LAN: Learning-based Approximate k-Nearest Neighbor Search in Graph Databases

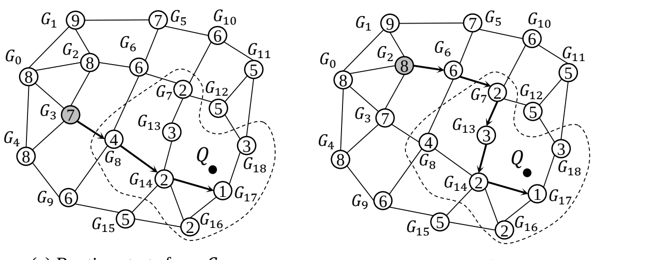
Yun Peng, Byron Choi, Tsz Nam Chan, Jianliang Xu
Department of Computer Science
Hong Kong Baptist University
{yunpeng, bchoi, edisonchan, xujl}@comp.hkbu.edu.hk

Abstract*—k-nearest neighbor (k-NN) search is fundamental in graph databases, which has numerous real-world applications, such as bioinformatics, computer vision, and software engineering. Graph edit distance (GED) and maximum common subgraph (MCS)-based distance are the most widely used distance measures in k-NN search. However, computing the exact k-NNs of a query graph Q using these measures is prohibitively time-consuming, as a large number of graph distance computations is needed, and computing GED and MCS are both NP-hard. In this paper, we study the *approximate k-nearest neighbor* (k-ANN) search for trading efficiency with a slight decrease of accuracy. Greedy routing on the proximity graph (PG) index is the state-of-the-art method for k-ANN search. However, such routing algorithms are not designed for graph databases, and a simple adoption is inefficient. The core reason is that the exhaustive neighbor exploration at each routing step incurs a large number of distance computations (NDC). In this paper, we propose a learning-based k-ANN search method to reduce NDC. First, we propose to prune unpromising neighbors from distance computations. We use a graph learning model to rank the neighbors at each routing step and explore only the top neighbors. For the accuracy of rank prediction, we propose the neighbor ranking model that works only in the neighborhood of Q. Second, we propose a learning-based method to select the initial node for the routing. The initial node selected has a high probability of being in the neighborhood of Q, such that the neighbor ranking model can be used. Third, we propose a compressed GNN-graph to accelerate the neighbor ranking model and initial node selection model. We prove that learning efficiency is improved without degrading the accuracy. Our extensive experiments show that our method is about 3.6x to 18.6x faster than the state-of-the-art methods on real-world datasets.

***Index Terms*—Graph database, Approximate k-NN search, Proximity graph, Learning to route, GNN acceleration**

I. Introduction

k-nearest neighbor (k-NN) search in graph databases, which finds the k most similar graphs to a query graph Q, is a fundamental problem and has a lot of applications, such as cheminformatics [1]–[3], bioinformatics [4], [5], pattern recognition [6]–[8], software engineering [9]–[11]. For example, in cheminformatics, chemists can use k-NN search to find the molecules having similar structures with the query molecule, as molecules with similar graph structures have similar functions [2]. In software engineering, since the control-flow of a code fragment can be modeled as a graph, software engineers can use k-NN search in a database of control-flow graphs to find code plagiarism issues [10]. Many similarity measures have been proposed to quantify the distance between two graphs, where graph edit distance (GED) and maximum common subgraph (MCS)-based distance are the most widely used measures in k-NN search in graph databases (*e.g.*, [3], [6], [12], [13]). However, computing GED and MCS are both NP-hard [6], [14]. It is impractical to use these measures to find the exact k-NNs. For example, in our preliminary experiments, we use the latest graph similarity search method [15] to find the exact 20-NNs of a randomly selected Q from a dataset AIDS that has 42,687 molecule graphs. This method does not finish after 10 hours. Therefore, this paper proposes *k-approximate nearest neighbor* (k-ANN) search in graph databases.



(a) Routing starts from G_3 (a) Routing starts from G_2

Fig. 1: Examples of routing on the proximity graph of a graph database $\mathcal{D} = \{G_0, G_1, ..., G_{18}\}$ to find the 1-NN G_{17} of Q (The numbers in circles are the distances to Q. The routings are marked by bold arrows. The dashed line marks the neighborhood of Q.)

The state-of-the-art index for k-ANN search has been the *proximity graph* (PG) [16]–[20]. It supports k-ANN search in a metric space, but it has not been studied with graph databases before. The general idea of PG is as follows. Given a database $\mathcal{D}$, the nodes of PG are the objects in $\mathcal{D}$ and two nodes have an edge if they fulfill a certain proximity property (*e.g.*, navigable small world property [17]). k-ANN search is evaluated by a greedy routing on PG. At each routing step, a router computes the distances between the query object Q and all neighbors of the current node of the router and routes to the neighbor that is the closest to Q. The router backtracks if the current node has no neighbor that is closer to Q than itself.

A baseline using PG and existing routing methods for k-ANN search in graph databases is, however, inefficient. The core reason is that at each routing step, the router exhaustively computes distances for all neighbors of the current node of the router. Since each node in PG has many neighbors, such exhaustive neighbor exploration incurs a large number of distance computations (NDC), which is too time-consuming, considering computing GED and MCS are both NP-hard. For example, in Fig. 1(a), the routing from G_3 to G_{17} needs to compute distances for all neighbors of G_3, G_8, G_{14}, and G_{17}. The NDC is 13, which is 3.25x of the routing length 4. In our preliminary experiments on AIDS, the NDC can be 20x of the routing length.

In this paper, we propose a learning-based routing method to address k-ANN search in graph databases. The overall approach is to use graph learning to prune unnecessary distance computations. This approach has three main technical issues, and our solutions can be summarized as follows.

First, we propose to route on PG with neighbor pruning. At each routing step, we prune the neighbors of the current node, which are far from the query graph Q, from computing the distances. It is motivated by the experimental observation that the routing seldom routes to the neighbors that are far from Q. For example, on AIDS dataset, at each routing step, we rank the neighbors of the current node by their distances to Q and prune the last 80% neighbors, NDC is reduced by 5.7x while the recall of search results does not decline. We have proved that *if we have an oracle to rank neighbors by their distances to Q, the routing with neighbor pruning uses a smaller NDC than the baseline, but produces the same search results.* We train a graph learning model M_{rk} to approximate the oracle. At each routing step, suppose G is the current node of the router, we use $M_{rk}(x)$ to predict the top $x\%$ neighbors of G. If the prediction of a neighbor G' of G is `false`, G' is pruned from distance computation. For prediction accuracy, we use M_{rk} only when the router has entered the neighborhood of Q. For the example in Fig. 1(a), the router can use $M_{rk}(30)$ at G_8 to prune G_6 and G_9 from distance computations, as G_6 and G_9 are not among the top 30% neighbors of G_8. Similarly, at G_{14} and G_{17}, the router can use $M_{rk}(30)$ to prune G_{13}, G_{15}, G_{16}, and G_{18}, respectively. The NDC is reduced from 13 to 7. Our experiments show that the efficiency of routing with our learning-based neighbor pruning is ~2.9x higher than that of the baseline on AIDS.

Second, we propose a learning-based method to select the initial node for the routing. The aim is that the initial node selected is in the neighborhood $\mathcal{N}_Q$ of Q, such that the neighbor pruning technique is used in most routing steps. Specifically, we train a graph learning model M_{nh} to predict that if a graph $G \in \mathcal{D}$ is in $\mathcal{N}_Q$. From the predicted neighborhood $\hat{\mathcal{N}_Q}$, we randomly sample s graphs and compute their distances to Q. The top samples are used as the initial nodes. We have proved that the initial node found has a high probability of being in $\mathcal{N}_Q$. To reduce the time complexity of initial node selection, we further propose a cluster-based method to prune unpromising graphs in $\mathcal{D}$. For the example in Fig. 1(a), if G_8 is selected as the initial node instead of G_3, the NDC is reduced from 7 to 3. Our experiments show that our initial node selection method can speed up the routing by ~2x when compared with randomly selecting a node as the initial node on AIDS.

Third, we propose to accelerate the graph learning used in the neighbor pruning and initial node selection. Both M_{rk} and M_{nh} take cross-graph learning between a data graph G and a query graph Q as the core module, as it is the latest method for graph distance learning [21], [22]. In the cross-graph learning, a node of G not only aggregates embeddings of its neighbors in G but also pays attention to the embeddings of all nodes in Q. Since many nodes have the same embedding, there are a lot of redundant computations in the cross-graph learning. We propose a compressed GNN-graph (CG) to group the nodes having the same embedding. We have proved that cross-learning on the CGs of G and Q is more efficient than that on G and Q without degrading the learning accuracy. Our experiments show that the k-ANN search is ~1.17x faster by using our cross-graph learning acceleration on AIDS. By using these techniques, the 20-ANN queries on AIDS dataset take ~40 seconds on average.

Contributions. The contributions of this paper are as below.

- We propose a learning-based neighbor pruning method to reduce the number of distance computations in the routing on PG of a graph database.
- We propose a learning-based initial node selection method. The initial node selected has a high probability of being in the neighborhood of Q.
- We propose a novel compressed GNN-graph (CG). Cross-learning on the CGs of G and Q is equivalent to but faster than that on G and Q.
- Our extensive experiments verify the effectiveness and efficiency of our proposed techniques.

Organizations. The rest of this paper is organized as follows. Sec. II discusses the related works. The preliminaries and problem definition are presented in Sec. III. Sec. IV presents the techniques of routing with neighbor pruning. The learning-based initial node selection is presented in Sec. V. Sec. VI presents the graph learning acceleration method. The experimental evaluation is presented in Sec. VII. Sec. VIII concludes this paper. All proofs are given in Appendix.

II. Related Work

In this section, we summarize the works that are closely related to this paper.

A. Proximity graph

Proximity graph (PG)-based method is the state-of-the-art index for k-ANN search (*e.g.*, [16]–[18], [20], [23]–[29]). Existing works of k-ANN search on PG can be categorized into two classes. The first class focuses on the proximity property of PG. For example, the navigable small world property is studied in [17]. The relative neighborhood relationship is studied in [18]. The second class focuses on the query evaluation on PG, which includes improving routing efficiency and selecting high-quality initial nodes for routing.

For efficiently routing on PG, Muñoz et al. [30] propose a quadrant-based method to prune unpromising neighbors from distance computations at each routing step. However, it cannot be used in graph data as graph data have no concept of quadrants. Baranchuk et al. [28] propose a learning-based routing method. However, it is tailor-made for 1-ANN search, and it is not clear how to extend it to efficiently support k-ANN search.

For initial node selection, several works [19], [31] randomly sample a node in PG as the initial node. `DLG` [32] and `HNSW` [17] construct a hierarchy of PGs and use the k-ANN search on the higher level PG to find the initial node for the lower level. However, these methods cannot bound the distance between Q and the initial node selected. Qin et al. [9] propose a learning-based approximate range query method in graph databases. However, if the range threshold is not small, it will take a large NDC, as it can be observed from our experiments.

There are also many indexes for k-ANN search not based on PG, *e.g.*, LSH [33] and inverted file index [16]. However, recent studies [20], [34], [35] report that PG outperforms these indexes. Therefore, we omit their details in this subsection.

B. Graph distance learning

Several recent works study using graph neural networks (GNNs) to learn the distance between two graphs. The common idea is to first learn the cross-graph embedding of G and Q and then feed the cross-graph embedding to multilayer perceptrons (MLPs) or convolutional neural networks (CNNs) to predict the distance between G and Q. For example, Li et al. [22] propose a cross-graph attention network `GMN`. Each node of G aggregates information from not only its neighbors in G but also all nodes of Q in cross-graph convolution. Bai et al. [36] use the matrix of the inner products of the embeddings of the nodes in G and Q as the cross-graph embedding. Peng et al. [37] propose to use a GNN on the association graph of G and Q to learn the cross-graph embedding of the two graphs. A recent work [21] integrates the idea of `GMN` in the A* search of GED computation.

C. Graph neural network acceleration

Many recent works study GNN acceleration. For example, `GraphSAGE` [38], `FastGCN` [39], and `Adapt` [40] sample a subset of neighbors in graph convolution. `SGCN` [41] removes the non-linear activation functions. Ye and Ji [42] propose the sparse attention technique to accelerate the graph attention network. `DegreeQuant` [43] and `#GNN` [44] quantize the node embeddings as integer vectors and binary vectors, respectively. However, these methods do not guarantee that the learning accuracy is preserved after acceleration.

The work that is most closely related to this paper is `HAG` [45], which accelerates GNN while preserving the accuracy. `HAG` aggregates the redundant sum operations in GNN learning. However, `HAG` cannot reduce the number of matrix multiplications, which is a bottleneck in cross-graph learning.

There are also works that accelerate GNN by taking advantage of the optimized graph primitives of certain platforms (*e.g.*, G^3 [46]) or optimizing GPU kernels (*e.g.*, `fuseGNN` [47]). These works are orthogonal to this paper, as we focus on reducing redundant computations in GNN learning.

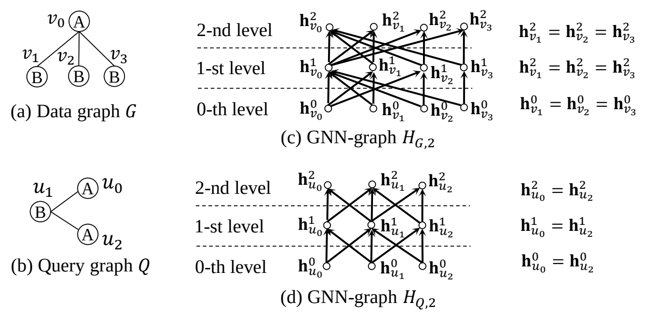

v_0 A
v_1 v_2 v_3
B B B
(a) Data graph G
2-nd level
1-st level
0-th level
$\mathbf{h}^2_{v_0}$ $\mathbf{h}^2_{v_1}$ $\mathbf{h}^2_{v_2}$ $\mathbf{h}^2_{v_3}$
$\mathbf{h}^1_{v_0}$ $\mathbf{h}^1_{v_1}$ $\mathbf{h}^1_{v_2}$ $\mathbf{h}^1_{v_3}$
$\mathbf{h}^0_{v_0}$ $\mathbf{h}^0_{v_1}$ $\mathbf{h}^0_{v_2}$ $\mathbf{h}^0_{v_3}$
$\mathbf{h}^2_{v_1} = \mathbf{h}^2_{v_2} = \mathbf{h}^2_{v_3}$
$\mathbf{h}^1_{v_1} = \mathbf{h}^1_{v_2} = \mathbf{h}^1_{v_3}$
$\mathbf{h}^0_{v_1} = \mathbf{h}^0_{v_2} = \mathbf{h}^0_{v_3}$
(c) GNN-graph $H_{G,2}$
u_1 A u_0
B
A u_2
(b) Query graph Q
2-nd level
1-st level
0-th level
$\mathbf{h}^2_{u_0}$ $\mathbf{h}^2_{u_1}$ $\mathbf{h}^2_{u_2}$
$\mathbf{h}^1_{u_0}$ $\mathbf{h}^1_{u_1}$ $\mathbf{h}^1_{u_2}$
$\mathbf{h}^0_{u_0}$ $\mathbf{h}^0_{u_1}$ $\mathbf{h}^0_{u_2}$
$\mathbf{h}^2_{u_0} = \mathbf{h}^2_{u_2}$
$\mathbf{h}^1_{u_0} = \mathbf{h}^1_{u_2}$
$\mathbf{h}^0_{u_0} = \mathbf{h}^0_{u_2}$
(d) GNN-graph $H_{Q,2}$

Fig. 2: An example of two graphs and their GNN-graphs (A and B are node labels)

III. Preliminaries and Problem Definition

This paper studies undirected graphs with node labels. A graph is denoted as $G = (V_G, E_G, \ell_G)$, where V_G and E_G are the node set and the edge set of G, respectively, and $\ell_G(v)$ is the label of node v. The set of neighbors of a node $v \in G$ is denoted by $N_G(v)$. The subscript may be omitted if it is clear from the context.

A. Search semantics

We focus on GED in this paper, as MCS is a special case of GED [48]. A graph G can be transformed to another graph G' by five edit operations: node and edge insertion, node and edge deletion, and node relabeling. Graph edit distance (GED) of G and G', denoted by $d(G, G')$, is the smallest number of edit operations that transform G to G'.

Example 1. *Fig. 2(a) and (b) show a data graph G and a query graph Q, respectively.* $d(G, Q) = 5$.

Approximate k-Nearest Neighbor Search Problem. Given a graph database $\mathcal{D}$, a query graph Q, and a parameter $k \ll |\mathcal{D}|$, approximate k-nearest neighbor (k-ANN) search is to find the k graphs from $\mathcal{D}$ whose distances to Q are the smallest.

B. Routing on proximity graph

Given a graph database $\mathcal{D}$, the nodes of PG are the graphs in $\mathcal{D}$ and two nodes have an edge if they fulfill a certain proximity property (*e.g.*, navigable small world property [17]). Algorithm 1, which is a a beam search on PG, presents the logic of a router for answering a k-ANN search. Line 1 initializes a priority queue W to store candidates, which is ordered by the distances of the candidates from Q. The binary flag G.explored marks if the router has computed distances for the neighbors of G. Line 3 selects the initial node for the routing. Lines 5-10 perform the routing steps. At each routing step, the router picks the unexplored node in W having the smallest distance to Q as the current node. When there is a tie, the node id is used to break the tie. Then, the router computes distances for all neighbors of the current node and adds all of them to W (Line 7). Then, W is resized to b (Line 9). During the resizing of W, for two nodes in W that have the same distance to Q, the unexplored one has a higher priority; if both are explored, the recently explored one has a higher priority; and if both are unexplored, the one with a smaller

Algorithm 1 Greedy routing on proximity graph (`baseline`)

```
Input: PG $\mathcal{G}$, query $Q$, beam size $b$, parameter $k$
Output: $k$-ANNs of $Q$
1: initialize a priority queue $W = \emptyset$ as the pool to store candidates
2: $G$.explored is false by default for each node $G$ in $\mathcal{G}$
3: $G_{init}$ = select an initial node in $\mathcal{G}$
4: $W$.add(($d(G_{init}, Q), G_{init}$))          ▷ $W$ is in ascending order of $d$
5: while $W$ has unexplored nodes do
6:     $G$ = the unexplored node in $W$ having the smallest GED to $Q$
7:     neigh_explore($\mathcal{G}$, $G$, $Q$)
8:     $G$.explored = true
9:     resize $W$ to size $b$
10: end while
11: $\mathcal{R}_{bs}$ = top-$k$ in $W$
12: return $\mathcal{R}_{bs}$
13: function neigh_explore($\mathcal{G}$, $G$, $Q$)
14: for each neighbor $G'$ of $G$ in $\mathcal{G}$ do
15:     add $(d(G', Q), G')$ into $W$
16: end for
```

node id has a higher priority. The routing stops if all nodes in W have been explored and the top-k graphs in W are returned.

C. Graph neural network

The main idea of Graph neural networks (GNNs) is to use graph convolutions to learn the embeddings of graphs. In this paper, we adopt the well-known `GIN` model [49]. Given a graph G, the graph convolution of `GIN` is as follows.[1] For a node u of G,

$$\mathbf{h}_u^l = ReLU(\mathbf{W}^l(\mathbf{h}_u^{l-1} + \sum_{v \in N(u)} \mathbf{h}_v^{l-1})) \tag{1}$$

where l denotes the layer id, $\mathbf{h}_u^l$ is the embedding of u at the l-th layer, v is a neighbor of u, and $\mathbf{W}^l$ is the trainable parameter matrix, respectively. $\mathbf{h}_u^0$ is the input feature of u (*e.g.*, the one-hot encoding of $\ell(u)$). If there are L layers, the embedding of G is $\mathbf{h}_G = mean_{u \in G}\mathbf{h}_u^L$.

The work [49] has proved that `GIN` is equivalent to the Weisfeiler-Lehman (WL) labeling, which is a well-known technique to test graph isomorphism. Given a graph $G = (V, E, \ell)$, WL labeling iteratively computes the WL labels of the nodes in G. The WL label of a node $u \in G$ at the l-th iteration, denoted by $wl^l(u)$, is as below.

$$wl^l(u) = wl^{l-1}(u), \{\{wl^{l-1}(v) | v \in N(u)\}\}, \text{and} \tag{2}$$

$$wl^0(u) = \ell(u), \tag{3}$$

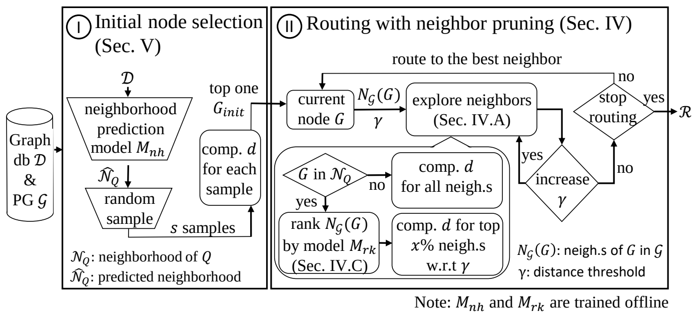
where $\{\{\}\}$ denotes multi-set.

The nodes in G having the same WL label at the l-th iteration of WL labeling must have the same embedding at the l-th layer of `GIN`.

D. GNN-graph

The work [45] represents the computation of graph convolutions of a GNN by a GNN-graph. Given a graph G and a GNN having L graph convolution layers, the GNN-graph of G is an $L+1$-level directed acyclic graph $H_{G,L} = (V_0, V_1, ..., V_L, E)$, where $V_l = \{\mathbf{h}_u^l | u \in G\}$.[2] For two nodes $\mathbf{h}_u^l$ and $\mathbf{h}_v^{l-1}$, $(\mathbf{h}_v^{l-1}, \mathbf{h}_u^l) \in H_{G,L}$ if $(u, v) \in G$. For each $u \in G$, $(\mathbf{h}_u^{l-1}, \mathbf{h}_u^l)$ is in $H_{G,L}$. Since many nodes of G have the same label, many nodes in $H_{G,L}$ have identical embeddings.

[1]The coefficient $1 - \epsilon$ of $\mathbf{h}_u^{l-1}$ is omitted for presentation simplicity and it does not affect the correctness of this paper.



I Initial node selection (Sec. V)
II Routing with neighbor pruning (Sec. IV)
$\mathcal{D}$
neighborhood prediction model M_{nh}
$\hat{\mathcal{N}}_Q$
random sample
s samples
top one G_{init}
comp. d for each sample
Graph db $\mathcal{D}$ & PG $\mathcal{G}$
route to the best neighbor
current node G
$N_{\mathcal{G}}(G)$ γ
explore neighbors (Sec. IV.A)
stop routing
no
yes
$\mathcal{R}$
G in $\mathcal{N}_Q$
no
yes
comp. d for all neigh.s
increase γ
rank $N_{\mathcal{G}}(G)$ by model M_{rk} (Sec. IV.C)
comp. d for top x% neigh.s w.r.t γ
$\mathcal{N}_Q$: neighborhood of Q
$\hat{\mathcal{N}}_Q$: predicted neighborhood
$N_{\mathcal{G}}(G)$: neigh.s of G in $\mathcal{G}$
γ: distance threshold
Note: M_{nh} and M_{rk} are trained offline

Fig. 3: Overview of our learning-based k-ANN search

Example 2. *For the graphs G and Q in Fig. 2(a) and (b), Fig. 2(c) and (d) shows the GNN-graphs $H_{G,2}$ and $H_{Q,2}$, respectively. Since $\ell(v_1) = \ell(v_2) = \ell(v_3)$ and $\ell(u_0) = \ell(u_2)$, $\mathbf{h}_{v_1}^l = \mathbf{h}_{v_2}^l = \mathbf{h}_{v_3}^l$ and $\mathbf{h}_{u_0}^l = \mathbf{h}_{u_2}^l$, for $l = 0, 1, 2$.*

E. Cross-graph learning

Cross-graph learning is the state-of-the-art technique for graph distance learning [21], [22]. Given two graphs G and Q, each node u of G not only aggregates embeddings from its neighbors in G but also pays attention to the embeddings of all nodes in Q. The cross-graph learning is defined as below.

Definition 1. *Given two graphs G and Q, for each node $u \in G$, the embedding of u is computed as follows.*

$$\mathbf{h}_u^l = ReLU(\mathbf{W}^l(\mathbf{h}_u^{l-1} + (\sum_{u' \in N(u)} \mathbf{h}_{u'}^{l-1}) + \boldsymbol{\mu}_u^{l-1}), \tag{4}$$

$$\boldsymbol{\mu}_u^{l-1} = \sum_{v \in Q} \alpha_{u,v}\mathbf{h}_v^{l-1}, \text{ and} \tag{5}$$

$$\alpha_{u,v} = \frac{exp(\mathbf{a} \cdot (\mathbf{h}_u^{l-1} || \mathbf{h}_v^{l-1}))}{\sum_{v' \in Q} exp(\mathbf{a} \cdot (\mathbf{h}_u^{l-1} || \mathbf{h}_{v'}^{l-1}))}, \tag{6}$$

where $||$ denotes concatenation, $\mathbf{a}$ is a trainable parameter vector, $\cdot$ denotes inner product, and α is attention weight. If there are L layers, $\mathbf{h}_G = mean_{u \in G}\mathbf{h}_G^L$. The cross-graph embedding of G and Q is $\mathbf{h}_{G,Q} = \mathbf{h}_G || \mathbf{h}_Q$.

F. Solution overview

Fig. 3 shows the overview of our learning-based k-ANN search. Given a graph database $\mathcal{D}$ and a PG $\mathcal{G}$ of $\mathcal{D}$, we propose a learning-based routing on $\mathcal{G}$ to efficiently find the k-ANNs of Q. The key is using graph learning models to prune unnecessary distance computations. Given a query graph Q, we maintain a priority queue W to store the candidates. (I) We use a model M_{nh} to predict the neighborhood $\mathcal{N}_Q$ of Q. From the predicted neighborhood $\hat{\mathcal{N}}_Q$, we randomly sample s graphs and compute their distances from Q. The top sample is

[2]We slightly abuse the symbol $\mathbf{h}_u^l$ to denote an embedding vector and a node in V_l of H.

Algorithm 2 Routing with neighbor pruning (`np_route`)

```
Input: PG $\mathcal{G}$, query $Q$, beam size $b$, answer count $k$, oracle $\mathcal{O}$, step size $d_s$
Output: $k$-ANNs of $Q$
1: initialize a priority queue $W = \emptyset$ as the pool to store candidates
2: $G$.explored is false by default for each node $G$ in $\mathcal{G}$
3: $G_{init}$ = select a node in $\mathcal{G}$ as the initial node
4: $W$.add(($d(G_{init}, Q), G_{init}$))          ▷ $W$ is in ascending order of $d$
5: $G$ = the node in $W$ having the smallest GED to $Q$
6: while $G$.explored = false do
7:     rank_expl($\mathcal{G}$, $G$, $Q$, $W$, $d(G, Q)$, $\mathcal{O}$)
8:     $G$.explored = true
9:     resize $W$ to $b$
10:    $G$ = the node in $W$ with the smallest $d$
11: end while
12: $G_{flo}$ = the node in $W$ with the smallest $d$          ▷ first local optimal
13: $\gamma = d(G_{flo}, Q) + d_s$
14: while true do
15:    for each $G$ that is explored do
16:        all_quali_neigh($\mathcal{G}$, $G$, $Q$, $W$, $\gamma$, $\mathcal{O}$)
17:    end for
18:    resize $W$ to $b$
19:    if all nodes in $W$ have been explored then
20:        break                                   ▷ stop condition of routing
21:    end if
22:    while $W$ has unexplored nodes with $d \leq \gamma$ do
23:        $G$ = the unexplored node in $W$ with the smallest $d$
24:        rank_expl($\mathcal{G}$, $G$, $Q$, $W$, $\gamma$, $\mathcal{O}$)
25:        $G$.explored = true
26:        resize $W$ to $b$
27:    end while
28:    $\gamma = \gamma + d_s$
29: end while
30: $\mathcal{R}_{np}$ = top-$k$ in $W$
31: return $\mathcal{R}_{np}$
```

used as the initial node G_{init} for the routing on $\mathcal{G}$ and G_{init} is added to W. (II) At each routing step, let G be the current node of the router. If G is not in $\mathcal{N}_Q$, we simply compute distances for all neighbors of G. If G is in $\mathcal{N}_Q$, we use a model M_{rk} to rank the neighbors of G and only compute distances for the top $x\%$ neighbors. $x\%$ is controlled by a distance threshold γ, which can be updated as the routing progresses, if needed. The neighbors whose distances from Q are computed are added to W. If the stopping condition is met, the routing stops and the top-k of W is returned as the search results $\mathcal{R}$. We remark that the construction of $\mathcal{G}$ and training of the models M_{nh} and M_{rk} are offline.

IV. Routing with Neighbor Pruning

The main idea of routing with neighbor pruning is that when the router reaches a node G in the PG, we do not compute distances for all neighbors of G at once. We rank the neighbors and only compute distances for the top $x\%$ neighbors of G. $x\%$ is controlled by a distance threshold, which can be updated as the routing progresses.

In the following, we first present the algorithm of routing with neighbor pruning under the assumption of an oracle for neighbor ranking. Then, we remove the assumption by proposing a graph learning model to approximate the oracle.

A. Algorithm of routing with neighbor pruning

Assume we have an oracle that can rank the neighbors of G in a negligible time. Then, we can use the oracle to partition

Algorithm 3 Get all qualified neighbors (`all_quali_neigh`)

```
Input: PG $\mathcal{G}$, node $G$ of $\mathcal{G}$, query $Q$, pool $W$, threshold $\gamma$, oracle $\mathcal{O}$
1: suppose $G$'s neighbors are ranked by $\mathcal{O}$ and partitioned to $B_0, B_1, ..., B_n$
2: suppose $B_0, B_1, ..., B_i$ have been opened
3: for $j = 0$ to $i$ do
4:     for each $G'$ in $B_j$ that is unexplored do
5:         add $(d(G', Q), G')$ into $W$
6:     end for
7:     if Line 5 adds a neighbor $G'$ in $B_j$ to $W$ and $d(Q, G') \geq \gamma$ then
8:         return
9:     end if
10: end for
11: for $j = i + 1$ to $n$ do
12:    for each neighbor $G'$ in $B_j$ do                ▷ open the batch $B_j$
13:        compute $d(G', Q)$ and add $(d(G', Q), G')$ into $W$
14:    end for
15:    if Line 13 adds a neighbor $G'$ in $B_j$ to $W$ and $d(Q, G') \geq \gamma$ then
16:        return
17:    end if
18: end for
19: return
```

Algorithm 4 Neighbor ranking and exploration (`rank_expl`)

```
Input: PG $\mathcal{G}$, node $G$ of $\mathcal{G}$, query $Q$, pool $W$, threshold $\gamma$, oracle $\mathcal{O}$
1: suppose $G$'s neighbors are ranked by $\mathcal{O}$ and partitioned to $B_0, B_1, ..., B_n$
2: suppose $B_0, B_1, ..., B_i$ have been opened
3: $G'$ is the neighbor farthest from $Q$ in the opened batches of $G$'s neighbors
4: if $d(Q, G') \geq \gamma$ then
5:     return
6: end if
7: for $j = i + 1$ to $n$ do
8:     for each neighbor $G'$ in $B_j$ do                ▷ open the batch $B_j$
9:         compute $d(G', Q)$ and add $(d(G', Q), G')$ into $W$
10:    end for
11:    if $B_j$ contains a neighbor $G'$ satisfying $d(Q, G') \geq \gamma$ then
12:        return
13:    end if
14: end for
```

the neighbors of G into batches $B_0, B_1, ..., B_n$ of equal size for controlling the percentage of neighbors in distance computation. Each batch has $y\%$ neighbors of G, where y is a tunable parameter, and all neighbors in B_i are farther from Q than all neighbors in B_{i-1}, for $i = 1, ..., n$. Given a distance threshold γ, we sequentially open the batches $B_0, B_1, ..., B_i$ and compute distances for the neighbors in the opened batches. If B_i has a neighbor G' satisfying $d(G', Q) > \gamma$, we do not open the batches after B_i. We call G is explored, *i.e.* G.explored = `true`, if at least a batch of the neighbors of G is opened.

`np_route` (Algorithm 2) presents the algorithm of routing with neighbor pruning using the oracle. The routing has two stages. The first stage is routing without backtracking until the first local optimal G_{flo} is reached (Lines 1-12). The second stage is routing with backtracking (Lines 13-29).

In the first stage, at each routing step, let G be the current node of the router, $d(Q, G)$ is used as the GED threshold to control the percentage of neighbor exploration at G (Line 7).

The second stage has several while-loops (Lines 14-29). In each while-loop (Lines 14-29), there is a GED threshold γ, and the routing is restricted in the nodes whose GEDs to Q do not exceed γ (Lines 22-27). To avoid missing a qualified node, for each explored node G, Lines 15-17 add all the unexplored neighbors of G whose distances to Q are no more than γ into W (note that G may not be in W). If W has no unexplored node that is within distance γ from Q, γ is increased by d_s (Line 28). The for-loop (Lines 15-17) adds all qualified neighbors to W w.r.t the new γ. The routing stops if all nodes in W are explored (Line 19), and the top-k in W are returned.

Example 3. *Consider the example shown in Fig. 1(b). Suppose $y = 30$, $b = 2$, $k = 1$, and $d_s = 1$. The first stage is the routing from the initial node G_2 to G_7 and the second stage is the routing from G_7 to G_{17}.*

The first stage: At G_2, $d(Q, G_2) = 8$ is used as the distance threshold for neighbor exploration. Since $y = 30$, the batches of the neighbors of G_2 are $B_{0,G_2} = \{G_6\}$, $B_{1,G_2} = \{G_3\}$, $B_{2,G_2} = \{G_0\}$, and $B_{3,G_2} = \{G_1\}$. Due to the distance threshold, B_{0,G_2} and B_{1,G_2} are opened in Line 7. At this point $W = [G_6, G_3, G_2]$. Line 9 updates W to be $[G_6, G_3]$. Then, Line 7 explores G_6. At G_6, $d(Q, G_6) = 6$, which is used as the distance threshold. The batches of the neighbors of G_6 are $B_{0,G_6} = \{G_7\}$, $B_{1,G_6} = \{G_8\}$, $B_{2,G_6} = \{G_5\}$, and $B_{3,G_6} = \{G_2\}$. B_{0,G_6} and B_{1,G_6} are opened in Line 7. At this point $W = [G_7, G_8, G_6, G_3]$. Line 9 updates W to be $[G_7, G_8]$. Then, Line 7 explores G_7. At G_7, $d(Q, G_7) = 2$, which is used as the distance threshold. The batches of the neighbors of G_7 are $B_{0,G_7} = \{G_{13}\}$, $B_{1,G_7} = \{G_{12}\}$, and $B_{2,G_7} = \{G_6, G_{10}\}$. B_{0,G_7} is opened in Line 7. Since $d(Q, G_{13}) > d(Q, G_7)$, G_7 is the first local optimal. At this point, $W = [G_7, G_{13}]$.

The second stage: Since $d_s = 1$, $\gamma = d(Q, G_7) + d_s = 3$, which is used as the distance threshold (Line 13). The for-loop (Lines 15-17) makes sure that no qualified neighbor of the explored nodes is missed. Line 16 for G_2 adds G_3 into W and Line 16 for G_6 adds G_8 into W. At this point, $W = [G_7, G_{13}, G_8, G_3]$. Line 18 updates W to be $[G_7, G_{13}]$. Since G_{13} is unexplored and $d(Q, G_{13}) \leq \gamma = 3$, Line 24 explores the neighbors of G_{13}. All the neighbors of G_{13} are in a batch $B_{0,G_{13}} = \{G_7, G_{14}\}$. $B_{0,G_{13}}$ is opened. At this point, $W = [G_{14}, G_7, G_{13}]$. Line 26 updates W to be $[G_{14}, G_7]$. Then, Line 24 explores G_{14}. The batches of the neighbors of G_{14} are $B_{0,G_{14}} = \{G_{17}\}$, $B_{1,G_{14}} = \{G_{16}, G_{13}\}$, $B_{2,G_{14}} = \{G_8\}$, and $B_{3,G_{14}} = \{G_{15}\}$. Since $\gamma = 3$, the batches $B_{0,G_{14}}$ and $B_{1,G_{14}}$ are opened. At this point $W = [G_{17}, G_{16}, G_{14}, G_7, G_{13}]$. Line 26 updates W to be $[G_{17}, G_{16}]$. Then, Line 24 explores G_{17}. At G_{17}, all the batches of the neighbors of G_{17} are opened and $W = [G_{17}, G_{16}, G_{14}, G_{18}]$. Line 26 updates W to be $[G_{17}, G_{16}]$. Since $d(Q, G_{16}) < \gamma = 3$, Line 24 explores G_{16}. All neighbors of G_{16} are added to W and at this point $W = [G_{17}, G_{16}, G_{14}, G_{15}]$. Line 26 updates W to be $[G_{17}, G_{16}]$. Since W has no unexplored node, Line 28 increases $\gamma = 3 + d_s = 4$. The for-loop (Lines 15-17) adds G_3, G_8, G_{12}, G_{15}, and G_{18} into W. At this point $W = [G_{17}, G_{16}, G_{18}, G_8, G_{15}, G_{12}, G_3]$. Line 18 updates W to be $[G_{17}, G_{16}]$. Since all nodes in W have been explored, Line 19 stops the routing and G_{17} is returned.

B. *Analysis of* `np_route` *with an oracle neighbor ranker*

We analyze the performance of `np_route` using the oracle neighbor ranker (Algorithm 2) by comparing the search results and the NDC with `baseline` (Algorithm 1).

Lemma 1. *Given the same initial node G_{init} and beam size b, the sequence of nodes explored by* `np_route` *is the same with that of* `baseline`*, where for a node G, the time when G is explored is the time when $G.explored$ is set to* `true`*.*

Theorem 1. *Let $\mathcal{R}_{np}$ and $\mathcal{R}_{bs}$ denote the sets of graphs returned by* `np_route` *and* `baseline`*, respectively. Given the same initial node G_{init} and beam size b, $\mathcal{R}_{np} = \mathcal{R}_{bs}$. The NDC used by* `np_route` *is no more than that of* `basline`*. The space cost of* `np_route` *is at most two times of that of* `baseline`*.*

C. *Learning-based neighbor ranking*

One may attempt to directly train a model to rank the neighbors of G. However, it is technically challenging to obtain such a model to fully rank the neighbors of G. Since each batch has $y\%$ of G's neighbors, we propose to train $100/y$ binary rankers and the i-th ranker M_{rk}^i partially ranks the neighbors of G. Specifically, M_{rk}^i classifies G's neighbors into two classes: the positive class consisting the top $iy\%$ neighbors and the negative class consisting the remain neighbors.

1) Model design: For a node G and a query Q, for each neighbor G' of G, M_{rk}^i first learns the cross-graph embedding $\mathbf{h}_{G',Q}$ of G' and Q (Sec. III-E) and then feeds the concatenation of $\mathbf{h}_{G',Q}$ and $\mathbf{h}_G$ to a multilayer perceptron (MLP) to make a binary classification. The binary cross-entropy with the regularizer in [37] is used as the loss function.

2) Model training: Assume we have a query workload $\mathcal{Q}$, which can be historical queries or sampled from the database $\mathcal{D}$ [9]. A basic method to train M_{rk}^i is that for each $Q \in \mathcal{Q}$, we use each node G in $\mathcal{G}$ and G's neighbors as the training data. Formally, the training data is $\{(Q, G', G) | Q \in \mathcal{Q}, G' \in N_{\mathcal{G}}(G), G \in \mathcal{G}\}$. The class label of (Q, G', G) is positive if G' is among the top $iy\%$ of G's neighbors; otherwise, negative. However, the training data can be huge. It may take a long time to train M_{rk}^i.

Recent studies [26], [35] find that most routing steps are in the neighborhood $\mathcal{N}_Q$ of Q. $\mathcal{N}_Q$ can be defined as $\{G | d(Q, G) < \gamma^*, G \in \mathcal{G}\}$, where γ^* is a tunable parameter. Motivated by this observation, we propose to use M_{rk}^i only after the router enters the neighborhood of Q. It makes that we only need to use $\{(Q, G', G) | Q \in \mathcal{Q}, G' \in N_{\mathcal{G}}(G), G \in \mathcal{N}_Q\}$ as the training data to train M_{rk}^i. Since $|\mathcal{N}_Q| \ll |\mathcal{G}|$, the size of training data is significantly reduced, and hence M_{rk}^i can be efficiently trained.

V. Learning-based Initial Node Selection

Since the neighbor ranking models work only in the neighborhood $\mathcal{N}_Q$ of Q, the initial node of the routing should be in $\mathcal{N}_Q$ in order to use the neighbor pruning technique in most routing steps. In the following, we first present the initial node selection method given a neighborhood prediction model M_{nh}. Then, we present the design of M_{nh}.

A. Initial node selection method

Suppose we have trained a model M_{nh} using the query workload $\mathcal{Q}$, such that for each graph G in $\mathcal{D}$, M_{nh} can predict if G is in the neighborhood of Q. Then, we randomly sample s graphs in the predicted neighborhood $\hat{\mathcal{N}}_Q$ and compute the GEDs to Q for the samples. The samples having the smallest GEDs to Q are used as the initial nodes. This method has a high probability of finding a graph in $\mathcal{N}_Q$.

Lemma 2. *Suppose the precision of the prediction of M_{nh} is p, i.e., $p = |\hat{\mathcal{N}}_Q \cap \mathcal{N}_Q|/|\hat{\mathcal{N}}_Q|$. The probability that the s samples have at least a graph in $\mathcal{N}_Q$ is $1-(1-p)^s$.*

B. Neighborhood prediction model

We first present a basic design of the neighborhood prediction model. Then, we propose an optimized design.

1) Basic design of M_{nh}: Given a data graph G and a query Q, M_{nh} first learns the cross-graph embedding $\mathbf{h}_{G,Q}$ (Sec. III-E) and then feeds $\mathbf{h}_{G,Q}$ to an MLP to make a binary prediction. If G is in $\mathcal{N}_Q$, it is positive; otherwise, negative. The binary cross entropy with the regularizer [37] is used as the loss function. Since the number of negative samples is much larger than that of positive samples, the negative class downsampling technique [50] is used for training M_{nh}.

This design of M_{nh} has a shortcoming. To predict $\mathcal{N}_Q$, we need to make a prediction for each graph in $\mathcal{D}$. The number of predictions is $O(|\mathcal{D}|)$, which is inefficient when $\mathcal{D}$ is large.

2) Optimized design of M_{nh}: We adopt the cluster-based learning framework [51] to reduce the number of predictions. The main idea is to cluster the graphs in $\mathcal{D}$ into a set of clusters $\mathcal{C}$. We train a model M_c to predict the size of the intersection between each cluster C in $\mathcal{C}$ and $\mathcal{N}_Q$ for each query Q in $\mathcal{Q}$. The distribution of the intersection size between $\mathcal{N}_Q$ and the clusters is skewed. Hence, we use a neural network to learn the distribution.

The number of predictions is reduced by using M_c. Specifically, for a query Q, we use M_c to make a prediction for each cluster. The top clusters are selected. For each selected cluster C, we use M_{nh} to make a prediction for each graph in C. The total number of predictions is reduced to $|\mathcal{C}|+\sum_{C\in\mathcal{C}'}|C|$, where $\mathcal{C}'$ denotes the selected clusters.

To cluster the graphs in $\mathcal{D}$, we can use existing graph embedding techniques (*e.g.*, [52]) to compute the embeddings of the graphs in $\mathcal{D}$ and then use KMeans for clustering.

VI. Cross-graph Learning Acceleration Based on GNN-graph Compression

The cross-graph learning using Definition 1 has many redundant computations. The reason is that many nodes in a graph have identical embedding (see Example 2). To eliminate the redundancy, in this section, we first propose a compressed GNN-graph (CG), where the nodes having the same embedding are grouped together. Then, we present the cross-graph learning method on two CGs. After that, we present the optimum CG construction method.

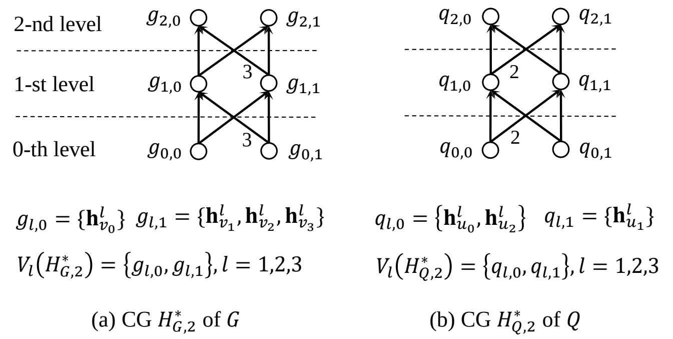

2-nd level
1-st level
0-th level
$g_{2,0}$
$g_{2,1}$
$g_{1,0}$
$g_{1,1}$
$g_{0,0}$
$g_{0,1}$
3
3
$g_{l,0}=\{\mathbf{h}^l_{v_0}\}$ $g_{l,1}=\{\mathbf{h}^l_{v_1},\mathbf{h}^l_{v_2},\mathbf{h}^l_{v_3}\}$
$V_l(H^*_{G,2})=\{g_{l,0},g_{l,1}\}, l=1,2,3$
(a) CG $H^*_{G,2}$ of G
$q_{2,0}$
$q_{2,1}$
$q_{1,0}$
$q_{1,1}$
$q_{0,0}$
$q_{0,1}$
2
2
$q_{l,0}=\{\mathbf{h}^l_{u_0},\mathbf{h}^l_{u_2}\}$ $q_{l,1}=\{\mathbf{h}^l_{u_1}\}$
$V_l(H^*_{Q,2})=\{q_{l,0},q_{l,1}\}, l=1,2,3$
(b) CG $H^*_{Q,2}$ of Q

Fig. 4: CGs of G and Q in Fig. 2 (edge weight = 1 is omitted for clarity)

A. Compressed GNN-graph

Definition 2. *Given an $L+1$-level GNN-graph $H_{G,L} = (V_0, V_1, ..., V_L, E)$ of a graph G, the compressed GNN-graph (CG) of G is an edge weighted $L+1$-level directed acyclic graph $H^*_{G,L} = (V_0, V_1, ..., V_L, E, w)$.*

- *For the l-th level, $l = 0, ..., L$, suppose the nodes in $V_l(H_{G,L})$ can be grouped into a set of groups, such that for any two nodes $\mathbf{h}^l_u$ and $\mathbf{h}^l_v$ in a group, the embedding vectors $\mathbf{h}^l_u$ and $\mathbf{h}^l_v$ are equal. $V_l(H^*_{G,L})$ has a node for each group. $|g|$ denotes the size of a group g.*[3]
- *For two nodes $g_{l-1,i} \in V_{l-1}(H^*_{G,L})$ and $g_{l,j} \in V_l(H^*_{G,L})$, $(g_{l-1,i}, g_{l,j}) \in E(H^*_{G,L})$ if $H_{G,L}$ has edges crossing $g_{l-1,i}$ and $g_{l,j}$.*
- *Let $\mathbf{h}^l_u$ be any node in the group $g_{l,j}$. The weight w of $(g_{l-1,i}, g_{l,j})$ is the number of incoming neighbors of $\mathbf{h}^l_u$ in the group $g_{l-1,i}$ in $H_{G,L}$.*

For presentation simplicity, in the following, we use g and q to denote the nodes in the CGs of G and Q, respectively.

Example 4. *Fig. 4(a) presents the CG $H^*_{G,2}$ of the GNN-graph $H_{G,2}$ in Fig. 2(c). Since the embedding vectors $\mathbf{h}^0_{v_1} = \mathbf{h}^0_{v_2} = \mathbf{h}^0_{v_3}$, the nodes of $V_0(H_{G,2})$ are grouped to two groups $g_{0,0} = \{\mathbf{h}^0_{v_0}\}$ and $g_{0,1} = \{\mathbf{h}^0_{v_1}, \mathbf{h}^0_{v_2}, \mathbf{h}^0_{v_3}\}$. Hence, $V_0(H^*_{G,2})$ has two nodes $g_{0,0}$ and $g_{0,1}$. Similarly, $V_1(H^*_{G,2})$ has two nodes $g_{1,0}$ and $g_{1,1}$, and $V_2(H^*_{G,2})$ has two nodes $g_{2,0}$ and $g_{2,1}$. The weighted edges are added based on Definition 2. For instance, $\mathbf{h}^1_{v_0}$ is in the group $g_{1,0}$. $\mathbf{h}^1_{v_0}$ has one and three incoming neighbors in the groups $g_{0,0}$ and $g_{0,1}$ in $H_{G,2}$, respectively. Hence, $w(g_{0,0}, g_{1,0}) = 1$ and $w(g_{0,1}, g_{1,0}) = 3$. For the GNN-graph $H_{Q,2}$ in Fig. 2(d), the CG $H^*_{Q,2}$ is shown in Fig. 4(b).*

B. Cross-graph learning on CGs

We define the cross-graph learning on the CGs of G and Q as follows.

Definition 3. *Given the CGs $H^*_{G,L}$ and $H^*_{Q,L}$ of G and Q, respectively, cross-graph learning computes the embedding*

[3]We slightly abuse the symbol g to denote a node in $H^*_{G,L}$ and a group of nodes in $H_{G,L}$.

Algorithm 5 CG construction

```
Input: graph $G$ and layer number $L$
Output: the CG $H^*_{G,L}$ of $G$
 1: initialize $H^*_{G,L} = (V_0, V_1, ..., V_L, E, w)$, $V_l = \emptyset$, $E = \emptyset$
 2: perform $L$ iterations of WL labeling
 3: for each $l = 0$ to $L$ do
 4:     group the nodes of $G$ by $wl^l$ and $V_l$ has a node for each group
 5: end for
 6: for each node $g_{l,j} \in V_l$ and each node $g_{l-1,i} \in V_{l-1}$ do
 7:     let $u$ be a node of $G$ in the group $g_{l,j}$
 8:     the weight of the edge $w(g_{l-1,i}, g_{l,j})$ = $|N_G(u) \cap g_{l-1,i}|$
 9:     $w(g_{l-1,i}, g_{l,j})$ += 1 if $u$ is also in the group $g_{l-1,i}$
10: end for
11: return $H^*_{G,L}$
```

$\mathbf{h}_{g_{l,i}}$ *of each node* $g_{l,i}$ *in the* l*-th level of* $H^*_{G,L}$, $l = 1, ..., L$, *as follows:*

$$\mathbf{h}_{g_{l,i}} = ReLU(\mathbf{W}^l(\mathbf{t}_{g_{l,i}} + \boldsymbol{\mu}_{g_{l,i}})), \tag{7}$$

$$\mathbf{t}_{g_{l,i}} = \sum_{g \in N^{in}_{H^*_{G,L}}(g_{l,i})} w(g, g_{l,i})\mathbf{h}_g, \tag{8}$$

$$\boldsymbol{\mu}_{g_{l,i}} = \sum_{q \in V_l(H^*_{Q,L})} |q|\alpha_{g_{l,i},q}\mathbf{h}_q, \text{ and} \tag{9}$$

$$\alpha_{g_{l,i},q} = \frac{exp(\mathbf{a} \cdot (\mathbf{t}_{g_{l,i}}||\mathbf{h}_q))}{\sum_{q' \in V_l(H^*_{G,L})} |q'| exp(\mathbf{a} \cdot (\mathbf{t}_{g_{l,i}}||\mathbf{h}_{q'}))}, \tag{10}$$

where N^{in} *denotes the incoming neighbors. For each node* $g_{0,i}$ *in the 0-th level of* $H^*_{G,L}$, $\mathbf{h}_{g_{0,i}} = \mathbf{h}^0_u$ *for any* $\mathbf{h}^0_u$ *in the group* $g_{0,i}$*. The embedding of* $H^*_{G,L}$ *is the weighted average of the embeddings of the nodes in the L-th level of* $H^*_{G,L}$*, i.e.,* $\mathbf{h}_{H^*_{G,L}} = (\sum_{g \in V_L(H^*_{G,L})} |g|\mathbf{h}_g)/(\sum_{g \in V_L(H^*_{G,L})} |g|)$*. The cross-graph embedding of G and Q is* $\mathbf{h}_{H^*_{G,L}}||\mathbf{h}_{H^*_{Q,L}}$.

Example 5. *Continue with Example 4,* $\mathbf{t}_{g_{1,0}} = \mathbf{h}_{g_{0,0}} + 3 \times \mathbf{h}_{g_{0,1}}$. $\boldsymbol{\mu}_{g_{1,0}} = 2 \times \alpha_{g_{1,0},q_{1,0}}\mathbf{h}_{q_{1,0}} + \alpha_{g_{1,0},q_{1,1}}\mathbf{h}_{q_{1,1}}$. $\mathbf{h}_{g_{1,0}} = ReLU(\mathbf{W}^1(\mathbf{t}_{g_{1,0}} + \boldsymbol{\mu}_{g_{1,0}}))$. $\mathbf{h}_{H^*_{G,L}} = (\mathbf{h}_{g_{2,0}} + 3 \times \mathbf{h}_{g_{2,1}})/4$. *Similarly,* $\mathbf{h}_{H^*_{Q,2}} = (2 \times \mathbf{h}_{q_{2,0}} + \mathbf{h}_{q_{2,1}})/3$. $\mathbf{h}_{H^*_{G,2}}||\mathbf{h}_{H^*_{Q,2}}$ *is the cross-graph embedding.*

Theorem 2. *Given two graphs G and Q,* $\mathbf{h}_{H^*_{G,L}}||\mathbf{h}_{H^*_{Q,L}}$ *computed by the cross-graph learning on the CGs* $H^*_{G,L}$ *and* $H^*_{Q,L}$ *(Definition 3) equals to* $\mathbf{h}_G||\mathbf{h}_Q$ *computed by the cross-graph learning on G and Q (Definition 1).*

Theorem 3. *The time complexity of cross-graph learning on two CGs* $H^*_{G,L}$ *and* $H^*_{Q,L}$ *is* $O((|V(H^*_{G,L})| + |V(H^*_{Q,L})|) + (|E(H^*_{G,L})| + |E(H^*_{Q,L})|) + \sum_{l=1}^{L} |V_l(H^*_{G,L})||V_l(H^*_{Q,L})|)$.

Corollary 1. *Given two graphs G and Q, the time complexity of the cross-graph learning on the CGs of G and Q (Definition 3) is no larger than that on G and Q (Definition 1).*

C. Construction of CG

In this subsection, we propose the algorithm to construct the optimum CG of G. Since the equivalence between the embedding of a node u at the l-th layer of `GIN` learning and the WL label of u at the l-th iteration of WL labeling (Sec. III-C), our idea is to use the WL label to group the nodes of G. The CG construction algorithm is shown in Algorithm 5. The time complexity of Algorithm 5 is $O(L(|V(G)| + |E(G)|))$.

Dataset	#graphs	avg $\lvert V\rvert$	avg $\lvert E\rvert$	#nlabel
AIDS	42,687	25.6	27.5	51
LINUX	47,239	35.5	37.7	36
PUBCHEM	22,794	48.2	50.8	10
SYN	1,000,000	10.1	15.9	5

TABLE I: Statistics of datasets

Theorem 4. *The CG constructed by Algorithm 5 is optimum to minimize the time complexity in Theorem 3.*

We remark that for the data graph G in $\mathcal{D}$, the CG of G can be precomputed. Although the CG of Q is computed on-the-fly, it is a one-off cost, as we need to perform cross-graph learning between Q and many data graphs in $\mathcal{D}$.

VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our proposed techniques.[4]

Datasets and query workload. We conduct the experiments on three real-world datasets that are widely used for graph similarity search [3], [9], [11], [15], [53]–[55]. AIDS is an antivirus screen compound graph dataset [3], [9], [15], [53]. LINUX is a set of control-flow graphs [9], [11], [54]. PUBCHEM is a set of chemical molecule graphs [15], [53], [55]. Following [3], [15], [53], [54], we also use a synthetic data SYN for scalability test. SYN is generated by the generator.[5] 20% of SYN is used by default, and 20%, 40%, 60%, 80%, and 100% of SYN are used in the scalability test. Table I shows the statistics of the datasets. Following [9], for each dataset, we sample 4,000 graphs as the query workload, which is split into training, validation and test data by 6:2:2.

For the ground-truth GED, we first compute the exact GED using the latest method [53]. If it does not finish in 10 seconds, following the approach of [9], [21], we use the best result of three approximate GED algorithms `VJ` [56], `Hung` [57], and `Beam` [58] as the ground-truth GED.

Metrics. The performance metrics follow the previous works [18], [19], [59]. Specifically, we use the recall at k ($recall@k$) to measure the search accuracy. $recall@k = |\mathcal{R} \cap \mathcal{R}'|/k$, where $\mathcal{R}$ is the result set of the k-ANN search algorithm and $\mathcal{R}'$ is the true result set. We use *queries per second* (QPS) to measure search efficiency. QPS is the number of queries finished in a second. We focus on search performance in high recall region.

Baseline methods. We compare our method `LAN` with `HNSW` [17], which is the latest method that supports k-ANN search in a metric space and `L2route` [28], which is the latest learning-based routing method on PG. Since `L2route` is designed for high-dimensional data, we first convert graphs into embedding vectors and then use `L2route` on the embedding vectors for k-ANN search. We use `LAN_IS` and `HNSW_IS` to denote the

[4]The source code is available at the project website https://github.com/csypeng/LAN

[5]https://www.cse.cuhk.edu.hk/~jcheng/graphgen1.0.zip

initial node selection methods of LAN and HNSW, respectively. Rand_IS denotes the method of randomly selecting a node as the initial node. We use LAN_Route and HNSW_Route to denote the routing methods of LAN and HNSW, respectively. For cross-graph learning acceleration, we compare with HAG [45], which is the latest method that can accelerate GNN learning without reducing accuracy.

Experimental settings. The experiments are conducted using PyTorch on a server with a Quad-Core AMD Opteron CPU, 800G RAM, and a GPU card NVIDIA Tesla V100S. The embedding dimensions are 128. We use the Adam optimizer. We set the initial learning rate to 0.005 and reduce it by 0.96 for every 5 epochs. We set the number of epochs to 1,000, and select the best model on validation data for testing. The neighborhood size parameter γ^* is set such that for 90% training queries, $\mathcal{N}_Q$ can contain the 200-NNs of Q and the batch size parameter y is 20. We focus on the performance for $k = 50$.

A. Comparison with existing methods

In this experiment, we compare the performance of LAN, HNSW, and L2route. Fig. 5 shows the results on AIDS, LINUX, PUBCHEM, and SYN.

From Fig. 5, we can observe that LAN significantly outperforms HNSW and L2route. In particular, at $recall$@50 = 0.95, the QPSs of LAN are ∼4.2x, ∼3.9x, ∼3.6x, and ∼9x higher than those of HNSW on AIDS, LINUX, PUBCHEM, and SYN, respectively. At $recall$@50 = 0.95, the QPSs of LAN are ∼16.1x, ∼18.6x, ∼17x, and ∼73.2x higher than those of L2route on AIDS, LINUX, PUBCHEM, and SYN, respectively.

From Fig. 5, we can also observe the margin of LAN decreases with the growth of recall. On one hand, it is reasonable as the higher recall we require, the fewer neighbors can be pruned. On the other hand, when $recall$@50 is as high as 0.98, the QPSs of LAN are still more than 3x higher than those of HNSW on AIDS, LINUX, and PUBCHEM, respectively, and more than 6x higher than that of HNSW on SYN.

B. Performance of routing with neighbor pruning

In this experiment, we compare the performances of LAN_Route and HNSW_Route. The initial node selection method of HNSW is used for both LAN_Route and HNSW_Route. The results are shown in Fig. 6.

From Fig. 6, we can observe that LAN_Route significantly outperforms HNSW_Route. In particular, at $recall$@50 = 0.95, the QPSs of LAN_Route are ∼2.9x, ∼2.6x, ∼2.5x, and ∼5.5x higher than those of HNSW_Route on AIDS, LINUX, PUBCHEM, and SYN, respectively. The margin of LAN_Route with respect to HNSW_Route reduces with the growth of recall. However, when $recall$@50 is as high as 0.98, the QPSs of LAN_Route are still more than 2.5x higher than those of HNSW_Route on AIDS, LINUX, and PUBCHEM, respectively, and more than 5x higher than that of HNSW_Route on SYN.

C. Performance of initial node selection

In this experiment, we compare the performances of LAN_IS, HNSW_IS, and Rand_IS. The results are presented in

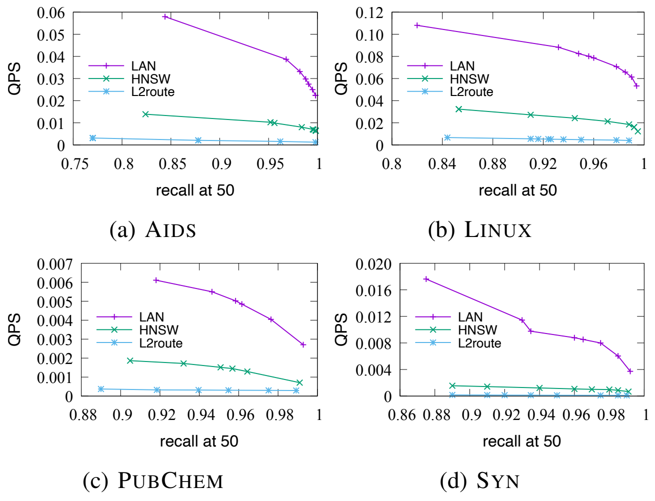


Fig. 5: Comparison with existing k-ANN search methods

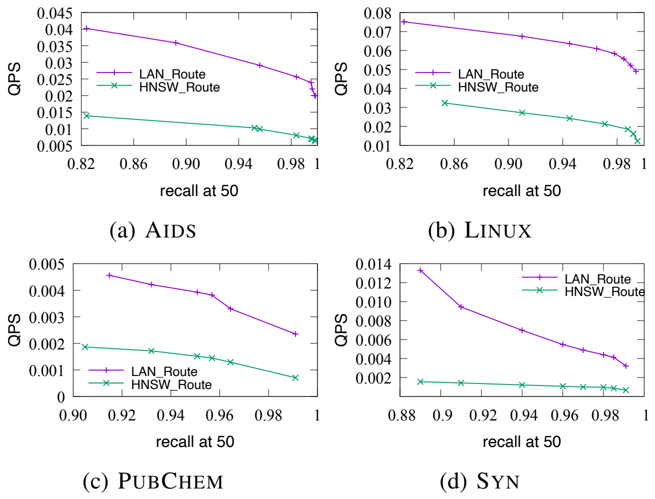
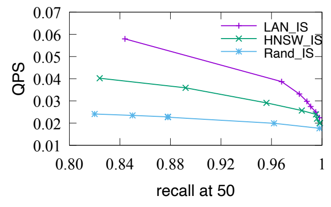


Fig. 6: Performance of routing with neighbor pruning (HNSW_IS is used for initial node selection)

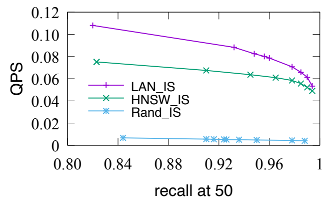
Fig. 7. LAN_IS uses the optimized design of M_{nh} (Sec. V-B2) as it is always faster than the basic design (Sec. V-B1).

Fig. 7 shows that LAN_IS outperforms HNSW_IS and Rand_IS. In particular, at $recall$@50 = 0.95, the QPSs of LAN_IS are 1.4x, 1.3x, 1.3x, and 1.7x higher than those of HNSW_IS on AIDS, LINUX, PUBCHEM, and SYN, respectively. At $recall$@50 = 0.95, the QPSs of LAN_IS are ∼2x, ∼17.2x, ∼1.5x, and ∼1.9x higher than those of Rand_IS on AIDS, LINUX, PUBCHEM, and SYN, respectively.

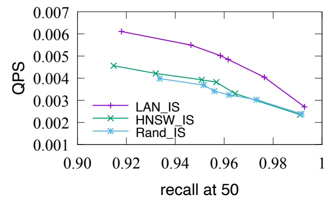
From Fig. 7, we can also observe that the margin of LAN_IS reduces with the growth of recall. However, at $recall$@50 is as high as 0.98, the QPSs of LAN_IS are more than 1x higher than those of HNSW_IS on AIDS, LINUX, and PUBCHEM, respectively, and more than 1.5x higher than that of HNSW_IS on SYN. At $recall$@50 = 0.98, the QPSs of LAN_IS are 1.5x, 15.1x, 1.3x, and 1.7x higher than those of Rand_IS on AIDS, LINUX, PUBCHEM, and SYN, respectively.



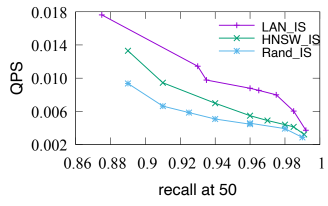
(a) AIDS



(b) LINUX



(c) PUBCHEM



(d) SYN

Fig. 7: Performance of initial node selection (`LAN_Route` is used for routing)

The precision of our initial node prediction model is shown in Fig. 8. From Fig. 8, we can observe that the precisions exceed 0.7 on AIDS, LINUX, PUBCHEM, and SYN, respectively. Since $1-(1-0.7)^4 > 0.99$, we only need to randomly sampling 4 nodes from $\hat{\mathcal{N}}_Q$, such that the probability that at least one sample is in $\mathcal{N}_Q$ is larger than 0.99.

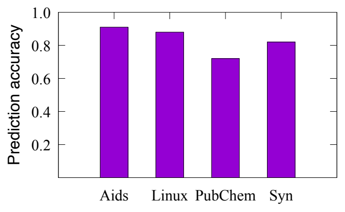


Fig. 8: Accuracy of initial node prediction

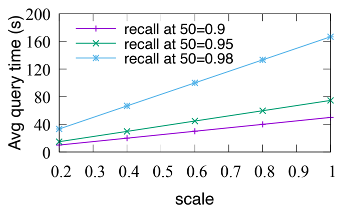
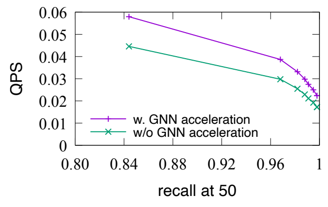


Fig. 9: Performance of scalability on SYN

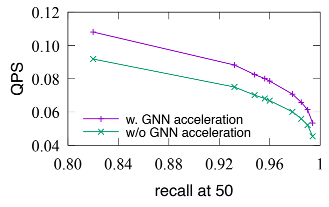
D. Scalability evaluation

In this experiment, we evaluate the scalability of `LAN`. We focus on the SYN dataset. Fig. 9 presents the results. In Fig. 9, the x-axis is the scale of the dataset and the y-axis is the average running time of a query.

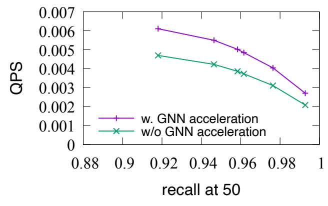
From Fig. 9, we can observe that `LAN` scales linearly with the size of the dataset. The reason is that following the approach of [18], [19], [59] to support large dataset, we randomly split the dataset into equal-size sub-datasets and sequentially perform k-ANN search on each sub-dataset. From Fig. 9, we can also observe that the gap between $recall@50$ = 0.9 and $recall@50$ = 0.95 is much larger than the gap between $recall@50$ = 0.95 and $recall@50$ = 0.98. It is consistent with the observations in previous experiments that the increase of running time is more than a linear function of the increase of recall.



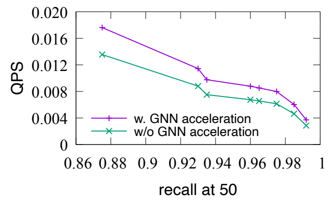
(a) AIDS



(b) LINUX



(c) PUBCHEM



(d) SYN

Fig. 10: Performance of cross-graph learning acceleration

E. Performance of cross-graph learning acceleration

Fig. 10 presents the effects of cross-graph learning acceleration on the efficiency of k-ANN search. From Fig. 10, we can observe that the QPS is increased after using our cross-graph learning acceleration technique. In particular, at $recall@50$ = 0.95, the QPSs are increased by ∼17%, ∼18%, ∼15%, and ∼17% on AIDS, LINUX, PUBCHEM, and SYN, respectively.

Fig. 11 shows the breakdown of the query time before using our cross-graph learning acceleration technique. From Fig. 11, we can observe that cross-graph learning accounts for ∼24%, ∼25%, ∼20%, and ∼29% of query time on AIDS, LINUX, PUBCHEM, and SYN, respectively.

Fig. 12 shows the speedup of cross-graph learning using our acceleration technique. We can observe that cross-graph learning is speeded up by ∼4x, ∼4.2x, ∼5.3x, and ∼3.1x on AIDS, LINUX, PUBCHEM, and SYN, respectively. In contrast, `HAG` cannot speedup the cross-graph learning.

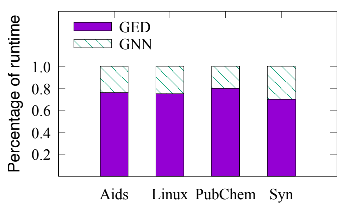


Fig. 11: Breakdown of k-ANN search time

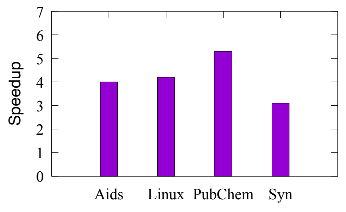


Fig. 12: Speedup of cross-graph learning

VIII. CONCLUSION

In this paper, we propose a learning-based k-ANN search method in graph databases. The core is using graph learning models to avoid unnecessary distance computations during the routing on PG. First, we propose an algorithm of routing with neighbor pruning. At each routing step, a graph learning model is used to rank the neighbors and only the top neighbors are explored. Second, we propose a graph learning model for initial node selection, such that the initial node selected has a high probability of being in the neighborhood of Q. Third, we propose a compressed GNN-graph to accelerate the cross-graph learning used in the neighbor ranking and initial node selection. The efficiency of learning is improved without degrading the accuracy. Our experiments show that our method is effective and significantly outperforms the state-of-the-art k-ANN search methods on real-world benchmark graph datasets.

In the future, we plan to study distributed k-ANN search method for supporting larger graph databases.

IX. Appendix

This appendix contains the proofs of the theorems and lemmas. Some detailed derivations are available on the project website.

A. *Proof of Lemma 1*

To prove Lemma 1, we first prove the following two Lemmas.

Lemma 3. *For any routing step of* `np_route`*, let S denote the set of explored nodes so far and $N_{\mathcal{G}}(S)$ denote the neighbors of the nodes in S in $\mathcal{G}$. The node G explored at this step must be the node that is the closest to Q among all unexplored nodes in $N_{\mathcal{G}}(S)$.*

Proof. It is trivial for the first stage of routing, *i.e.*, the routing steps from the initial node to the first local optimal (Lines 1-12 of Algorithm 4). The reason is that at each routing step, let G be the current node, all neighbors of G whose distances are no more than $d(Q, G)$ are added to W, and the node in W that is the closest to Q is explored at the next routing step.

For the second stage, *i.e.*, the routing steps from the first local optimal to the stop of the routing, the while-loop (Lines 22-27) performs the routing steps. Suppose the distance threshold for current routing step is γ (Line 22). Let $N^{ue}_{\mathcal{G}}(S, \gamma)$ denote the unexplored nodes in $N_{\mathcal{G}}(S)$ whose distances to Q are no more than γ. Lines 15-17 make sure that all nodes in $N^{ue}_{\mathcal{G}}(S, \gamma)$ are added to W. Line 18 may squeeze some nodes in $N^{ue}_{\mathcal{G}}(S, \gamma)$ out of W. If the node G in $N^{ue}_{\mathcal{G}}(S, \gamma)$ that is the closest to Q is squeezed out, the routing stops. Otherwise, G must be the first to be explored in the while-loop (Lines 22-27). □

Lemma 4. *For any routing step of* `baseline`*, let S denote the set of explored nodes so far and $N_{\mathcal{G}}(S)$ denote the neighbors of the nodes in S in $\mathcal{G}$. The node G explored at this step must be the node that is the closest to Q among all unexplored nodes in $N_{\mathcal{G}}(S)$.*

Proof. It is trivial as at each routing step, let G be the current node, all neighbors of G are added to W and the node in W that is the closest to Q is explored at the next routing step. □

Proof sketch of Lemma 1 is as follows.

Proof. Let seq^i_{bs} and seq^i_{np} denote the sequences of explored nodes of `baseline` and `np_route` from the 0-th step to the i-th step, respectively. We prove this lemma by induction.

For the 0-th step, it is trivial as $seq^0_{bs} = seq^0_{np} = [G_{init}]$.

Suppose $seq^{i-1}_{bs} = seq^{i-1}_{np}$. Due to Lemma 3 and Lemma 4, the nodes explored by `baseline` and `np_route` at the i-th step must be the same. Therefore, $seq^i_{bs} = seq^i_{np}$. □

Detailed proof of Lemma 1 is as follows.

Proof. It is trivial to prove that this Lemma holds in the first stage of routing, *i.e.*, the routing steps from the initial node to the first local optimal. The reason is that at each routing step, both `np_route` and `baseline` route to the best neighbor of the current node of the router. In the following, we prove that this Lemma holds in the second stage of routing, *i.e.*, the routing steps from the first local optimal G_{flo} to the stop of the routing.

Since the routing has backtracking in the second stage, the key is to prove that the backtrackings of `np_route` and `baseline` are the same.

We first define some terms. A backtracking is called as a *record-breaking backtracking* (rbb) if the backtracking is to explore a node whose GED is larger than that of the previous rbb. The 0-th rbb is the first backtracking after the first local optimal. A rbb may have more than one while-loops (Lines 14-29). A *superstep* of routing is the routing steps between two consecutive rbbs. The i-th superstep is the routing steps between the i-th rbb (included) and the $(i + 1)$-th rbb (excluded). For presentation simplicity, we use W_{np} to denote the W of `np_route` and W_{bs} to denote the W of `baseline`.

We prove the lemma by the following steps. i) We prove the sequence of nodes explored until the first local optimal is the same. ii) We prove the first backtracking is the same. iii) We prove the sequence of nodes explored in every superstep is the same. iv) We prove the number of supersteps is the same.

i) Let $seq_{flo,np}$ and $seq_{flo,bs}$ denote the sequences of nodes explored until the first local optimal by `np_route` and `baseline`, respectively. $seq_{flo,np} = seq_{flo,bs}$ as there is no backtracking until the first local optimal.

ii) The first backtracking of `np_route` and that of `baseline` are the same. Suppose the first backtracking of `baseline` explores G. We prove G must be the first backtracking of `np_route` by the following cases.

ii.a If $d(G, Q) < d(G_{flo}, Q) + d_s$,

ii.a.1 If G is in W_{np}, G must be the unexplored node in W_{np} having the smallest d. It can be proved by contradiction. Suppose G' is the unexplored node in W_{np} s.t. $d(G', Q) < d(G, Q)$. G' must be a neighbor of some node in $seq_{flo,np}$. Since $seq_{flo,np} = seq_{flo,bs}$, G' is also a neighbor of some node in $seq_{flo,bs}$. Since G is in W_{bs} and $d(G', Q) < d(G, Q)$, G' must be in W_{bs}. Therefore, `baseline` should explore G' before G. It is a contradiction. Hence, G is the unexplored node in W_{np} having the smallest d. G must be the first explored node in the while-loop (Lines 22-27) of `np_route`.

ii.a.2 If G is not in W_{np}, the for-loop (Lines 15-17) must add G into W_{np}. The for-loop (Lines 15-17) may also add other nodes into W_{np}. But, G must be the node in W_{np} having the smallest d, since otherwise, G is not the first backtracking of `baseline`. Since G is the node in W_{np} having the smallest d, G must be the first explored node in the while-loop (Lines 22-27) of `np_route`.

ii.b If $d(G,Q) \geq d(G_{flo},Q) + d_s$, there must be no unexplored node G' in W_{np} that is closer to Q than G, since otherwise, `baseline` will explore G' before G. The while-loop (Lines 14-29) will increase γ. While $\gamma < d(G,Q)$, no node will be added to W_{np} by the for-loop (Lines 15-17). It can be proved by contradiction. Suppose G' is added to W_{np} before $\gamma \geq d(G,Q)$. It means that $d(G',Q) < d(G,Q)$. `baseline` should explore G' before G, which is a contradiction. Once $\gamma \geq d(G,Q)$, G must be in W_{np} and the while-loop (Lines 22-27) will explore G.

iii) We prove the sequence of nodes explored in every superstep is the same by induction.

iii.a We prove the sequence of nodes explored in the 0-th superstep is the same. The prove is also by induction. Let $seq_{np}^0 = [G_{np,0}^0, G_{np,1}^0, ..., G_{np,x_0}^0]$ and $seq_{bs}^0 = [G_{bs,0}^0, G_{bs,1}^0, ..., G_{bs,y_0}^0]$ be the sequences of nodes explored in the 0-th superstep of `np_route` and `baseline`, respectively.

iii.a.1 $G_{np,0}^0 = G_{bs,0}^0$ as they are the first backtrackings of `np_route` and `baseline`, respectively, and we have already proved they are equal in ii).

iii.a.2 Suppose $[G_{np,0}^0, G_{np,1}^0, ..., G_{np,j-1}^0] = [G_{bs,0}^0, G_{bs,1}^0, ..., G_{bs,j-1}^0]$. We prove $G_{np,j}^0 = G_{bs,j}^0$. $G_{np,j}^0$ is the unexplored node having the smallest d in all the neighbors of all nodes in $[seq_{flo,np}, G_{np,0}^0, G_{np,1}^0, ..., G_{np,j-1}^0]$. $G_{bs,j}^0$ is the unexplored node having the smallest d in all the neighbors of all nodes in $[seq_{flo,bs}, G_{bs,0}^0, G_{bs,1}^0, ..., G_{bs,j-1}^0]$. Since $seq_{flo,np} = seq_{flo,bs}$, $G_{np,j}^0 = G_{bs,j}^0$.

iii.a.3 $x_0 = y_0$ can be proved by contradiction. Suppose $x_0 \neq y_0$. If $x_0 < y_0$, `np_route` should explore G_{bs,x_0+1}^0 after G_{np,x_0}^0, as $d(G_{bs,x_0+1}^0, Q) < d(G_{np,0}^1, Q)$, where $G_{np,0}^1$ is the 0-th node explored in the 1-st rbb, *i.e.*, `np_route` should not finish the 0-th rbb after examining G_{np,x_0}^0. Similarly, if $x_0 > y_0$, `baseline` should explore G_{np,y_0+1}^0 after G_{bs,y_0}^0. Hence, $x_0 = y_0$.

iii.b Suppose $[seq_{flo,np}, seq_{np}^0, seq_{np}^1, ..., seq_{np}^{i-1}] = [seq_{flo,bs}, seq_{bs}^0, seq_{bs}^1, ..., seq_{bs}^{i-1}]$. We prove $seq_{np}^i = seq_{bs}^i$. Let $seq_{np}^i = [G_{np,0}^i, G_{np,1}^i, ..., G_{np,x_i}^i]$ and $seq_{bs}^i = [G_{bs,0}^i, G_{bs,1}^i, ..., G_{bs,y_i}^i]$. We prove $seq_{np}^i = seq_{bs}^i$ by induction.

iii.b.1 $G_{np,0}^i = G_{bs,0}^i$ can be proved by the same logic of ii).

iii.b.2 Suppose $G_{np,j-1}^i = G_{bs,j-1}^i$. We can prove $G_{np,j}^i = G_{bs,j}^i$ by the same logic of iii.a.2).

iii.b.3 $x_i = y_i$ can be proved by the same logic of iii.a.3).

iv) We prove `np_route` and `baseline` have the same number of supersteps. Let x and y denote the numbers of supersteps of `np_route` and `baseline`, respectively. We prove $x = y$ by contradiction. Without loss of generality, we assume $x < y$. It means that `np_route` stops after the $x-1$-th superstep. Let $G_{bs,0}^x$ be the first node explored in the x-th superstep of `baseline`. $G_{bs,0}^x$ must be is in W_{bs} after the x supersteps of `baseline`.

- If $G_{bs,0}^x$ is in W_{np} after the x supersteps of `np_route`, the while-loop (Lines 22-27) will explore $G_{bs,0}^x$ after γ is increased to be no less than $d(G_{bs,0}^x, Q)$.
- If $G_{bs,0}^x$ is not in W_{np} after the x supersteps of `np_route`, the for-loop (Lines 15-17) will add $G_{bs,0}^x$ into W_{np} after γ is increased to be no less than $d(G_{bs,0}^x, Q)$ and the while-loop (Lines 22-27) will explore $G_{bs,0}^x$.

Therefore, `np_route` should not stop after the $x-1$-th superstep, which is a contradiction. Therefore, $x = y$.

Finally, we conclude that the sequence of nodes explored by `np_route` is the same with `baseline`. □

B. Proof of Theorem 1

Proof. Since the sequences of nodes explored by `np_route` and `baseline` are the same, the search results $\mathcal{R}_{np}$ returned by `np_route` must be the same with the search result $\mathcal{R}_{bs}$ of `baseline`.

Since `np_route` prunes some neighbors, the number of distance computations must be no more than that of `baseline`.

The space cost of `baseline` is the linear to $|S| + |N_{\mathcal{G}}(S)|$, where S denotes the explored nodes and $N_{\mathcal{G}}(S)$ denotes the neighbors of the nodes S in $\mathcal{G}$. Since `np_route` needs to record the batches, the space cost of `np_route` is higher than that of `baseline`. Extremely, the size of each batch is 1, where space cost of `np_route` is $|S| + 2 \times |N_{\mathcal{G}}(S)|$, as the explored nodes of `np_route` are the same with that of `baseline` (see Lemma 1). Therefore, the space cost of `np_route` is at most two times of that of `baseline`. □

C. Proof of Lemma 2

Proof. If we randomly sample a graph G in $\mathcal{N}_Q$, G is in $\mathcal{N}_Q$ with probability p. Since the s samples are sampled independently, the probability that all the s samples are not in $\mathcal{N}_Q$ is $(1-p)^s$. Therefore, the probability that at least a sample is in $\mathcal{N}_Q$ is $1-(1-p)^s$. □

D. Proof of Theorem 2

Proof. To prove $\mathbf{h}_{H_{G,L}^*} || \mathbf{h}_{H_{Q,L}^*} = \mathbf{h}_G || \mathbf{h}_Q$, we only need to prove $\mathbf{h}_{H_{G,L}^*} = \mathbf{h}_G$ and $\mathbf{h}_{H_{Q,L}^*} = \mathbf{h}_Q$. In the following, we only prove $\mathbf{h}_{H_{G,L}^*} = \mathbf{h}_G$ as $\mathbf{h}_{H_{Q,L}^*} = \mathbf{h}_Q$ can be proved in the same way.

To prove $\mathbf{h}_{H_{G,L}^*} = \mathbf{h}_G$, the key is to prove that in each layer of cross-graph learning, if a node is in a group, the embedding of the node computed by the cross-graph learning on G and Q equals to the embedding of the group computed by the cross-graph learning on the CG of G and the CG of Q.

In the l-th layer cross-graph learning on G and Q, for a node u of G, u needs to i) compute $\mathbf{t}_u^l$, *i.e.*, aggregate embeddings from its neighbors in G, ii) compute $\boldsymbol{\mu}_u^l$, *i.e.*, aggregate weighted embeddings from all nodes in Q, and iii) multiply $\mathbf{W}^l$.

In the l-th layer cross-graph learning on the CG of G and the CG of Q, a node g at the l-th level of the CG of G needs to i) compute $\mathbf{t}_g^l$, *i.e.*, aggregate weighted embeddings from its incoming neighbors in the $l-1$-th level of the CG of G, ii) compute $\boldsymbol{\mu}_g^l$, *i.e.*, aggregate weighted embeddings from all nodes at the l-the level of the CG of Q, and iii) multiply $\mathbf{W}^l$.

By Definition 3, if the node u is in the group g, $\mathbf{t}_u^l = \mathbf{t}_g^l$ and $\boldsymbol{\mu}_u^l = \boldsymbol{\mu}_g^l$. Therefore, $\mathbf{h}_u^l = \mathbf{h}_g$.

In the L-th layer, since $\mathbf{h}_G = mean_{u \in G}\mathbf{h}_u^L$, $\mathbf{h}_{H^*_{G,L}}$ is the weighted average of the embeddings of the nodes in the L-th level of $H^*_{G,L}$, where the weight is group size, and each node u in G must belong to a node in the L-th level of $H^*_{G,L}$, $\mathbf{h}_G = \mathbf{h}_{H^*_{G,L}}$ must hold. □

E. Proof of Theorem 3

Proof. The cross-graph learning at the l-th level of $H^*_{G,L}$, $l = 1, ..., L$, has three steps: i) collect information from the incoming neighbors in the $l-1$-th level of $H^*_{G,L}$, ii) compute attention for the nodes in the l-th level of $H^*_{Q,L}$, and iii) multiply $\mathbf{W}$. It is the same for the l-th level of $H^*_{Q,L}$.

For i), the time cost is linear to the number of incoming edges of the l-th level. The total time for L levels is $O(|E(H^*_{G,L})| + |E(H^*_{Q,L})|)$.

For ii), the time cost is linear to the number of pairs of the nodes in the l-th level of $H^*_{G,L}$ and the nodes in the l-th level of $H^*_{Q,L}$. The total time for L levels is $O(\sum_{l=1}^{L} |V_l(H^*_{G,L})||V_l(H^*_{Q,L})|)$. Note that we do not need to compute the attention for the 0-th level.

For iii), the time cost is linear to the number of nodes. Hence, the total time is $O(|V(H^*_{Q,L})| + |V(H^*_{G,L})|)$. □

F. Proof of Corollary 1

Proof. It is trivial as for a graph G, the number of edges between two levels of the CG of G is no larger than $E(G)$ and the number of nodes in each level of the CG of G is no larger than $V(G)$. □

G. Proof of Theorem 4

Proof. The key argument is that the WL label of a node is equivalent to the embedding of the node. If the nodes have been grouped by their WL labels, it is impossible to further reduce the number of groups while making sure that all nodes in a group always have the same embedding. □

References

[1] A. A. Collado, R. Carrasco-Velar, N. García-Pedrajas, and G. Cerruela-García, "Maximum common property: A new approach for molecular similarity," *Journal of Cheminformatics*, vol. 12, no. 61, 2020.

[2] H. Du, "Chemical molecules search based on graph similarity measure," in *Proc. of the International Conference on Biomedical Engineering and Computer Science*, 2010, pp. 1–4.

[3] X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang, "Efficient structure similarity searches: a partition-based approach," *The VLDB Journal*, vol. 27, no. 1, pp. 53–78, 2018.

[4] C. Garcia-Hernandez, A. Fernández, and F. Serratosa, "Ligand-based virtual screening using graph edit distance as molecular similarity measure," *Journal of Chemical Information and Modeling*, vol. 59, no. 4, pp. 1410–1421, 2019.

[5] M. Naderi, C. Alvin, Y. Ding, S. Mukhopadhyay, and M. Brylinski, "A graph-based approach to construct target-focused libraries for virtual screening," *Journal of cheminformatics*, vol. 8, no. 1, pp. 1–16, 2016.

[6] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng, "Answering top-k graph similarity queries in graph databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1459–1474, 2019.

[7] R. Wang, T. Zhang, T. Yu, J. Yan, and X. Yang, "Combinatorial learning of graph edit distance via dynamic embedding," in *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 5241–5250.

[8] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proc. of the International Conference on Pattern Recognition Applications and Methods*, 2015, pp. 271–278.

[9] Z. Qin, Y. Bai, and Y. Sun, "GHashing: Semantic graph hashing for approximate similarity search in graph databases," in *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2062–2072.

[10] S. Cesare and Y. Xiang, "Malware variant detection using similarity search over sets of control flow graphs," in *Proc. of the International Conference on Trust, Security and Privacy in Computing and Communications*, 2011, pp. 181–189.

[11] X. Wang, X. Ding, A. K. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *Proc. of the IEEE International Conference on Data Engineering*, 2012, pp. 210–221.

[12] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng, "Finding top-k similar graphs in graph databases," in *Proc. of the International Conference on Extending Database Technology*, 2012, pp. 456–467.

[13] Z. Abu-Aisheh, R. Raveaux, and J.-Y. Ramel, "Efficient k-nearest neighbors search in graph space," *Pattern Recognition Letters*, vol. 134, pp. 77–86, 2020.

[14] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 25–36, 2009.

[15] J. Kim, "Boosting graph similarity search through pre-computation," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2021, pp. 951–963.

[16] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2539–2554.

[17] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[18] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," vol. 12, no. 5, pp. 461–474, 2019.

[19] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021.

[20] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2020.

[21] L. Yang and L. Zou, "Noah: Neural-optimized A* search algorithm for graph edit distance computation," in *Proc. of IEEE International Conference on Data Engineering (ICDE)*, 2021.

[22] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc. of International Conference on Machine Learning (ICML)*, 2019, pp. 3835–3845.

[23] B. Harwood and T. Drummond, "Fanng: Fast approximate nearest neighbour graphs," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 5713–5722.

[24] L. C. Shimomura, R. S. Oyamada, M. R. Vieira, and D. S. Kaster, "A survey on graph-based methods for similarity searches in metric spaces," *Information Systems*, vol. 95, p. 101507, 2021.

[25] J. Qin, W. Wang, C. Xiao, and Y. Zhang, "Similarity query processing for high-dimensional data," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3437–3440, 2020.

[26] L. Prokhorenkova and A. Shekhovtsov, "Graph-based nearest neighbor search: From practice to theory," in *Proc. of the International Conference on Machine Learning*, vol. 119, 2020, pp. 7803–7813.

[27] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, 2020, pp. 1033–1044.

[28] D. Baranchuk, D. Persiyanov, A. Sinitsin, and A. Babenko, "Learning to route in similarity graphs," in *Proc. of the International Conference on Machine Learning*, 2019, pp. 475–484.

[29] P. Lin and W.-L. Zhao, "A comparative study on hierarchical navigable small world graphs," *ArXiv*, vol. abs/1904.02077, 2019.

[30] J. Vargas Muñoz, M. A. Gonçalves, Z. Dias, and R. da S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition (PR)*, vol. 96, p. 106970, 2019.

[31] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[32] K. Aoyama, A. Ogawa, T. Hattori, and T. Hori, "Double-layer neighborhood graph based similarity search for fast query-by-example spoken term detection," in *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 5216–5220.

[33] K. Lu, H. Wang, W. Wang, and M. Kudo, "VHP: Approximate nearest neighbor search via virtual hypersphere partitioning," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1443–1455, 2020.

[34] L. Prokhorenkova and A. Shekhovtsov, "Graph-based nearest neighbor search: From practice to theory," in *Proc. of the International Conference on Machine Learning*, 2020, pp. 7803–7813.

[35] H. Wang, Z. Wang, W. Wang, Y. Xiao, Z. Zhao, and K. Yang, "A note on graph-based nearest neighbor search," 2020.

[36] Y. Bai, H. Ding, K. Gu, Y. Sun, and W. Wang, "Learning-based efficient graph similarity computation via multi-scale convolutional set matching." in *Proc. of AAAI Conference on Artificial Intelligence (AAAI)*, 2020, pp. 3219–3226.

[37] Y. Peng, B. Choi, and J. Xu, "Graph edit distance learning via modeling optimum matchings with constraints," in *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2021, pp. 1534–1540.

[38] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. of the International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[39] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *Proc. of the International Conference on Learning Representations*, 2018.

[40] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Proc. of the International Conference on Neural Information Processing Systems*, 2018, pp. 4563–4572.

[41] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *Proc. of the International Conference on Machine Learning*, vol. 97, 2019, pp. 6861–6871.

[42] Y. Ye and S. Ji, "Sparse graph attention networks," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021.

[43] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," in *Proc. of the International Conference on Learning Representations*, 2021.

[44] W. Wu, B. Li, C. Luo, and W. Nejdl, "Hashing-accelerated graph neural networks for link prediction," in *Proceedings of the Web Conference 2021*, 2021, pp. 2910–2920.

[45] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, "Redundancy-free computation for graph neural networks," in *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 997–1005.

[46] H. Liu, S. Lu, X. Chen, and B. He, "G3: when graph neural networks meet parallel graph processing systems on gpus," *Proc. of the VLDB Endowment*, vol. 13, no. 12, pp. 2813–2816, 2020.

[47] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie, "fuseGNN: Accelerating graph convolutional neural network training on gpgpu," in *Proc. of the IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[48] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, no. 8, pp. 689–694, 1997.

[49] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proc. of the International Conference on Learning Representations, ICLR*, 2019.

[50] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, "Practical lessons from predicting clicks on ads at facebook," in *Proc. of the International Workshop on Data Mining for Online Advertising*, 2014, pp. 1–9.

[51] C.-Y. Chiu, A. Prayoonwong, and Y.-C. Liao, "Learning to index for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 8, pp. 1942–1956, 2020.

[52] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 855–864.

[53] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, and D. Ouyang, "Speeding up ged verification for graph similarity search," in *Proc. of IEEE International Conference on Data Engineering (ICDE)*, 2020, pp. 793–804.

[54] K. Gouda and M. Hassaan, "CSI_GED: An efficient approach for graph edit similarity computation," in *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, 2016, pp. 265–276.

[55] Y. Peng, Z. Fan, B. Choi, J. Xu, and S. S. Bhowmick, "Authenticated subgraph similarity searchin outsourced graph databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1838–1860, 2015.

[56] S. Fankhauser, K. Riesen, and H. Bunke, "Speeding up graph edit distance computation through fast bipartite matching," in *Proc. of International Workshop on Graph-Based Representations in Pattern Recognition*, 2011, pp. 102–111.

[57] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image and Vision computing*, vol. 27, no. 7, pp. 950–959, 2009.

[58] M. Neuhaus, K. Riesen, and H. Bunke, "Fast suboptimal algorithms for the computation of graph edit distance," in *Structural, Syntactic, and Statistical Pattern Recognition*, 2006, pp. 163–172.

[59] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 1964–1978, 2021.