An Efficient Graph Indexing Method

Xiaoli Wang [#], Xiaofeng Ding [*], Anthony K.H. Tung [#], Shanshan Ying [#], Hai Jin [*]

[#] *School of Computing, National University of Singapore, Singapore*
Email: {xiaoli, atung, shanshan}@comp.nus.edu.sg
[*] *School of Computer Science, Huazhong University of Science and Technology, P. R. China*
Email: {xfding, hjin}@hust.edu.cn

***Abstract*—Graphs are popular models for representing complex structure data and similarity search for graphs has become a fundamental research problem. Many techniques have been proposed to support similarity search based on the graph edit distance. However, they all suffer from certain drawbacks: high computational complexity, poor scalability in terms of database size, or not taking full advantage of indexes. To address these problems, in this paper, we propose SEGOS, an indexing and query processing framework for graph similarity search. First, an effective two-level index is constructed off-line based on sub-unit decomposition of graphs. Then, a novel search strategy based on the index is proposed. Two algorithms adapted from TA and CA methods are seamlessly integrated into the proposed strategy to enhance graph search. More specially, the proposed framework is easy to be pipelined to support continuous graph pruning. Extensive experiments are conducted on two real datasets to evaluate the effectiveness and scalability of our approaches.**

I. Introduction

Graphs are widely used to model complex entities in many applications including bio-informatics [1], chem-informatics [2], and pattern recognition [3], etc. Managing a large amount of graph data in these domains is a very challenging problem. It is essential to process graph queries efficiently. The classical query processing is often formulated as the (sub)graph isomorphism problem (e.g., [4], [5]). However, this kind of exact matching is too restrictive, as real objects are often affected by noises. Therefore, similarity search has become a basic operation in graph databases.

To manage graph data based on similarity, a number of similarity measures have been proposed in the literature (e.g., [3], [6], [7]). Among them, graph edit distance (GED) is a popular measure for evaluating graph similarity. Essentially, GED is the minimum number of edit operations required to transform one graph into another [8]. This motivates our studies in this paper on GED based graph similarity search problem, and our focuses on graph range query which is one of the most widely studied problem. This problem can be described as follows: *given a graph database* $D = \{g_1, g_2, \ldots, g_{|D|}\}$ *and a query graph* q*, find all* $g_i \in D$ *that are similar to* q *within a GED threshold denoted by* τ. Scanning the whole database D to compute the GED between q and each $g_i \in D$ is very expensive, due to the high complexity of GED computation, which is proved to be NP-hard [9]. Facing this difficulty, several existing works use upper and lower bounds of GED to prune off unlikely candidates. Although these methods allow more efficient bound computations, they still suffer from certain drawbacks. First, the most efficient approaches proposed in [9] and [10] are still very expensive. Second, they do not take full advantage of indexes, and require a full scan of the whole database. These bring in poor scalability in databases with a large number of graphs.

Facing these difficulties, it is natural to consider building an effective index structure to reduce complex computations. Our basic idea is to break graphs into sub-units (sub-unit is used as a small substructure derived from a graph in our paper), and to index them as filtering features using inverted lists. This idea of structure decomposing is similar to many existing methods for filtering sequences (using q-grams) [11], trees (using binary branches) [12], and graphs (using paths, trees or subgraphs to test for graph isomorphism) [4], [5], [13], [14], [15], [16]. Among these existing methods, filtering is done by performing exact matching on the sub-units and then inferring the edit distance bound through those sub-units that exactly match the queried structure. For example, in **κ-AT** method [14], the edit distance to a query graph is approximated by looking at the *κ-adjacent tree* patterns that match exactly to those patterns obtained from the query graph. These exact matches can be found by transforming adjacent trees into sorted sequences and then using hash-based indexing. Unfortunately, these existing works have several common disadvantages. Some of them require enumerating sub-units exhaustively with high space and time overhead, and some of them do not capture the attributes on vertices or edges which are continuous values on graphs and often suffer from poor pruning power. This is because filtering features in them need to be matched exactly with the features in the query graph.

To overcome the above drawbacks, we develop a novel sub-unit based index. In our approach, we decompose each database graph into sub-units, and each sub-unit contains a vertex and discriminative information about its neighboring vertices and edges. To avoid exhaustive enumerations, discriminative information for a sub-unit only contains the most neighboring information. To enhance filtering power, the decomposed sub-units in our method are compared against the sub-units generated from the query graph using the Hungarian algorithm [17]. Formulated as a bipartite matching problem, each sub-unit in database graphs can have only partial matching with each sub-unit in the query graph. The need arises to find highly similar sub-units that not only match exactly but also are highly similar to the sub-units from the query.

To support such functionality, we propose a novel query processing framework, called **SEGOS** (**SEarching similar**

Graphs based On Sub-units). In this framework, a two-level inverted index is constructed based on the decomposed sub-units. In the upper-level index, sub-units derived from the graph database are used to index all graphs using inverted lists. In the lower-level index, each sub-unit is further broken into multiple vertices and indexed in inverted lists. This two-level inverted index is preprocessed to maintain a global order for sub-units and graphs. This order ensures that sub-units or graphs can be accessed in increasing dissimilarity to a query sub-unit or graph. Given a query, our strategy follows a novel, cascaded framework: in the lower level, top-k similar sub-units to each sub-unit of the query can be returned quickly; in the upper level, graph pruning is done based on the top-k results from the lower level. Two search algorithms, based on the paradigm of the TA and the CA methods [18] are proposed for retrieving sub-units and graphs. By deploying the summation of sub-unit distances as the aggregation function, sorted lists can be easily constructed to guarantee the global orders on increasing dissimilarity for graphs. The CA based methods can enhance similarity search by avoiding access to graphs with high dissimilarity. It is clear that the top-k sub-units returned from the lower-level sub-unit search can be automatically used as the input to the upper-level graph search. Therefore, these two search stages are easy to be pipelined to support continuous graph pruning.

In summary, our main contributions are:

- We propose a novel two-level inverted index to speed up graph similarity search. The lower-level index is first used to efficiently find top-k similar sub-units. With the top-k results, the upper-level index is retrieved to construct a list of graphs that are sorted based on the similarity score.
- We propose a better search strategy following a cascade framework using the novel index. Search algorithms adapted from the TA and the CA methods [18] are proposed to improve efficiency by dramatically reducing accesses to sub-units and graphs with high dissimilarity.
- **SEGOS** can be applied to enhance existing works like **C-Star** [9] developed for evaluating graph edit distance using sub-units.
- **SEGOS** is easy to be pipelined into three processing stages: the lower-level top-k sub-unit search, the upper-level graph sorted list processing, and the dynamic graph mapping distance computation.

The rest of this paper is organized as follows. Section II provides related work and Section III introduces several preliminary concepts and filtering principles. Section IV illustrates how the two-level inverted index can be constructed based on the sub-unit decomposition of graphs. Then, our novel search strategy and the enhanced pipeline algorithm are introduced in Section V. We provide experimental results in Section VI and conclude the paper in Section VII. Finally, several proofs are shown in the Appendix.

II. Related Work

A. Graph Edit Distance

The GED problem has been extensively studied in many previous works, and a detailed survey can be found in [8]. GED is widely defined as the minimum number of edit operations needed to transform one graph into another. An edit operation can be an insertion, a deletion or a substitution of a vertex/edge. Algorithms for computing the GED can be classified into two classes: exact and approximate algorithms.

Exact algorithms calculate the exact GED between two graphs. Many optimal error-correcting subgraph isomorphism algorithms have been proposed, and A^*-based algorithms [19] are the most widely used ones. However, since GED computation is in NP-hard [20], these algorithms have exponential complexity and are only feasible for small graphs [21].

To avoid expensive GED computations, approximate algorithms are developed to compute lower and upper bounds of GED for graph filtering. In [10] and [22], GED computation is formulated as a BLP problem and a lower bound and an upper bound for GED can be computed with time complexity of $O(n^7)$ and $O(n^3)$ respectively. A recent method proposed in [9] computes both lower and upper bounds in cubic time, by breaking graphs into multisets of sub-units, and applying a novel algorithm to bound GED for filtering. However, these algorithms suffer from the scalability problem since they have to scan the whole database to compute bounds for each graph.

B. Graph Isomorphism Search

In graph isomorphism and subgraph isomorphism search, the aim is to find graphs that are either isomorphic or contain a subgraph that is isomorphic to the query graph. In this regard, the matching must be exact and there is no query relaxation of any form. Algorithms for isomorphism search includes **FG-index** [13], **TreePi** [16] and **Tree+Delta** [23]. These methods differ only in the features that they use for pruning candidates. These techniques however cannot be easily generalized to handle graph similarity search which requires certain amount of error tolerance in the matching graphs.

C. Graph Similarity Search

There is a great amount of literatures on graph similarity search. However, few developed indexes for searching by graph edit distance. Here we list these works based on the similarity function that they adopted.

Feature Counting Since graph alignment is NP-hard, various heuristical feature counting methods have been developed to compare graphs. **GraphGrep** proposed in [4] compares graphs by counting the number of matching paths between two graphs. Signatures are generated for all the paths in a graph up to a threshold length and inserted into an index to facilitate searching and counting of paths. In [24], features are generated by merging each node in a graph together with its neighbouring vertices information. Graph similarity is judged by counting the number of features that are sufficiently from both graphs and a B^+-tree is used to index the features of the graphs in the database. However, none of these methods can

guarantee that edit distance is minimized for graphs that are returned as query results.

Edge Relaxation Given two graphs g_1 and g_2, if c_{12} is the maximum common subgraph of g_1 and g_2, then the substructure similarity between g_1 and g_2 is defined by $\frac{|E(c_{12})|}{|E(g_2)|}$ and $1 - \frac{|E(c_{12})|}{|E(g_2)|}$ is called the edge relaxation ratio. In [15], the **gIndex** is developed to support similarity search by edge relaxation. The **gIndex** adopts discriminative frequent subgraphs as basic indexing structures and involves complex feature extraction for each query. Adopting edge relaxation as a similarity measure implicitly excludes node substitution as a graph edit operation [9] and is thus not general enough to handle search by edit distance.

Edit Distance As far as we know, there are few works that provide an index for searching by graph edit distance. The **C-Tree** [5] is one of such pieces of work. In **C-Tree**, an R-tree like index structure is used to organize graphs hierarchically in a tree. Each internal node in the tree summarizes its descendants by a graph closure. By approximating the graph edit distance against the graph closures that are stored in the internal nodes, **C-tree** tries to avoid accessing individual graphs that are too dissimilar based on the GED. A most recent work κ**-AT** [14] decomposes graphs into κ*-adjacent tree* patterns and indexes them using inverted lists. A lower bound is also proposed to filter out graphs that do not sharing sufficient common patterns with a query graph.

In our experiments, we will compare our approach against **C-Tree** and κ**-AT**.

III. Preliminaries and Principles

In this paper, we focus on a database D of undirected, simple graphs whose vertices are labelled. A graph is defined as a 4-tuple $g = (V, E, \Sigma, l)$, where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, Σ is a finite alphabet of vertex labels and $l : V \to \Sigma$ is a labelling function assigning a label to a vertex. Figure 1 shows an example of a graph database with five data graphs from g_1 to g_5. The size of a graph g, denoted by $|g|$, is the number of vertices in g, and other common notations used in our paper are shown in Table I.

TABLE I
Notations

Notation	Description
$deg(v)$	$\lvert\{u \lvert (u,v) \in E\}\rvert$, the degree of v
$\delta(g)$	$\max_{v \in V(g)} deg(v)$
$\delta(D)$	$\max_{g \in D} \delta(g)$
$\lambda(g_1, g_2)$	the edit distance between graphs g_1 and g_2
$\lambda(s_1, s_2)$	the edit distance between stars s_1 and s_2
$\mu(g_1, g_2)$	the mapping distance between g_1 and g_2
$\zeta(g_1, g_2)$	the overall score of g_2 obtained from g_1

A. Graph Decomposing Method

To estimate GED bounds effectively, we employ the idea proposed in [9] to decompose a graph into multiple sub-units like star. A star is defined as a labelled, single-level and rooted tree which can be represented by a 3-tuple $s = (r, L, l)$, where r is the root, L is the set of leaves and l is a labelling function. For each v_i in the graph, we construct a star $s_i = (v_i, L_i, l)$,

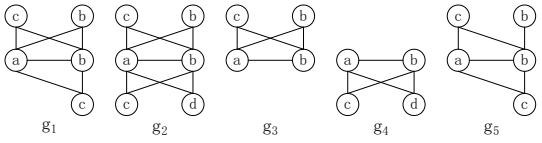


Fig. 1. A Sample Graph Database

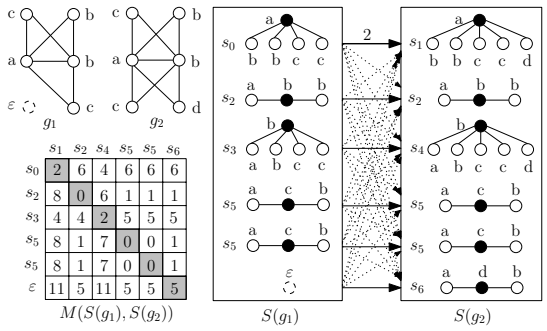


Fig. 2. Mapping Distance Computation Between g_1 and g_2

where L_i is the label set of v_i's neighbors. A graph g with $|g|$ vertices can be decomposed into a multiset of $|g|$ stars. In Figure 2, two graphs g_1 and g_2 are transformed into two star representations: $S(g_1)$ and $S(g_2)$. With this transformation, [9] has given a lemma to compute the star edit distance.

Lemma 1: (Star Edit Distance)(SED) Given two stars s_1 and s_2, the edit distance between them is computed as

$$\lambda(s_1, s_2) = T(r_1, r_2) + d(L_1, L_2)$$

where $T(r_1, r_2) = 0$ if $l(r_1) = l(r_2)$, otherwise $T(r_1, r_2) = 1$.

$$d(L_1, L_2) = \big||L_1| - |L_2|\big| + M(L_1, L_2)$$

$$M(L_1, L_2) = \max\{|\Psi_{L_1}|, |\Psi_{L_2}|\} - |\Psi_{L_1} \cap \Psi_{L_2}|$$

Ψ_L is the multiset of vertex labels in L. Assuming that the alphabet Σ of vertex labels has a total order, we can compute SED between two stars in only $\Theta(n)$ time, if Ψ_{L_1} and Ψ_{L_2} are sorted. For example, to compute the distance between s_0 of $S(g_1)$ and s_1 of $S(g_2)$ in Figure 2, it is obvious that $T(r_1, r_2) = 0$, for $l(r_1) = l(r_2) = a$. Having $|L_1| = 4$, $\Psi_{L_1} = \{b, b, c, c\}$, $|L_2| = 5$ and $\Psi_{L_2} = \{b, b, c, c, d\}$, we get that $\lambda(s_0, s_1) = 0 + |4 - 5| + 5 - 4 = 2$.

Definition 1: (Mapping Distance) Given two star representations $S(g_1)$ and $S(g_2)$ with the same cardinality, assume $P : S(g_1) \to S(g_2)$ is a bijection, then the distance between them is defined as

$$\mu(g_1, g_2) = \min_P \sum_{s_i \in S(g_1)} \lambda(s_i, P(s_i))$$

The computation of mapping distance is equivalent to finding an optimal mapping between two star representations. Zeng *et al.* [9] constructs a weighted matrix for each pair of stars from two graphs, and applies the Hungarian algorithm [17] to get the optimal solution in cubic time. The weight between two stars is the SED. If two graphs are of different size, ϵ node is inserted for normalization. In Figure 2, the bottom left matrix $M(S(g_1), S(g_2))$ is the weight matrix between star sets $S(g_1)$ and $S(g_2)$. Cells in gray denote the

optimal matching between $S(g_1)$ and $S(g_2)$, i.e. $\mu(g_1, g_2) = 2 + 0 + 2 + 0 + 0 + 5 = 9$. To have a clear view, two sets of stars are shown on the right, and the optimal matching is marked with solid arrows.

As shown in [9], the mapping distance can be used to bound GED effectively, and a lower bound $L_m(g_1, g_2)$ and a upper bound $U_m(g_1, g_2)$ can be derived as below.

Lemma 2: Suppose $\mu(g_2, g_1)$ is the mapping distance between g_1 and g_2. Then,

$$L_m(g_1, g_2) = \frac{\mu(g_2, g_1)}{\max\{4, [\max\{\delta(g_1), \delta(g_2)\} + 1]\}} \leq \lambda(g_1, g_2)$$

Lemma 3: Suppose P is a mapping between $V(g_1)$ and $V(g_2)$ obtained from Hungarian algorithm when computing $\mu(g_1, g_2)$. Then $U_m(g_1, g_2) = C(g_1, g_2, P) \geq \lambda(g_1, g_2)$, where $C(g_1, g_2, P)$ is the cost to transform g_1 to g_2 using the mapping P [10].

This paper employs the above decomposing method to build the index, hereafter, a sub-unit refers to a star structure, and SED can also denote the sub-unit edit distance. The sub-unit is also represented as a sequence of labels for simplicity. For example, in Figure 5, "s_0: abbcc" represents the sub-unit s_0 as its label sequence of "abbcc". As shown above, computing mapping distance takes cubic time on graph size. The existing filtering strategy proposed in [9] suffers from poor scalability as it has to scan a large graph database, and compute mapping distance between each data graph and the query graph for pruning. Facing this problem, two ways can be developed to enhance the graph search: using dynamic mapping distance computation and a better filtering strategy.

B. Dynamic Mapping Distance Computation

To reduce complex mapping distance computations, this paper proposes a novel computing method as below.

Theorem 1: Given two graphs g_1 and g_2 and their sub-unit representations $S(g_1)$ and $S(g_2)$. Suppose $S'(g_2)$ contains several sub-units derived from g_2 and $S'(g_2) \subseteq S(g_2)$. Then we have

$$\mu(S(g_1), S'(g_2)) \leq \mu(g_1, g_2)$$

In Figure 3, $M(S(g_1), S'(g_2))$ is a different cost matrix defined for computing $\mu(S(g_1), S'(g_2))$. For the ϵ sub-unit, we define its distance to any existing sub-unit s_i in $S(g_1)$ as 0 instead of $\lambda(s_i, \epsilon)$. We apply the Dynamic Hungarian [25] to find the minimum cost and matching on $M(S(g_1), S'(g_2))$. After that, the incremental part for computing full $\mu(g_1, g_2)$ uses the original definition of cost matrix with $\lambda(s_i, \epsilon)$. With this definition, it is clear that $\mu(S(g_1), S'(g_2)) \leq \mu(S(g_1), S(g_2))$.

This property allows us to compute bounds for the GED between two graphs even if only a subset of a graph's sub-units are available. If $\mu(S(q), S'(g))$ is sufficiently large, there is no need to compute the bound based on the full set of sub-units between graphs.

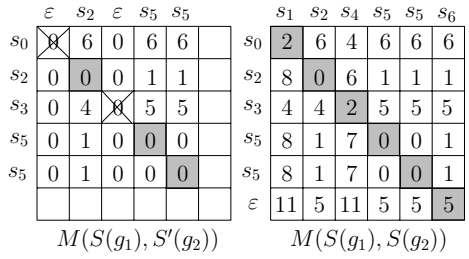


Fig. 3. An Example for Computing $\mu(S(g_1), S'(g_2))$

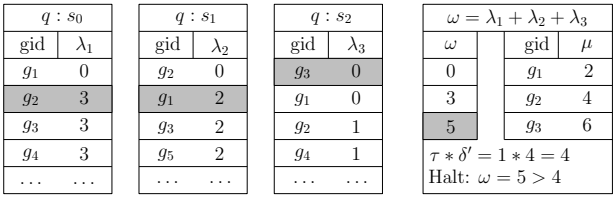


Fig. 4. A Simple Example for CA-based Filtering Strategy

C. Filtering Strategy

To reduce the complex computations of GED bounds, it is natural for us to consider a more efficient filtering strategy. In this paper, we propose a novel search strategy based on the paradigm of the TA and the CA methods proposed in [18][1].

Figure 4 shows a simple example that helps to illustrate our CA-based filtering strategy for range query on the graph database in Figure 1. Consider the three score sorted lists on the left which consist of sub-units from q. Each entry in the lists records the graph identity g_i and the SED between the corresponding sub-unit in g_i and the sub-unit of q. We use the summation of SEDs as the score aggregation function and assume that for an unseen graph g, $\mu(g, q) \geq \omega$ where ω is the summation of SEDs seen currently (we also call this assumption as *monotonic assumption*). Then, in this example, the search algorithm halts when $\omega = \lambda_1 + \lambda_2 + \lambda_3 = 5 > \tau * \delta' (= 4)$. Hereafter, we denote $\delta' = \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$ where D' is the set containing all unseen graphs. Here, g_4 and g_5 are filtered out without computing their mapping distances, since their values of μ are no less than ω. From Lemma 2, for an unseen graph g with $\mu(g, q) > \tau * \delta'$, we have $\lambda(g, q) > \tau$ and g can be safely filtered out.

Accordingly, our search strategy must overcome the following challenges: 1) An effective indexing structure is needed for constructing the score-sorted lists. 2) Since graphs in score indexing lists are sorted according to their SEDs to the sub-unit of the query, an efficient search algorithm must be developed to obtain sub-units that are highly similar to the query sub-unit. 3) Score indexing lists must be sorted to guarantee the correctness of halting based on monotonic assumption of the TA or the CA based search strategy.

[1]As far as we know, such TA and CA based methods had never been previously applied for matching complex structures like sequences (using qgrams) [11], trees (using binary branches) [12], or graphs [4], [5], [15], [16], [13], [14]. This is because all these previous methods simply use the number of exact matches among the sub-units to bound the edit distance and compute the exact edit distance for all candidate that pass through the filter. For cases in which such filters are not effective (eg. range query with a very loose edit distance threshold), our approach here provide an elegant way to avoid computing the exact edit distance for large number of candidates.

IV. INDEX BUILDING IN SEGOS

To handle the above problems, a two-level inverted index based on the sub-unit decomposition is constructed.

A. The Upper-Level Inverted Index

Given a database with graphs and their sub-unit representations, an inverted index can be constructed. For example, given a database of g_1 and g_2 in Figure 2, we can construct an inverted index for all sub-units derived from data graphs in this database as shown in Figure 5. This index is made up of two main parts: an index for all distinct sub-units from the given database, and an inverted list below each unit. Here, the sub-units are sorted in alphabetical order. Each entry in the inverted lists contains the graph identity and the frequency of the corresponding unit. All lists are sorted in increasing order of the graph size. In Figure 5, since $|g_1| < |g_2|$, g_1 is located before g_2 in the lists.

With this index, it is very convenient to fetch out graphs that contain a given sub-unit. Then, given a query, if we can quickly access sub-units that are highly similar to the sub-units from the query in increasing dissimilarity, graphs can also be accessed in globally increasing dissimilarity to the query. Therefore, a lower-level index for sub-units is built.

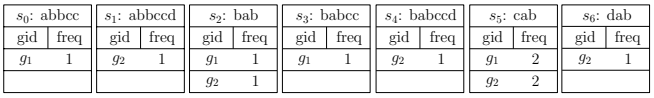
B. The Lower-Level Inverted Index

We construct the lower-level inverted index for all sub-units based on vertex labels. A sub-unit is broken into a multiset of labels excluding its root label. For example, s_0 in Figure 5 is decomposed into $\Psi_{s_0} = \{b, b, c, c\}$. With this decomposition, it is easy for us to build an inverted index for sub-units based on labels. The index also contains two components: a label index in increasing order and inverted lists below labels recording the sub-unit identities and the frequencies of corresponding labels in the leaves of the sub-unit. Entries in each list are first grouped based on the leaf size of $|\Psi_s|$ and then sorted in decreasing frequencies within each group. For example, in Figure 6, the list below label b has three groups sorted in increasing leaf size. In the first group, s_2, s_5 and s_6 all have leaf sizes of 2. In the second group, s_0 and s_3 have leaf sizes of 4. In the last group, s_1 and s_4 have leaf sizes of 5. In each group, frequencies are sorted decreasingly. Considering in the last group, the frequency of s_1 is 2 which is larger than that of s_4 (=1). Moreover, the last list without a label index is an extended list storing the sizes of all sub-units in increasing leaf size.

With this index, it is convenient to search similar sub-units for a query sub-unit based on the sub-unit edit distance. We will present the details of the search algorithm in next section.

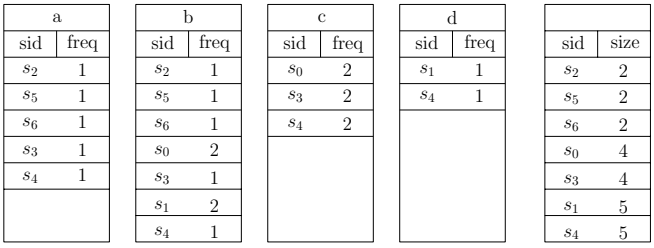
C. Index Maintenance

While employing a more complex two-level index in this paper, it is worth noting that both these levels are inverted indexes and the features like sub-units and labels can be easily generated from individual graphs. As observed in [26], such inverted indexes can be implemented either with a special purpose inverted list engine or in commercial relational database systems. For the latter case, we will be building on various query optimization, concurrency control techniques that had been developed over the years [2] to update our indexes. For the earlier case, we will describe our operations here.



s_0: abbcc		s_1: abbccd		s_2: bab		s_3: babcc		s_4: babccd		s_5: cab		s_6: dab	
gid	freq	gid	freq	gid	freq	gid	freq	gid	freq	gid	freq	gid	freq
g_1	1	g_2	1	g_1	1	g_1	1	g_2	1	g_1	2	g_2	1
				g_2	1					g_2	2		

Fig. 5. Upper-Level Inverted Index for Graphs



a		b		c		d			
sid	freq	sid	freq	sid	freq	sid	freq	sid	size
s_2	1	s_2	1	s_0	2	s_1	1	s_2	2
s_5	1	s_5	1	s_3	2	s_4	1	s_5	2
s_6	1	s_6	1	s_4	2			s_6	2
s_3	1	s_0	2					s_0	4
s_4	1	s_3	1					s_3	4
		s_1	2					s_1	5
		s_4	1					s_4	5

Fig. 6. Lower-Level Inverted Index for Sub-units

There are essentially seven kinds of updates for graph data: (1) inserting a new graph, (2) deleting a data graph, (3) inserting an edge into a graph, (4) deleting an edge of a graph, (5) inserting a new vertex into a graph, (6) deleting a vertex from a graph, and (7) relabelling a vertex in a graph. To support these updates, four kinds of operations occur in our two-level inverted index:

1) $Op1$: Inserting or deleting the graph information into an inverted list below a sub-unit in the upper-level index.
2) $Op2$: Inserting or deleting the sub-unit information in an inverted list below a label in the lower-level index.
3) $Op3$: Create a new list for a new generated unit, or delete a unit from the upper-level index when its list is empty.
4) $Op4$: Create a new list for a new label, or delete a label from the lower-level index when its list is empty.

Assuming that the inverted index is properly implemented and optimized over a B-tree (or B^+-tree) [28], all the operations above will take at most $O(logN)$ page accesses. Building on these operations, our index can easily support various types of updates as below: 1) Inserting a graph needs us to decompose this graph into a multiset of sub-units, and then perform $Op1$. For a new generated unit, we will perform $Op3$ followed by $Op2$. If a new label is detected, perform $Op4$. 2) Deleting a graph requires us to remove all the graph information in the upper-level index. 3) Inserting or deleting an edge of a graph affects two sub-units. Therefore, the graph information below two original sub-units is removed and they are inserted into two new lists. Furthermore, sub-unit information is also updated in the lower-level index. 4) Inserting or deleting a vertex only affects one unit. The operations are similar to update 3). 5) Relabelling a vertex will affect the sub-unit rooted by this vertex and those sub-units rooted by its neighbors. These operations are similar to updates 3) and 4).

[2]This approach is also adopted in [27] where qgrams are stored in a relational database to support approximate string join.

V. Searching in SEGOS

Based on the proposed two-level inverted index, we develop **SEGOS**, a cascade query processing framework, to employ the **dynamic mapping distance computation** and the **filtering strategy** proposed in Section III to enhance the graph search. The novel framework contains two search steps: the top-k sub-unit search and the graph similarity search. As shown in Figure 7, in the lower level, top-k similar sub-units to each sub-unit of the query can be returned quickly by using the TA search algorithm; in the upper level, graph pruning is done based on the top-k results from the lower level. To support continuous graph pruning, the CA graph search algorithm can be further divided into two stages: sorted list processing and dynamic graph mapping distance computation. In this step, sub-units for each data graph can be output with round-robin scan through the score sorted lists, and used as input to run dynamic mapping distance computation for seen data graphs with the query. This section will show how this framework work for graph pruning, and TA, CA, and DC denote the three stages in our framework.

A. Searching for Top-K Sub-units

Given a query graph q, we need to efficiently find sub-units that are highly similar to each sub-unit from q in the TA stage. A full scan of the database to compute the sub-unit edit distance (SED) between each sub-unit and a query sub-unit can be very expensive. In this paper, we propose a *top-k sub-unit searching algorithm* based on TA method [18]. The TA filtering strategy can help to avoid access to sub-units with high dissimilarity to the query sub-unit, but the score-sorted lists constructed need to guarantee the correctness of the TA halting monotonic assumption. From Definition 1 in Section III-A, the SED between a query sub-unit s_q and any database sub-unit s_i can be represented as below:

$$\lambda(s_q, s_i) = \begin{cases} T(r_q, r_i) + 2 * |L_q| - (\psi + |L_i|), & if\ |L_i| \leq |L_q| \\ T(r_q, r_i) - |L_q| - (\psi - 2 * |L_i|), & if\ |L_i| > |L_q| \end{cases} \tag{1}$$

where $\psi = |\Psi_{L_q} \cap \Psi_{L_i}|$ denotes the common leave labels between s_q and s_i. From the above two equations, if we ignore the difference between the roots of sub-units $T(r_q, r_i)$, the SED increases when the value of $(\psi + |L_i|)$ or $(\psi - 2 * |L_i|)$ decreases. Therefore, two aggregation functions can be derived as $\omega = 2 * |L_q| - (\psi + |L_i|)$ and $\omega = -|L_q| - (\psi - 2 * |L_i|)$ and we need to construct two sets of score-sorted lists to apply the above two functions. That means, sub-units with leaf sizes no more than $|L_q|$ and those with leaf sizes larger than $|L_q|$ must be processed separately.

Fortunately, the lower-level index can be used to conveniently construct these two sets of score-sorted lists. We know that each lower-level index list has been grouped increasingly according to sub-units' leaf sizes. Maintaining a leaf size array denoted by AL pointing to positions of all leaf size groups, it is easy to find the position that after which the leaf sizes are larger than that of the query sub-unit in $O(log|AL|)$ time. Since each group has been sorted based on decreasing frequencies, all groups within a leaf size range can be directly merged into one list in $O(|AL| \times |SL|)$ time ($|SL|$ is the maximum length of all leaf size groups). Generally, $|AL|$ is a constant smaller number compared to $|SL|$, so the merge complexity

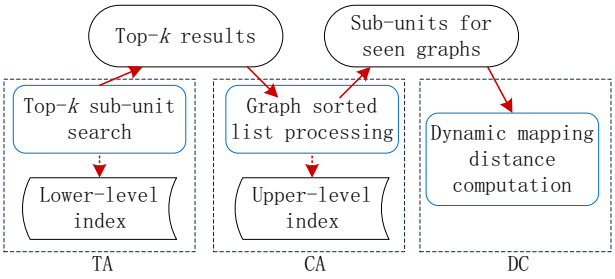


Fig. 7. The Cascade Search framework

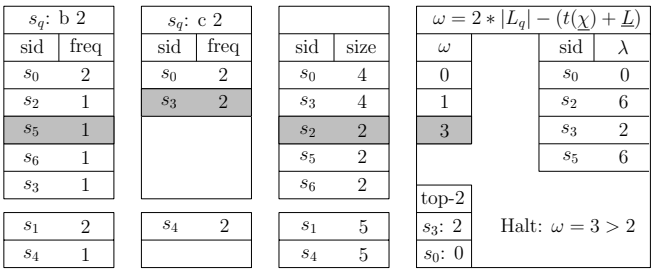


Fig. 8. A Top-k Sub-unit Searching Example for $s_q = abbcc$

can be considered to be linear. The detail of the merge function is given in Algorithm 1.

Figure 8 shows the score-sorted lists obtained for $s_q = abbcc$ using the index in Figure 6. The query sub-unit s_q has leaf labels b and c. For the label b, we fetch out the inverted list under "b" in Figure 6. Then a size bound larger than $|L_q| = 4$ can be found in position 5 pointing to $(s_1, 2)$. From here, groups with leaf sizes no larger than 4 are merged into one single list of $\{(s_0, 2), (s_2, 1), (s_5, 1), (s_6, 1), (s_3, 1)\}$. Another list of $\{(s_1, 2), (s_4, 1)\}$ with leaf size larger than 4 is also formed. Similarly, two lists below c are formed as $\{(s_0, 2), (s_3, 2)\}$ and $\{(s_4, 2)\}$. The size list is also split into two parts, but the one with leaf sizes no larger than 4 should be reversely accessed decreasingly.

Given the score-sorted lists for s_q, suppose s_q has m distinct leaf labels with frequencies of $(c_1, c_2, \ldots, c_m)$. We compute $\psi = t(\chi)$ as the number of common leaf labels between s_q and any s_i.

$$t(\chi) = \sum_{j=1}^{m} \min\{c_j, \chi_j\} \tag{2}$$

where χ_j represents the frequency corresponding to s_i in the j^{th} score list of s_q. If s_i does not appear in that list, $\chi_j = 0$. Accordingly, given m distinct label sorted lists and one size sorted list for s_q, the steps of our searching algorithm are:

1) Do sorted access in a round-robin schedule to each sorted list. If a sub-unit s_i is seen, compute $\lambda(s_q, s_i)$. Maintain a queue of top-k sub-units with the lowest λ values.
2) For each label list SL_j, let $\underline{\chi}_j$ be the frequency last seen under sorted access. Let $\underline{L}$ be the size last seen in the size list. For the score-sorted lists with smaller size, $\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L})$. Otherwise, $\omega = -|L_q| - (t(\underline{\chi}) - 2 * \underline{L})$. If the top-$k$ values are at most equal to ω, then halt. Otherwise, go to step 1.

The detail is shown in Algorithm 2 and its correctness is shown in Appendix A. Figure 8 shows an example to search top-2 similar sub-units to $s_q = abbcc$ on score-sorted lists containing sub-units with lower leaf sizes. Sub-units are accessed in a round-robin way from the list below label b to the size list. SED is calculated for each sub-unit seen and a top-2 queue is maintained. Algorithm halts in the positions with gray shadows because $\omega = 2 * 4 - (1 + 2 + 2) = 3 \geq 2$, where 2 is the maximum value in the top-2 queue. Obviously, the top-2 results are returned without access to s_6.

Algorithm 1 Merge function

Require: A list L and a size index array A of length n
Ensure: A score-sorted list SL

```
end ← true, max ← 0, p ← 0
initialize an array A' with values of A;
while true do
    for i = 0 to n − 1 do
        if A'[i] == A[i + 1] then
            continue;
        end ← false;
        if max < L[A'[i]].freq then
            max ← L[A'[i]].freq;
            p ← i;
    if end == true then
        break;
    SL.push_back(L[A'[p]]);
    A'[p] + +;
```

Algorithm 2 Top-k sub-unit searching algorithm

Require: m sorted lists SL and 1 size list L for s_q; low
Ensure: The top-k similar sub-units

```
top − k ⇐ ∅;
for all sorted lists with j = 1 . . . m + 1 do
    if j ≤ m then
        s_id ⇐ SL_j.getNext();
        χ_j ⇐ s_id.freq;
    else
        s_id ⇐ L.getNext();
        L ⇐ s_id.size;
    if s_id is not seen before then
        calculate λ(s_q, s_id);
        if |top − k| < k then
            Maintain top − k and continue;
        if λ(s_q, s_id) < max{λ|λ ∈ top − k} then
            Maintain new top − k;
    if low is true then
        ω = 2 ∗ |L_q| − (t(χ) + L);
    else
        ω = −|L_q| − (t(χ) − 2 ∗ L);
    if ω ≥ max{λ|λ ∈ top − k} then
        return top − k;
return top − k;
```

B. Score-Sorted Lists Construction for Graph Search

The above algorithm provides us an efficient way to return highly similar sub-units to a query sub-unit. Then graph score sorted lists can be easily formed by combining a set of lists fetched from the upper-level index below the corresponding top-k results.

Given a query graph q, for each query sub-unit s_q, its top-k queue is returned from the lower-level TA stage. Then, for each sub-unit s_i in the queue, a graph inverted list indexed by s_i can be directly fetched from the upper-level index. Therefore, k graph lists will be returned for each query sub-unit s_q. Later the k graph lists will be split into two segments: those with graph sizes larger than $|q|$, and those not. Segments within a graph size range will be combined into one group. Within each group, graphs are naturally ordered in terms of SEDs according to the top-k values. Furthermore, in the group with smaller sizes, the segments having SED larger than $\lambda(s_q, \epsilon)$ are discarded. Since the upper-level index lists have been sorted by increasing graph sizes, finding size range position takes $O(log|GL|)$ time ($|GL|$ is the maximum size of all graph size index arrays).

For example, given a query $q = g_1$ in Figure 1, the top-2 similar sub-units for the query sub-unit s_5 are s_5 and s_2, in Figure 9. Then

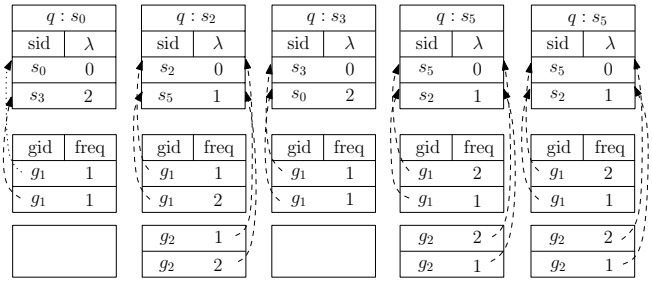


Fig. 9. The Sorted Lists for $q = g_1$

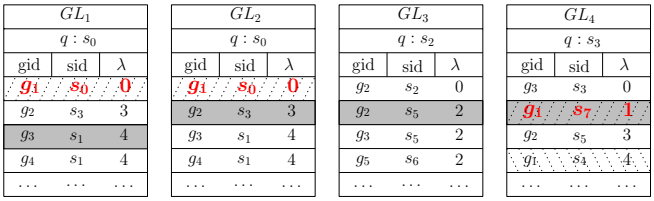


Fig. 10. An Example for Computing CA Bounds

two graph lists indexed by s_5 and s_2 are extracted from the upper-level index in Figure 5: $\{(g_1, 2), (g_2, 2)\}$ and $\{(g_1, 1), (g_2, 1)\}$. Since the query is of size 5, each graph list is divided into two segments. For example, the list below s_5 is split into $\{(g_1, 2)\}$ with $|g_1| \leq 5$ and $\{(g_2, 2)\}$ with $|g_2| > 5$. Similarly, the list below s_2 is split into $\{(g_1, 1)\}$ and $(g_2, 1)\}$. After that, segments $\{(g_1, 2)\}$ and $\{(g_1, 1)\}$ with smaller sizes are combined into one list $\{(g_1, 2), (g_1, 1)\}$. Since $\lambda(s_5, s_5) = 0 \leq \lambda(s_5, s_2) = 1$, $(g_1, 2)$ is located before $(g_1, 1)$. In Figure 9, if a graph is fetched from a list below a sub-unit, it is connected to that sub-unit using a dashed arrow.

Based on the constructed graph score sorted lists, the CA stage accesses sub-units for data graphs using a round-robin scan. Using the summation of SEDS as an aggregation function, the halting condition and several aggregation bounds can be directly derived.

C. Bounds from Aggregation Function

Given m score lists of a query graph q, we compute the overall score of a graph g having been seen, denoted by $\zeta(q, g)$ as

$$\zeta(q, g) = t'(\chi_1, \ldots, \chi_m) = \sum_{j=1}^{m} \chi_j$$

χ_j is a local minimum SED of graph g having been seen below the j^{th} list of q. The computation of χ_j is as below.

Definition 2: Let $Se_j = \{e_1, \ldots, e_x\}$ including all SEDs of a graph g below the j^{th} list. Then the corresponding χ_j of g is computed as

$$\chi_j = \min_{e_i \in Se_j} \{e_i\}$$

Generally, if Se_j is empty, $\chi_j = 0$.

Example 1: As shown in Figure 10, a graph g_1 has been seen blow three lists GL_1, GL_2, and GL_4 of q (this can be seen in cells with slashes in the figure). We have its local minimum SED in each list as $\chi_1 = 0$, $\chi_2 = 0$, and $\chi_4 = 1$. Since Se_3 is empty, $\chi_3 = 0$. Therefore, the overall score of g_1 obtained from q is $\zeta(q, g_1) = 0+0+0+1 = 1$.

Suppose $l(g) = \{l_1, \ldots, l_y\} \subseteq \{1, 2, \ldots, m\}$ is a set of known lists of g having been seen below q. Let $\underline{\chi}(g)$ be the multiset of distances corresponding to the distinct sub-units of g last seen.

- **Aggregation Lower Bound** denoted by $L_\mu(q, g)$ is obtained by substituting the missing lists $j \in \{1, 2, \ldots, m\} \setminus l(g)$ with $\underline{\chi}_j$ (the distance last seen under the j^{th} list) in $\zeta(q, g)$. That is, $\chi_j = \underline{\chi}_j$ when Se_j is empty.
- **Aggregation Upper Bound** denoted by $U_\mu(q, g)$ is computed as $U_\mu(q, g) = t'(\underline{\chi}(g)) + \overline{\chi} * (max\{|q|, |g|\} - |\underline{\chi}(g)|)$.

Here, $\overline{\chi} = max_{s \in S(q) \cup S(g)}\{\lambda(s, \epsilon)\}$. As shown in Example 1, we have $\zeta(q, g_1) = 1$. Suppose the cells with gray shadows are the current positions accessed, the distances last seen below the lists of q is $\{4, 3, 2, 1\}$. To replace the unseen value χ_3 of g_1 with $\underline{\chi}_3 = 2$, $L_\mu(q, g_1) = \zeta(q, g_1) + \underline{\chi}_3 = 1 + 2 = 3$. It can be seen from Figure 10, the distinct sub-unit set of g_1 last seen is $\underline{\chi}(g_1) = \{s_0, s_7\}$. Suppose $|g_1| = 3$, a remaining sub-unit s_4 has not been accessed (the cell with back slash), and the maximum distance between sub-units in q and g_1 is $\overline{\chi} = max_{s \in S(q) \cup S(g_1)}\{\lambda(s, \epsilon)\} = 11$. To substitute the value of unseen sub-units from g_1 to q with $\overline{\chi}$, $U_\mu(q, g_1) = t'(\underline{\chi}(g_1)) + \overline{\chi} * (max\{|q|, |g_1|\} - |\underline{\chi}(g_1)|) = 0 + 1 + 11 * (4 - 2) = 23$.

Theorem 2: Let g_1 and g_2 be two graphs, the bounds obtained as above satisfy the following:

$$\zeta(g_1, g_2) \leq L_\mu(g_1, g_2) \leq \mu(g_1, g_2) \leq U_\mu(g_1, g_2)$$

The proof of this theorem can be seen in Appendix B.

D. Graph Pruning Algorithm

Our graph pruning algorithm is a CA-based algorithm. Its filtering strategy is similar to the top-k sub-unit search, while using a different aggregation function. It also employs the above aggregation bounds and dynamic mapping distance computation algorithm to reduce the graph mapping distance computation. A simple example of graph sorted lists processing can be seen in Figure 4 in Section III. The main steps of our CA-based algorithm are shown as blow. Given m sorted lists for a graph query q and a threshold τ,

1) Perform sorted retrieval in a round-robin schedule to each sorted list. At each depth h of lists:
   - Maintain the lowest values $\underline{\chi}_1, \ldots, \underline{\chi}_m$ encountered in the lists. Maintain a distance accumulator $\zeta(q, g_i)$ and a multiset of retrieved sub-units $S'(g_i) \subseteq S(g_i)$ for each g_i seen under lists.
   - For each g_i that is retrieved but unprocessed, if $\zeta(q, g_i) > \tau * \delta_{g_i}$ ($\delta_{g_i} = \max\{4, [\max\{\delta(q), \delta(g_i)\} + 1]\}$), filter out the graph; if $L_\mu(q, g_i) > \tau * \delta_{g_i}$, filter out the graph; if $U_\mu(q, g_i) \leq \tau * \delta_{g_i}$, add the graph to the candidate set. Otherwise, if $\mu(S(q), S'(g_i)) > \tau * \delta_{g_i}$, filter out the graph. If all the above bounds are useless, run the Dynamic Hungarian algorithm to obtain $L_m(q, g_i)$ and $U_m(q, g_i)$ for filtering.
2) When a new distance is updated, compute a new ω. If $\omega = t'(\underline{\chi}) = \sum_{j=1}^{m} \underline{\chi}_j > \tau * \delta'$, then halt. Otherwise, go to step 1.

The details of CA-based range query are shown in Algorithm 3 and its correctness is shown in Appendix C. Obviously, the CA method performs the pruning test only for every h index entries accessed, and aggregation bounds can be accumulated in constant time. For data graphs having very similar sub-units to the query, aggregation upper bounds are small enough to output them as candidates; while for those having very dissimilar sub-units, aggregation lower bounds are large enough to prune them. Therefore, aggregation bounds take negligible constant time for early filtering.

As described before, our whole search strategy includes the TA, CA, and DC stages. Previously, we have provided the complexity analysis of some steps. Here, we present a more complete analysis. First, in the TA stage, constructing sorted lists for each queried sub-unit is decided by the merge time, which takes $O(|AL| \times |SL|)$ time as shown in Section V-A, and a simple study of the TA search complexity is in [18]. The worst case of this step takes $O(kd|SL|)$ (k is the value of top-k results and d is the average degree of sub-units) time for sorted access and takes $O(Nlogk)$ (N is the number of graphs accessed) time for maintaining a heap. Second, in the CA stage, graph sorted lists are combined by top-k results. As stated in Section V-B, it takes $O(log|GL|)$ time. Third, in the DC stage, the CA search complexity is similar to the TA search. We compute the dynamic mapping distance in $\Theta(n^3)$ (n is the average size of graphs) time and do the sub-unit difference operation in $O(logn)$ time.

Algorithm 3 CA-based range query algorithm

Require: m sorted lists GL for q, τ and h
Ensure: All g_i s.t. $\lambda(q, g_i) \leq \tau$

```
candidate ⇐ ∅; flag ⇐ false;
for all sorted lists GL_j with j = 1 ... m do
    g_id ⇐ GL_j.getNext();
    χ_j ⇐ g_id.dist;
    maintain the distance accumulator ζ(q, g_id);
    maintain the multiset for seen sub-units S'(g_id);
    if scandepth%h == 0 then
        for all g_id seen and unprocessed do
            if ζ(q, g_id) > τ * δ_{g_i} then
                filter it out and continue;
            if L_μ(q, g_id) > τ * δ_{g_i} then
                filter it out continue;
            if U_μ(q, g_id) > τ * δ_{g_i} then
                further compute other bounds;
            if μ(S(q), S'(g_id)) > τ * δ_{g_i} then
                filter it out and continue;
            Filtering with L_m(q, g_id) and U_m(q, g_id);
    if ω = t'(χ) > τ * δ' then
        flag ⇐ true and break;
if flag ≠ true then
    post process the remaining graphs not appeared;
```

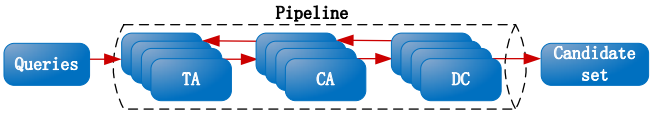


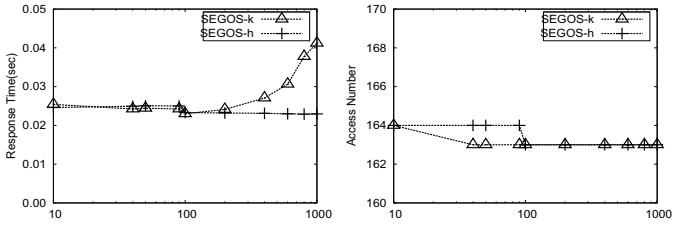
Fig. 11. The Pipeline of Query Processing Framework

E. Query Processing Pipelining Algorithm

As shown in Figure 7, the above graph pruning algorithm can be divided into two stages: graph sorted list processing (CA) and dynamic graph mapping distance computation (DC). In step 1, we only use aggregation bounds, and output accessed graphs with seen sub-unit multisets to the separate DC stage for mapping distance computations. The main advantage of our query processing framework lies in reducing the complex GED bounds computations by avoiding accessing highly dissimilar graphs.

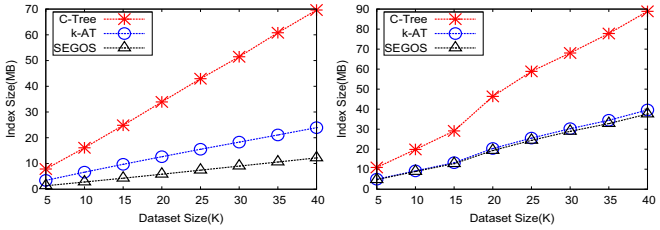
Moreover, the proposed approaches can be further improved by pipelining. It is easy to pipeline the whole query processing framework in Figure 7 into three consecutive stages: TA, CA, and DC. As shown in Figure 11, given graph queries, they are first decomposed into multiple sub-unit multisets. Then, each sub-unit is input to the TA stage to get its top-k similar sub-units. The output of top-k results for each query graph is fed to the input of the CA stage for building the graph score sorted lists. After that, the CA stage retrieves graph score sorted lists for each query graph in a round-robin schedule. When CA halts or the ends of all lists have been reached, all the accessed sub-units for seen data graphs are arranged to be the input of the DC stage. In the DC stage, we compute partial mapping distance when the accessed sub-units for the data graph are more than 50%, and run dynamic computation for graphs which have been processed but not filtered out. Moreover, there is no need to further return top-k results in the TA stage when the CA halts.

The pipelining algorithm can avoid parameter tunings for the k value in the TA stage and the h value in the CA stage. The k value can be fixed as a small number like 20, and h is not needed since the CA stage does not control the dynamic computations. To reduce dynamic computation overhead, we run partial matching only when more than 50% sub-units of a graph have been accessed. Further consideration will be illustrated in Section VI. To differentiate our algorithms, hereafter **SEGOS** means our original CA search algorithm without pipeline, and **SEGOS-Pipeline** refers to the pipelining one.



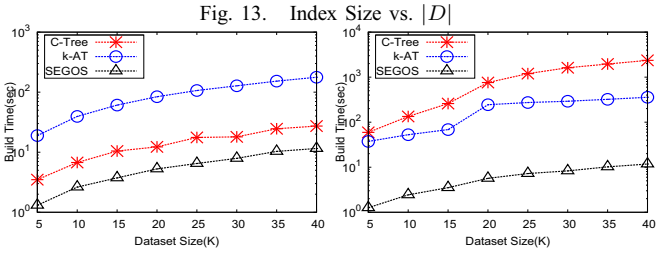
(a) Response Time (b) Access number

Fig. 12. Sensitivity Test on AIDS Dataset



(a) on AIDS Dataset (b) on Linux Dataset

Fig. 13. Index Size vs. $|D|$



(a) on AIDS Dataset (b) on Linux Dataset

Fig. 14. Construction Time vs. $|D|$

VI. Experimental Study

In this section, we compare our methods with two state-of-the-art approaches **C-Tree**[5] and κ**-AT**[14] on two real datasets. **SEGOS** was compiled with gcc 4.4.3 in Red hat Linux Operating System, and all experiments were run on a server with Quad-Core AMD Opteron(tm) Processor 8356, 128GB memory, running RHEL 4.7AS. In the experiments, we randomly selected 20 graphs from the dataset as query graphs and present the average result.

AIDS Dataset. This dataset is a DTP AIDS Antiviral Screen chemical compound dataset, published by National Cancer Institute (http://dtp.nci.nih.gov/docs/aids/aids_data.html). This dataset has been widely used in many existing works [4], [5], [13], [14], [15], [16]. It consists of $42,687$ chemical compounds, with an average of 46 vertices. Compounds are labelled with 63 unique vertex labels.

Linux Dataset. *Program Dependence Graph (PDG)* is an ideal static representation of the data flow and control dependency within a procedure, with each vertex assigned to one statement and each edge representing the dependency between two statements. PDG is widely used in software engineering for clone detection, optimization, debugging, etc (e.g., [29], [30]). Here, we use CodeSurfer 2.1pl to generate the PDG dataset (http://www.grammatech.com). First we maximize the configuration of the Linux kernel and then dump the *Program Dependence Graph* using CodeSurfer 2.1pl with strict error limitation. This Linux kernel procedure dataset has in total 48,747 graphs, with an average of 45 vertices. The graphs are labelled with 36 unique labels, representing the roles of vertices in the procedure, such as "declaration", "expression", "control-point", etc.

Taken from different applications, AIDS is a sparse database with near normal size distribution while Linux is that with near uniform size distribution. Table II presents five major parameters used in our experiments, including their descriptions and values (with default values in bold). Hereafter, the default values will be used in all the experiments if not particularly indicated.

TABLE II
Parameter Settings

Parameter	Description	Value
k_s	k value for the TA stage	$10, 20, .., \mathbf{100}, 200, .., 1000$
h	h value for the CA stage	$10, 20, .., 100, 200, .., \mathbf{1000}$
$\|D\|$	dataset graph number	5K,10K,15K,**20K**,25K,30K,35K,40K
$\|q\|$	query vertex number	$10, 20, 30, 40, 50, 60, 70, 80$
τ	distance threshold	$0, 2, 4, 6, 8, \mathbf{10}, 12, 14, 16, 18, 20$

A. Sensitivity Study

We first conduct a series of parameter sensitivity analysis on our non-pipeline algorithm **SEGOS**. The impact of different parameters on the access number and the response time is presented. Access number here is defined as the number of graphs accessed to compute mapping distances for a query graph.

In Figure 12 **SEGOS-k** and **SEGOS-h** respectively correspond to the sensitivity of parameters k_s and h. It can be seen that, when k_s is small, the lists of top-k sub-units are quite short. In this case, our algorithm filters out few graphs after scanning through the lists, and the dynamic algorithm has to be applied on more sub-units for the remaining graphs. As k_s increases from 50 to 100, the lists of top-k sub-units become larger for the CA stage and more graphs are pruned off early. When k_s is larger than 100, there is little change in the access number, since CA has reached its halting conditions.

Meanwhile, when h is small, the chance of retrieving the whole set of sub-units is low. As h grows, more sub-units will be seen, allowing more graphs to be pruned without being fully accessed. As such, both the response time and the access number decrease as h increases from 10 to 100. The response time will be stable when h is large enough to hit the halting condition of the CA stage.

We exclude results on the Linux dataset since it shows very similar trends. However, the sensitive values for k_s are larger in this dataset because its size distribution is more uniform than the AIDS dataset. Generally, our method achieves good performance by setting k_s as about 1% of the total sub-unit number and h as in the order of a few hundred. Without loss of generality, we simply use the default values in Table II for both two real datasets in the following experiments.

B. Index Construction Performance

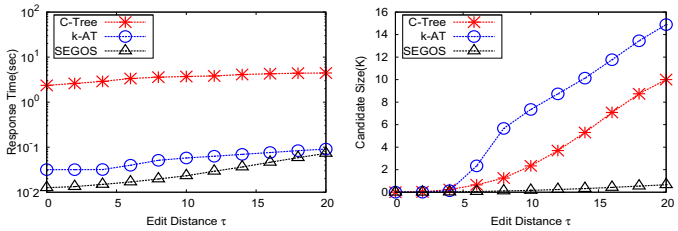
In this subsection, we evaluate index construction performance of **SEGOS**, κ**-AT** and **C-Tree** w.r.t the dataset size. To build the index for κ**-AT**, we first conduct a sensitivity test and find that κ**-AT** performs the best by setting $\kappa = 2$ on both datasets.

Figure 13 and 14 show the index size and index construction time on both datasets, with $|D|$ varying from $5K$ to $40K$. We can see that **SEGOS** needs the shortest construction time and takes up the smallest space among all the three index structures, for it is sufficient for **SEGOS** to build two simple inverted indexes with only one dataset scan. For the other two index strategies, we find that κ**-AT** has to scan the dataset up to κ times to build a κ-layer feature table for each graph, and index these elements in all feature tables, and **C-Tree** uses one complex R-Tree like index structure, making it the most expensive one in index construction and the largest one in index size. In summary, **SEGOS** outperforms κ**-AT** and **C-Tree** in terms of index size and build time.

C. Query Performance

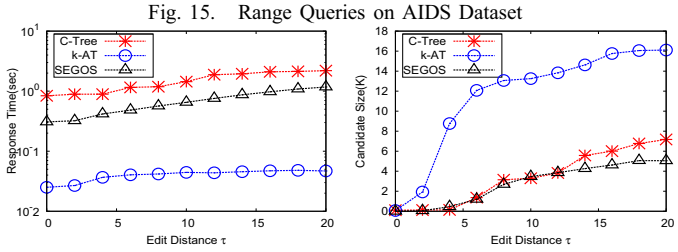
We next investigate the performance of our range query algorithms compared against those of **C-Tree** and κ**-AT**. Figure 15 and 16 show the results of range queries with τ varying from 0 to 20, $|D| = 20K$.

From Figure 15 we can see that **SEGOS** always returns the smallest number of candidates while incurring shortest response time. On the AIDS dataset, it outperforms κ**-AT** by up to two orders of magnitude in terms of candidate set size, and beats **C-Tree** in terms of filtering efficiency by two orders of magnitude.



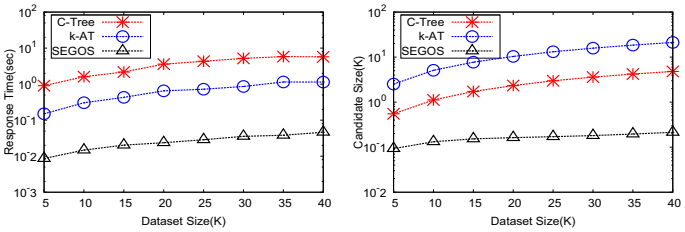
(a) Response Time vs. τ (b) Candidate Size vs. τ

Fig. 15. Range Queries on AIDS Dataset



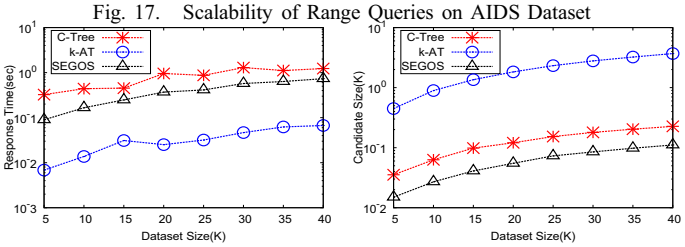
(a) Response Time vs. τ (b) Candidate Size vs. τ

Fig. 16. Range Queries on Linux Dataset



(a) Response Time vs. $|D|$ (b) Candidate Size vs. $|D|$

Fig. 17. Scalability of Range Queries on AIDS Dataset



(a) Response Time vs. $|D|$ (b) Candidate Size vs. $|D|$

Fig. 18. Scalability of Range Queries on Linux Dataset

Figure 16(b) shows that κ-**AT** has the poorest filtering ability, although it is the fastest one when it comes to filter as shown in Figure 16(a). Even when τ is as small as 6, κ-**AT** gives 800 more candidates than **SEGOS**. Here we can conclude that although the simplistic filtering adopted by κ-**AT** gives it higher efficiency, it's filtering power is however much weaker than the other two. In a more concrete term, κ-**AT** is fast simply because it does not do much filtering. Compared to **C-Tree**, it is clear that **SEGOS** dominates **C-Tree** w.r.t response time and candidate size. The superiority of **SEGOS** becomes more significant when τ grows larger. We can see that **C-Tree** returns $2K$ more candidates than **SEGOS** which is about $1/10$ of the entire dataset size.

There are two reasons for the best result of our algorithm in Figure 15(a). First, the number of accessed graphs for mapping distance computation is much smaller on the AIDS dataset than the Linux dataset. Second, the randomly selected queries include graphs with smaller sizes or with high dissimilarity to most graphs in the AIDS dataset which can be fast completed in **SEGOS**.

Note that candidates verification using the GED is an extremely expensive process (NP-Hard). If we take into consideration that the GED computation for each of acquired candidates (of average size 40) is in thousand of seconds, then the extra candidates generated by κ-**AT** (eg. 800) will cost an additional hundreds of thousands seconds. From our observation, the total response time including filtering time and verification time increases as the candidate number becomes larger. This result is also presented in several existing works. As such, it makes sense to sacrifice a little more time to filter out as many candidates as possible, as **SEGOS** does.

D. Scalability Study

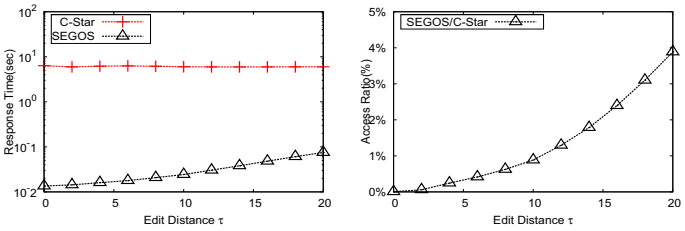
We conduct two groups of experiments to evaluate the scalability of our algorithm in terms of the dataset size over two real datasets.

Figure 17 and 18 illustrate the scalability of the algorithms with respect to the dataset size $|D|$, ranging from $5K$ to $40K$. Here, we choose $\tau = 2$ for Linux dataset, and $\tau = 10$ for the AIDS dataset. This is because there are many similar graphs in the Linux dataset and a small τ is sufficient to show the difference in performance (**SEGOS** also performs better than the others when τ is large). On the contrary, since the AIDS dataset does not have that many similar graphs, a larger τ is more appropriate to reveal the difference.

Figure 17 shows that **SEGOS** outperforms the other two algorithms over the entire range of dataset sizes. Furthermore, as the dataset size grows, **SEGOS**'s response time increases only from $8ms$ to $40ms$, which is only 0.1% that of **C-Tree** and 50% that of κ-**AT**. On the Linux dataset, **SEGOS** is still the most effective one in candidate filtering and costs moderate response time. From these two figures, we can see that **SEGOS** is better than κ-**AT** on the AIDS dataset, and **C-Tree** on both the AIDS and the Linux datasets. Though **SEGOS** needs more time than κ-**AT** on the Linux dataset, it filters out more candidates than κ-**AT**, by two orders of magnitude.

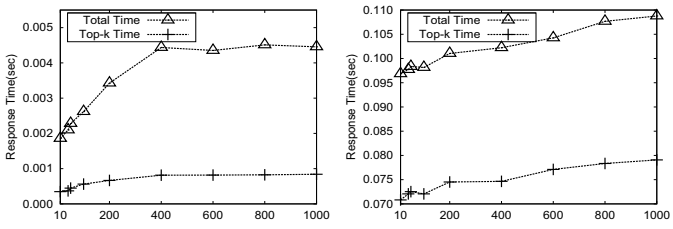
E. Effects of *SEGOS* on *C-Star*

To show how much **SEGOS** can enhance **C-Star**, we conduct a set of experiments to see the response time and the access ratio of **SEGOS**, compared to **C-Star**. $20K$ graphs are randomly selected from two real datasets, and 10 graphs are extracted as queries. Figure 19 shows that **SEGOS** can enhance **C-Star** by dramatically reducing mapping distance computations by two orders of magnitude on average. We also investigate queries which have a mass of similar graphs in the database, since in this special case our method may degrade to the linear case of **C-Star** while taking extra overhead for the TA stage. However, we find that the overhead can be negligible, even in the worst case, this overhead takes less than 0.1% of the overall response time. A result showing the overhead with various k_s values is presented in Figure 20.



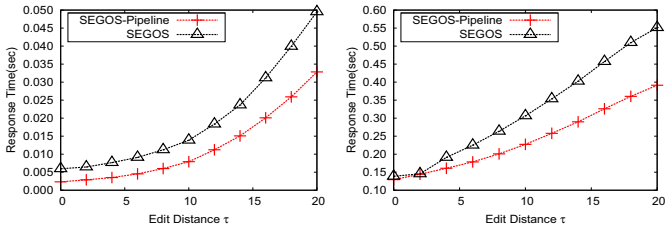
(a) Response Time vs. τ (b) Access Ratio vs. τ

Fig. 19. Quality of **SEGOS**



(a) on AIDS Dataset (b) on Linux Dataset

Fig. 20. Overhead Testing of Top-k Sub-unit Search on Range Queries



(a) on AIDS Dataset (b) on Linux Dataset

Fig. 21. Effects of Pipeline on **SEGOS**

F. Effects of the Pipelining Algorithm

We also implement a simple pipelining algorithm for **SEGOS**, denoted by **SEGOS-Pipeline**. Since this algorithm is implemented with multi-threading, we only compare it to our non-pipeline method to study its effects. In our implementation, we dispatch two parallel threads to respectively run the TA and the CA stage, and two threads to run the DC stage respectively for two parallel parts: partial matching computations and sub-unit multiset difference computations. With this, the overhead of **SEGOS** can be reduced by parallel processing. **SEGOS-Pipeline** fixes the k_s value to be 20, and CA feeds its output into the DC stage when it finishes processing sorted lists constructed by top-20 results from TA. Therefore, the h parameter can be removed. Figure 21 shows one group of results on range queries, varying τ from 0 to 20. In this experiment, we randomly select $20K$ data graphs and 20 query graphs from two real datasets. The results show that the pipelined algorithm can further speed up the graph search. The access number for queries does not exceed 700, which is not significant enough to show the high enhancement. However, the trend shows that with τ increasing, the enhancement also becomes higher as the access number becomes larger.

VII. Conclusions and Future Work

In this study, we investigated an important problem of GED based similarity graph search. Different from previous works, we propose **SEGOS**, an efficient indexing and pipeline query processing framework based on sub-units. A two-level inverted index is constructed and preprocessed to maintain a global similarity order both for sub-units and graphs. With this blessing property, graphs can be accessed in increasing dissimilarity, and any GED based lower/upper bound can be used as filtering features. With this, two algorithms adapted from TA and CA are seamlessly integrated into the framework to speed up the search, and it is easy to pipeline the proposed framework to process continuously graph pruning. The top-k result in the TA stage is automatically fed into the CA stage, and the accessed sub-units of each graph from the CA stage are output to the DC stage.

Experimental results on two real datasets show that the proposed approach outperforms the state-of-the-art works with best filtering power. Although κ-**AT** is fast to answer queries but its loose bound causes it to suffer very poor filtering power. Since GED verification is extremely expensive, it makes sense to sacrifice a few more milliseconds to prune as many candidates as possible. **SEGOS** also can highly improve **C-Star** [9] by avoiding accessing the whole database. We also implement a simple pipelining algorithm to see its potential for enhancing the proposed framework, and future work can be done in this topic for handling huge graph databases with GPU or MapReduce system. Besides, interesting topics can be opened up to handle more general sub-unit based methods by providing appropriate aggregation functions for the TA or CA search. Another interesting topic can be further to do is that with bounds adaption our work also can support the sub-graph matching problems.

Acknowledgment

Xiaofeng Ding and Hai Jin were supported by NSF China grant 61100060.

References

[1] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou, "Mining coherent dense subgraphs across massive biological network for functional discovery," *Bioinfomatics*, vol. 1, no. 1, pp. 1–9, 2005.

[2] X. Yan, P. S. Yu, and J. Han, "Substructure similarity search in graph databases," in *SIGMOD*, 2005, pp. 766–777.

[3] B. T. Messmer and H. Bunke, "A new algorithm for error-tolerant subgraph isomorphism detection," *IEEE TPAMI*, vol. 20, no. 5, pp. 493–504, 1998.

[4] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *ICPR*, 2002, pp. 112–115.

[5] H. He and A. K. Singh, "Closure-tree: an index structure for graph queries," in *ICDE*, 2006, pp. 38–38.

[6] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognition Letter*, vol. 19, no. 3-4, pp. 255–259, 1998.

[7] M.-L. Fernández and G. Valiente, "A graph distance metric combining maximum common subgraph and minimum common supergraph," *Pattern Recognition Letter*, vol. 22, no. 6-7, pp. 753–758, 2001.

[8] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Anl. & Applic.*, 2009.

[9] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: on approximating graph edit distance," *PVLDB*, vol. 2, no. 1, pp. 25–36, August 2009.

[10] D. Justice, "A binary linear programming formulation of the graph edit distance," *IEEE TPAMI*, vol. 28, no. 8, pp. 1200–1214, 2006.

[11] C. Li, B. Wang, and X. Yang, "Vgram: improving performance of approximate queries on string collections using variable-length grams," in *VLDB*, 2007, pp. 303–314.

[12] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *SIGMOD*, 2005, pp. 754–765.

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

[14] G. Wang, B. Wang, X. Yang, and G. Yu, "Efficiently indexing large sparse graphs for similarity search," *IEEE TKDE*, vol. 99, no. PrePrints, 2010.

[15] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *SIGMOD*, 2004, pp. 335–346.

[16] S. Zhang, M. Hu, and J. Yang, "Treepi: a novel graph indexing method," in *ICDE*, 2007, pp. 966–975.

[17] H. W. Kugn, "The hungarian method for the assignment problem," *Naval Research Logistics*, vol. 2, pp. 83–97, 1955.

[18] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001, pp. 102–113.

[19] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. SSC*, vol. 4, no. 2, pp. 100–107, 1968.

[20] M. Garey and D. Johnson, *Computers and intractability*. Freeman San Francisco, 1979.

[21] M. Neuhaus, K. Riesen, and H. Bunke, "Fast suboptimal algorithms for the computation of graph edit distance," in *SSSPR*, 2006, pp. 163–172.

[22] H. A. Almohamad and S. O. Duffuaa, "A linear programming approach for the weighted graph matching problem," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 5, pp. 522–525, 1993.

[23] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree + delta >= graph," in *VLDB*, 2007, pp. 938–949.

[24] Y. Tian and J. Patel, "Tale: A tool for approximate large graph matching," in *ICDE*. IEEE, 2008.

[25] I. H. Toroslu and G. íçoluk, "Incremental assignment problem," *Inf. Sci.*, vol. 177, no. 6, pp. 1523–1529, 2007.

[26] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *SIGMOD*. ACM, 2001, pp. 425–436.

[27] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*. Citeseer, 2001, pp. 491–500.

[28] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance," in *ACM SIGIR*. ACM, 1989, pp. 405–411.

[29] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[30] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, New York, NY, USA, 2008, pp. 321–330.

APPENDIX

A. Proof of Algorithm 2

Proof: We show that the algorithm really returns the exact top-k result to a query sub-unit s_q when halting. Suppose we have m sorted lists for s_q. In fact, this algorithm can halt on two conditions: 1) The value of ω is no less than the maximum value in top-k. Since the top-k queue is maintained by the top-k minimum SEDs to s_q, when halting, they are naturally the top-k values among all sub-units having been retrieved. If we can prove that all remaining unseen sub-units have SEDs no less than the maximum value in top-k, the result is sure to be correct.

1.1) When processing the lists with smaller size graphs, we have

$$\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L}) \geq max\{top - k\}$$

For any unseen sub-unit s_i, we have

$$\lambda(s_q, s_i) = T(r_q, r_i) + 2 * |L_q| - (t(\chi) + |L_i|)$$

where $T(r_q, r_i) \geq 0$. Since all lists in this case are sorted in decreasing orders, we have

$$t(\underline{\chi}) + \underline{L} = \sum_{j=1}^{m} \underline{\chi}_j + \underline{L} \geq t(\chi) + |L_i|$$

where all $\chi_x \in \chi$ and L_i are located below the halting positions. Therefore, $\omega \leq \lambda(s_q, s_i)$, i.e., unseen sub-units have $\lambda \geq \omega \geq max\{top\text{−k}\}$. The top-$k$ results are the real k minimum values.

1.2) When running on sorted list with larger size graphs, we have

$$\omega = -|L_q| - (t(\underline{\chi}) - 2 * \underline{L}) \geq max\{top - k\}$$

In this case, for any unseen sub-unit s_i, we have

$$\lambda(s_q, s_i) = T(r_q, r_i) - |L_q| - (t(\chi) - 2 * |L_i|)$$

where $T(r_q, r_i) \geq 0$. Since label lists are sorted decreasingly while size list is sorted increasingly, we have

$$t(\underline{\chi}) - 2 * \underline{L} = \sum_{j=1}^{m} \underline{\chi}_j - 2 * \underline{L} \geq t(\chi) - 2 * |L_i|$$

where all $\chi_x \in \chi$ and L_i are located below the halting positions. Therefore, $\omega \leq \lambda(s_q, s_i)$, i.e., unseen sub-units have $\lambda \geq \omega \geq max\{top\text{−k}\}$. The top-$k$ results are correct to be the k minimum values.

2) Algorithm halts when all sorted lists have been accessed to the ends. In this case, with post processing, the top-k result is sure to be correct because they are the k minimum values among all sub-units. ■

B. Proof of Theorem 2

Proof: From the aggregation bounds definitions in Section V-C, it is clear that $\zeta(g_1, g_2) \leq L_\mu(g_1, g_2) \leq U_\mu(g_1, g_2)$. Now we prove $L_\mu(g_1, g_2) \leq \mu(g_1, g_2)$. Suppose P is an optimal alignment between $S(g_1)$ and $S(g_2)$. Then,

$$\mu(g_1, g_2) = \sum_{s_i \in S(g_1)} \lambda(s_i, P(s_i))$$

where $P(s_i)$ is each sub-unit in g_2 aligned to s_i in g_1 and $P(s_i) \in S(g_2) \cup \{\varepsilon\}$. Let $\zeta(g_1, g_2)$ of g_2 be the overall score obtained by computing the summation of all local minimum SED of g_2 below m sorted lists for g_1.

1) For those lists below $S'(g_1)$ including entries of g_2, since they contain the top-k lowest scores, we have

$$\sum_{s_i \in S'(g_1)} \min_{e_i \in Se} \{e_i\} = \sum_{s_i \in S'(g_1)} \min_{s_j \in S(g_2)} \{\lambda(s_i, s_j)\} \leq \sum_{s_i \in S'(g_1)} \lambda(s_i, P(s_i))$$

2) For those below $S''(g_1) = S(g_1) \backslash S'(g_1)$:

$$\sum_{s_i \in S''(g_1)} \min\{\underline{\chi}_i, \lambda(s_i, \varepsilon)\} \leq \sum_{s_i \in S''(g_1)} \min_{s_j \in S(g_2) \cup \{\varepsilon\}} \{\lambda(s_i, s_j)\} \leq \sum_{s_i \in S''(g_1)} \lambda(s_i, P(s_i))$$

Accordingly, we obtain $L_\mu(g_1, g_2)$ and $\mu(g_1, g_2)$ as,

$$L_\mu(g_1, g_2) = \sum_{s_i \in S'(g_1)} \min_{e_i \in Se} \{e_i\} + \sum_{s_i \in S''(g_1)} \min\{\underline{\chi}_i, \lambda(s_i, \varepsilon)\}$$

$$\mu(g_1, g_2) = \sum_{s_i \in S'(g_1)} \lambda(s_i, P(s_i)) + \sum_{s_i \in S''(g_1)} \lambda(s_i, P(s_i))$$

Therefore, $L_\mu(g_1, g_2) \leq \mu(g_1, g_2)$.

3) We prove $U_\mu(g_1, g_2) \geq \mu(g_1, g_2)$. As described in **Aggregation Upper Bound**, $\underline{\chi}(g_2)$ is a multiset of distances corresponding to the sub-units of g_2 last seen in known lists without duplicates, and

$$\overline{\chi} = max_{s \in S(g_1) \cup S(g_2)} \{\lambda(s, \epsilon)\}$$

Suppose $S'(g_2) \subseteq S(g_2)$ is the sub-units corresponding to $\underline{\chi}(g_2)$, and $S'(g_1)$ contains sub-units of g_1 aligned to $S'(g_2)$ due to $\underline{\chi}(g_2)$. If $S'(g_2) \subseteq \{P(s_i) | s_i \in S'(g_1)\}$, we have

$$t'(\underline{\chi}(g_2)) = \sum_{s_i \in S'(g_1)} \lambda(s_i, P(s_i))$$

$$\overline{\chi} * (max\{|g_1|, |g_2|\} - |\underline{\chi}(g_2)|) \geq \sum_{s_i \in S(g_1) \backslash S'(g_1)} \lambda(s_i, P(s_i))$$

If $S'(g_2) \not\subseteq \{P(s_i) | s_i \in S'(g_1)\}$, we have

$$t'(\underline{\chi}(g_2)) \geq \sum_{s_i \in S'(g_1)} \lambda(s_i, P(s_i))$$

$$\overline{\chi} * (max\{|g_1|, |g_2|\} - |\underline{\chi}(g_2)|) \geq \sum_{s_i \in S(g_1) \backslash S'(g_1)} \lambda(s_i, P(s_i))$$

Accordingly, we obtain $U_\mu(g_1, g_2) \geq \mu(g_1, g_2)$. ■

C. Proof of Algorithm 3

Proof: We prove that our candidate set includes all positive results when algorithm halts.

1) The algorithm halts with $\omega > \tau * \delta'$.

1.1) Running on the sorted lists with smaller size graphs, entries in each list below $s_j \in q$ have distances $\chi_j \leq \lambda(s_j, \epsilon)$. From the halting condition, we have $\omega = t'(\underline{\chi}) > \tau * \delta'$. Then, for any unseen graph $g_i \subseteq D'$, suppose P is the optimal alignment between $S(q)$ and $S(g_i)$. From Definition 1, we have

$$\mu(q, g_i) = \sum_{s_j \in S(q)} \lambda(s_j, P(s_j))$$

where $P(s_j)$ is each sub-unit in g_i aligned to s_j in q and $P(s_j) \in S(g_i) \cup \{\varepsilon\}$. Since $|g_i| \leq |q|$, and g_i locates below halting positions of ω. We have $\underline{\chi}_j \leq \lambda(s_j, P(s_j))$. Hence, $\omega \leq \mu(q, g_i)$. Therefore, for any $g_i \subseteq D'$, we have

$$L_m(q, g_i) = \frac{\mu(q, g_i)}{\delta_{g_i}} \geq \frac{\mu(q, g_i)}{\delta'} \geq \frac{\omega}{\delta'} > \tau$$

Any unseen $g_j \subseteq D'$ can be safely filtered out.

1.2) Similarly, if algorithm runs on the sorted list with larger size graphs, any unseen $g_i \subseteq D'$ also can be safely filtered out.

2) Algorithm halts when the ends of all sorted lists have been reached. In this case, this algorithm will become a linear scan algorithm by postprocessing the remaining unseen graphs, which guarantees that we have the correct candidate set without false negative. ■