MSQ-Index: A Succinct Index for Fast Graph Similarity Search

Xiaoyang Chen, Hongwei Huo, *Senior Member, IEEE,* Jun Huan, *Senior Member, IEEE,* Jeffrey Scott Vitter, *Fellow, IEEE,* Weiguo Zheng and Lei Zou

Abstract—Graph similarity search under the graph edit distance constraint has received considerable attention in many applications, such as bioinformatics, data mining, pattern recognition and social networks. Existing methods for this problem have limited scalability because of the huge amount of memory they consume when handling very large graph databases with tens of millions of graphs. In this paper, we present a succinct index that incorporates succinct data structures and hybrid encoding to achieve improved query time performance with minimal space usage. Specifically, the space usage of our index requires only 5%–15% of the previous state-of-the-art indexing size while at the same time achieving several times acceleration in query time on the tested data. We also improve the query performance by augmenting the global filter with range searching, which allows us to perform similarity search in a reduced region. In addition, we propose two effective lower bounds together with a boosting technique to obtain the possible smallest candidate set. Extensive experiments demonstrate that our proposed approach is superior both in space and filtering to the state-of-the-art approaches. To the best of our knowledge, our index is the first in-memory index for this problem that successfully scales to cope with the large dataset of 25 million chemical structure graphs from the PubChem dataset. The source code is available online.

Index Terms—Graph indexing, similarity search, filter boosting, succinct index, hybrid encoding

1 Introduction

Graphs are widely used to model complicated data objects in many disciplines, such as bioinformatics [24], social networks [25], software and data engineering [35]. Effective analysis and management of graph data become increasingly important. Many graph-based queries have been investigated, which can be roughly divided into two broad categories: graph exact search [2], [34] and graph similarity search [19], [28], [39]. Compared with exact search, similarity search can provide a robust solution that permits error-tolerant and supports for searching not precisely defined patterns.

Similarity computation between two labeled graphs is a core operation of graph similarity search. There are at least four similarity metrics being well investigated: graph edit distance [12], [32], [36], [39], maximal common subgraph distance [4], [10], graph alignment [1], [33], and graph kernel function [29], [31]. In this paper, we focus on the graph edit distance (GED) because it is applicable to virtually all types of data graphs and can also capture structural differences. GED has been widely used in various applications, including pattern recognition [9], graph classification [23] and chemistry analysis [22].

The graph edit distance between two graphs h and g, denoted by $ged(h, g)$, is the minimum length of an edit path between h and g, where an *edit path* is a sequence of edit operations that transforms one graph to another. Typical edit operations [18] are inserting and deleting a vertex or an edge, and relabeling a vertex or an edge.

Based on the metric, GED, we study the following graph similarity search problem: Given a graph database $\mathcal{G}$, a query graph h and a threshold τ, this problem aims to find all graphs g in $\mathcal{G}$ such that $ged(h, g) \leq \tau$. Unfortunately, computing GED is known to be an NP-hard problem [36]. Thus, the basic solution for this problem that computes GED for all pairs of h and data graphs $g \in \mathcal{G}$ may be lead to unsatisfactory computational efficiency.

Most of existing methods [6], [32], [37], [38], [39] adopt the filtering-and-verification framework to speed up graph similarity search. In the filtering phase, GED lower bounds are employed to prune as many false-positive graphs from $\mathcal{G}$; this phase can be efficiently accomplished with specified index structures. The remaining unpruned graphs constitute a candidate set, $\mathcal{C}$, and are validated with expensive GED computations in the verification phase.

1.1 Limitations

Lots of graph similarity search methods have been proposed [32], [36], [37], [38], [39] and gained promising results. However, these methods still suffer from huge storage and expensive query cost when dealing with large transaction graph databases [4].

Some methods have to build an index structure to efficiently prune graphs in the filtering phase. For instance,

- *Xiaoyang Chen and Hongwei Huo are with Xidian University, Xi'an 710071, Shaanxi, China. E-mail: xychen1991@stu.xidian.edu.cn and hwhuo@mail.xidian.edu.cn*
- *Jun Huan is with Baidu Research, Baidu Technology Park, No. 10 Xibeiwang East Road, Haidian District, Beijing 100094, China. E-mail: huanjun@baidu.com*
- *Jeffrey Scott Vitter is with the University of Mississippi, University, MS 38677-1848, USA. E-mail: jsv@OleMiss.edu*
- *Weiguo Zheng is with School of Data Science, Fudan University, Shanghai 200433, China. E-mail: zhengweiguo@fudan.edu.cn*
- *Lei Zou is with Peking University, Beijing 100080, China. E-mail: zoulei@pku.edu.cn*

Manuscript received XX XX, 2017; revised XX XX, 201X

GSimJoin [37] built an inverted index for all path-based q-grams, and Mixed [39] proposed an index structure, *u-tree*, storing all branch and disjoint substructures. These index structures usually store each entry in an int type. This may yield an unaffordable storage cost for large graph databases.

On the other hand, some may produce expensive query cost owing to loose GED lower bounds and inefficient filtering. For the former, they would produce a large candidate set, leading to unacceptable verification cost. For the later, they filter data graphs from the whole database while usually only a small proportion of data graphs are similar to the query graph.

1.2 Contributions

To solve the above issues, we propose a space-efficient index structure, called MSQ-Index, by incorporating succinct data structures and hybrid encoding. MSQ-Index consists of multiple *succinct q-gram trees*, where each tree is compressed through a hybrid encoding schema. Meanwhile, several auxiliary succinct data structures with a little storage cost are proposed to ensure that each compressed entry can be accessed in constant time.

MSQ-Index can also provide efficient query processing, which benefits from the following two aspects: (1) two GED lower bounds together with a boosting technique are proposed to obtain the smallest possible candidate set; (2) a preprocessing method is provided to help MSQ-Index perform similarity search only on a small percentage of data graphs in the database.

In summary, our contributions are summarized below.

- We propose a succinct index structure, called *succinct q-gram tree*, which combines succinct data structures and hybrid encoding to achieve efficient similarity search with minimal space usage. Each entry in this tree is compressed and needs only several bits to store, which takes much fewer bits than that used to store an int type in existing indexing methods.
- We propose two effective GED lower bounds, called *degree-based q-gram counting lower bound* and *degree-sequence lower bound*, to prune data graphs. Moreover, we provide a boosting technique to improve these lower bounds.
- We propose a preprocessing method, which helps us perform similarity search only on a small percentage of data graphs in the database.
- We have conducted extensive experiments on both small and large datasets to evaluate the index size, construction time, filtering ability, and response time. The results confirm the effectiveness and efficiency of our proposed method and show that it can scale well to cope with the large dataset consisting of 25 million chemical compounds from the PubChem dataset.
- The source code is available online [5].

The rest of this paper is organized as follows: In Section 2, we investigate research works related to this paper. In Section 3, we introduce the problem definition and the filtering principle. In Section 4, we present our indexing method MSQ-Index. In Section 5, we give the theoretical analysis of MSQ-Index. In Section 6, we report the experimental results. Finally, we make concluding remarks in Section 7.

2 Related Work

Recently, the graph similarity search problem has received considerable attentions, and existing methods to this problem can be found in the literatures [6], [11], [12], [21], [30], [32], [36], [37], [38], [39].

Filters. Inspired by the q-gram concept in string similarity queries, Wang et al. firstly proposed a tree-based q-gram counting filter in κ-AT [32], where a tree-based q-gram is defined as a κ-adjacent subtree consisting of a vertex and paths whose length is less than κ starting from this vertex. While, Zhao et al. considered a simple path as a path-based q-gram in GSimJoin [37]. The principle of the q-gram counting filter is stated as follows: If $ged(h, g) \leq \tau$, then the number of common q-grams between two graphs h and g satisfies $|Q(h) \cap Q(g)| \geq \max\{|Q(h)| - D_s(h) \cdot \tau, |Q(g)| - D_s(g) \cdot \tau\}$, where $Q(\cdot)$ is the q-gram multiset and $D_s(\cdot)$ is the maximum number of q-grams that can be affected by an edit operation. Clearly, the q-gram counting filter can be efficiently finished in $\mathcal{O}(\max\{|Q(h)|, |Q(g)|\})$ time. Nevertheless, this counting filter may suffer from poor filtering ability when $D_s(h)$ and $D_s(g)$ are large.

Another class of filter is the mapping distance-based filter, which derives GED lower bound by computing the mapping distance between two graphs. In C-Star [36], Zeng et al. decomposed a graph into star structures (i.e., 1-adjacent subtrees) and then computed the mapping distance of star structures of h and g through the bipartite matching. While, Zheng et al. employed the branch structures (i.e., star structures without end vertices) in Mixed [39]. The Hungarian algorithm [20] is employed to compute the bipartite matching, whose time complexity is $\mathcal{O}(|V|^3)$, where $|V| = \max\{|V_h|, |V_g|\}$. This class of filter may be inefficient when it is performed pairwise computations between h and all data graphs g in the database [30].

The substructures in the aforementioned filters are fixed-size, whereas Zhao et al. introduced a partition-based filter in Pars [38], which divided each data graph g into $\tau + 1$ non-overlapping substructures and pruned g if there exists no substructure that is subgraph isomorphic to h. Later, Liang et al. proposed a parameterized, partition-based lower bound that can be instantiated into a series of tight lower bounds in ML-Index [21]. Whether it is Pars or ML-Index, it needs to perform subgraph isomorphism test during similarity search. However, subgraph isomorphism test is an NP-hard problem [34] and may consume a large amount of time. Therefore, both Pars and ML-Index may suffer from inefficient filtering for large graph databases.

It is worth to mention that the above filters show different performance on different datasets and one can hardly prove the merits of them in theory [11].

Indexing Techniques. Several indexing techniques have been proposed to speed up the computation of the above filters through maximizing computation sharing. In κ-AT, GSimJoin, and Pars, they employed an inverted index to store the tree-based q-grams, path-based q-grams, and disjoint substructures, respectively. For Mixed, it used an R-tree [14] like index structure u-tree to store all branch

TABLE 1: Notations

Symbol	Description
$\lvert\cdot\rvert$	size of a set, or an array.
$\mathcal{G}$	graph database
τ	threshold of graph edit distance
g (or h)	data (or query) graph
$ged(h,g)$	graph edit distance between h and g
$D(g)$ (or $L(g)$)	degree-based (or label-based) q-gram multiset
σ_g	degree-sequence of g
δ	the boosting parameter
$\lambda^D(h,g)$	degree-based q-gram counting lower bound
$\lambda^L(h,g)$	label-based q-gram counting lower bound
$\lambda^S(h,g)$	degree-sequence lower bound
$\xi_i^D(h,g)$	the i-th boosted lower bound of $\lambda^D(h,g)$
$\xi_i^L(h,g)$	the i-th boosted lower bound of $\lambda^L(h,g)$
$\xi_i^S(h,g)$	the i-th boosted lower bound of $\lambda^S(h,g)$
$\mathcal{A}$	the whole region formed by $\mathcal{G}$
$\mathcal{Q}_h$	the query region formed by h

and disjoint substructures. SEGOS [30] introduced a two-level index structure to speed up C-Star. In ML-Index, they designed a multi-layered index structure, where each layer employed an inverted index to store the partitioned substructures. The above index structures all employ an int type to store each entry. This may yield their index sizes too large to fit into the main memory when dealing with large graph databases.

GED Computation. A widely used method to compute GED is based on the A* algorithm [15], [27]. Zhao et al. [37], [38] designed several heuristic functions to improve A*. Recently, Gouda et al. proposed a novel edge-based mapping method for exact GED computation, called CSI_GED [12], based on common substructure isomorphism. CSI_GED employs the backtracking search combined with three specific heuristics, gaining an excellent performance. Later, Chen et al. [7] introduced a beam-stack search based method for GED computation.

3 Problem Definition and Filtering Principle

In this section, we first provide formal definitions of graph edit distance and graph similarity search in section 3.1 and then introduce the filtering principle in section 3.2. Table 1 lists some notations used in the paper.

3.1 Problem Definition

For ease of presentation, we only focus on simple, undirected graphs without multi-edges or self-loops. Let Σ be a set of discrete-valued labels. We define a labeled graph as a triplet $g = (V_g, E_g, l_g)$, where V_g is the set of vertices, $E_g \subseteq V_g \times V_g$ is the set of edges, $l_g : V_g \cup E_g \to \Sigma$ is the labeling function that assigns a label to a vertex or an edge. For a vertex u, we use $l_g(u)$ to denote its label. Similarly, $l_g(e(u,v))$ is the label of edge $e(u,v)$. $\Sigma_{V_g} = \{l_g(u) : u \in V_g\}$ and $\Sigma_{E_g} = \{l_g(e(u,v)) : e(u,v) \in E_g\}$ are the label multisets of V_g and E_g, respectively. The graph size refers to $|V_g|$ in this paper.

Definition 1 (Subgraph Isomorphism [34])**.** *Given graphs g and h, g is subgraph isomorphic to h, denoted by $g \subseteq h$, if there exists an injective function $f : V_g \to V_h$, such that (1) $\forall v \in V_g$, $f(v) \in V_h$ and $l_g(v) = l_h(f(v))$; (2) $\forall e(u,v) \in E_g$, $e(f(u), f(v)) \in E_h$ and $l_g(e(u,v)) = l_h(e(f(u), f(v)))$. If $g \subseteq h$ and $h \subseteq g$, then g is graph isomorphic to h (or vice versa), denoted by $g \cong h$.*

Six edit operations [3] can be used to transform one graph to another, including inserting/deleting a vertex or an edge, and substituting the label of a vertex or an edge. An *edit path* $P = \langle p_1, p_2, \ldots, p_k \rangle$ is a sequence of edit operations that transforms graph h to graph g (or vice versa), denoted as $h = h^0 \xrightarrow{p_1} \ldots \xrightarrow{p_k} h^k \cong g$, where edit operation p_i is applied to graph h^{i-1} to obtain the graph h^i, for $1 \le i \le k$. We define the number of edit operations in P as the length of the edit path and call P *optimal* only when it has the minimum length among all possible edit paths.

Definition 2 (Graph Edit Distance)**.** *Given two graphs h and g, the graph edit distance between them, denoted by $ged(h,g)$, is the length of an optimal edit path between them, or the minimum number of edit operations needed to transform one graph to another.*

Problem statement: Given a graph database $\mathcal{G} = \{g_1, g_2, \ldots, g_{|\mathcal{G}|}\}$, a query graph h and a threshold τ, the problem is to find all data graphs g in $\mathcal{G}$ such that $ged(h,g) \le \tau$.

Example 1. Fig. 1 shows a query graph h and three data graphs g_1, g_2 and g_3. We can compute $ged(h, g_1) = 5$, $ged(h, g_2) = 4$, and $ged(h, g_3) = 3$. If $\tau = 3$, then g_3 is the required graph.

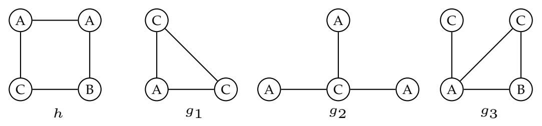


Fig. 1: Query graph h and data graphs g_1, g_2, and g_3.

3.2 Filtering Principle

In this section, we first propose two effective GED lower bounds and then provide a technique to boost them.

3.2.1 *Q-Gram Counting Lower Bounds*

Definition 3 (Degree-based q-gram)**.** *Let $D_v = (l_g(v), N(v), d(v))$ be the degree structure (also called branch structure [39]) of a vertex v, where $l_g(v)$ is the label of v, $N(v)$ is the multiset of edge labels for those edges adjacent to v, and $d(v)$ is the degree of v. The degree-based q-gram of v is defined as D_v, and the degree-based q-gram multiset of g is $D(g) = \{D_v : v \in V_g\}$.*

As discussed in [39], a vertex edit operation (i.e., inserting, deleting or substituting a vertex) affects one degree-based q-gram, and an edge edit operation (i.e., inserting, deleting or substituting an edge) affects two degree-based q-grams. Based on the principle of q-gram counting filter [32], [37], we then establish the following degree-based q-gram counting lower bound.

Theorem 1 (Degree-based q-gram counting lower bound)**.** *Given two graphs h and g, then we have $ged(h,g) \ge \lambda^D(h,g)$, where $\lambda^D(h,g) = \max\{|V_h|, |V_g|\} - \frac{1}{2}(|\Sigma_{V_h} \cap \Sigma_{V_g}| + |D(h) \cap D(g)|)$.*

Proof. See Appendix A in supplementary materials. □

On the other hand, considering the label of a vertex or an edge as a label-based q-gram, we then obtain the label-based q-gram counting lower bound: $ged(h,g) \ge \lambda^L(h,g)$, where $\lambda^L(h,g) = \max\{|V_h|, |V_g|\} + \max\{|E_h|, |E_g|\} - |L(h) \cap L(g)|$,

$L(h) = \Sigma_{V_h} \cup \Sigma_{E_h}$, and $L(g) = \Sigma_{V_g} \cup \Sigma_{E_g}$. This lower bound is a rewritten form of the label counting filter [37].

Fig. 2 shows the degree-based and label-based q-gram multisets of graphs shown in Fig. 1, where the vertex degree in each degree-based q-gram is omitted. The number on the left of each q-gram is the times of this q-gram occurring in the graph.

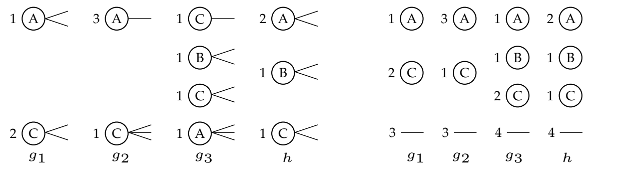


Fig. 2: Degree-based(left) and label-based(right) q-gram multisets.

Example 2. Let $\tau = 2$. Considering the graphs shown in Fig. 1, we compute $\lambda^D(h, g_2) = 2.5$ and $\lambda^L(h, g_2) = 2$. Clearly, $\lambda^D(h, g_2) > \tau$, thus we can filter g_2 out. Similarly, we obtain $\lambda^D(h, g_1) = 2$ and $\lambda^L(h, g_1) = 3$, thus we also can filter g_1 out since $\lambda^L(h, g_1) > \tau$. However, for g_3 we have $\lambda^D(h, g_3) = 2 \leq \tau$ and $\lambda^L(h, g_3) = 1 \leq \tau$, which means that g_3 passes these two q-gram counting lower bounds.

3.2.2 Degree-sequence Lower Bound

In this section, we propose a *degree-sequence lower bound*, which is capable of filtering g_3 out. The idea behind is that if h is isomorphic to g, then they must have the same degree sequence; thus, the distance between the degree sequences of h and g is a lower bound of $ged(h, g)$.

Definition 4 (Degree sequence)**.** *The degree sequence of graph g, denoted by σ_g, is defined as a monotonic non-increasing sequence consisting of the degree of vertices of g.*

Definition 5 (Degree-sequence distance)**.** *Given two degree sequences σ_h and σ_g, the distance between them is defined as $dis(\sigma_h, \sigma_g) = \lceil \frac{1}{2} \sum_{\sigma_g[i] > \sigma_h[i]} (\sigma_g[i] - \sigma_h[i]) \rceil + \lceil \frac{1}{2} \sum_{\sigma_g[i] \leq \sigma_h[i]} (\sigma_h[i] - \sigma_g[i]) \rceil$.*

The distance $dis(\sigma_h, \sigma_g)$ is the lower bound of the sum of the number of edge insertion and deletion operations in an optimal edit path between h and g. We multiply by $\frac{1}{2}$ in $dis(\sigma_h, \sigma_g)$ since one edge insertion/deletion operation changes degrees of two vertices. Combining with the lower bound of vertex edit operations, we then establish the following degree-sequence lower bound.

Theorem 2 (Degree-sequence lower bound)**.** *Given two graphs h and g, then we have $ged(h, g) \geq \lambda^S(h, g)$, where $\lambda^S(h, g) = \max\{|V_h|, |V_g|\} - |\Sigma_{V_h} \cap \Sigma_{V_g}| + dis(\sigma'_h, \sigma'_g)$, $\sigma'_h = [\sigma_h[1], \ldots, \sigma_h[|V_h|], 0_1, \ldots, 0_{|V|-|V_h|}]$ and $\sigma'_g = [\sigma_g[1], \ldots, \sigma_g[|V_g|], 0_1, \ldots, 0_{|V|-|V_g|}]$.*

Proof. See Appendix B in supplementary materials □

Example 3. For graphs h and g_3 shown in Fig. 1, we have $\sigma_h = [2\ 2\ 2\ 2]$ and $\sigma_{g_3} = [3\ 2\ 2\ 1]$, and then compute $dis(\sigma'_h, \sigma'_g) = \lceil \frac{1}{2}(3-2) \rceil + \lceil \frac{1}{2}(2-1) \rceil = 2$. Finally, we obtain $\lambda^S(h, g) = 3 > \tau = 2$, and thus can filter g_3 out.

3.2.3 Boosting

In this section, we provide a technique to boost these three lower bounds $\lambda^D(h, g)$, $\lambda^L(h, g)$, and $\lambda^S(h, g)$ above. Without loss of generality, we assume that $|V_h| \geq |V_g|$ and use $\lambda(,)$ to denote any one of these three lower bounds (i.e., $\lambda \in \{\lambda^D, \lambda^L, \lambda^S\}$) to explain the boosting technique.

Let P be the optimal edit path that transforms h to g. It is trivial that Δ vertex deletions exist in P, where $\Delta = |V_h| - |V_g|$. Accordingly, there is a sequence of graphs $h = h^0 \rightarrow h^1 \rightarrow \cdots \rightarrow h^\Delta \rightarrow \cdots \rightarrow h^k \cong g$, where $h^i \subseteq h^{i-1}$ is an *induced subgraph* of h^{i-1} and obtained by deleting a vertex in h^{i-1} and edges adjacent to this vertex, for $1 \leq i \leq \Delta$; clearly, h^i contains $|V_h| - i$ vertices.

Let $\Psi(h, i)$ be the set of h's induced subgraphs containing $|V_h| - i$ vertices, for $1 \leq i \leq \Delta$. We have $h^i \in \Psi(h, i)$. Let $\xi_i(h, g) = \min_{o \in \Psi(h,i)}\{ged(h, o) + \lambda(o, g)\}$. Then $\xi_i(h, g)$ is a lower bound of $ged(h, g)$, which boosts the original lower bound $\lambda(h, g)$.

Theorem 3. *Given two graphs g and h, then we have $\lambda(h, g) = \xi_0(h, g) \leq \xi_1(h, g) \leq \cdots \leq \xi_\Delta(h, g) \leq ged(h, g)$, where $\lambda \in \{\lambda^D, \lambda^L, \lambda^S\}$, $\Delta = |V_h| - |V_g|$, and $\xi_i(h, g) = \min_{o \in \Psi(h,i)}\{ged(h, o) + \lambda(o, g)\}$.*

Proof. see Appendix C in supplementary materials. □

Theorem 3 states that we can obtain a sequence of boosting lower bounds. Hereafter, we use $\xi_i^D(h, g)$, $\xi_i^L(h, g)$ and $\xi_i^S(h, g)$ to denote the ith boosted lower bounds when taking $\lambda(,)$ as $\lambda^D(,)$, $\lambda^L(,)$ and $\lambda^S(,)$, respectively.

Example 4. For the graphs h and g_1 shown in Fig. 1, we compute $\lambda^D(h, g_1) = 2$, $\lambda^L(h, g_1) = 3$ and $\lambda^S(h, g_1) = 3$. Using the boosting technique, we compute that the first boosted lower bounds are $\xi_1^D(h, g_1) = 4.5$, $\xi_1^L(h, g_1) = 5$ and $\lambda_1^S(h, g_1) = 5$. Clearly, all of the boosted lower bounds are tighter than the original ones.

Based on Theorem 3, we propose the filtering method filterGraph in Alg. 1 to determine whether pruning a data graph g, where $\delta \leq \tau$ is a user-given boosting parameter, k (line 1) is the maximum layer that the boosted lower bounds can be applied.

Algorithm 1: filterGraph(h, g, τ, δ)

```
1  $k \leftarrow \max\{0, \min\{\delta, |V_h| - |V_g|\}\}$;
2  $pruned \leftarrow$ flase;
3  for $i \leftarrow 0$ to $k$ do
4      compute $\xi_i^D(h, g)$ and $\xi_i^L(h, g)$ ;
5      if $\xi_i^D(h, g) > \tau$ or $\xi_i^L(h, g) > \tau$ then
6          $pruned \leftarrow$ true ;
7      else
8          compute $\xi_i^S(h, g)$;
9          if $\xi_i^S(h, g) > \tau$ then
10             $pruned \leftarrow$ true ;
11     if $pruned =$ true then
12         return true ;
13 return false ;
```

Complexity Analysis. For an induced subgraph $o \in \Psi(h, i)$, for $0 \leq i \leq k$, we can compute $\lambda^D(o, g)$, $\lambda^L(o, g)$, and $\lambda^S(o, g)$ in $\mathcal{O}(|V_o|)$, $\mathcal{O}(|V_o| + |E_o|)$, and $\mathcal{O}(|V_o| \log |V_o|)$ time, respectively. Since $\Psi(h, i)$ contains $\binom{i}{|V_h|} \leq |V_h|^i$ induced subgraphs, we can compute $\xi_i^D(h, g)$, $\xi_i^L(h, g)$, and $\xi_i^S(h, g)$

in $\mathcal{O}(|V_h|^{i+1})$, $\mathcal{O}((|V_h|+|E_h|)|V_h|^i)$, and $\mathcal{O}(|V_h|^{i+1}\log|V_h|)$ time, respectively. Thus, the time complexity of Alg. 1 is $\mathcal{O}(\sum_{i=0}^{k}|V_h|^i(2|V_h|+|E_h|+|V_h|\log|V_h|)) = \mathcal{O}(|V_h|^{\delta+2})$, for $|E_h| \le |V_h|^2$ and $k \le \delta$.

4 MSQ-Index

In this section, we propose a succinct index structure, called MSQ-Index (**M**ultiple **S**uccinct **Q**-Gram Tree **Index**), which supports for fast similarity search on a large database $\mathcal{G}$. MSQ-Index consists of the following three steps:

(1) **Preprocessing.** For each graph g in $\mathcal{G}$, we map it to a vertex-edge-based 2D point, $(|V_g|, |E_g|)$. Clearly, these points can form a rectangle region. Meanwhile, we convert the number counting filter [36] to a query rectangle. By dividing the whole region into non-overlapping subregions, we can perform similarity search in a reduced query region rather than the whole region.

(2) **Succinct index construction.** We build a q-gram tree over each subregion in which leaf nodes store q-gram information of data graphs and internal nodes summary its child nodes, and then compress each q-gram tree to minimize the space usage. Meanwhile, we create auxiliary succinct data structures to support fast query.

(3) **Query processing.** We perform similarity search over succinct q-gram trees built only in the reduced query region. The remaining unpruned graphs constitute a candidate set $\mathcal{C}$ and we can employ existing GED computation methods [7], [12], [27] to verify graphs in $\mathcal{C}$.

4.1 Preprocessing

Existing indexing methods such as [30], [32], [37], [38], [39] perform similarity search on the whole database $\mathcal{G}$. However, this may be inefficient since typically only a small part of data graphs in $\mathcal{G}$ are similar to a given query graph h. Here we propose a preprocessing method, which helps us search only on some data graphs.

Transformation. For each data graph g, we map it to a vertex-edge-based 2D point, $(|V_g|, |E_g|)$, where the x-coordinate and y-coordinate denote the number of vertices and edges in g, respectively. Then, we obtain a set of points $\{(|V_g|, |E_g|) : g \in \mathcal{G}\}$, and these points can form a rectangle region $\mathcal{A} = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, where $x_{\min}/y_{\min}$ and $x_{\max}/y_{\max}$ are the smallest and largest numbers of vertices/edges in $\mathcal{G}$, respectively.

Division. Given a division point (x_0, y_0) and a length $\ell > 0$, we divide $\mathcal{A}$ into non-overlapping subregions as follows: First, we compute an initial square subregion $\mathcal{A}_{0,0}$ formed by the point set $\{(x,y) : |x - x_0| + |y - y_0| \le \ell\}$. Then, we make extensions along the surrounding of $\mathcal{A}_{0,0}$ to obtain subregions $\mathcal{A}_{i,j}$ of the same size as $\mathcal{A}_{0,0}$, where i and j denote the relative offsets of the extensional subregion w.r.t. $\mathcal{A}_{0,0}$ in lines $y = x$ and $y = -x$, respectively. Finally, we repeat the above extension process until all points in $\mathcal{A}$ are exhausted.

Reduced Query Region. Once we have partitioned $\mathcal{A}$ into non-overlapping subregions such that $\mathcal{A} = \bigcup_{i,j}\mathcal{A}_{i,j}$ and $\mathcal{A}_{i,j} \cap \mathcal{A}_{i',j'} = \emptyset$ for all $i \ne i'$ and $j \ne j'$, we can reduce the query region from the whole region $\mathcal{A}$ to a reduced region $\mathcal{Q}_h$ below.

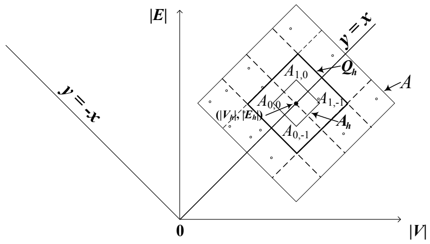


Fig. 3: Illustration of $\mathcal{A}_h$, $\mathcal{Q}_h$ and $\mathcal{A}$.

Definition 6 (Query rectangle and region)**.** *Given a query graph h and a threshold τ, the query rectangle $\mathcal{A}_h$ is defined as the rectangle formed by the point set $\{(x,y) : |x - |V_h|| + |y - |E_h|| \le \tau\}$. The query region $\mathcal{Q}_h$ is the union of all subregions intersecting with $\mathcal{A}_h$, that is, $\mathcal{Q}_h = \bigcup_{i,j}\mathcal{A}_{i,j}$, s.t. $\mathcal{A}_{i,j} \cap \mathcal{A}_h \ne \emptyset$.*

For a data graph g, if $ged(h,g) \le \tau$, then we have $||V_g| - |V_h|| + ||E_g| - |E_h|| \le \tau$ [36]; consequently, $(|V_g|, |E_g|) \in \mathcal{A}_h$. Since $\mathcal{A}_h \subseteq \mathcal{Q}_h$, we have $(|V_g|, |E_g|) \in \mathcal{Q}_h$. Therefore, searching only in $\mathcal{Q}_h$ will not produce false positives. In the example of Fig. 3, $\mathcal{Q}_h = \{\mathcal{A}_{0,0}, \mathcal{A}_{1,0}, \mathcal{A}_{0,-1}, \mathcal{A}_{1,-1}\}$ is the region in which we need to perform similarity search.

Given a point (x,y), its coordinates in lines $y = x$ and $y = -x$ are $\frac{1}{\sqrt{2}}(x+y, y-x)$. Thus, the relative offsets of point (x,y) w.r.t. the division point (x_0, y_0) are $d_x = \frac{1}{\sqrt{2}}((x+y) - (x_0+y_0))$ in line $y = x$ and $d_y = \frac{1}{\sqrt{2}}((y-x) - (y_0 - x_0))$ in line $y = -x$, respectively. As the side length of a subregion is $\sqrt{2}\ell$, the point (x,y) belongs to the subregion $\mathcal{A}_{i,j}$ satisfying $i = \lfloor\frac{d_x}{\sqrt{2}\ell}\rfloor$ and $j = \lfloor\frac{d_y}{\sqrt{2}\ell}\rfloor$. By Definition 6, we know that subregions in $\mathcal{Q}_h$ are adjacent and can use the following formula to compute $\mathcal{Q}_h$.

$$\mathcal{Q}_h = \bigcup_{i,j}\mathcal{A}_{i,j} \text{ for all } i_1 \le i \le i_2 \text{ and } j_1 \le j \le j_2, \quad (1)$$

where $i_1 = \lfloor(|E_h| - \tau + |V_h| - (x_0 + y_0))/2\ell\rfloor$ and $j_1 = \lfloor(|E_h| - \tau - |V_h| - (y_0 - x_0))/2\ell\rfloor$ are the relative positions of the subregion in the lower left corner of $\mathcal{Q}_h$ w.r.t. $\mathcal{A}_{0,0}$ in lines $y = x$ and $y = -x$, respectively, $i_2 = \lfloor(|E_h| + \tau + |V_h| - (x_0 + y_0))/2\ell\rfloor$ and $j_2 = \lfloor(|E_h| + \tau - |V_h| - (y_0 - x_0))/2\ell\rfloor$ are the relative positions of the subregion in the top right corner of $\mathcal{Q}_h$ w.r.t. $\mathcal{A}_{0,0}$ in lines $y = x$ and $y = -x$, respectively. Since $\lfloor z_1/\ell\rfloor - \lfloor z_2/\ell\rfloor \le (z_1 - z_2)/\ell + 1$, for any z_1 and z_2, we obtain $i_2 - i_1 \le \tau/\ell + 1$ and $j_2 - j_1 \le \tau/\ell + 1$. So we compute $\mathcal{Q}_h$ in $\mathcal{O}((\tau/\ell+2)^2)$ time, which is almost constant.

In practice, we compute modes of data distributions of $(|V_g|, \cdot)$ and $(\cdot, |E_g|)$, respectively, and then take them as the division point (x_0, y_0). For the subregion length ℓ, we will discuss it in the experiment.

4.2 Succinct Index Construction

In this section, we introduce a succinct q-gram tree index, which incorporates succinct data structures and hybrid encoding to achieve improved query time performance with minimal space.

4.2.1 *Tree Structure*

Let $\mathcal{U}_D$ and $\mathcal{U}_L$ be the sets of all distinct degree-based and label-based q-grams occurring in $\mathcal{G}$, respectively, where $\mathcal{U}_D[i]$ and $\mathcal{U}_L[i]$ are the ith most frequently occurring degree-based and label-based q-grams, respectively. Then, we define the q-gram profile of a graph as follows:

Definition 7 (q-gram profile)**.** *The q-gram profile of a graph g is defined as a four-tuple $LD = (F_D, F_L, n_v, n_e)$, where n_v and n_e are the numbers of vertices and edges in g, respectively, F_D and F_L are two arrays to store the degree-based and label-based q-gram multisets $D(g)$ and $L(g)$, respectively, such that $F_D[i]$ and $F_L[i]$ are the numbers of occurrences of $\mathcal{U}_D[i]$ in $D(g)$ and $\mathcal{U}_L[i]$ in $L(g)$, respectively.*

Definition 8. *Given two q-gram profiles LD and LD', the union operator "$\sqcup$" between them is defined as: $LD \sqcup LD' = (F_D \oplus F'_D, F_L \oplus F'_L, \min\{n_v, n'_v\}, \min\{n_e, n'_e\})$, where*

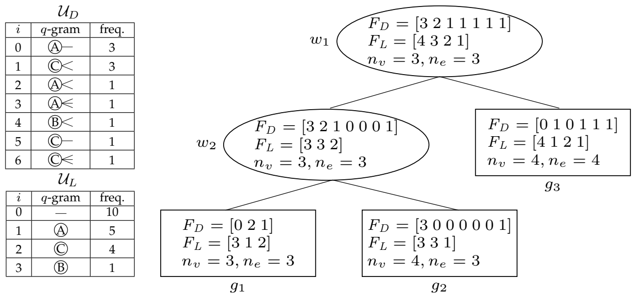
$$(F_D \oplus F'_D)[i] = \begin{cases} \max\{F_D[i], F'_D[i]\} & \text{if } i < \min\{|F_D|, |F'_D|\}; \\ F_D[i] & \text{if } |F'_D| \le i < |F_D|; \\ F'_D[i] & \text{otherwise.} \end{cases}$$

and similar definition for $F_L \oplus F'_L$.

Based on Definition 8, we can generalize the union operator "$\sqcup$" of two q-gram profiles to that of multiple q-gram profiles.

Definition 9 (q-gram tree)**.** *A q-gram tree is a balanced tree such that each leaf node stores a q-gram profile of a data graph and each internal node is the union of its child nodes.*

Imagining the q-gram profile as the *minimum bounding rectangle* used to represent an object in R-tree [14], we can construct the q-gram tree like the way of building the R-tree. Fig. 4 gives an example of a q-gram tree built on graphs g_1, g_2 and g_3 shown in Fig. 1.



$\mathcal{U}_D$

i	q-gram	freq.
0	Ⓐ−	3
1	Ⓒ<	3
2	Ⓐ<	1
3	Ⓐ≤	1
4	Ⓑ<	1
5	Ⓒ−	1
6	Ⓒ≤	1

$\mathcal{U}_L$

i	q-gram	freq.
0	—	10
1	Ⓐ	5
2	Ⓒ	4
3	Ⓑ	1

Fig. 4: Example of a q-gram tree.

4.2.2 *Succinct Representation*

Considering a q-gram tree, F_D and F_L in each node take up most of the space. In order to reduce the occupied space, we compress F_D and F_L while maintaining the query efficiency. Next, we regard X as D or L to explain the proposed method.

For a q-gram tree, we traverse it in a depth-first order to obtain a sequence of nodes $w_1, \ldots, w_{\mathcal{N}}$, where w_i is the ith node in this traversal and $\mathcal{N}$ is the number of nodes in the tree. Let $w_i.F_X$ be the array F_X in the node w_i. Then we concatenate all $w_i.F_X$ to obtain I_X as follows:

$$I_X = w_1.F_X \circ w_2.F_X \circ \cdots \circ w_{\mathcal{N}}.F_X,$$

where "$\circ$" is a concatenation operator. For instance, for the q-gram tree shown in Fig. 4, we can obtain $I_D =$ $[\underbrace{3\ 2\ 1\ 1\ 1\ 1\ 1}_{w_1.F_D}\ \underbrace{3\ 2\ 1\ 0\ 0\ 0\ 1}_{w_2.F_D}\ \underbrace{0\ 2\ 1}_{g_1.F_D}\ \underbrace{3\ 0\ 0\ 0\ 0\ 0\ 1}_{g_2.F_D}\ \underbrace{0\ 1\ 0\ 1\ 1\ 1}_{g_3.F_D}]$.

Instead of directly using an int type to store each entry in I_X, we compress I_X based on the following observations.

Observation 1. There are many zeros in I_X.

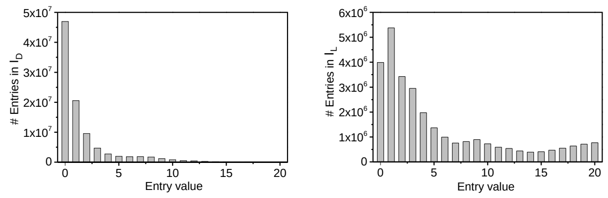


Fig. 5: Distribution of entries in I_D (left) and I_L (right).

For instance, we empirically tested the q-gram tree built on 5 million graphs from the PubChem dataset and showed the top 90% of entries of I_D and I_L in Fig. 5. From this figure, we know that more than 50% and 20% of the entries in I_D and I_L are zeros, respectively.

We use a bit array B_X and an array V_X to represent I_X to reduce the space. Specifically, if $I_X[j] = 0$ then we set $B_X[j] = 0$; otherwise, $B_X[j] = 1$. Meanwhile, we set $V_X[j]$ to the jth nonzero entry in I_X. For example, for the above array I_D, we use $B_D =$ [1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 1] and $V_D =$ [3 2 1 1 1 1 1 3 2 1 1 2 1 3 1 1 1 1 1] to represent it.

Observation 2. The small entries occupy a large proportion of I_X.

We also find that there are lots of small entries in I_X (e.g., more than 90% of entries are smaller than 20 in Fig. 5). Thus an efficient encoding strategy for small entries is key to further reduce the space.

We use the following strategy to encode small nonzero entries stored in V_X: First, we divide V_X into fixed-length blocks of size b. Then, we encode each block by choosing one from two encoding methods so that the *encoded bit sequence*, S_X, has the minimum space. One encoding method is the fixed-length encoding, which uses $\lfloor \log b_{\max} \rfloor + 1$ bits to encode each entry in a block, where $b_{\max}$ is the maximum value in this block. The other method uses Elias γ-encoding [8] to encode each entry in a block.

Also, we build several auxiliary structures to support for accessing each entry in I_X:

- SB_X is used to store the start position of the encoding of each block in S_X;
- *flag*$_X$ is used to mark the encoding method used for each block: if the kth block is a γ-encoding block, then *flag*$_X[k] = 0$; otherwise, it is a fixed-length encoding block, and *flag*$_X[k] = 1$.
- *words*$_X$ is used to indicate the number of bits required for each entry in a fixed-length encoding block.

The process of compressing I_X is illustrated in Fig. 6.

In addition, in each node, we retain the numbers of vertices and edges (i.e., n_v and n_e) and add the left and right boundaries (i.e., l_X and r_X) that F_X is located in I_X. Replacing $X = D$ and $X = L$, we can obtain the succinct representation of a q-gram tree, which is also called *succinct q-gram tree*. Fig. 7 shows the succinct representation of the q-gram tree shown in Fig. 4.

4.2.3 *Accessing*

In this section, we discuss how to access an entry in a succinct q-gram tree, which is the basic operation when searching on this tree.

Given a node w_i, the jth entry in $w_i.F_X$ is $w_i.F_X[j] = I_X[k]$, where $k = w_i.l_X + j$ is the position that $w_i.F_X[j]$ is

$$I_X \xrightarrow{\text{is represented by}} \begin{cases} V_X \xrightarrow{\text{is encoded by}} \begin{cases} S_X \\ SB_X \\ \mathit{flag}_X \\ \mathit{words}_X \end{cases} \\ B_X \end{cases}$$

Fig. 6: Process of compressing I_X, where $X = D$ or $X = L$.

located in I_X. We know that I_X is represented by B_X and V_X (see Fig. 6), and can use them to access $I_X[k]$ as follows:

$$I_X[k] = \begin{cases} 0 & \text{if} B_X[k] = 0; \\ V_X[rank_1(B_X, k)] & \text{otherwise.} \end{cases} \qquad (2)$$

where $rank_1(\cdot, k)$ returns the number of 1's in the first k bits. If $B_X[k] = 0$, then $I_X[k] = 0$; otherwise, we use the rank operation $rank_1(B_X, k)$ to determine the position that $I_X[k]$ is stored in V_X and then obtain $I_X[k] = V_X[rank_1(B_X, k)]$.

However, we cannot directly access an entry in V_X since V_X is encoded by four structures, S_X, SB_X, flag_X and words_X (see Fig. 6). In order to retrieve an entry $V_X[z]$, we perform the decoding operation $\mathit{decompress}$ on S_X and then obtain

$$V_X[z] = \mathit{decompress}(S_X,\ \mathit{flag}_X[\lfloor z/b \rfloor],\ SB_X[\lfloor z/b \rfloor], \\ \mathit{words}_X[rank_1(\mathit{flag}_X, \lfloor z/b \rfloor)], (z \bmod b) + 1), \qquad (3)$$

where b is the block size. The decoding operation $\mathit{decompress}$ contains the following two steps:

(1) We query $\mathit{flag}_X[\lfloor z/b \rfloor]$ and $SB_X[\lfloor z/b \rfloor]$ to determine the encoding method used and decoding position for the $\lfloor z/b \rfloor$th block to which $V_X[z]$ belongs, respectively;

(2) If $\mathit{flag}_X[\lfloor z/b \rfloor] = 0$, then the $\lfloor z/b \rfloor$th block is a γ-encoding block, and we decode S_X for $(z \bmod b) + 1$ times starting from the $SB_X[\lfloor z/b \rfloor]$th bit. The last decoding value is $V_X[z]$. If $\mathit{flag}_X[\lfloor z/b \rfloor] = 1$, then the $\lfloor z/b \rfloor$th block is a fixed-length encoding block. We use $\mathit{words}_X[rank_1(\mathit{flag}_X, \lfloor z/b \rfloor)]$ to determine the number of bits used to encode each entry in this block, and then directly decode the $((z \bmod b) + 1)$th fixed-length encoded entry as $V_X[z]$.

We can finish the decoding operation $\mathit{decompress}$ in constant time using a lookup table [16]. Also, we can compute $rank_1(,)$ in constant time with a rank dictionary [13], [17]. Thus, we obtain $I_X[k]$ from Formula (2) in constant time.

Example 5. We show how to access $g_2.F_D[0]$ shown in Fig. 4 by using the succinct representation shown in Fig. 7, where $b = 4$. Clearly, $g_2.F_D[0] = I_D[g_2.l_D + 0] = I_D[17]$. According to Formula (2), $I_D[17] = V_D[rank_1(B_D, 17)] = V_D[13]$. Then, we compute $V_D[13]$ through Formula (3) as follows: (1) We determine the encoding method by $\mathit{flag}_D[\lfloor 13/b \rfloor] = \mathit{flag}_D[3] = 0$ and the decoding position by $SB_D[\lfloor 13/b \rfloor] = SB_D[3] = 22$; (2) As $\mathit{flag}_D[3] = 0$, $V_D[13]$ is in a γ-encoding block. Starting form the 22th bit of S_D, we decode two times and the second decoding value is $V_D[13] = 3$. So, we obtain $g_2.F_D[0] = 3$.

4.3 Query Processing

Our query process consists of two phases: the computation of the reduced query region $\mathcal{Q}_h$ from Formula (1), described in Section 4.1, and similarity search on the succinct q-gram trees built in $\mathcal{Q}_h$.

4.3.1 Query on the Succinct Q-Gram Tree

Consider a node w in the succinct q-gram tree. Let $C_D(w, h)$ and $C_L(w, h)$ be the number of common degree-based and label-based q-grams between w and h, respectively. We have $C_X(w, h) = \sum_i \min\{w.F_X[i], h.F_X[i]\}$, where X is D or L. Then we establish Theorem 4 to safely prune w.

Theorem 4. *If $C_L(w, h) < \max\{w.n_v, |V_h|\} + \max\{w.n_e, |E_h|\} - \tau$ or $C_D(w, h) < \max\{w.n_v, |V_h|\} - 2\tau$, then we can safely prune all the w's child nodes.*

Proof. See Appendix D in supplementary materials. □

Query Algorithm. We give the query method on a succinct q-gram tree in Alg. 2, where $root$ is the root node and δ is the boosting parameter.

```
Algorithm 2: searchSQTree($root, h, \tau, \delta$)
1 $\mathcal{C} \leftarrow \emptyset$;
2 searchTree ($root, h, \tau, \delta, \mathcal{C}$);
3 return $\mathcal{C}$;
  procedure searchTree($h, w, \tau, \delta, \mathcal{C}$)
1     compute $C_D(w, h)$ and $C_L(w, h)$;
2     if $C_D(w, h) \geq \max\{n_v, |V_h|\} - 2\tau$ or $C_L(w, h) \geq$
        $\max\{w.n_v, |V_h|\} + \max\{w.n_e, |E_h|\} - \tau$ then
3         if $w$ is an internal node then
4             foreach $w$'s child node $w_j$ do
5                 searchTree ($w_j, h, \tau, \delta, \mathcal{C}$);
6         else
7             $flag$ ←filterGraph($h, w, \tau, \delta$);
8             if $flag$ is false then
9                 $\mathcal{C} \leftarrow \mathcal{C} \cup \{w\}$;
```

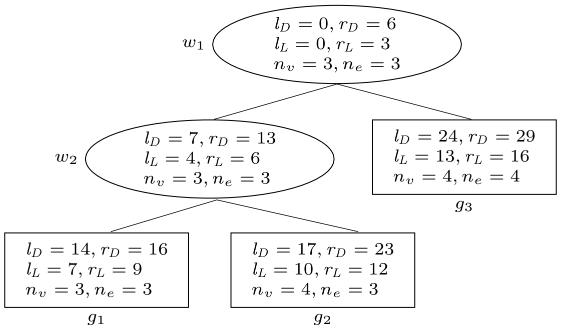
The search process on a succinct q-gram tree is similar to that on an R-tree. Starting from the root node, $root$, we traverse this tree. For a node w, we compute $C_D(w, h)$ and $C_L(w, h)$ and then determine whether pruning w according to Theorem 4. If $C_D(w, h) < \max\{w.n_v, |V_h|\} - 2\tau$ or $C_L(w, h) < \max\{w.n_v, |V_h|\} + \max\{w.n_e, |E_h|\} - \tau$, then we safely prune w; otherwise, we access each subtree of w.

Furthermore, when reaching a leaf node w, namely w is a data graph and has not been pruned, we check whether it passes the filtering method filterGraph introduced in Section 3.2. Notice that we have not stored the degree sequence σ_w in the succinct q-gram tree; when calling filterGraph, we need to calculate σ_w through $w.F_D$'s nonzero entries that contain the vertex degree information.

Complexity Analysis. Let $\mathcal{V}$ be the set of nodes visited in the search. For a node $w \in \mathcal{V}$, we can compute $C_D(w, h)$ in linear time of the number of nonzero entries in $w.F_D$ and $h.F_D$. Since $h.F_D$ contains at most $|V_h|$ degree-based q-grams, we can compute $C_D(w, h)$ in $\mathcal{O}(|V_h|)$ time. Similarly, we can compute $C_L(w, h)$ in $\mathcal{O}(|V_h| + |E_h|)$ time. Thus, these $|\mathcal{V}|$ nodes takes $\mathcal{O}(|\mathcal{V}|(|V_h| + |E_h|))$ time. Let $\mathcal{L} \subset \mathcal{V}$ be the set of leaf nodes unpruned in the search. For a leaf node in $\mathcal{L}$, we execute filterGraph to determine whether filtering it out, which consumes $\mathcal{O}(|V_h|^{\delta+2})$ time. Thus, these $|\mathcal{L}|$ leaf nodes takes $\mathcal{O}(|\mathcal{L}||V_h|^{\delta+2})$ time. So, the time complexity of Alg. 2 is $\mathcal{O}(|\mathcal{V}|(|V_h| + |E_h|) + |\mathcal{L}||V_h|^{\delta+2}) = \mathcal{O}(|V_h|^2(|\mathcal{V}| + |\mathcal{L}||V_h|^{\delta}))$, for $|E_h| \leq |V_h|^2$.

4.3.2 Whole Query Algorithm

Alg. 3 gives the whole query method over MSQ-Index, where (x_0, y_0) is the division point, ℓ is the subregion length, δ is



(a)

1111111 1110001 011 1000001 010111 B_D

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V_D	3	2	1	1	1	1	1	3	2	1	1	2	1	3	1	1	1	1	1
S_D	11 10 01 01				1 1 1 011				10 01 01 10				1 011 1 1				1 1 1		
SB_D	0				8				14				22				28		
$flag_D$	1				0				1				0				1		
$words_D$	2				-				2				-				1		

(b)

1111 111 111 111 1111 B_L

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V_L	4	3	2	1	3	3	2	3	1	2	3	3	1	4	1	2	1
S_L	100 011 010 001				11 11 10 11				01 10 11 11				1 00100 1 010				1
SB_L	0				12				20				28				38
$flag_L$	1				1				1				0				1
$words_L$	3				2				2				–				1

(c)

Fig. 7: Succinct representation of a q-gram tree.

the boosting parameter, and $root_{i,j}$ is the root node of the succinct q-gram tree built in the subregion $\mathcal{A}_{i,j}$.

Algorithm 3: search($h, \tau, \delta, x_0, y_0, l$)

```
1 $\mathcal{C} \leftarrow \emptyset$, $ans \leftarrow \emptyset$ ;
2 $\mathcal{Q}_h \leftarrow \bigcup_{i,j} \mathcal{A}_{i,j}$ for all $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$ ;
3 foreach $\mathcal{A}_{i,j} \subseteq \mathcal{Q}_h$ do
4     $\mathcal{C}_{i,j} \leftarrow$searchSQTree($root_{i,j}, h, \tau, \delta$);
5     $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_{i,j}$;
6 foreach $g \in \mathcal{C}$ do
7     if $ged(h, g) \leq \tau$ then
8         $ans \leftarrow ans \cup \{g\}$ ;
9 return $ans$ ;
```

In Alg. 3, we first compute the query region $\mathcal{Q}_h$ from Formula (1). Then, we call searchSQTree to perform similarity search on the succinct q-gram trees built in the subregions contained in $\mathcal{Q}_h$. Finally, the remaining unpruned graphs constitute a candidate set $\mathcal{C}$ and we employ existing GED computation methods [12], [27] to verify the graphs in $\mathcal{C}$.

5 Cost Estimation

In this section, we analyze the performance of MSQ-Index. As we know MSQ-Index consists of multiple succinct q-gram trees, for simplicity we analyze the cost of using a succinct q-gram tree built on $\mathcal{G}$ since this cost has the same order of magnitude of that using multiple succinct q-gram trees built on $\mathcal{G}$'s subsets.

5.1 Storage Cost Estimation

The succinct q-gram tree built on $\mathcal{G}$ can be decomposed into three parts (a), (b) and (c), as follows:

(a) left and right boundaries (i.e., l_D/l_L and r_D/r_L), #vertices and #edges (i.e., n_v and n_e) and child-pointers in all nodes;
(b) structures B_D, S_D, SB_D, $flag_D$, and $words_D$ used to compress I_D;
(c) structures B_L, S_L, SB_L, $flag_L$, and $words_L$ used to compress I_L.

For parts (a), (b) and (c), we denote their occupied space by $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_c$, respectively. An example of these three parts is illustrated in Fig. 7.

For a node in the tree, we use $\lfloor \log |B_D| \rfloor + 1$ bits to store l_D and r_D, respectively, and $\lfloor \log |B_L| \rfloor + 1$ bits to store l_L and r_L, respectively, since $l_D \leq r_D \leq |B_D|$ and $l_L \leq r_L \leq |B_L|$. We also use $\lfloor \log v_m \rfloor + 1$ and $\lfloor \log e_m \rfloor + 1$ bits to store n_v and n_e, respectively, where $v_m = \max_{g \in \mathcal{G}}\{|V_g|\}$ and $e_m = \max_{g \in \mathcal{G}}\{|E_g|\}$.

Let d be the average fan-out of each node in the tree. The total number of nodes, $\mathcal{N}$, is bounded by $\mathcal{N} = \sum_{h=0}^{\log_d |\mathcal{G}|} \frac{|\mathcal{G}|}{d^h} \leq \frac{d|\mathcal{G}|}{d-1}$. Thus, we can use $\lfloor \log \frac{d|\mathcal{G}|}{d-1} \rfloor + 1$ bits to store each child-pointer. So the space $\mathcal{S}_a$ is bounded by

$$\begin{aligned}
\mathcal{S}_a &= \mathcal{N}(2(\lfloor \log |B_D| \rfloor + 1) + 2(\lfloor \log |B_L| \rfloor + 1) + \lfloor \log v_m \rfloor + 1 \\
&\qquad + \lfloor \log e_m \rfloor + 1 + \lfloor \log \frac{d|\mathcal{G}|}{d-1} \rfloor + 1) \\
&\leq \frac{d|\mathcal{G}|}{d-1}(2\log(|B_D||B_L|) + \log(v_m e_m) + \log \frac{d|\mathcal{G}|}{d-1} + 7) \\
&\leq 2|\mathcal{G}|(2\log(|B_D||B_L|) + 2\log I_m + \log(2|\mathcal{G}|) + 7) \\
&\leq 4|\mathcal{G}|(\log(I_m|B_D||B_L||\mathcal{G}|) + 4) \\
&= \mathcal{O}(|\mathcal{G}| \log(I_m|B_D||B_L||\mathcal{G}|))
\end{aligned}$$

where $I_m = \max\{v_m, e_m\}$. The second inequality is due to the fact that $d \geq 2$.

Regarding X as D or L, we then consider the space required by S_X, B_X, SB_X, $flag_X$, and $words_X$.

First, we analyze the space needed by the encoded bit sequence S_X. Let E^γ and E^f be the collections of Elias γ-encoding and fixed-length encoding blocks, respectively, and $|\gamma(b_i)|$ and $|f(b_i)|$ be the number of bits required to encode the ith block b_i using γ-encoding and fixed-length encoding, respectively. By our hybrid encoding scheme, the number of bits needed by S_X is bounded by

$$\begin{aligned}
&\sum_{i=1}^{|V_X|/b} \min\{|\gamma(b_i)|, |f(b_i)|\} \\
&= \sum_{i \in E^\gamma} |\gamma(b_i)| + \sum_{i \in E^f} |f(b_i)| \leq \sum_{i \in E^\gamma} |f(b_i)| + \sum_{i \in E^f} |f(b_i)| \\
&\leq \sum_{i \in E^\gamma \cup E^f} b(\lfloor \log b_X^m \rfloor + 1) \leq \frac{|V_X|}{b} b(\lfloor \log b_X^m \rfloor + 1) \\
&\leq |V_X| \log b_X^m + |V_X|,
\end{aligned}$$

where b is the block size and b_X^m is the maximum value in V_X. The first inequality is due to the fact that $|\gamma(b_i)| \leq |f(b_i)|$ when $i \in E^\gamma$. The second inequality is due to the fact that the number of bits required to encode b_i through fixed-length encoding is bounded by $b(\lfloor \log V_X^m \rfloor + 1)$. The third inequality is due to the fact that $|E^\gamma| + |E^f| = |V_X|/b$.

Second, we analyze the space required by the structures B_X, SB_X, $flag_X$, and $words_X$.

For B_X, it contains $|B_X|$ bits. Besides, we build the rank dictionary over it, which needs $o(|B_X|)$ bits [17]. Thus, B_X totally requires $|B_X| + o(|B_X|)$ bits.

For SB_X, it requires $\frac{|V_X|}{b}(\log(|V_X|\log b_X^m + |V_X|) + 1)$ bits in the worst case because each entry in SB_X needs at most $\lfloor\log(|V_X|\log b_X^m + |V_X|)\rfloor + 1$ bits and there are $|V_X|/b$ blocks.

For $flag_X$, it requires $|V_X|/b + o(|V_X|/b)$ bits, because each block takes up one bit and there are $|V_X|/b$ blocks, and the rank dictionary built over $flag_X$ needs $o(|V_X|/b)$ bits.

For $words_X$, it requires $\frac{|V_X|}{b}(\lfloor\log b_X^m\rfloor + 1)$ bits since each entry requires $\lfloor\log b_X^m\rfloor + 1$ bits and there are at most $|V_X|/b$ entries.

Putting all the space required for B_X, SB_X, $flag_X$, and $words_X$ together, we then obtain

$$\begin{aligned}&|B_X| + o(|B_X|) + \frac{|V_X|}{b}\log(|V_X|\log b_X^m + |V_X|) + \\ &\qquad \frac{|V_X|}{b}\log b_X^m + 3\frac{|V_X|}{b} + o(\frac{|V_X|}{b}) \\ &= |B_X| + o(|B_X|) + o(|V_X|) = |B_X| + o(|B_X|)\ ,\end{aligned}$$

for $b = \log^2 |V_X|$. The second equality is due to the fact that $|V_X| \le |B_X|$. Thus, the total space of S_X, B_X, SB_X, $flag_X$, and $words_X$ is at most $|V_X|(\log b_X^m + 1) + |B_X| + o(|B_X|)$ bits.

Considering a degree-based q-gram, it occurs at most $|V_g|$ times in a data graph g, thus we have $b_D^m \le v_m$. Similarly, $b_L^m \le \max\{v_m, e_m\}$. Replacing X with D and substituting $b_D^m \le v_m$, we obtain that

$$\begin{aligned}\mathcal{S}_b &\le |V_D|(\log v_m + 1) + |B_D| + o(|B_D|) \\ &\le |B_D|(\log I_m + 2) + o(|B_D|) = \mathcal{O}(|B_D|\log I_m)\end{aligned}$$

where the second inequality holds because $v_m \le I_m = \max\{v_m, e_m\}$ and $|V_D| \le |B_D|$. Similarly, Replacing X with L and substituting $b_L^m \le \max\{v_m, e_m\}$, we obtain that

$$\begin{aligned}\mathcal{S}_c &\le |V_L|(\log\max\{v_m, e_m\} + 1) + |B_L| + o(|B_L|) \\ &\le |B_L|(\log I_m + 2) + o(|B_L|) = \mathcal{O}(|B_L|\log I_m)\end{aligned}$$

By summing the space required by $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_c$, and ignoring the lower-order terms, we obtain the space bound on the succinct q-gram tree: $\mathcal{O}(|\mathcal{G}|\log(I_m|B_D||B_L||\mathcal{G}|) + (|B_D| + |B_L|)\log I_m)$ bits.

Note. As we know, $I_m = \max\{v_m, e_m\}$ is the maximum number of vertices or edges in $\mathcal{G}$, which is usually a small value. Therefore, the space in bits required by the succinct q-gram tree is dominated by the addition of the "linearithmic" function of the database size $|\mathcal{G}|$ and the linear function of $|B_D| + |B_L|$.

5.2 Query Cost Estimation

Alg. 3 gives the query method over MSQ-Index, where the query process contains the following three parts: (i) computing the query region $\mathcal{Q}_h$ from Formula (1) (line 2); (ii) searching on the succinct q-gram trees built in $\mathcal{Q}_h$ to obtain a candidate set $\mathcal{C}$ (lines 3–5); (iii) verifying graphs in $\mathcal{C}$ (lines 6–8). Let $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ be the cost incurred by these three parts above, respectively. Then the total query cost $\mathcal{T}$ can be formulated as

$$\mathcal{T} = \mathcal{T}_1 + \mathcal{T}_2 + \mathcal{T}_3 = \mathcal{T}_Q + \mathcal{T}_V,$$

where $\mathcal{T}_Q = \mathcal{T}_1 + \mathcal{T}_2$ is the filtering cost, $\mathcal{T}_V = \mathcal{T}_3 = |\mathcal{C}| \cdot \mathcal{T}_{GED}$ is the verifying cost, and $\mathcal{T}_{GED}$ is the average GED computation time.

For (i), the cost of computing $\mathcal{Q}_h$ from Formula (1) is $\mathcal{T}_1 = \mathcal{O}((\tau/\ell + 2)^2)$, which is almost constant and can be negligible.

For (ii), for simplicity we analyze the cost of using a succinct q-gram tree built in the whole region. As discussed in Section 4.3.1, the cost is $\mathcal{T}_2 = \mathcal{O}(|V_h|^2(|\mathcal{V}| + |\mathcal{L}||V_h|^\delta))$, where $\mathcal{V}$ is the set of nodes visited and $\mathcal{L}$ is the set of leaf nodes unpruned.

Therefore, the filtering cost is $\mathcal{T}_Q = \mathcal{T}_1 + \mathcal{T}_2 = \mathcal{O}(|V_h|^2(|\mathcal{V}| + |\mathcal{L}||V_h|^\delta))$. Putting the verifying cost $\mathcal{T}_V$ together, we obtain $\mathcal{T} = \mathcal{T}_Q + \mathcal{T}_V = \mathcal{O}(|V_h|^2(|\mathcal{V}| + |\mathcal{L}||V_h|^\delta)) + |\mathcal{C}| \cdot \mathcal{T}_{GED}$. Clearly, $\mathcal{T}$ is mainly determined by $|\mathcal{C}|$, $|\mathcal{L}|$ and $|\mathcal{V}|$. Next, we discuss how to estimate them.

Estimating $|\mathcal{C}|$ and $|\mathcal{L}|$. Let $\mathcal{X}_\mathcal{C}(h, g)$ be the GED lower bound such that $\mathcal{X}_\mathcal{C}(h, g) = \max\{\xi_k^D(h, g), \xi_k^L(h, g), \xi_k^S(h, g)\}$, where $k = \max\{0, \min\{|V_h| - |V_g|, \delta\}\}$, and $\xi_k^D(h, g)$, $\xi_k^L(h, g)$ and $\xi_k^S(h, g)$ are introduced in Section 3.2.3. A data graph g cannot be filtered out by the filtering method filterGraph (i.e., Alg. 1) when $\mathcal{X}_\mathcal{C}(h, g) \le \tau$. Thus, the candidate set is $\mathcal{C} = \{g \in \mathcal{G} : \mathcal{X}_\mathcal{C}(h, g) \le \tau\}$.

Let τ be a random variable. Considering $\mathcal{X}_\mathcal{C}(,)$ as a distance metric, we define a cumulative probability distribution of τ as $\mathcal{F}_\mathcal{C}(\tau) = P_r[\mathcal{X}_\mathcal{C}(g, g') \le \tau] = \frac{\#Num_\mathcal{C}(\tau)}{|\mathcal{G}|\cdot(|\mathcal{G}|-1)}$, where $\#Num_\mathcal{C}(\tau)$ is the number of data graph pairs (g, g') such that $\mathcal{X}_\mathcal{C}(g, g') \le \tau$, for any $g, g' \in \mathcal{G}$ and $g \ne g'$. We can reasonably assume that data graphs similar to h always exist in $\mathcal{G}$, and then use $\mathcal{F}_\mathcal{C}(\tau)$ to approximate the probability that a data graph g belongs to $\mathcal{C}$. Consequently, we can use $|\mathcal{G}| \cdot \mathcal{F}_\mathcal{C}(\tau)$ to estimate $|\mathcal{C}|$.

From our empirical observation, $\mathcal{F}_\mathcal{C}$ exhibits a typical "S"-shaped curve. The possible reason is that $\mathcal{X}_\mathcal{C}(,)$ distance distribution of data graphs in the database $\mathcal{G}$ is not uniform. (see our empirical test, Appendix E1 in supplementary materials). We can employ a sigmoid-like function to estimate $\mathcal{F}_\mathcal{C}$ as follows:

$$\mathcal{F}_\mathcal{C}(\tau) = \frac{1}{1 + a \cdot c^{-\tau}},$$

where $a, c > 0$ are two parameters. When $\tau \to \infty$, $\mathcal{F}_\mathcal{C}(\tau) = 1$ coincides with the fact that $\mathcal{C}$ contains all data graphs in this case. Also, we can rewrite the above formula as $\ln(\frac{1}{\mathcal{F}_\mathcal{C}(\tau)} - 1) = \ln a + \tau \cdot (-\ln c)$. By using the least-squares method, we can estimate a and c.

Let $\mathcal{X}_\mathcal{L}(h, g)$ be the GED lower bound such that $\mathcal{X}_\mathcal{L}(h, g) = \max\{\lambda^L(h, g), \lambda^{D'}(h, g)\}$, where $\lambda^{D'}(h, g) = \frac{1}{2}(\max\{|V_h|, |V_g|\} - |D(h) \cap D(g)|) \le \lambda^D(h, g)$. According to Theorem 4, a leaf node g cannot be pruned when $\mathcal{X}_\mathcal{L}(h, g) \le \tau$. Thus, the set of unpruned leaf nodes in the search is $\mathcal{L} = \{g \in \mathcal{G} : \mathcal{X}_\mathcal{L}(h, g) \le \tau\}$. Similarly, we can also consider $\mathcal{X}_\mathcal{L}(,)$ as a distance metric to estimate $|\mathcal{L}|$ like the way of estimating $|\mathcal{C}|$ above.

Estimating $|\mathcal{V}|$. Let d be the average fan-out of each node in the tree. The number of nodes in the ith level is d^i, for $0 \le i \le t$, where $t = \log_d |\mathcal{G}|$ is the height of this tree. Assuming

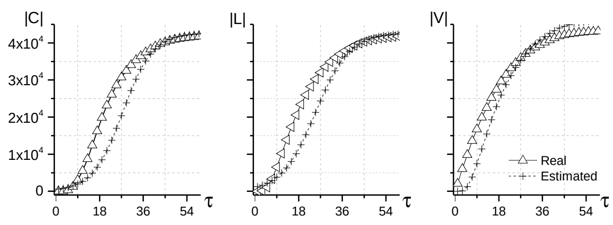


Fig. 8: Estimation of $|\mathcal{C}|$, $|\mathcal{L}|$ and $|\mathcal{V}|$ on AIDS dataset.

that visited nodes in each level are evenly distributed. Let $P_r(i)$ be the probability of visiting a node in the ith level. Then the number of nodes visited in the ith level is $d^i \cdot P_r(i)$, and thus we have $|\mathcal{V}| = \sum_{i=0}^{t} d^i \cdot P_r(i)$.

For a leaf node g, the probability of g belonging to $\mathcal{L}$ is $\frac{|\mathcal{L}|}{|\mathcal{G}|}$. For an internal node w in the ith level, it contains d^{t-i} descendant leaf nodes; as long as one of these descendant leaf nodes belongs to $\mathcal{L}$, w will be visited. Thus, the possibility of visiting w is $1-(1-\frac{|\mathcal{L}|}{|\mathcal{G}|})^{d^{t-i}}$. Since visited nodes in each level are evenly distributed, we obtain $P_r(i) = 1-(1-\frac{|\mathcal{L}|}{|\mathcal{G}|})^{d^{t-i}}$. So, we can estimate $|\mathcal{V}|$ as

$$|\mathcal{V}| = \sum_{i=0}^{t} d^i \cdot (1-(1-\frac{|\mathcal{L}|}{|\mathcal{G}|})^{d^{t-i}}).$$

Empirical test. We consider the AIDS dataset as the tested dataset and randomly select 10^4 graphs from this dataset to make up the query graphs. Fig. 8 shows the real and estimated values of $|\mathcal{C}|$, $|\mathcal{L}|$ and $|\mathcal{V}|$ on the average, respectively. From Fig. 8, we know that

(1) the estimated value of $|\mathcal{C}|$ (or $|\mathcal{L}|$ or $|\mathcal{V}|$) is closed to the real value.

(2) when τ is small (e.g., $\tau \leq 9$), both the estimated and real values of $|\mathcal{C}|$ (or $|\mathcal{L}|$ or $|\mathcal{V}|$) are small.

In practice, we consider τ as a small value because users are typically more inclined to search for similar graphs to a given query graph. Thus, the query cost of MSQ-Index is acceptable, and thereby, MSQ-Index can efficiently finish the similarity search.

6 Experiments and Discussions

6.1 Datasets and Settings

Datasets. We choose several real and synthetic datasets from different domains in the experiment, described as follows:

(1) AIDS[1] is an antivirus screen compound dataset from the Development and Therapeutics Program in NCI/NIH, which contains 42,687 chemical compounds.

(2) COIL [26] consists of 7,200 images of different objects, where each image is converted into a region adjacency graph. Vertices represent different regions, and we randomly assign four labels to them. Edges represent the adjacency of regions and are labeled with the length (in pixels) of the common border of two adjacent regions.

(3) NASA[2] is an XML dataset that contains 36,790 data graphs, where each graph stores the metadata of an astronomical. We randomly assign 10 vertex labels to each graph according to the way described in [38].

(4) PubChem[3] is an NIH funded project to record experimental data of chemical that interactions with biological systems. We randomly select 25,000,000 chemical compounds to make up the large dataset, Pub-25M, used in the experiment.

(5) Synthetic. The synthetic dataset is generated by the synthetic graph data generator GraphGen[4], which allows us to specify various parameters, including the dataset size, the average graph density $\rho = \frac{2|E|}{|V|(|V|-1)}$, the number of edges, and the number of vertex and edge labels. We first generate several synthetic datasets and then merge them to obtain a large dataset, Sync-5M, which contains 5,000,000 graphs.

For each dataset, we randomly select 100 graphs from it to make up the query graphs. Table 2 summarizes some general characteristics of the above five datasets.

TABLE 2: Dataset Statistics

Dataset	$\lvert\mathcal{G}\rvert$	avg. $\lvert V\rvert$	avg. $\lvert E\rvert$	$\lvert\Sigma_V\rvert$	$\lvert\Sigma_E\rvert$
AIDS	42,687	25.6	27.5	62	3
COIL	7,200	21.5	54	4	2
NASA	36,790	33.2	32.2	10	1
Sync-5M	5,000,000	27.5	38.4	5	3
Pub-25M	25,000,000	23.4	25.2	101	3

Compared Methods. We perform comprehensive experimental studies for MSQ-Index by comparing it with the state-of-the-art indexing methods, GSimJoin [37], Pars [38], and Mixed [39]. For Pars, we implemented it with a random partition method because the authors did not share their implementations.

For each compared method, we employ the GED computation method, CSI_GED [12], as the GED verifier except for GSimJoin that has implemented A* as its verifier in the executable binary file.

Evaluation Metrics. In the experimental studies, we consider the following three metrics: (1) *Index construction cost*, including the index size and construction time; (2) *Candidate set size*, which is the number of data graphs that have not been filtered out; (3) *Response time*, which is the sum of the filtering time and verification time. The results obtained in (2) and (3) are the *average* candidate set size and the *average* response time for the 100 query graphs, respectively.

We have conducted all experiments on a HP Z800 PC with a 2.67 GHz CPU and 24GB main memory, running Ubuntu 12.04 operating system. We implemented MSQ-Index in C++, with –O3 to compile and run. For MSQ-Index, we set the boosting parameter $\delta = 2$, subregion length $\ell = 2$ and block size $b = 16$, respectively, as the default parameters. For the other tested methods, we adopt their default parameters.

6.2 Evaluating MSQ-Index

As described earlier in this paper, several techniques are proposed in MSQ-Index, including: (1) the succinct representation of a q-gram tree (Section 4.2), which is used to decrease the index size; (2) two lower bounds together with a boosting technique (Section 3.2), which aims to obtain a candidate set as small as possible; (3) the preprocessing method (Section 4.1), which ensures that similarity search is preformed in a reduced query region. Thus, it is necessary to study the contributions of these techniques to MSQ-Index.

Evaluating Succinct Representation. In this part, we evaluate the effectiveness of our succinct representation.

1. http://dtp.nci.nih.gov/docs/aids/aidsdata.html
2. http://www.cs.washington.edu/research/xmldatasets/
3. http://pubchem.ncbi.nlm.nih.gov/
4. http://www.cse.ust.hk/graphgen/

First, we evaluate the effectiveness of the method of compressing I_D and I_L. The compressing process contains the following two steps (see Fig. 6): (1) using B_X and V_X to represent I_X; (2) employing S_X, SB_X, $flag_X$, and $words_X$ to encode V_X, where X is D or L.

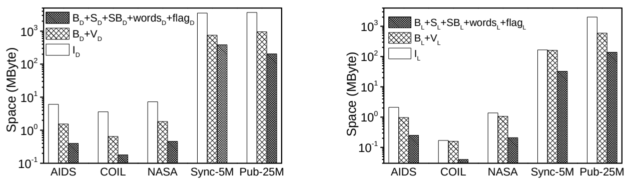


Fig. 9: Space of compressing I_D (left) and I_L (right).

In Fig. 9, we report the space of each step. In the first step, we can reduce I_D's space by more than 50%; this reduction is due to the fact that I_D contains lots of zeros. In the second step, we can further reduce I_D's space by more than 80%. This is because that each encoded entry takes about 4–6 bits (see the experiment on encoding, Appendix E2, in supplementary materials), which takes much fewer bits than that used to store an int type (i.e., typical 32 bits). As a result, using B_D, S_D, SB_D, $flag_D$, and $words_D$ to compress I_D, we can reduce the space by more than 90%. For I_L, we have a similar result.

TABLE 3: Space usage (MByte)

Dataset	q-gram tree			succinct q-gram tree		
	$\mathcal{M}_a$	$\mathcal{M}_b$	$\mathcal{M}_c$	$\mathcal{S}_a$	$\mathcal{S}_b$	$\mathcal{S}_c$
AIDS	0.29	6.09	2.11	0.88	0.41	0.25
COIL	0.09	3.62	0.17	0.15	0.18	0.04
NASA	0.44	7.28	1.38	0.85	0.46	0.21
Sync-5M	59.6	3527.8	236.3	141.9	415.3	41.8
Pub-25M	343.6	3669.2	2014.1	649.1	327.3	191.9

Second, we report the space of a q-gram tree and its succinct representation in Table 3. For a q-gram tree (see Fig. 4), we decompose its space into three parts $\mathcal{M}_a$, $\mathcal{M}_b$, and $\mathcal{M}_c$, where $\mathcal{M}_a$ is the space of n_v, n_e and child-pointers of all nodes, and $\mathcal{M}_b$ and $\mathcal{M}_c$ are the space of F_D and F_L of all nodes, respectively. Correspondingly, we obtain three parts $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_c$ in the succinct representation (see Fig. 7).

From Table 3, we know that $\mathcal{M}_b$ and $\mathcal{M}_c$ take up most of the space of a q-gram tree. By compressing them, we reduce their space to about 10% of the original size, see $\mathcal{S}_b$ and $\mathcal{S}_c$ in the last two columns. As a result, the space of the succinct q-gram tree, namely the sum of $\mathcal{S}_a$, $\mathcal{S}_b$, and $\mathcal{S}_c$, is less than 20% of that of the original q-gram tree (i.e., the sum of $\mathcal{M}_a$, $\mathcal{M}_b$, and $\mathcal{M}_c$).

Evaluating Filters. In this part, we evaluate the effectiveness of the proposed lower bounds and boosting technique under different thresholds τ setting, that is, $\tau \in \{1, 3, 5, 7, 9\}$. Notice that $\tau = 9$ is the maximum threshold value used in the current indexing-based methods [32], [36], [37], [39], [21], [38]. Here we only display the experimental results on the large dataset Pub-25M; for the results on other datasets, see Appendix E3 in supplementary materials.

First, we fix the boosting parameter $\delta = 0$ and then evaluate the effectiveness of the proposed lower bounds. Fig. 10 shows the obtained results, where "QF" denotes that we employ the label-based and degree-based q-gram counting lower bounds, and "DF" denotes the improved version of QF by incorporating the degree-sequence lower bound.

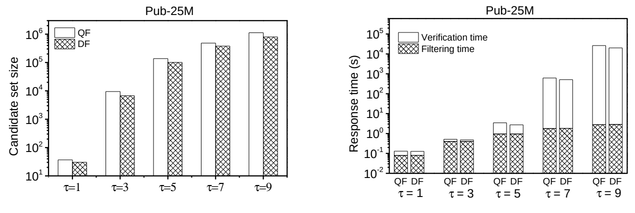


Fig. 10: Performance of multiple lower bounds on Pub-25M.

As depicted in Fig. 10, we can use these two q-gram counting lower bounds to filter most of the dissimilar graphs out. For instance, when $\tau = 3$, QF produces less than 1.0×10^4 candidate graphs, which indicates that more than 99.9% of data graphs are pruned. Moreover, through incorporating with the degree-sequence lower bound, DF further reduces the candidate set size by about 30%–60%.

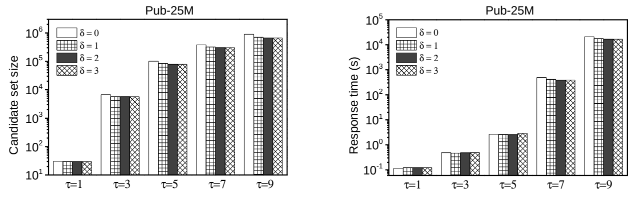


Fig. 11: Performance of boosting on Pub-25M.

Second, we evaluate the effectiveness of the proposed boosting technique. From Fig. 11, we know that the candidate set size is getting smaller and smaller with the increasing of δ. For instance, when $\tau = 9$, the candidate set size of $\delta = 2$ is about 67% of that of $\delta = 0$. Also, we observe that the response time first decreases and then increases, as δ increases; it achieves the minimum when $\delta = 2$ in most cases. Two factors may contribute to this trend: (1) When δ is too small, we obtain a large candidate set and spend lots of GED verification time; (2) When δ is too large, we spend too much filtering time. Thus, we set $\delta = 2$ as the default parameter in MSQ-Index.

Evaluating Preprocessing. In this part, we evaluate the effectiveness of the proposed preprocessing method.

To measure how much the query region $\mathcal{Q}_h$ gets smaller w.r.t. the whole region $\mathcal{A}$, we define a metric *access ratio* as $\frac{|\mathcal{Q}_h|}{|\mathcal{G}|}$, where $|\mathcal{Q}_h|$ is the number of points contained in $\mathcal{Q}_h$. Correspondingly, we obtain the *speedup* in filtering time, which is computed as the filtering time of SQ-Index divided by that of MSQ-Index, where SQ-Index is a succinct q-gram tree built in $\mathcal{A}$. Table 4 lists the access ratio and speedup on the average for the 100 query graphs.

TABLE 4: Average access ratio and speedup

Metric	Dataset	$\tau = 1$	$\tau = 3$	$\tau = 5$	$\tau = 7$	$\tau = 9$
access ratio	AIDS	9.9%	19.2%	25.5%	33.2%	38.7%
	COIL	4.5%	8.4%	11.3%	14.8%	17.6%
	NASA	4.7%	8.1%	12.1%	15.5%	21.1%
	Sync-5M	4.0%	7.6%	14.1%	22.9%	32.3%
	Pub-25M	3.4%	6.35%	8.6%	17.2%	22.8%
speedup	AIDS	8.73	4.24	2.31	1.90	1.37
	COIL	9.61	4.75	2.24	1.97	1.51
	NASA	6.88	3.69	2.12	1.83	1.54
	Sync-5M	7.58	3.44	2.69	1.88	1.46
	Pub-25M	9.14	4.49	3.04	2.69	1.71

Table 4 shows that $\mathcal{Q}_h$ contains a small number of points; for example, it contains about 25% of the points when $\tau = 5$. This means that MSQ-Index searches only on a small percentage of data graphs in the database. As a result, MSQ-Index achieves 1.4x–9.7x speedup in filtering time compared with SQ-Index.

Besides, we observe that the access ratio increases with the increasing of τ. This is because that a larger τ will produce a larger $\mathcal{Q}_h$. For the experimental results of the subregion length ℓ, see Appendix E4 in supplementary materials.

6.3 Comparing with Existing Methods

In this section, we compare MSQ-Index with existing indexing methods GSimJoin [37], Pars [38], and Mixed [39] to evaluate its performance.

Index Construction. For the above methods, we test their index construction performance and list the obtained results in Table 5, where "-O" means that the memory consumption is out of the 24 GB main memory during the index construction.

TABLE 5: Index size (MByte) and building time (s)

Dataset	GSimJoin		Pars		Mixed		MSQ-Index	
	size	time	size	time	size	time	size	time
AIDS	33.8	5.8	15.6	29.2	37.5	3.2	1.7	2.5
COIL	11.9	2.7	3.1	3.2	7.5	1.5	0.4	0.8
NASA	16.7	12.9	17.9	25.4	47.3	4.2	1.5	3.6
Sync-5M	-O	-O	3859.2	4407.6	6952.9	722.2	597.7	1821.8
Pub-25M	-O	-O	-O	-O	-O	-O	1168.1	2509.5

Table 5 shows that the succinct index MSQ-Index takes much less space than GSimJoin, Pars, and Mixed; its index size is only 5%–15% of other methods. For Pars, it has the longest construction time since there are many subgraph isomorphism tests during index construction. For Mixed, it stores all branch and disjoint substructures, consuming the most space in most cases. Besides, we observe that GSimJoin requires more space than other methods on the COIL dataset. This is because that COIL contains relatively dense graphs and the number of paths increases exponentially in these dense graphs. It is worth to mention that although MSQ-Index is stored in a compressed form, its building time is shortest in most cases.

For the large dataset Pub-25M, GSimJoin, Pars, and Mixed all throw the memory error, and only MSQ-Index can be successfully constructed. At the same time, MSQ-Index's index size is less than 1.2 GB and the index building time is less than 45 minutes, achieving an excellent performance.

Query Performance. We evaluate the query performance of all tested methods under different thresholds τ setting. Fig. 12 shows the *average* candidate set size and the *average* response time for the 100 query graphs.

From Fig. 12, we know that MSQ-Index produces the smallest candidate set. GSimJoin does not perform well because the path-based q-grams have too many overlapping. Mixed performs better than GSimJoin and Pars in most cases. Compared with Mixed, the candidate set size produced by MSQ-Index can decrease by about 67%, 79%, 28% and 25% on the AIDS, COIL, NASA and Sync-5M datasets, respectively, when $\tau = 5$. Notice that for Sync-5M, GSimJoin throws the memory error and we do not display the results.

For the response time of GSimJoin, Pars, Mixed and MSQ-Index, denoted by "G", "P", "M" and "S", respectively, MSQ-Index takes the shortest response time in most cases, which benefits from two aspects: (1) it searches in a reduced query region, requiring the shortest filtering time; (2) it generates the smallest candidate set, consuming the shortest verification time. For Pars, it takes the longest filtering time because it preforms lots of subgraph isomorphism tests. For GSimJoin, it has the shortest response time when $\tau = 1$; this is because that GSimJoin's verifier A* has a better performance than other methods' verifier CSI_GED [12]. However, when τ becomes large (e.g., $\tau > 5$), A* consumes a large amount of memory and running time; as a result, A* cannot finish the verification phase at this time. So we prefer CSI_GED as the default verifier in Pars, Mixed and MSQ-Index. Notice that when $\tau > 5$, we do not show the response time of GSimJoin because its verifier cannot properly run for the memory error.

To evaluate MSQ-Index on the large dataset Pub-25M, we compare it with the online similarity search method, CSI_GED, which is the only method (as far as we know) that can successfully run on this dataset. Fig. 13 shows the average candidate set size and response time.

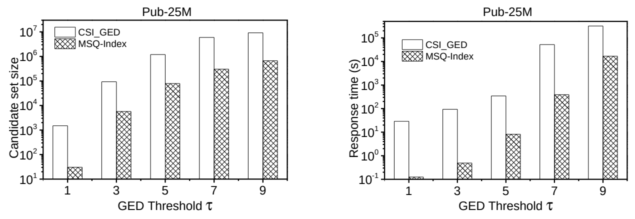


Fig. 13: Results on Pub-25M.

From Fig. 13, we know that the non-indexing method CSI_GED does not perform well. The reasons are as follows: (1) the lower bound employed in CSI_GED has a weak pruning ability, leading to a large candidate set; thus the verification cost is unacceptable. (2) CSI_GED performs pairwise computations to prune data graphs, resulting in a large amount of filtering time. Compared with CSI_GED, MSQ-Index can achieve several hundred times speedup.

6.4 Scalability

In this section, we fix $\tau = 5$ and then evaluate the scalability of the tested methods on the real and synthetic datasets.

Varying $|\mathcal{G}|$. We vary the size of Pub-25M from 500K (kilo) to 25M (million) to study the effect of the database size $|\mathcal{G}|$. As the database size increases, the candidate set size $|\mathcal{C}|$ produced by MSQ-Index shows a non-linear increasing trend. For instance, when increasing the database size from 5M to 10M, $|\mathcal{C}|$ increases by 2.58 times, while increasing from 10M to 20M, it increases by 3.88 times. This naturally leads to a problem: why $|\mathcal{C}|$ does not increases linearly with the size of the database for a fixed threshold $\tau = 5$? The possible reason is that the distribution of data graphs in Pub-25M is uneven (see Appendix E5 in supplementary materials). From Appendix E5, on the uniformly distributed dataset obtained by randomly shuffling the Pub-25M dataset, MSQ-Index shows a linear scalability. Furthermore, when $|\mathcal{G}|$ reaches 5M, 10M, and 10M, GSimJoin, Pars, and Mixed cannot properly run, respectively, because of the memory error. Among these indexing methods, only MSQ-Index can scale to deal with such an extensive database.

Varying $|V_h|$. We vary the query graph size $|V_h|$ from 10 to 60 to study $|V_h|$'s effect. The tested dataset is a subset of Pub-25M, which contains 5 million randomly selected graphs. Fig. 15 shows the average candidate set size and filtering time, where we do not display GSimJoin's results since this method cannot properly run.

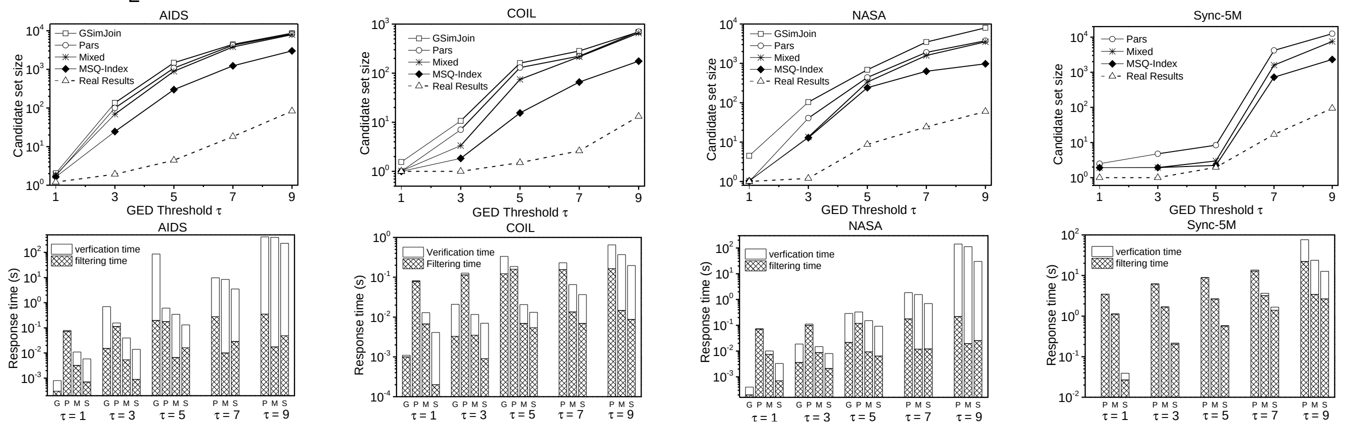


Fig. 12: Average candidate size and response time under different τ.

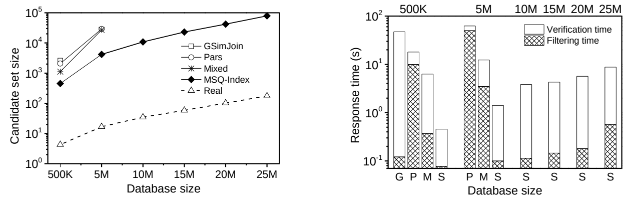


Fig. 14: Scalability *vs.* $|\mathcal{G}|$.

From Fig. 15, as $|V_h|$ increases, the candidate set size of all tested methods first increases and then decreases. The reason is that the distribution of data graphs in the database is not uniform, where the number of graphs whose size near 30 occupies a relatively large proportion. As $|V_h|$ increases, the filtering time of Pars increases steadily, while for MSQ-Index the filtering time incurred first increases and then decreases. This is because that MSQ-Index searches only in a reduced query region, which contains few graphs when $|V_h|$ is smaller than 20 or larger than 50.

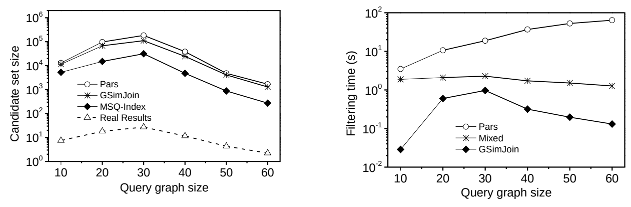


Fig. 15: Scalability *vs.* $|V_h|$.

Varying $|\Sigma_V|$. We generate a group of synthetic datasets to study the effect of the number of vertex labels. Precisely, we fix the dataset size be 100K, the average graph density $\rho = 50\%$, and the number of edges in each data graph be 30, respectively, and then vary the number of vertex labels. Fig. 16 shows the average candidate size. Clearly, as the number of vertex labels increases, the candidate set size of all tested methods decreases; this is because that more label information can be used to filter graphs out.

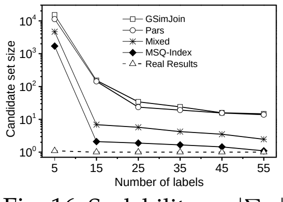


Fig. 16: Scalability *vs.* $|\Sigma_V|$

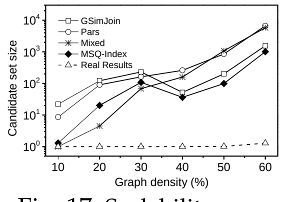


Fig. 17: Scalability *vs.* ρ.

Varying ρ. We fix the synthetic dataset size, the number of edges, and the number of vertex labels be 100K, 30, and 5, respectively, and then vary the graph density ρ to study the effect of graph density. Fig. 17 displays the average candidate set size; it shows that as ρ increases, the candidate set size increases in most cases. This is because that all tested methods only considering the local structures have a weak filtering ability when dealing with the density graphs.

7 Conclusions and Future Work

In this paper, we present a space-efficient index structure for the graph similarity search problem. The proposed succinct index structure incorporates succinct data structures and hybrid encoding, significantly reducing the space usage while at the same time keeping fast query performance. Each entry in the index structure is compressed and requires about 4–6 bits to store on the tested datasets, which takes much fewer bits than that used to store an int type in the previous indexing methods. However, there is still a room for improvement on the space bound. The design of a representation of the q-gram tree that achieves the entropy-compressed space bound while still preserving query efficiency is left as a future work.

8 Acknowledgments

The authors thank the editor and anonymous reviewers for their careful reading and their constructive comments, which have considerably improved the quality and readability of the paper, and thank Xiang Zhao and Xuemin Lin for their codes. This work was supported in part by the National Natural Science Foundation of China grants 61741215, 61373044 and 61173025, and by the U.S. National Science Foundation grant CCF-1017623. Hongwei Huo is the corresponding author.

References

[1] J. Berg and M. Lassig. Local graph alignment and motif search in biological networks. *PNAS*, 101(41):14689–14694, 2004.

[2] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*, pages 1199–1214, 2016.

[3] D. B. Blumenthal and J. Gamper. Improved lower bounds for graph edit distance. *IEEE Trans. Knowl Data Eng.*, 30(3):503–516, 2018.

[4] J. Cheng, Y. Ke, A. W. C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. VLDB. J, 20(4):521–539, 2011.

[5] X. Chen, H. Huo, J. Huan, J. S. Vitter, W. Zheng, and L. Zou. Source code for MSQ-Index. https://github.com/Hongweihuo-Lab/MSQ-Index.

[6] X. Chen, H. Huo, J. Huan, and J. S. Vitter. Efficient Graph Similarity Search in External Memory. *IEEE Access*, 5(1): 4551–4560, 2017.

[7] X. Chen, H. Huo, J. Huan, and J. S. Vitter. An efficient algorithm for graph edit distance computation. *Knowl.-Based Syst.*, 163:762–775, 2019.

[8] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inform Theory.*, 21(2):194–203, 1975.

[9] F. Emmert-Streib, M. Dehmer, and Y. Shi Fifty years of graph matching, network alignment and network comparison. *Inform. Sciences*, 346:180–197, 2016.

[10] M. L. Fernández and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognit Lett.*, 22(6):753–758, 2001.

[11] K. Gouda and M. Arafa. An improved global lower bound for graph edit similarity search. *Pattern Recognit Lett.*, 58:8–14, 2015.

[12] K. Gouda and M. Hassaan. CSI-GED: An efficient approach for graph edit similarity computation. In *ICDE*, pages 256–275, 2016.

[13] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst Sci Cybernetics.*, 4(2):100–107, 1968.

[16] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *DCC*, pages 292–301, 2014.

[17] G. Jacobson. *Succinct data structures*. Carnegie Mellon University, 1989.

[18] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Trans. Pattern Anal Mach Intell.*, 28(8):1200–1214, 2006.

[19] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. In *PVLDB*, pages 181–192, 2013.

[20] H. W. Kuhn. The hungarian method for the assignment problem. Nav. Res. Log., 2:83–97, 1955.

[21] Y. Liang and P. Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *ICDE*, pages 783–794, 2017.

[22] R. M. Marín, N. F. Aguirre and E. E. Daza. Graph theoretical similarity approach to compare molecular electrostatic potentials. *J.Chem.Inf.Model.*, 48(1):109–118, 2008.

[23] M. Neuhaus and H. Bunke. Edit distance-based kernel functions for structural pattern classification. *Pattern Recogn.*, 39(10):1852–1863, 2006.

[24] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[25] M. Rahman, M. A. Bhuiyan and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Trans. Knowl Data Eng.* , 26(10): 2466–2478, 2014.

[26] K. Riesen and H. Bunke. IAM graph database repository for graph based pattern recognition and machine learning. In *SSPR*, pages 287–297, 2008.

[27] K. Riesen, S. Emmenegger, and H. Bunke. A novel software toolkit for graph edit distance computation. In *GbRPR*, pages 142–151, 2013.

[28] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 903–914, 2010.

[29] N. Shervashidze and K. M. Borgwardt. Fast subtree kernels on graphs. In *NIPS*, pages 1660–1668, 2009.

[30] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *ICDE*, pages 210–221, 2012.

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

[32] G. Wang, B. Wang, X. Yang, and G. Yu. Efficiently indexing large sparse graphs for similarity search. *IEEE Trans. Knowl Data Eng.*, 24(3):440–451, 2012.

[33] N. Weskamp, E. Hullermeier, D. Kuhn, and G. Klebe. Multiple graph alignment for the structural analysis of protein active sites. *IEEE/ACM Trans. Comput Biol Bioinformatics.*, 4(2):310–320, 2007.

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

[35] S. J. Yen and A. L. P. Chen. A graph-based approach for discovering various types of association rules. *IEEE Trans. Knowl Data Eng.* , 13(5):839–845, 2001.

[36] 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, 2009.

[37] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. VLDB. J, 22(6):727–752, 2013.

[38] X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang. Efficient structure similarity searches: a partition-based approach. VLDB. J, 27(1):53–78, 2018.

[39] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Efficient graph similarity search over large graph databases. *IEEE Trans. Knowl Data Eng.*, 27(4):964–978, 2015.

Dr. Xiaoyang Chen received the Ph.D. degree in computer science from Xidian University in 2019. His research interests include graph indexing and search, design and analysis of algorithms, external memory algorithms, and compressed indexes.

Dr. Hongwei Huo (SM'17) received the B.S. degree in mathematics from Northwest University, China, and the M.S. degree in computer science and the Ph.D. degree in electronic engineering from Xidian University. She is currently a Professor in School of Computer Science and Technology, Xidian University. Her research interests include the design and analysis of algorithms, graph indexing and search, external memory algorithms and compressed indexes, genome compression, and algorithm engineering.

Dr. Jun (Luke) Huan (SM'11) directs the Baidu Big Data Lab in Beijing. He works on Data Science, AI, Machine Learning and Data Mining. He has published more than 130 peer-reviewed papers in leading conferences and journals. He was a recipient of the US National Science Foundation Faculty Early Career Development Award in 2009. His group won several best paper awards from leading international conferences. Dr. Huan service record includes Program Co-Chair of IEEE BIBM in 2015 among others.

Dr. Jeffrey Scott Vitter (F'93) received the B.S. degree (Hons.) in mathematics from the University of Notre Dame in 1977, the Ph.D. degree in computer science from Stanford University in 1980, and an M.B.A degree from Duke University in 2002. Dr. Vitter is a Fellow of the Guggenheim Foundation, the National Academy of Inventors, the American Association for the Advancement of Science, the ACM, and the IEEE. He is currently Distinguished Professor of Computer and Information Science at the University of Mississippi. His research interests span the design and analysis of algorithms, big data, external memory algorithms, data compression, databases, compressed data structures, parallel algorithms, and machine learning.

Dr. Weiguo Zheng received the Ph.D. degree in computer science from Peking University in July 2015. He is currently an Associate Professor in School of Data Science, Fudan University. His research interests include graph database, knowledge graph management, natural language question answering, and similarity search.

Dr. Lei Zou received his B.S. degree and Ph.D. degree in Computer Science at Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is a full professor in Institute of Computer Science and Technology of Peking University. His research interests include graph database and semantic data management.