THE NATIONAL UNIVERSITY
of SINGAPORE



School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA6/10

SINGS: Similarity Search in Large Graph Databases

Xiaoli Wang, Xiaofeng Ding, Anthony K.H. Tung, Shanshan Ying and Hai Jin

June 2010

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

SINGS: Similarity Search in Large Graph Databases

Xiaoli Wang[1], *Xiaofeng Ding*[2], *Anthony K.H. Tung*[1], *Shanshan Ying*[1], *Hai Jin*[2]
[1]{xiaoli, atung, shanshan}@comp.nus.edu.sg
School of Computing, National University of Singapore
[2]{xfding, hjin}@hust.edu.cn
School of Computer Science, Huazhong University of Science and Technology



NUS
National University of Singapore

School of Computing

NUS SOC Technical Report
Number TRA6/10
June 24, 2010

School of Computing
National University of Singapore
13 Computing Drive
Singapore 117417
Telephone: +65 6516 2727
Fax: +65 6779 4580
Homepage: http://www.comp.nus.edu.sg

SINGS: Similarity Search in Large Graph Databases

Xiaoli Wang[1], Xiaofeng Ding[2], Anthony K.H. Tung[1], Shanshan Ying[1], Hai Jin[2]
[1]{xiaoli, atung, shanshan}@comp.nus.edu.sg
School of Computing, National University of Singapore
[2]{xfding, hjin}@hust.edu.cn
School of Computer Science, Huazhong University of Science and Technology

National University of Singapore, School of Computing, TRA6/10
June 24, 2010

Abstract

Graphs are popular models for representing complex structured data and similarity search for graphs has become a fundamental research problem. Many techniques have been proposed to support similarity search on graphs based on the graph edit distance. However, they all suffer from certain drawbacks: high computational complexity, poor scalability in terms of the number of graphs and the size of the graph, or not taking full advantage of indexes. To address these problems, in this paper, we propose **SINGS**, an indexing and query processing framework for graph similarity search. An effective two-level index based on *star structures* is constructed and a novel search strategy based on the new indexing structure is also proposed. Two algorithms adapted from the TA and CA algorithms are seamlessly integrated into this proposed strategy for speeding up graph search. Extensive experiments are conducted on both real and synthetic datasets to evaluate the effectiveness and scalability of our approaches.

1 Introduction

Graphs are widely used to model complex data in many applications, like bio-informatics [10], chem-informatics [19], and pattern recognition [14]. Consequently, the management of large amount of graph data becomes a very important problem. It is thus essential to process graph queries efficiently. The classical query processing is often formulated as the (sub)graph isomorphism problem(e.g., [7, 9]). However, this kind of exact matching is too restrictive, as real objects are often affected by noise. Therefore, graph similarity search has become a basic operation in graph databases.

To manipulate graph data based on similarity, a number of graph similarity measures have been proposed in the literature (e.g., [2, 5, 14, 15]). Among them, graph edit distance (GED) has become quite popular for measuring the graph dissimilarity. It is often defined as the minimum number of edit operations to transform one graph into another [6]. GED has attracted a lot of attention from researchers for its wide usages. Consequently, this paper focuses on the graph similarity search problem based on GED. This problem can be described as follows: given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a query graph q, find all $g_i \in D$ that are similar to q based on the GED. Scanning the whole graph database D to compute GED between q and each $g_i \in D$ is very expensive. This is due to the high complexity of GED computation, which has been shown to be in NP-hard [21]. There are several existing works trying to compute the lower and upper bounds of GED and using them to prune off graphs that cannot be in the answer set. Although allowing more efficient bound computations, these methods still suffer from certain drawbacks. First, the most efficient approaches are still in cubic time [11, 21] which is still relatively expensive for large graphs. Second, they do not take full advantage of indexes, and require a scan on the whole database. This brings poor scalability over a large graph database.

To overcome these difficulties, it is natural to consider building an effective index structure for enhancing graph search. Our basic approach is to break the graphs into smaller units and use these smaller units as filtering features by indexing them using inverted lists. While this approach is similar to many existing

methods for filtering sequences (using q-grams)[13], trees (using binary branches) [20] and graphs (using paths, trees or subgraphs to test for graph isomorphism) [7, 18, 9, 22, 3], one fundamental difference exists. In all these methods, filtering is done by performing exact matching on the smaller units and then trying to infer the edit distance bound through these smaller units that exactly match the queried structure. For example, in [13], the edit distance to a query sequence is approximated by looking at the variable length q-grams that match the query sequence exactly. These exact matches can be found by using hash-based indexing which can be done in almost constant time. In our approach however, we break each graph in the database into stars [21] which are then matched against the stars generated from the query graph using the **Hungarian** [12] algorithm. Formulated as a bipartite matching problem, each star in the database graphs can have only partial matching with each star in the queried graph. Accordingly, this brings about the need to find not only stars that match exactly to the query graph but also stars that are highly similar to the queried graph.

To support such a requirement, we propose a novel indexing and query processing framework, called **SINGS** (**Star INdexing for Graph Search**). In our approach, a two-level inverted index is constructed based on stars. In the top-level index, stars derived from the graph database are used to index all graphs using inverted lists. In the lower-level index, each star is further broken into multiple labels and indexed in a inverted list. This two-level inverted index is preprocessed to maintain a global order for stars and graphs. That means, stars or graphs can be accessed in increasing dissimilarity. With this property, a novel strategy is proposed for graph search. Given a query, our strategy follows a cascaded framework: in the lower level, top-k similar stars to each star of the query can be returned quickly; in the top level, graph pruning is based on the top-k results from the lower level. Since our novel index structure is sorted, our search algorithms for stars and graphs are based on the paradigm of TA and CA methods, proposed in [4]. By deploying edit distance as aggregation function, sorted lists can be easily constructed in certain orders, and guarantee the global increasing dissimilarity orders for graphs. TA or CA based methods can enhance similarity search by avoiding accessing to graphs with high dissimilarity.

In summary, our main contributions are:

- We propose a novel two-level inverted index to speed up graph search. The lower-level index is first used to efficiently find top-k similar stars. With the top-k results, the top-level index is retrieved to construct a list of graphs that are sorted based on the similarity score.
- We show that if two graphs are within an edit distance threshold τ, their star sets also have enough similarity related to τ. We also show that if two stars have lower edit distance, they must share enough leaf labels. Based on them, we propose a better search strategy following a cascade framework using the novel index structure.
- Searching algorithms adapted from TA and CA methods [4] are efficient by dramatically reducing accesses to star structures in the whole database. Especially, we apply **Dynamical Hungarian** [17] to the CA-based algorithm, which can improve TA-based method by avoiding running the full **Hungarian** [12] algorithm for all non-candidates.
- Our algorithms can handle different graph queries in a large graph database. In this paper, we focus on range query and KNN query. Extensive experiments are also conducted to show their effectiveness and scalability.

The rest of this paper is organized as follows. In Section 2, we provide related work and Section 3 introduces several preliminary concepts. Section 4 illustrates how the two-level inverted index can be constructed based on the star structures. Our novel search strategy is introduced in Section 5 and the experimental results are shown in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work

2.1 Graph Edit Distance

The GED problem has been extensively studied in many previous works, and [6] has given a detailed survey. 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. Basically, its comptuation algorithms can be classified into two categories: exact and approximate algorithms.

The exact algorithms calculate the exact GED between two graphs. Many optimal error-correcting subgraph isomorphism algorithms have been proposed, and A^* based algorithms (e.g., [8]) are the most widely used ones. However, since GED computation is proved to be in NP-Hard[21], these algorithms often have exponential complexity and are only feasible for graphs of small size [16].

To avoid as many complex GED computations, the approximate algorithms focus on finding proper lower and upper bounds for graph filtering. [1] and [11] formulates the problem as a BLP problem and computes a lower bound in $O(n^7)$ time and an upper bound in $O(n^3)$ time. The stat-of-art method proposed in [21] computes both lower and upper bounds in cubic time, by breaking graphs into multisets of stars, and applying a novel distance to bound GED for filtering. However, all these algorithms suffer from the scalability problem for they have to scan the whole database to compute bounds for each graph.

2.2 Graph Search Problem

There is a great amount of literature on graph search problem(e.g., [7, 9]). Since (sub)graph isomorphism detection is known to be in NP-complete, various indexing structures have been proposed. Generally, they fall into three groups based on paths, trees, and frequent subgraphs.

GraphGrep proposed in [7] is a typical example of *path-based indexing approach*. It enumerates all existing paths to index graphs. Since path only is not discriminative enough, more complex indexing is developed as *subgraph-based indexing approach*. **gIndex**[18] adopts the discriminative frequent subgraphs as basic index structures, but it involves complex feature extraction for each query. **C-Tree** [9] employs an R-tree like index structure, organizing graphs hierarchically in a tree, each of whose node summarizes its descendants by a graph closure. **FG-index** [3] is based on set frequent graphs. It builds an outer inverted-index on the set of defined features and an inner one on the cluster of frequent subgraphs. Though graph-based indexing enjoys strong pruning power, the index itself is complex and expensive to construct. Furthermore, **TreePi** [22] and **Tree+Delta** [23] are based on frequent subtrees.

But all these index methods demand lots of preprocessing efforts and are specifically designed for graphs(or sometimes trees) search only. In this paper, we propose a quite different feature for indexing, and build a flexible similarity graph search framework. It not only can integrate existing bounds for filtering, but also have the potential be applied to search other data, such as trees and sequences with minor modifications.

3 Preliminaries

In this paper, we focus on the huge dataset D of undirected labeled simple graphs. 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 labeling function assigning a label to a vertex. The size of a graph g, denoted by $|g|$, is defined as the number of vertices in g. Figure 1 shows an example of two graphs g_1 and g_2, and other major notations used in our paper are in Table 1.

3.1 Star Representation of Graphs

A star is defined as a labeled, single-level and rooted tree which can be represented by a 3-tuple $s = (r, L, l)$, where r is the root vertex, L is the set of leaves and l is a labeling function. For each v_i in the graph, we construct a star $s_i = (v_i, L_i, l)$, where L_i is the label set of v_i's neighbors. A graph g with n vertices can be represented as a multiset of n stars. In Figure 1, two graphs g_1 and g_2 are transformed into two star representations: $S(g_1)$ and $S(g_2)$. With this transformation, we cite the definitions of novel distances between stars and star representations from [21] as follows:

Definition 3.1. *(Star Edit Distance(SED)) Given two star structures s_1 and s_2, the edit distance between them is defined as*

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

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
$L_m(g_1, g_2)$	the lower bound of $\lambda(g_1, g_2)$
$U_m(g_1, g_2)$	the upper bound of $\lambda(g_1, g_2)$
$\zeta(g_1, g_2)$	the overall score of g_2 obtained from g_1
ω	the halting threshold for TA or CA methods

Table 1: Notation

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

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

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

In the above equation, Ψ_L is the multiset of vertex labels in L. Assume that the alphabet Σ of vertex labels has a total order[1], 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 1, it is straight forward that $T(r_1, r_2) = 0$, for $r_1 = r_2 = a$. 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 3.2. *(Mapping Distance) Given two star representations of* g_1 *and* g_2 *as* $S(g_1)$ *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 is equivalent to finding an optimal mapping between two star representations. Zeng *et al.* [21] constructs a weighted matrix for each pair of stars from two graphs, and applies *Hungarian algorithm* to get the optimal solution in $O(n^3)$ time. Here weight between two stars is their SED. If two graphs are of different size, ϵ node is inserted for normalization. In Figure 1, the left bottom 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 hand side of Figure 1, and the optimal matching is marked with solid arrows.

4 Star Indexing in SINGS

In this Section, we propose **SINGS**, a novel indexing and query processing framework for graph similarity search. A two-level inverted index based on stars is constructed and used to build score-sorted lists for processing graph queries.

4.1 The Top-Level Inverted Index

Given a huge database with graphs and their star representations, a star inverted index can be constructed. First, we represent a star as a *label sequence* $l_r l_1..l_n$, where l_r is the label of root vertex and $l_1..l_n$ is a sequence composed by the labels of its n leaves sorted in ascending orders. With this representation, stars also have a global order. In Figure 1, s_0 is represented as *abbcc* and all stars have an alphabetical order.

[1]Hereafter, we assume the alphabet of vertex labels in a database have a total order.

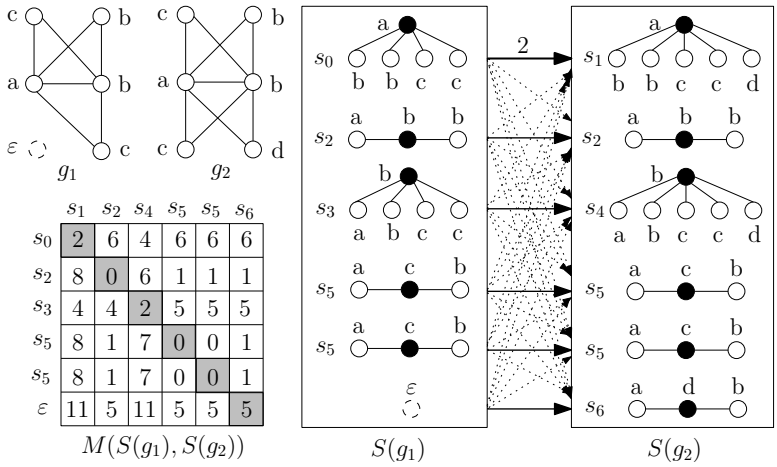


Figure 1: Mapping between Two Star Representations

s_i represents the ith star. Hence, given a graph dataset with g_1 and g_2 in Figure 1, we can construct an inverted-list index for all stars in the database as shown in Figure 2. This structure is made up of two main parts: an index for all distinct stars in the given dataset, and an inverted list for each star which stores the information of graphs containing this star. Here, the star index is sorted in increasing order according to label sequences of stars. Each entry in the inverted lists contains the identity of each graph and the number of occurrences of the star. All lists are sorted in increasing orders of graph size. In Figure 2, since $|g_1| < |g_2|$, g_1 is located before g_2 in each list.

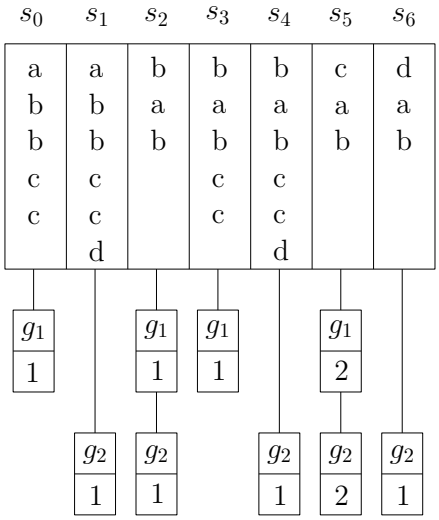


Figure 2: Top-Level Inverted Index for Graphs

With this index, it is very convenient to fetch out graphs containing a given star. Then, given a query, if we can quickly access stars which are similar to each star of the query in increasing dissimilarity, graphs can also be accessed in globally increasing dissimilarity to the query. This is why we build a lower-level index for all stars in the database.

4.2 The Lower-Level Inverted Index

Similarly, we construct the lower-level inverted index for all stars based on vertex labels. A star is broken into a multiset of labels excluding its root label, and we call this its *label representation*. For example, s_0

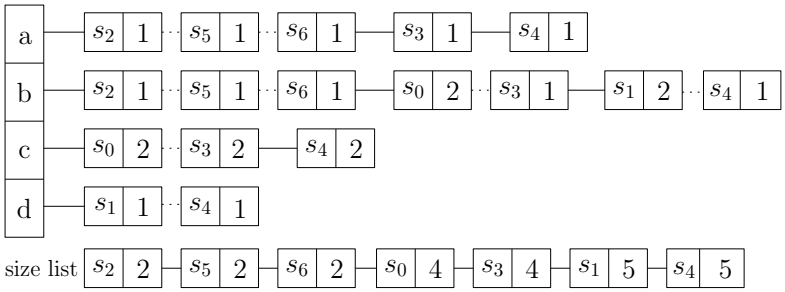


Figure 3: Lower-Level Inverted Index for Stars: entries connected by dot line are grouped together; within a group, entries are sorted by decreasing occurrences

in Figure 2 is represented as $\Psi_{s_0} = \{b, b, c, c\}$. With this representation, it is easy for us to construct an inverted index for stars based on labels. The index also contains two components: a label index in increasing orders and inverted lists below labels recording stars' identities and the occurrences of labels in their label representations. Entries in each inverted list are first grouped based on the star leaf size of $|\Psi_s|$ and then sorted by decreasing orders of occurrences within each group. For example, in Figure 3, 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 size of 4 and their occurrences are sorted in decreasing orders. Moreover, the last list without a label index is an extended list storing the size information of all stars. The entry in this list includes the identity and leaf size of a star. The list is sorted in increasing order of leaf sizes. With this index structure, stars can be accessed in a global increasing dissimilarity to a particular star.

5 Searching in SINGS

We now present how the inverted index can be used to enhance graph similarity search. Our approach follows a cascade framework. First, the lower-level inverted index is used to find top-k similar stars. Then, the top-level index is used to construct sorted lists of graphs based on the top-k results.

5.1 Searching for Top-K Stars

The top-k star query processing is based on TA method [4]. However, instead of random access to lower-level inverted list entries, we fetch a star using its identity. Lists sorted based on stars can be easily constructed from the lower-level index. Given a query q of size m, it is first processed as a star representation of $S(q) = \{s_0, s_1, \ldots, s_m\}$. Each star s_i of leaf size n is represented as $\Psi_{s_i} = \{l_1, \ldots, l_n\}$. To find top-k similar stars to s_i, a merge function is used to construct a sorted list for each distinct label in Ψ_{s_i}, and SED is adapted as the aggregation function. Two conditions are considered for computing its distance to any s_j.

$$\lambda(s_i, s_j) = \begin{cases} T(r_i, r_j) + 2 * n - (\psi + |L_j|), \ if \ |L_i| \geq |L_j| \\ T(r_i, r_j) - n - (\psi - 2 * |L_j|), \ if \ |L_i| < |L_j| \end{cases} \quad (1)$$

The Proof A.1 is shown in the appendix. Here, $\psi = |\Psi_{L_i} \cap \Psi_{L_j}|$. Suppose s_i has m distinct leaf labels with occurences of $(c_1, c_2, \ldots, c_m)$. We compute $\psi = t(\chi)$ as the number of common leaf labels between s_i and s_j.

$$t(\chi) = \sum_{p=1}^{m} \min\{c_p, \chi_p\} \quad (2)$$

χ_p represents the occurrence corresponding to s_j in a list below the p^{th} label of s_i. If s_j does not appear in that list, $\chi_p = 0$. If we ignore the root information between two stars, λ increases when the value of $(\psi + |L_j|)$ or $(\psi - 2 * |L_j|)$ decreases. Thus, aggregation functions are adapted by ignoring the root information $T(r_i, r_j)$ in the above two equations. Accordingly, each sorted list is composed of two segments due to the above two conditions. Since the lower-level index lists have been grouped based on star leaf sizes, if we maintain an array to store pointers to each group, positions of leaf size bounds are easy to be found by binary search.

Since each group has been sorted based on label occurrences, each segment can be obtained by merging all groups within the size range in linear time. The detail of the merge function is given in Algorithm 1 in Appendix B.

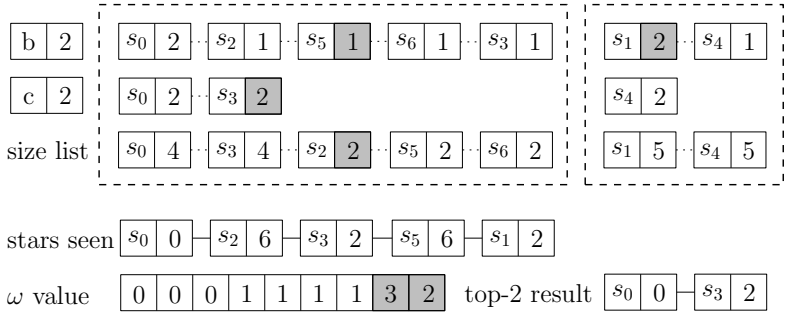


Figure 4: Top-k star searching example. The upper is sorted Lists for a Star $s = abbcc$: stars in the left box have sizes no more than 4; stars in the right box have sizes larger than 4

Figure 4 shows the sorted lists obtained for $s = abbcc$ with leaf size 4 using the index in Figure 3. For the label b, we find a size bound of 4 in position 5 with entry $(s_1, 2)$. Then groups with leaf sizes no more than 4 are merged into one group of $\{(s_0, 2), (s_2, 1), (s_5, 1), (s_6, 1), (s_3, 1)\}$. Another group of $\{(s_1, 2), (s_4, 1)\}$ with leaf size larger than 4 is also formed. Similarly, two groups below c are formed as $\{(s_0, 2), (s_3, 2)\}$ and $\{(s_4, 2)\}$. Furthermore, the size list is also split into two groups and the group with sizes no more than 4 is resorted in decreasing orders. Subsequently, TA-based searching algorithm runs on each group of sorted lists respectively using the above two aggregation functions. Given m distinct label sorted lists and 1 size sorted list for s_i with leaf size n, the steps of our searching algorithm are shown as follows:

1. Do sorted access in a round-robin schedule to each sorted list. If a star s_j is seen, compute $\lambda(s_i, s_j)$. Maintain a set of top-k stars with the lowest λ values.

2. For each label list SL_p, let $\underline{\chi}_p$ be the occurrence of the last star seen under sorted access. Let $\underline{L}$ be the last seen size in the size list. Define the threshold ω. For the group with smaller size, $\omega = 2 * n - (t(\underline{\chi}) + \underline{L})$. For another group with larger size, $\omega = -n - (t(\underline{\chi}) - 2 * \underline{L})$. Here, $t(\underline{\chi})$ is computed by Equation (2). If the top-k values are at most equal to ω, then halt. Otherwise, go to 1.

The detail is shown in Algorithm 2 in Appendix B. Although the aggregation functions have ignored the root information of stars, our algorithm is also correct to obtain the exact top-k result when halting. The correctness is shown in Proof A.2 in the appendix. Take Figure 4 for an example, the top-2 similar stars to s is returned. First, for lower size sorted lists, stars are sorted accessed in a round-robin way from the list below label b to the size list, as shown in the "stars seen" list. SED is calculated for each star 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 top-2 queue. Subsequently, larger size sorted lists are accessed similarly. From this example, it is clear that this algorithm is efficient for finding top-k stars by avoiding accessing to every star in the database.

5.2 Graph Searching Algorithms with Score-Sorted Lists

As shown in previous section, the top-k star searching algorithm provides us an efficient way to access stars in increasing dissimilarity with respect to a query star. Our graph searching algorithm can also enjoy such nice property once the underlying star lists are well organized. Two algorithms adapted from CA and TA are proposed to make this possible. Before giving the algorithms in detail, we introduce how to build appropriate sorted lists for these algorithms.

Given a query q of size m, we first extract m top-k similar star lists, one for each star in q. For query star s_q, each entry in the top-k list consists of two parts: star ID and its distance to the query star. Then, for each entry $(s_i, \lambda(s_q, s_i))$ in the top-k list, graph list GL_j indexed by s_i is fetched from the top-level index. In total, k sorted graph lists will be returned for each query star s_q. Later each of these k graph lists will be

split into two segments with respect to graph size: those with graph size larger than query size, and those not. Segments belonging to the same category will be combined into one group. Within each group, graphs are naturally ordered in terms of SED between the star s_i the graph is indexed by, and the query star s_q. Furthermore, in the group with smaller size, those segments having SED larger than $\lambda(s_q, \epsilon)$ are discarded.

Given a query $q = g_1$ in Figure 1, the top-2 similar stars for query star s_5 are s_5 and s_2, in Figure 5. Then two graph lists indexed by s_5 and s_2 are extracted from the top-level index in Figure 2: $\{(g_1, 2), (g_2, 2)\}$ and $\{(g_1, 1), (g_2, 1)\}$. Noticing the query is of size 4, the graph lists are divided into two groups: $\{(g_1, 2), (g_1, 1)\}$ in the left dashed box, and $\{(g_2, 2), (g_2, 1)\}$ in the right dashed box. Since $\lambda(s_5, s_5) = 0 \leq \lambda(s_5, s_2) = 1$, $(g_1, 2)$ comes before $(g_1, 1)$ in the same group.

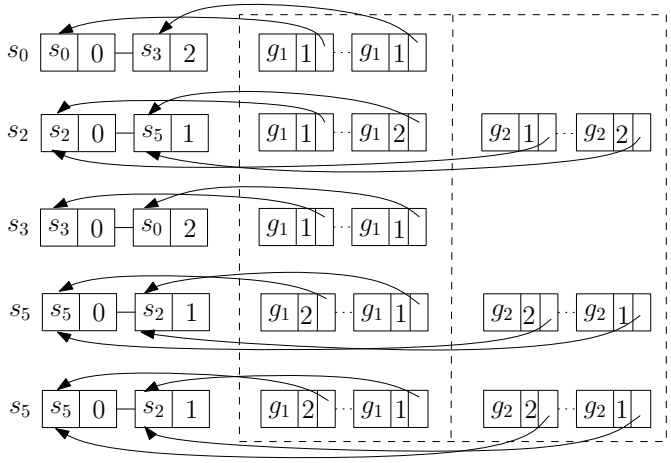


Figure 5: The sorted lists for $q = g_1$

5.2.1 TA-based Searching Algorithm

Now we present our TA based graph searching algorithm. The steps are similar to the top-k star searching algorithm. The difference is that this algorithm uses mapping distance as aggregation function, and compute $\omega = t'(\chi)$. $t'(\chi) = \sum_{p=1}^{m} \chi_p$ is a summation function on m sorted lists. Actually, all exiting bounds for GED can be directly adopted in our searching algorithms as filter features. In this paper, we take a lower and a upper bound, denoted by $L_m(q, g_i)$ and $U_m(q, g_i)$, based on mapping distance proposed in [21] as an example.

Range Query Given m sorted lists of a query q and a GED threshold τ, the steps of TA-based range query is shown as below:

1. Do sorted access in a round-robin schedule to each sorted list. For a graph g_i being seen, if $L_m(q, g_i) > \tau$, filter out this graph; if $U_m(q, g_i) \leq \tau$, return this graph in the final result.

2. For each sorted list SL_p, let $\underline{\chi}_p$ be the corresponding distance of the last graph seen under sorted access. If $\omega = t'(\underline{\chi}) > \tau * \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$, then halt. Otherwise, go to 1.

KNN Query Given m sorted lists of a query q, the steps of TA-based KNN query is shown as below:

1. Do sorted access in a round-robin schedule to each sorted list. For a graph g_i being seen, if $L_m(q, g_i) \geq \overline{U}_m$, add it to the candidate set; if $U_m(q, g_i) < \tau$, update $top - k$.

2. For each sorted list SL_p, let $\underline{\chi}_p$ be the corresponding distance of the last graph seen under sorted access. If $\omega = t'(\underline{\chi}) \geq \overline{U}_m * \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$, then halt. $\overline{U}_m$ is the maximum value in top-k. Otherwise, go to 1.

In step 2, D' is the set of graphs which have not been accessed. It should be noted that a post processing phase is needed to guarantee the correctness of the algorithms. If algorithms halt by reaching to the ends of all lists, those graphs which did not appear should be accessed to avoid false negative. The detail of range

query algorithm and its correctness are shown in Algorithm 3 and Proof A.3 in the Appendix. KNN query algorithm and its correctness are shown in Algorithm 4 and Proof A.4 in the Appendix. The improvement of these two algorithms is like top-k star searching. The parameter ω helps us to avoid access to graphs with high dissimilarity. To avoid computing mapping distance for all accessed graphs, we later propose CA-based algorithm to improve TA-based method. In the following, we first present some useful bounds and theorems used in CA methods.

5.2.2 Bounds of Mapping Distance

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

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

where, χ_p is a local minimum SED of graph g having been seen below the i^{th} list of q. We define the computation of χ_p as below.

Definition 5.1. *Let $Se_p = \{e_1, \ldots, e_x\}$ including all SEDs of a graph g below the p^{th} list. Then the corresponding χ_p of g is computed as*

$$\chi_p = \min_{e_i \in Se_p} \{e_i\}$$

Generally, if Se_p is empty, $\chi_p = 0$. Suppose $l(g) = \{l_1, \ldots, l_y\} \subseteq \{1, 2, \ldots, m\}$ is a set of known lists of g below q. Let $\underline{\chi}(g)$ be the multiset of distances corresponding to the stars of g last seen in known lists without duplicates. Then we have,

- **Lower bound** denoted by $L_\mu(q, g)$ is obtained by substituting the missing lists $p \in \{1, 2, \ldots, m\} \setminus l(g)$ with $\underline{\chi}_p$ (the bottom distance of last graph seen under the p^{th} list) in $\zeta(q, g)$. That is, $\chi_p = \underline{\chi}_p$ when Se_p is empty.
- **Upper bound** denoted by $U_\mu(q, g)$ is computed as $U_\mu(q, g) = t'(\underline{\chi}(g)) + \overline{\chi} * (|g| - |\underline{\chi}(g)|)$.

The value of $\overline{\chi}$ is computed as $\overline{\chi} = max_{s \in S(q) \cup S(g)}\{\lambda(s, \epsilon)\}$. For example, if we have seen graph g in a set of lists $\{1, \ldots, y\}$, we have $\zeta(q, g) = \chi_1 + \chi_2 + \cdots + \chi_y$. Then lower bound can be computed as $L_\mu(q, g) = \zeta(q, g) + \underline{\chi}_{y+1} + \ldots + \underline{\chi}_m$. It is obvious that $\zeta(q, g) \leq L_\mu(q, g) \leq U_\mu(q, g)$.

Theorem 5.1. *Let g_1 and g_2 be two graphs, the bounds obtained as above satisfies the following:*

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

and the mapping distance $\mu(g_1, g_2)$ satisfies the following:

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

The proof of this theorem is shown in Proof A.5 in the Appendix.

Theorem 5.2. *Given m sorted lists for q, suppose $S'(g) \subseteq S(g)$ is a multiset of stars in g seen below lists. Then we have*

$$\mu'(S(q), S'(g)) \leq \mu(q, g)$$

The computation of $\mu'(S(q), S'(g))$ and an informal proof of the theorem is shown in Proof A.6 in the Appendix. This nice property enlightens us the idea of computing the subset of stars first to reduce as much cost. That is to say, if $\mu'(S(q), S'(g))$ is large enough, there is no need to compute the whole graph; otherwise, the whole star set will be computed. Since originally **Hungarian** [12] algorithm is used to get the optimal matching, **Dynamic Hungarian**[17] provides us a perfect way to process these graphs incrementally, which only takes $O(kn^2)$ time and k is the number of remaining stars. With this pruning strategy, we can stop early when two graphs are quite different from each other.

5.2.3 CA-based Searching Algorithm

Now we present our CA based algorithms. The main steps are similar to the TA-based algorithms. The above bounds are deployed to enhance the search. The steps are shown as below:

Range Query Given m sorted lists for q and a threshold τ,

1. Do sorted access in a round-robin schedule to each sorted list. At each depth h of lists:
   - Maintain the bottom values $\underline{\chi}_1, \ldots, \underline{\chi}_m$ encountered in the lists. Maintain a distance accumulator $\zeta(q, g_i)$ and a multiset of stars seen $S'(g_i) \subseteq S(g_i)$ for each g_i seen under lists.
   - For each graph g_i seen and unprocessed, if $\zeta(q, g_i) > \tau$, filter out this graph; if $L_\mu(q, g_i) > \tau$, filter out this graph; if $U_\mu(q, g_i) \leq \tau$, return this graph as a candidate. Otherwise, if $\mu'(S(q), S'(g_i)) > \tau$, filter out this graph. If all the above bounds have no use, then run Dynamic Hungarian to obtain $L_m(q, g_i)$ and $U_m(q, g_i)$ for filtering.
2. When a new distance is accessed, compute a new ω. If $\omega = t'(\underline{\chi}) > \tau * \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$, then halt. Otherwise, go to 1.

KNN Query Given m sorted lists of a query q,

1. Do sorted access in a round-robin schedule to each sorted list. At each depth h of lists:
   - Maintain the bottom values $\underline{\chi}_1, \ldots, \underline{\chi}_m$ encountered in the lists. Maintain a distance accumulator $\zeta(q, g_i)$ and a multiset of stars seen $S'(g) \subseteq S(g)$ for each g_i seen under lists. Let $top-k$ contains the smallest U_m values seen so far, and $\overline{U}_m = max\{U_m | U_m \in top-k\}$.
   - For each graph g_i seen and unprocessed, compute bounds to maintain the new $top-k$ values. If $\zeta(q, g_i) \geq \overline{U}_m$, $L_\mu(q, g_i) \geq \overline{U}_m$, or $\mu'(S(q), S'(g_i)) \geq \tau$, add this graph to the candidate set. Otherwise, run Dynamic Hungarian to obtain $L_m(q, g_i)$ and $U_m(q, g_i)$ to filter as TA method.
2. When a new distance is accessed, compute a new ω. If $\omega = t'(\underline{\chi}) \geq \overline{U}_m * \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$, then halt. Otherwise, go to 1.

The details of CA-based range query and KNN query algorithms are shown in Algorithm 5 and 6 in the Appendix. The proof of their correctness is similar to Proof A.3 and A.4.

6 Experimental results

In this section, we conduct extensive experiments to verify the effectiveness and scalability of our proposed methods on both synthetic and real datasets. All the experiment programs are compiled by gcc 3.4.3 in Red hat Linux Operating system and run on Quad-Core AMD Opteron(tm) Processor 8356, 128GB memory, running RHEL 4.7AS.

Synthetic Datasets. We use graph data generator to produce synthetic datasets. The number of graphs, average graph size, and the number of vertex labels can be specified to various values. In our experiments, we set vertex label number to be 63, and the average graph size to be 30.

Real Datasets. Without loss of generality [18, 21], we use AIDS, which consists of 42,687 compounds and 63 vertex labels. The average graph size is 46.

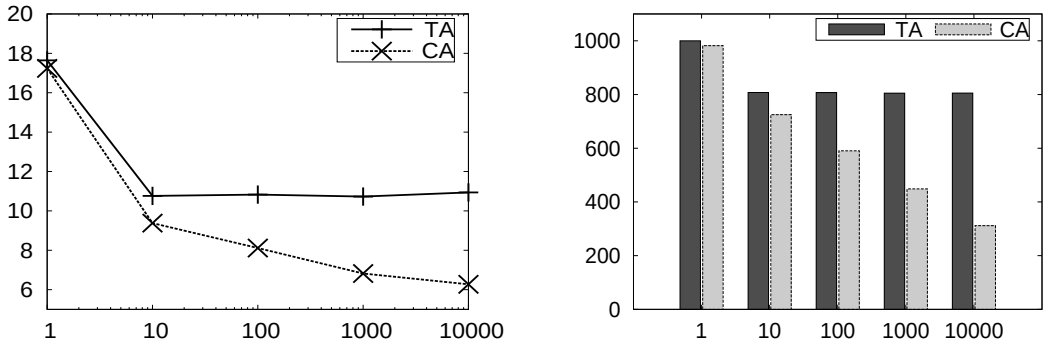
Parameter	Description	Value
k_s	number for top-k stars	1,10,100,**1000**,10000
h	depth for CA	1,10,100,**1000**,10000
N	dataset cardinality	**1000**,2000,3000,4000,5000
τ	distance threshold	0,**5**,10,15,20
k_g	number for KNN	1,**5**,10,15,20,25

Table 2: Experimental Settings

The experimental settings are summarized in Table 2, the default values are in bold. In all the following algorithms, parameters are set to be the default values except the various one. Note that, 10 queries are randomly selected from the datasets for each experiment, and the results are all averaged on the queries. We adopt the CPU time consumption and the access number as performance measures. The access number is defined as the number of graphs, which are accessed to compute the mapping distance. We compare our methods with the most current and efficient linear algorithm proposed in [21].

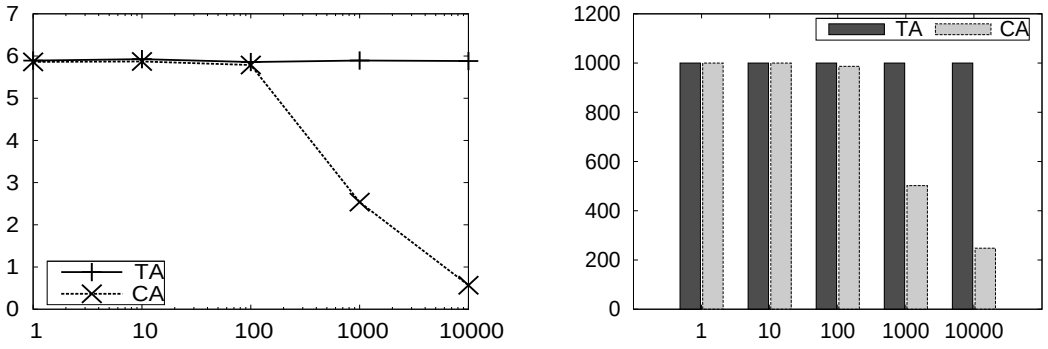
6.1 Sensitivity Test on Synthetic Datasets

In the first set of experiments, a serious of sensitivity analysis to the parameters of our algorithms is carried out over synthetic datasets. The dataset is generated including 1000 graphs. Figure 6 to Figure 9 shows the performance of our methods for various parameter settings.



(a) Runtime vs k_s (b) Access number vs k_s

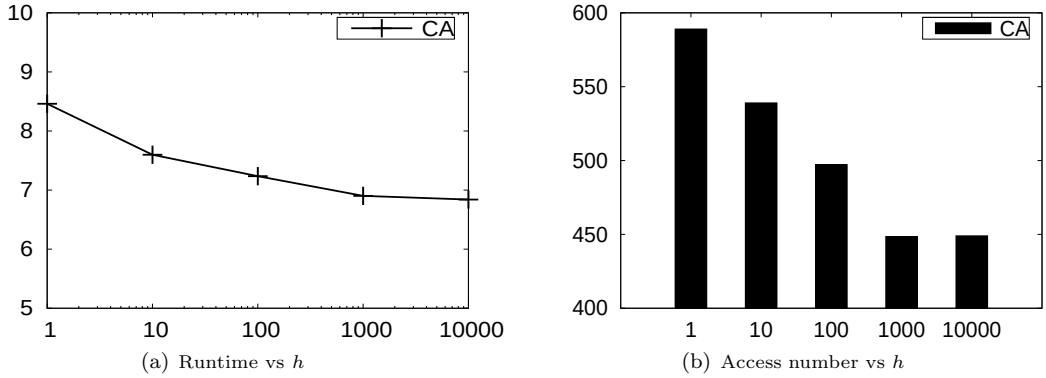
Figure 6: Sensitivity of k_s for Range Queries



(a) Runtime vs k_s (b) Access number vs k_s

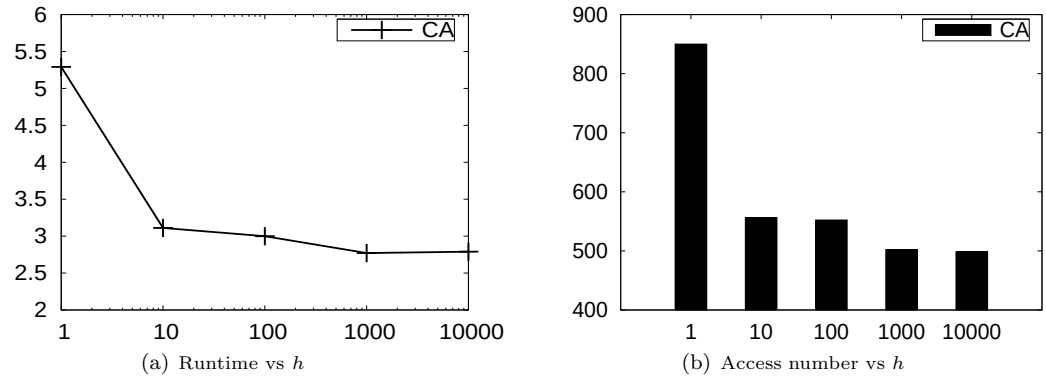
Figure 7: Sensitivity of k_s for KNN Queries

Figure 6 and 7 show the sensitivity of k_s to two query algorithms handled by TA and CA based methods. The run time of both TA and CA methods generally decrease as k_s increases from 1 to 10000. This is reasonable since the access number has decreased when k_s increases. In fact, when k_s is small, sorted lists can be very short. TA or CA algorithm might run through the whole lists without halt, and the remaining graphs can be checked one by one. Especially when $k_s = 0$, our methods are equal to the linear scan. Note that, since CA method adopted the lower and upper bounds of mapping distance for pruning, its performance can be better than that of TA method. Furthermore, when k_s becomes very large, runtime becomes stable because halt condition holds. However, for KNN queries in our experiment, the runtime and the access number become unchanged with various k_s. Due to the characteristic of graph distribution and the large upper bound of GED that TA adopted for KNN search, its hard to find k_s number of graphs with upper bounds smaller than the current graph's lower bound, thus TA method's KNN query performance is almost not affected by parameter k_s.



(a) Runtime vs h (b) Access number vs h

Figure 8: Sensitivity of h for Range Queries



(a) Runtime vs h (b) Access number vs h

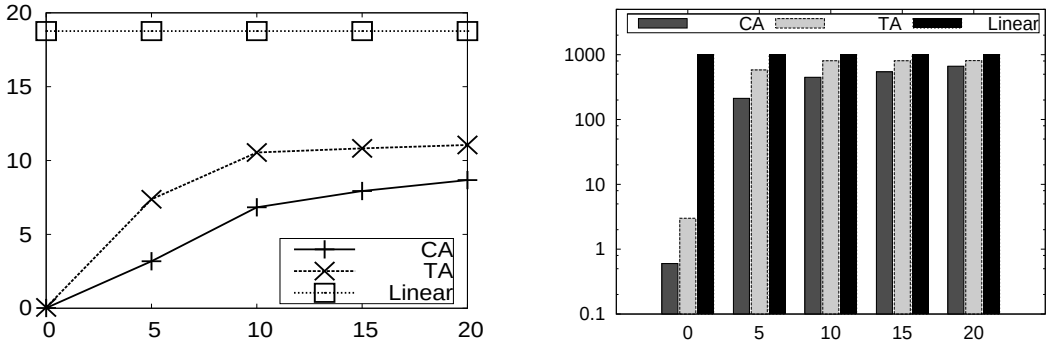
Figure 9: Sensitivity of h for KNN Queries

Figure 8 and 9 show the sensitivity of depth h used in CA method to two query algorithms. From the figures, we can see that the run time and the access number of CA method decrease when h is increased from 1 to 1000, and increase when h increases from 1000 to 10000. This is because in every h step, the CA method performs random access to get the missing stars of some potential graphs. Thus, the Dynamic Hungarian [17] is applied for some graphs on its subset of stars and this lower bound can be used to filter out non-candidates. If h is appropriate, full Hungarian algorithm running can be highly reduced. Furthermore, some bounds obtained with the overall score of graphs might have certain filtering capacity. All these can help to improve the TA method and makes the CA method have better performance than TA. Later experiments will also show this.

6.2 Query Performance on Range Query

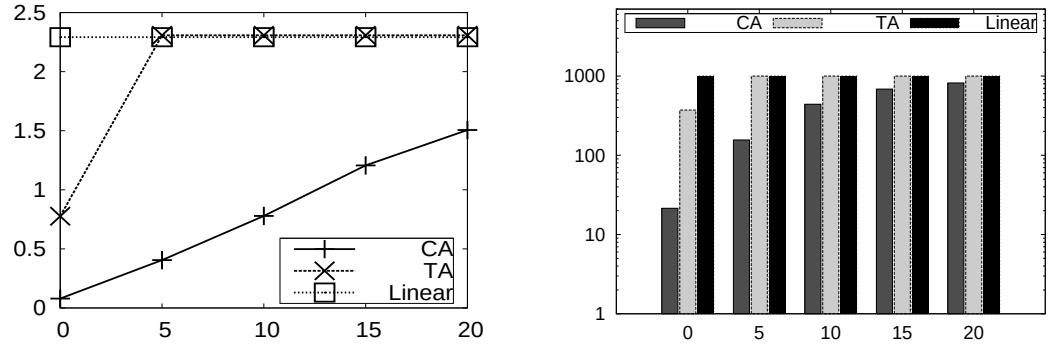
Figure 10 and 11 show the results of range queries with various τ over synthetic and real datasets. Our CA based method shows better performance than the linear algorithm and TA method also shows better performance when τ is small. This is because when τ is small the ω threshold can be useful in our methods. It is natural that the run time and the access number of our methods increase simultaneously when τ varies from 0 to 20. The reason is that TA or CA methods halt only when the accessing threshold $\omega > \tau$. Thus, when τ is getting larger, algorithms might access more graphs to meet the halt condition.

Figure 12 and 13 show the scalability of range queries over synthetic and real datasets with N from 1000 to 5000. The runtime and the access number of our methods are both high less than linear algorithm over synthetic dataset. For real dataset, CA shows better performance than TA and linear algorithm. This result is reasonable and consistent with our previous sensitivity test on them. As database size increases, CA shows further good performance. This indicates that the CA method clearly have better prune power than TA, and linear algorithm always shows worse performance.



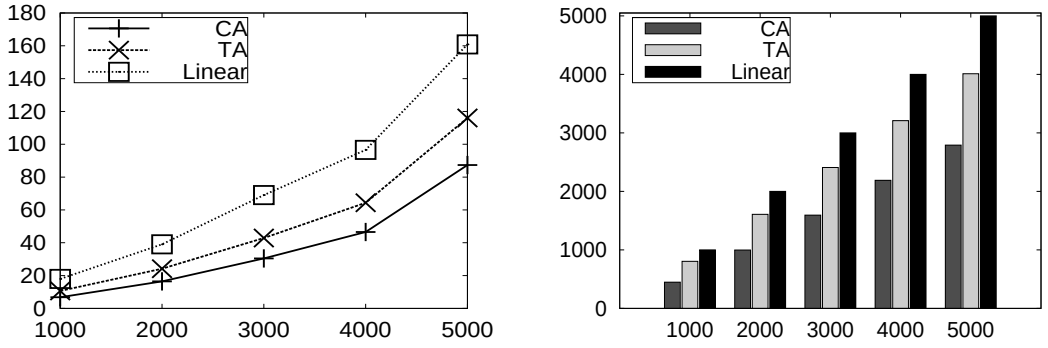
(a) Runtime vs τ (b) Access number vs τ

Figure 10: Range Queries with Varying τ over Synthetic Datasets



(a) Runtime vs τ (b) Access number vs τ

Figure 11: Range Queries with Varying τ over Real Datasets



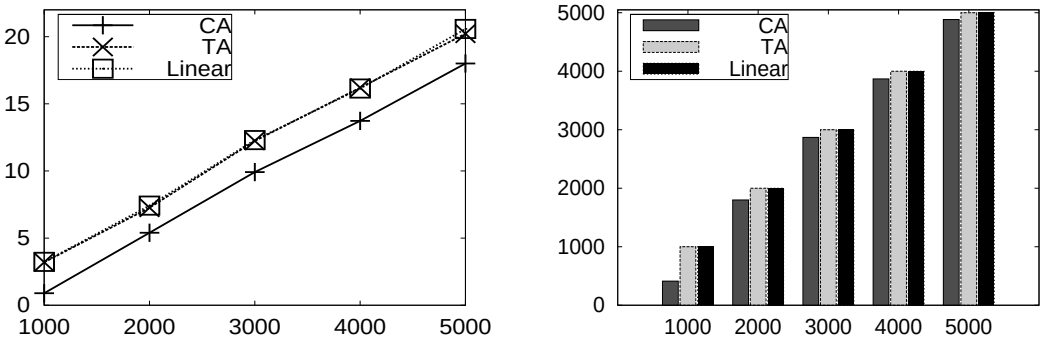
(a) Runtime vs N (b) Access number vs N

Figure 12: Scalability of Range Queries over Synthetic Datasets

6.3 Query Performance on KNN Query

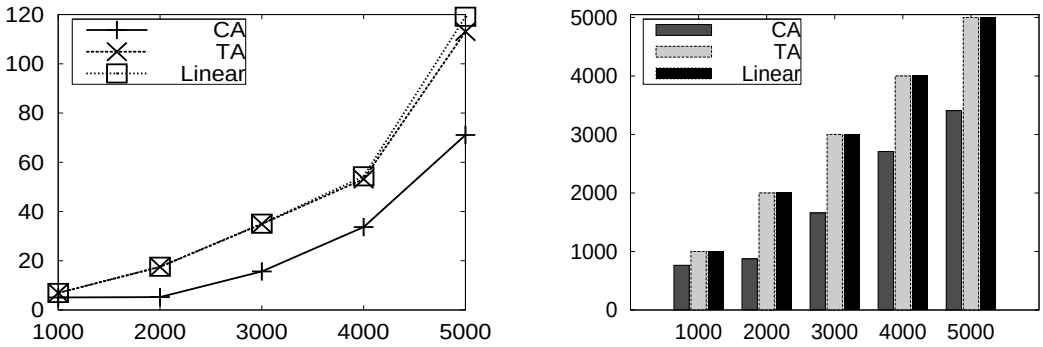
Figure 14 and 15 show the scalability of KNN queries over synthetic and real datasets with N from 1000 to 5000. The results for CA methods are similar to previous results on range query that CA always shows better performance than TA and linear algorithms. For KNN, performance can be highly affected by the data distribution and the filtering capacity of bounds. Thus, TA might not halt with the loose ω bound and falls to linear scan. Hence, our methods also can be further improved by deploying some new proposed tighter bounds.

With various k_g from 1 to 25, Figure 16 and 17 display the KNN query results on synthetic and real datasets. It can be seen that the CA method accessed much less data, especially when k_g is no larger than 5. When k_g is getting larger, TA method achieves its worst case performance, as almost the same as that of



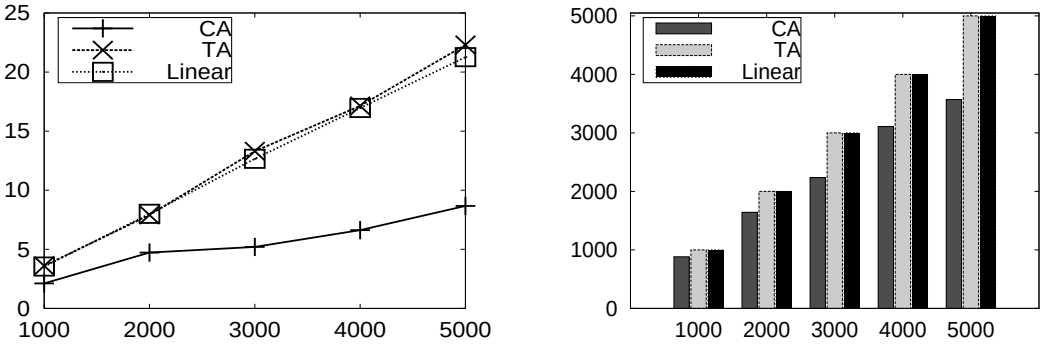
(a) Runtime vs N (b) Access number vs N

Figure 13: Scalability of Range Queries over Real Datasets



(a) Runtime vs N (b) Access number vs N

Figure 14: Scalability of KNN Queries over Synthetic Datasets



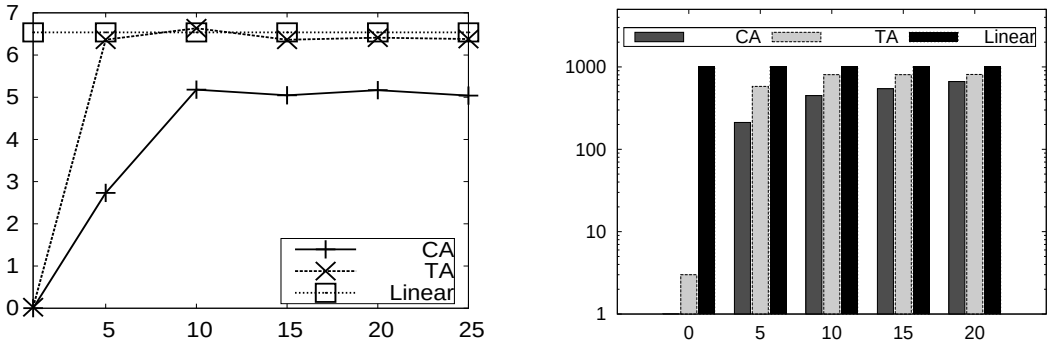
(a) Runtime vs N (b) Access number vs N

Figure 15: Scalability of KNN Queries over Real Datasets

linear algorithm. The reason is similar to that for previous results.

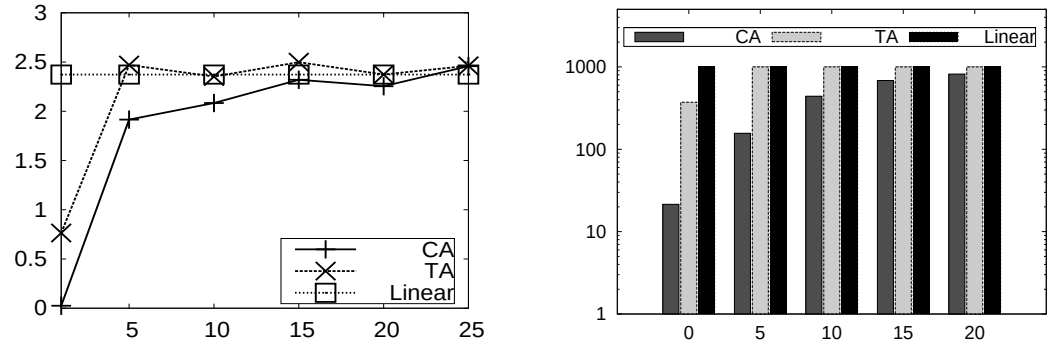
7 Conclusion

In this study, we have investigated an important problem of similarity search in large graph databases based on GED. Different from the previous works, our method proposes an efficient indexing and query processing framework that based on stars. A two-level inverted index is constructed and preprocessed to maintain a global similarity order both for stars and graphs. With this property, graphs can be accessed in increasing dissimilarity and lower or upper bounds of GED can be used as filtering features. In this paper, we cite two bounds based on mapping distance proposed by Zeng et. al. [21] to show the effectiveness of our



(a) Runtime vs k_g (b) Access number vs k_g

Figure 16: KNN Queries with Varying k_g over Synthetic Datasets



(a) Runtime vs k_g (b) Access number vs k_g

Figure 17: KNN Queries with Varying k_g over Real Datasets

framework on stars. With the sort property of our indexing structure, two algorithms adapted from TA and CA methods are developed to speed up the search, and these algorithms can be seamlessly integrated into the framework. Extensive experiments have been conducted to show the effectiveness and scalability of our proposed solutions.

Besides, the proposed general framework is not restrictive to star transformation only. It can also be applied to other smaller-units based methods with an appropriate aggregation function on searching algorithms.

References

[1] H. A. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):522–525, 1993.

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

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

[4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[5] M.-L. Fernández and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letter*, 22(6-7):753–758, 2001.

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

[7] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR (2)*, pages 112–115, 2002.

[8] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. SSC*.

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

[10] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological network for functional discovery. *Bioinfomatics*, 1(1):1–9, 2005.

[11] D. Justice. A binary linear programming formulation of the graph edit distance. *IEEE TPAMI*, 28(8):1200–1214, 2006.

[12] H. W. Kugn. The hungarian method for the assignment problem. *Naval Research Logistics*, 2:83–97, 1955.

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

[14] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE TPAMI*, 20(5):493–504, 1998.

[15] R. Myers, R. C. Wilson, and E. R. Hancock. Bayesian graph edit distance. *IEEE TPAMI*, 22(6):753–758, 2000.

[16] M. Neuhaus, K. Riesen, and H. Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *SSSPR*, pages 163–172, 2006.

[17] I. H. Toroslu and G. íçoluk. Incremental assignment problem. *Inf. Sci.*, 177(6):1523–1529, 2007.

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

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

[20] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.

[21] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: on approximating graph edit distance. *PVLDB*, 2(1):25–36, August 2009.

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

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

A Proofs

A.1 Proof of Equation (1)

Proof. The Equation (1) can be easily obtained from Definition 3.2. Given two stars s_i of leaf size $|L_i| = n$ and s_j, $\lambda(s_i, s_j)$ is computed as

$$T(r_i, r_j) + ||L_i| - |L_j|| + \max\{|\Psi_{L_i}|, |\Psi_{L_j}|\} - |\Psi_{L_i} \cap \Psi_{L_j}| = T(r_i, r_j) + |n - |L_j|| + \max\{n, |L_j|\} - \psi$$

$\psi = |\Psi_{L_i} \cap \Psi_{L_j}|$. Therefore, it is easily to obtain euqtions as follows:

$$\lambda(s_i, s_j) = \begin{cases} T(r_i, r_j) + 2 * n - (\psi + |L_j|), \ if \ |L_i| \geq |L_j| \\ T(r_i, r_j) - n - (\psi - 2 * |L_j|), \ if \ |L_i| < |L_j| \end{cases}$$

□

A.2 Proof of Correctness of Algorithm 2

Proof. To prove the correctness of Algorithm 2, we only need to prove that it really returns the exact top-k result to a query star s_i of leaf size n when halting. Suppose we have m sorted lists for s_i. In fact, this algorithm can halt in two conditions:

1) The value of ω is no less than the maximum value of the top-k candidates. Since the top-k queue is maintained by the top-k minimum SED to s_i, when halting, they are naturally the top-k values among all stars having been accessed. If we can improve that all remaining unseen stars having SED no less than the maximum value in top-k, the result is sure to be correct.

1.1) when running on smaller size sorted lists, we have

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

For any unseen star s_j, we have

$$\lambda(s_i, s_j) = T(r_i, r_j) + 2 * n - (t(\chi) + |L_j|), \ where \ T(r_i, r_j) \geq 0$$

Since all lists in this case are sorted in decreasing orders, we have

$$t(\underline{\chi}) + \underline{L} = \sum_{k=1}^{m} \underline{\chi}_k + \underline{L} \geq t(\chi) + |L_j|$$

where all $\chi_i \in \chi$ and L_j are below the halting positions. Therefore, $\omega \leq \lambda(s_i, s_j)$, i.e., unseen stars have $\lambda \geq \omega \geq max\{top - k\}$. The top-k results are correct to be the k minimum values.

1.2) when running on lager size sorted lists, we have

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

In this case, for any unseen star s_j, we have

$$\lambda(s_i, s_j) = T(r_i, r_j) - n - (t(\chi) - 2 * |L_j|), \ where \ T(r_i, r_j) \geq 0$$

Since label lists are sorted in decreasing orders while size list is sorted in increasing orders, we have

$$t(\underline{\chi}) - 2 * \underline{L} = \sum_{k=1}^{m} \underline{\chi}_k - 2 * \underline{L} \geq t(\chi) - 2 * |L_j|$$

where all $\chi_i \in \chi$ and L_j are below the halting positions. Therefore, $\omega \leq \lambda(s_i, s_j)$, i.e., unseen stars have $\lambda \geq \omega \geq max\{top - k\}$. 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 stars. □

A.3 Proof of Correctness of Algorithm 3

Proof. We prove that our candidate set includes all those positive graphs when algorithm halts. Two conditions are considered:

1) Algorithm halts with ω. In this case, let

$$\delta'^{2} = \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$$

where D' is a set containing all unseen graphs.

1.1) If algorithm runs on smaller size sorted lists, entries in each list below s_i have distance $\chi_i \leq \lambda(s_i, \epsilon)$ according to the construction restriction. From the halting condition, we have $\omega = t'(\underline{\chi}) > \tau * \delta'$. Then for any unseen graph $g_j \subseteq D'$, suppose P is the optimal alignment between $S(q)$ and $S(g_j)$. From Definition 3.2, we have

$$\mu(q, g_j) = \sum_{s_i \in S(q)} \lambda(s_i, P(s_i))$$

where $P(s_i)$ is each star in g_j aligned to s_i in q and $P(s_i) \in S(g_j) \cup \{\varepsilon\}$. Since $|g_j| \leq |q|$, and g_j locates below halting positions of ω. We have $\underline{\chi}_i \leq \lambda(s_i, P(s_i))$. Hence, $\omega \leq \mu(q, g_j)$. Then for any $g_j \subseteq D'$, we have

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

Therefore, any unseen $g_j \subseteq D'$ can be safely filtered out.

1.2) Similarly, if algorithm runs on larger size sorted lists, we also can prove that any unseen $g_j \subseteq D'$ can be safely filtered out.

2) Algorithm halts when all sorted lists have been accessed to the ends. In this case, this algorithm will become a linear scan algorithm, which guarantees a correct candidate set without false negative. □

A.4 Proof of Correctness of Algorithm 4

Proof. For KNN query, it is easy for us to prove its correctness based on the Proof A.2 and A.3. We eliminate this proof here. □

A.5 Proof of Theorem 5.1

Proof. 1) From definitions in Section 5.2.2, it is easy to obtain $\zeta(g_1, g_2) \leq L_\mu(g_1, g_2) \leq U_\mu(g_1, g_2)$.

2) We improve $L_\mu(g_1, g_2) \leq \mu(g_1, g_2)$. Suppose P is an optimal alignment between $S(g_1)$ and $S(g_2)$. From Definition 3.2, we have

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

where $P(s_i)$ is each star 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. Then,

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

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

2.2) For the remaining lists below $S''(g_1) = S(g_1) \backslash S'(g_1)$, it is obvious that

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

[2]Note that, δ' is used in all Algorithms hereafter.

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 improve $U_\mu(g_1, g_2) \geq \mu(g_1, g_2)$. From definition of $U_\mu(g_1, g_2)$, we have

$$U_\mu(g_1, g_2) = t'(\underline{\chi}(g_2)) + \overline{\chi} * (|g_2| - |\underline{\chi}(g_2)|)$$

where $\underline{\chi}(g_2)$ be a multiset of distances corresponding to the stars 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 stars corresponding to $\underline{\chi}(g_2)$, and $S'(g_1)$ contains stars of g_1 aligned to $S'(g_2)$ due to $\underline{\chi}(g_2)$.

3.1) If $S'(g_2) \subseteq \{P(s_i) | s_i \in S'(g_1)\}$, we have

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

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

3.2) If $S'(g_2) \nsubseteq \{P(s_i) | s_i \in S'(g_1)\}$, we have

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

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

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

A.6 Proof of Theorem 5.2

Proof. In Figure 18, a different cost matrix $M(S(g_1), S'(g_2))$ is defined for computing $\mu'(S(g_1), S'(g_2))$. For the ϵ star, we define its distance to any existing star s_i in $S(g_1)$ as 0, instead of $\lambda(s_i, \epsilon)$. We apply the Dynamic Hungarian on the cost matrix 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 $\mu'(S(g_1), S'(g_2)) \leq \mu(S(g_1), S(g_2))$.

□

A.7 Other Theorems and Their Proofs

Different bounds and filtering techniques can be easily adopted in our framework to speed up the search. Here, we add some filtering techniques which can be used in our framework. Some are proposed in previous works, and some are new techniques we have found that can used as filtering features.

Theorem A.1. *Let g_1 and g_2 be two graphs, then the number of common star structures between $S(g_1)$ and $S(g_2)$, denoted by $\phi(g_1, g_2)$, satisfies the following:*

$$\phi(g_1, g_2) \geq \max\{size(g_1), size(g_2)\} - \max\{2, [\max\{\delta(g_1), \delta(g_2)\} + 1]\} \cdot \lambda(g_1, g_2)$$

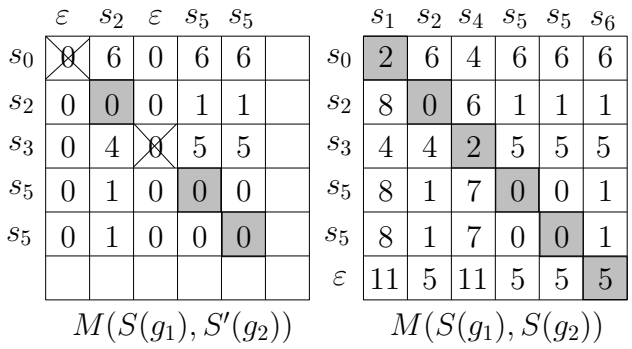


Figure 18: One example for computing $\mu'(S(g_1), S'(g_2))$

Proof. We show that at most $\max\{2, [\max\{\delta(g_1), \delta(g_2)\}+1]\} \cdot k$ star structures differ between two graphs by k edit operations. Assume that edit operations e_1, e_2, ..., e_k transform g_1 to g_2. Then there is a sequence of graphs $g_1 = g'_0 \to g'_1 \to \ldots \to g'_k = g_2$, where $g'_{i-1} \to g'_i$ via e_i for $1 \leq i \leq k$. Assume there are k_1 edge insertion/deletion operations, k_2 vertex insertion/deletion operations and k_3 vertex relabeling operations. We have $k_1 + k_2 + k_3 = k$. We prove our theorem using the analysis as following.

1. Assume that e_i is a edge insertion or deletion operation between two vertices v_i and v_j. Only two star structures rooted by v_i and v_j are affected. Hence, each this kind of operation will change two star structures.

2. Assume that e_j is a insertion or deletion operation on vertex v_i. For insertion, v_i is a new vertex inserted without any edge to other vertices. It can be seen as one star structure ε is changed into one star structure rooted by v_i. Similarly, deletion can be seen as changing one structure rooted by v_i to one star structure ε.

3. A vertex relabeling operation on v_i can change $deg(v_i)+1$ star structures. This is because, in addition to the star structure rooted by v_i, all the star structures derived by the $deg(v_i)$ vertices connected with v_i are also affected by this operation.

Accordingly, k edit operations can at most change $2 \cdot k_1 + 1 \cdot k_2 + (\max\{\delta(g_1), \delta(g_2)\}+1) \cdot k_3$ star structures to transform g_1 to g_2. Thus, g_1 and g_2 must share at least the following number of common star structures:

$$\begin{aligned} \phi(g_1, g_2) &\geq \max\{size(g_1), size(g_2)\} - [2 \cdot k_1 + 1 \cdot k_2 + (\max\{\delta(g_1), \delta(g_2)\}+1) \cdot k_3] \\ &\geq \max\{size(g_1), size(g_2)\} - \max\{2, [\max\{\delta(g_1), \delta(g_2)\}+1]\} \cdot (k_1 + k_2 + k_3) \\ &\geq \max\{size(g_1), size(g_2)\} - \max\{2, [\max\{\delta(g_1), \delta(g_2)\}+1]\} \cdot \lambda(g_1, g_2) \end{aligned}$$

□

Proposition 1. *Let* $diff(g_1, g_2) = ||V(g_1)| - |V(g_2)|| + ||E(g_1)| - |E(g_2)||$. *If* $diff(g_1, g_2) > k$, *then* $\lambda(g_1, g_2) > k$.

Proposition 2. *If* $\phi(g_1, g_2) < \max\{size(g_1), size(g_2)\} - \max\{2, [\max\{\delta(g_1), \delta(g_2)\}+1]\} \cdot k$, *then* $\lambda(g_1, g_2) > k$.

Proof. We prove using the contrapositive proposition: If $\lambda(g_1, g_2)$ is not larger than k, then the size difference $diff(g_1, g_2) \leq k$. Using edit operations to transform g_1 to g_2, it is clear that the size of the transformed graph from g_1 must be equal to $size(g_2)$. Therefore, at least $||V(g_1)| - |V(g_2)|| + ||E(g_1)| - |E(g_2)||$ edit operations are needed for the transformation, i.e. $diff(g_1, g_2) \leq \lambda(g_1, g_2)$. Thus, $diff(g_1, g_2) \leq k$. □

B Algorithms

This section provides detailed algorithm pseudocodes to supplement Section 5. The description steps of these algorithms can be seen in Section 5.

Algorithm 1 Merge

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

```
1: end ← true, max ← 0, p ← 0
2: initialize an array A' with values of A;
3: while end == true do
4:    for i = 0 to n − 1 do
5:       if A'[i] == A[i + 1] then
6:          continue;
7:       end ← false;
8:       if max < L[A'[i]].count then
9:          max ← L[A'[i]].count;
10:         p ← i;
11:   l.push_back(L[A'[p]]);
12:   A'[p] + +;
```

Algorithm 2 TA-based star searching algorithm

Require: m sorted lists l for s with leaf size n and a flag low
Ensure: The top-k similar stars

```
1: top − k ⇐ ∅;
2: for all sorted lists l_i with i = 1 . . . m do
3:    s_id ⇐ l_i.getNext().id;
4:    if i < m then
5:       χ_i ⇐ l_i.getNext().occurrence
6:    else
7:       L ⇐ l_i.getNext().occurrence
8:    if s_id is not seen before then
9:       calculate λ(s, s_id);
10:      if The size of top − k is less than k then
11:         top − k.push(s_id);
12:         continue;
13:      if λ(s, s_id) < max{λ|λ ∈ top − k} then
14:         top − k.pop();
15:         top − k.push(s_id);
16:   if low is true then
17:      ω = 2 ∗ n − (t(χ) + L);
18:   else
19:      ω = −n − (t(χ) − 2 ∗ L);
20:   if ω ≥ max{λ|λ ∈ top − k} then
21:      return top − k;
22: if all lists are accessed to end then
23:   return top − k;
```

Algorithm 3 TA-based range query algorithm

Require: A database D, m sorted lists l for q and a threshold τ
Ensure: All $g_i \in D$ s.t. $\lambda(q, g_i) \leq \tau$

```
1: candidate ⇐ ∅; flag ⇐ false;
2: for all sorted lists l_i with i = 1...m do
3:    g_id ⇐ l_i.getNext().id;
4:    χ_i ⇐ l_i.getNext().dist;
5:    if g_id is not seen before then
6:       if L_m(q, g_id) > τ then
7:          continue;
8:       if U_m(q, g_id) ≤ τ then
9:          g_id is reported in final result;
10:         continue;
11:      candidate.push_back(g_id);
12:      ω = t'(χ);
13:   if ω = t'(χ) > τ * δ'} then
14:      flag ⇐ true
15:      break;
16: if flag! = true then
17:   post process the remaining graphs not appeared;
```

Algorithm 4 TA-based KNN query algorithm

Require: A database D, m sorted lists l for q and a threshold τ
Ensure: All $g_i \in D$ containing top-k graphs

```
1: candidate ⇐ ∅; top − k ⇐ ∅; flag ⇐ false;
2: for all sorted lists l_i with i = 1...m do
3:    g_id ⇐ l_i.getNext().id;
4:    χ_i ⇐ l_i.getNext().dist;
5:    if g_id is not seen before then
6:       candidate.push_back(g_id);
7:       if The size of top − k is less than k then
8:          top − k.push(g_id);
9:          continue;
10:      if L_m(q, g_id) ≥ max{U_m|U_m ∈ top − k} then
11:         continue;
12:   ω = t'(χ);
13:   if ω = t'(χ) > max{U_m|U_m ∈ top − k} * δ'} then
14:      flag ⇐ true
15:      break;
16: if flag! = true then
17:   post process the remaining graphs not appeared;
```

Algorithm 5 CA-based range query algorithm

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

```
1: candidate ⇐ ∅; flag ⇐ false;
2: for all sorted lists l_i with i = 1...m do
3:    g_id ⇐ l_i.getNext().id;
4:    χ_i ⇐ l_i.getNext().dist;
5:    maintain a distance accumulator ζ(q, g_id) for each graph;
6:    maintain a set for stars seen S'(g_id) for each graph;
7:    if scandepth%h == 0 then
8:       for all g_id is not seen before do
9:          if ζ(q, g_id) > τ then
10:            continue;
11:         if L_μ(q, g_id) > τ then
12:            continue;
13:         if U_μ(q, g_id) > τ then
14:            g_id is reported in final result;
15:            continue;
16:         if μ'(S(q), S'(g_id)) > τ then
17:            continue;
18:         L_m(q, g_id) and U_m(q, g_id) are similar to TA;
19:         candidate.push_back(g_id);
20:   ω = t'(χ);
21:   if ω = t'(χ) ≥ τ * δ'} then
22:      flag ⇐ true
23:      break;
24: if flag! = true then
25:   post process the remaining graphs not appeared;
```

Algorithm 6 CA-based KNN query algorithm

Require: a database D, m sorted lists l for q and h
Ensure: All $g_i \in D$ containing top-k graphs

1: $candidate \Leftarrow \emptyset$; $top-k \Leftarrow \emptyset$; $flag \Leftarrow false$;
2: **for all** sorted lists l_i with $i = 1 \ldots m$ **do**
3: $g_{id} \Leftarrow l_i.getNext().id$;
4: $\underline{\chi}_i \Leftarrow l_i.getNext().dist$;
5: maintain a distance accumulator $\zeta(q, g_{id})$ for each graph;
6: maintain a set for stars seen $S'(g_{id})$ for each graph;
7: **if** $scandepth\%h == 0$ **then**
8: **for all** g_{id} is not seen before **do**
9: $candidate.push_back(g_{id})$;
10: **if** The size of $top-k$ is less than k **then**
11: $top-k.push(g_{id})$;
12: continue;
13: **if** $\zeta(q, g_{id}) > max\{U_m | U_m \in top-k\} * \delta'$ **then**
14: continue;
15: **if** $L_\mu(q, g_{id}) > max\{U_m | U_m \in top-k\} * \delta'$ **then**
16: continue;
17: **if** $\mu'(S(q), S'(g_{id})) > max\{U_m | U_m \in top-k\} * \delta'$ **then**
18: continue;
19: $L_m(q, g_{id})$ and $U_m(q, g_{id})$ are similar to TA;
20: $\omega = t'(\underline{\chi})$;
21: **if** $\omega = t'(\underline{\chi}) \geq max\{U_m | U_m \in top-k\} * \delta'\}$ **then**
22: $flag \Leftarrow true$
23: break;
24: **if** $flag! = true$ **then**
25: post process the remaining graphs not appeared;