top-k Query Time (AIDS) (b) Top-k Query Time (Chem)

FIGURE 9. Response times of the hybrid top-*k* search.

C. EVALUATION OF TOP-K SEARCH WITH AN INDEX

In this subsection, we evaluate top-k search algorithms using the NassIndex [37], which is based on the pre-computed GEDs of data graphs.

Figure 8 shows experimental results of top-k searches using NassIndex. TopK-Basic and TopK-LB were well suited with the index. For $k > 20$, these algorithms outperformed the baseline algorithm. On the AIDS dataset, for example, TopK-LB was about 2 times faster than the baseline algorithm when $k > 30$. For $k \leq 20$, however, the baseline algorithm exhibited very good performance. This was because the existing threshold-based similarity search techniques and GED computation algorithms ran extremely fast when a GED threshold is low [29], [37]. For smaller k values, top-k results tend to be found within a low GED threshold. Nonetheless, TopK-Basic and TopK-LB were still competitive with the baseline algorithm for the smaller k values. As we discussed in Section III-D, TopK-UB hardly utilized the index and exhibited very poor performance compared with other algorithms. It was about three times slower than the baseline algorithm for a large k value.

D. EVALUATION OF HYBRID TOP-K SEARCH

We applied the hybrid search framework to all the top-k algorithms with the parameter $\tau_c = 3$. Figure 9 shows the results. As shown in the figure, TopK-UB were surprisingly fast when we used the hybrid search framework. This is because the threshold-based search finds result graphs having low GEDs and it takes advantage of the index using those results to reduce the number of candidates. As a result, TopK-UB consistently outperformed all other algorithms on both datasets. For example, it was about 2 times faster than TopK-Basic and TopK-LB on the PubChem dataset for $k > 30$. The hybrid approach also improved the performance of TopK-Basic. However, the performance of TopK-LB was slightly degraded with the hybrid search framework. The reason is as follows. TopK-LB found the graph having the smallest GED first. TopK-LB could greatly reduce the number of candidate using the first result. Therefore, it hardly took advantage of the

TABLE 2. Improvement when using the index (in %).

Dataset	Algorithm	*k*				
		10	20	30	40	50
AIDS	Baseline	189	278	262	294	302
	TopK-Basic	297	316	355	351	378
	TopK-LB	142	224	268	321	339
	TopK-UB	289	344	364	385	428
Chem	Baseline	223	235	379	653	648
	TopK-Basic	303	382	498	591	603
	TopK-LB	45	105	311	550	557
	TopK-UB	156	191	403	892	912

threshold-based search of the hybrid framework and the overhead of the threshold-based search degraded the performance, which was negligible though.

The hybrid search framework are well designed to utilize the index so that the index effectively reduced the number of graphs requiring GED computation and significantly improved the performance. On the AIDS dataset, for example, TopK-UB with the index was up to 5 times faster than that without the index. On the PubChem dataset, it was up to 10 times faster. Table 2 summarizes the improvement of the all algorithms on the both datasets when using the index.

V. RELATED WORK

A. FILTERING AND INDEXING

Existing feature-based filtering techniques are categorized into two groups. The first group utilizes small fixed-length substructures to establish a necessary condition to meet a given threshold. *k*-AT [11], path-based *q*-gram [14], [30], c-star [27] and branch [15] are proposed in this group. They build an offline inverted index on fixed-length features extracted from data graphs. The second group utilizes large variable-length features to capture structural differences. To make a feature as large as possible, they partition each data graph into disjoint subgraphs and filter out dissimilar data graphs using the pigeonhole principle based on the observation in string similarity search techniques (*e.g.*, [38], [39]). Pars [13] and MLIndex [9] build an offline inverted index on partition features extracted from data graphs. Recently, pre-computed pairwise GED-based index are proposed to address the fundamental limitations of feature-based index [37].

B. GED COMPUTATION AND VERIFICATION

The most representative GED computation algorithm is A*-GED [36], which traverse the search tree using A* algorithm. BLP-GED [40] formulates the GED computation problem as a binary linear program. DF-GED [32], and CSI_GED [34] traverse the search tree in a depth-first manner. HGED [35] mixes both the depth-first and A* search techniques to improve GED computation. To verify candidate graphs in graph similarity search techniques, A*-GED is the most widely used. To reduce the search space, Inves [29] uses the bridge differences to obtain tight GED lower bounds, and AStar$^+$-LSa [33] removes blank vertices from a vertex mapping. NassGED integrates existing filtering techniques into GED computation to tighten the lower bound of a partial mapping.

C. TOP-K GRAPH SIMILARITY SEARCH

ParsK [31] is the first attempt to the top-*k* graph similarity search with graph edit distance constraints. Based on the observation of the top-*k* string similarity search technique proposed in [41], it built a hierarchical partition index structure to support top-*k* graph similarity queries. We differ by developing top-*k* search algorithms that exploit precise GED lower and upper bounds based on incremental GED computation. Our hybrid search framework are also designed to utilizes the latest index based on pre-computed pairwise GEDs of data graphs. An MCS-based top-*k* graph similarity search technique has been proposed in [28]. In [42], they formulated the problem of top-*k* representative queries on graph databases focusing on representativeness modeling.

VI. CONCLUSION

In this paper, we have proposed novel approaches for top-*k* graph similarity search. We utilizes GED lower and upper bounds to efficiently obtain the best *k* results for a query. To precisely compute a target GED bound, we have developed incremental GED computation algorithms. We have adapted a recent index structure based on pre-computed pairwise GEDs to top-*k* graph similarity search by designing a hybrid similarity search framework that are applied to our top-*k* search techniques. We have conducted extensive performance studies using real datasets to test the proposed techniques.

REFERENCES

[1] L. Liu, Y. Lu, and C. Y. Suen, "Retrieval of envelope images using graph matching," in *Proc. Int. Conf. Document Anal. Recognit.*, Sep. 2011, pp. 99–103.

[2] R. Wessel, S. Ochmann, R. Vock, I. Blümel, and R. Klein, "Efficient retrieval of 3D building models using embeddings of attributed subgraphs," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2011, pp. 2097–2100.

[3] Y. Zhang, X. Yang, H. Qiao, Z. Liu, C. Liu, and B. Wang, "A graph matching based key point correspondence method for lunar surface images," in *Proc. 12th World Congr. Intell. Control Autom. (WCICA)*, Jun. 2016, pp. 1825–1830.

[4] M. Neuhaus and H. Bunke, "A graph matching based approach to fingerprint classification using directional variance," in *Proc. Audio-Video-Based Biometric Person Authentication (AVBPA)*, 2005, pp. 191–200.

[5] E. Ozdemir and C. Gunduz-Demir, "A hybrid classification model for digital pathology using structural and statistical pattern recognition," *IEEE Trans. Med. Imag.*, vol. 32, no. 2, pp. 474–483, Feb. 2013.

[6] F. Niedermann, "Deep business optimization: Concepts and architecture for an analytical business process optimization platform," Ph.D. dissertation, Univ. Stuttgart, Stuttgart, Germany, 2015.

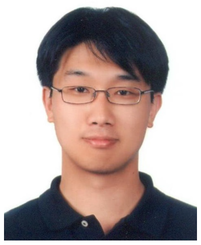
[7] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate comparison of binary executables," in *Proc. 2nd ACM SIGPLAN Program Protection Reverse Eng. Workshop (PPREW)*, 2013, pp. 1–10.

[8] O. Kostakis, J. Kinable, H. Mahmoudi, and K. Mustonen, "Improved call graph comparison using simulated annealing," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2011, pp. 1516–1523.

[9] Y. Liang and P. Zhao, "Similarity search in graph databases: A multi-layered indexing approach," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 783–794.

[10] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang, "Connected substructure similarity search," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2010, pp. 903–914.

[11] G. Wang, B. Wang, X. Yang, and G. Yu, "Efficiently indexing large sparse graphs for similarity search," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 3, pp. 440–451, Mar. 2012.
[12] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *Proc. IEEE 28th Int. Conf. Data Eng.*, Apr. 2012, pp. 210–221.
[13] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang, "A partition-based approach to structure similarity search," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 169–180, Nov. 2013.
[14] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, "Efficient processing of graph similarity queries with edit distance constraints," *VLDB J.*, vol. 22, no. 6, pp. 727–752, Dec. 2013.
[15] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao, "Graph similarity search with edit distance constraint in large graph databases," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage. (CIKM)*, 2013, pp. 1595–1600.
[16] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph indexing and approximate matching in large graphs," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1185–1194, Sep. 2010.
[17] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu, "TreeSpan: Efficiently computing similarity all-matching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2012, pp. 529–540.
[18] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognit. Lett.*, vol. 19, nos. 3–4, pp. 255–259, Mar. 1998.
[19] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel, "SAGA: A subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, Nov. 2006.
[20] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke, "Approximation of graph edit distance based on Hausdorff matching," *Pattern Recognit.*, vol. 48, no. 2, pp. 331–343, 2015.
[21] A. Fischer, K. Riesen, and H. Bunke, "Improved quadratic time approximation of graph edit distance by combining Hausdorff matching and greedy assignment," *Pattern Recognit. Lett.*, vol. 87, pp. 55–62, Feb. 2017.
[22] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Anal. Appl.*, vol. 13, no. 1, pp. 113–129, Feb. 2010.
[23] K. Gouda and M. Arafa, "An improved global lower bound for graph edit similarity search," *Pattern Recognit. Lett.*, vol. 58, pp. 8–14, Jun. 2015.
[24] K. Riesen and H. Bunke, "Improving bipartite graph edit distance approximation using various search strategies," *Pattern Recognit.*, vol. 48, no. 4, pp. 1349–1363, Apr. 2015.
[25] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, no. 3, pp. 353–362, May 1983.
[26] F. Serratosa, "Computation of graph edit distance: Reasoning about optimality and speed-up," *Image Vis. Comput.*, vol. 40, pp. 38–48, Aug. 2015.
[27] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 25–36, Aug. 2009.
[28] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng, "Answering top-*k* graph similarity queries in graph databases," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 8, pp. 1459–1474, Aug. 2020.
[29] J. Kim, D. Choi, and C. Li, "Inves: Incremental partitioning-based verification for graph similarity search," in *Proc. 22nd Int. Conf. Extending Database Technol. (EDBT)*, 2019, pp. 229–240.
[30] X. Zhao, C. Xiao, X. Lin, and W. Wang, "Efficient graph similarity joins with edit distance constraints," in *Proc. IEEE 28th Int. Conf. Data Eng.*, Apr. 2012, pp. 834–845.
[31] X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang, "Efficient structure similarity searches: A partition-based approach," *VLDB J.*, vol. 27, no. 1, pp. 53–78, Feb. 2018.
[32] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proc. Int. Conf. Pattern Recognit. Appl. Methods*, 2015, pp. 271–278.
[33] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, and D. Ouyang, "Speeding up GED verification for graph similarity search," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 793–804.
[34] K. Gouda and M. Hassaan, "CSI_GED: An efficient approach for graph edit similarity computation," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 265–276.
[35] J. Kim, "HGED: A hybrid search algorithm for efficient parallel graph edit distance computation," *IEEE Access*, vol. 8, pp. 175776–175787, 2020.
[36] K. Riesen, S. Fankhauser, and H. Bunke, "Speeding up graph edit distance computation with a bipartite heuristic," in *Proc. Mining Learn. With Graphs (MLG)*, 2007, pp. 1–4.
[37] J. Kim, "Boosting graph similarity search through pre-computation," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 951–963.
[38] J. Kim, C. Li, and X. Xie, "Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 169–180.
[39] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition-based method for similarity joins," *Proc. VLDB Endowment*, vol. 5, no. 3, pp. 253–264, Nov. 2011.
[40] J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam, "Exact graph edit distance computation using a binary linear program," in *Proc. Structural, Syntactic, Stat. Pattern Recognit. (S+SSPR)*, 2016, pp. 485–495.
[41] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng, "Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 519–530.
[42] S. Ranu, M. Hoang, and A. Singh, "Answering top-k representative queries on graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2014, pp. 1163–1174.



JONGIK KIM received the B.S. and M.S. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1998 and 2000, respectively, and the Ph.D. degree in computer engineering from Seoul National University, South Korea, in 2004. He was a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI), from 2004 to 2007. He was a Professor at Jeonbuk National University, from 2007 to 2021. He is currently a Professor with Chungnam National University. His research interests include semi-structured database, flash-memory data management, stream data processing, and similarity query processing. He was in the program committee of DASFAA 2020 and VLDB 2021 (industrial track).