TaGSim: Type-aware Graph Similarity Learning and Computation

Jiyang Bai
Florida State University
Tallahassee, Florida, USA
bai@cs.fsu.edu

Peixiang Zhao
Florida State University
Tallahassee, Florida, USA
zhao@cs.fsu.edu

ABSTRACT

Computing similarity between graphs is a fundamental and critical problem in graph-based applications, and one of the most commonly used graph similarity measures is graph edit distance (GED), defined as the minimum number of graph edit operations that transform one graph to another. Existing GED solutions suffer from severe performance issues due in particular to the NP-hardness of exact GED computation. Recently, deep learning has shown early promise for GED approximation with high accuracy and low computational cost. However, existing methods treat GED as a global, coarse-grained graph similarity value, while neglecting the type-specific transformative impacts incurred by different types of graph edit operations, including node insertion/deletion, node relabeling, edge insertion/deletion, and edge relabeling. In this paper, we propose a *type-aware* graph similarity learning and computation framework, TaGSim (**T**ype-**a**ware **G**raph **Sim**ilarity), that estimates GED in a fine-grained approach *w.r.t.* different graph edit types. Specifically, for each type of graph edit operations, TaGSim models its unique transformative impacts upon graphs, and encodes them into high-quality, type-aware graph embeddings, which are further fed into type-aware neural networks for accurate GED estimation. Extensive experiments on five real-world datasets demonstrate the effectiveness and efficiency of TaGSim, which significantly outperforms state-of-the-art GED solutions.

PVLDB Reference Format:
Jiyang Bai and Peixiang Zhao. TaGSim: Type-aware Graph Similarity Learning and Computation. PVLDB, 15(2): 335 - 347, 2022.
doi:10.14778/3489496.3489513

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jiyangbai/TaGSim.

1 INTRODUCTION

In the modern network era, graphs have been ubiquitous in numerous high-impact areas, including social media, bioinformatics, artificial intelligence, and critical infrastructures [15]. At the core of myriad graph-based applications lies a common and fundamental problem of graph similarity computation, which demands efficient discovery of highly similar graphs. Specifically, graph edit distance (GED) defined by a minimum number of graph edit operations between a pair of graphs has by far been the most well-studied graph similarity metric thanks to its generality and broad applicability [3, 10, 35]. However, the exact GED computation has proven to be NP-hard [8], and simply intractable even for small graphs in practice [5].

Existing solutions for GED computation have focused primarily on pruning unpromising search spaces, or filtering dissimilar graph pairs based on GED lower bounds, before the exact GED verification is triggered [10, 22, 26, 51]. These methods, however, suffer from severe performance vulnerabilities especially in large graph datasets: (1) Existing GED lower bounds demonstrate limited filtering capabilities, and a large number of dissimilar graph pairs fail to be identified and filtered, thus rendering a large amount of fruitless computation; (2) The algorithms that evaluate such GED lower-bounds tend to incur high-order polynomial, or even exponential, computational costs, thus imposing another performance barrier for GED computation [3, 4, 26, 35, 36, 51].

With the great success of deep learning techniques, some recent works have treated GED computation as a learning problem, and applied graph neural networks (GNN) for GED approximation [2, 11, 30]. Specifically, neural networks are trained to map a pair of graphs to an approximate GED score with an objective to minimize the discrepancy between the estimated GED and the ground-truth, exact GED. Such learning-based methods have achieved higher GED estimation accuracy than the traditional, search-based methods and GED lower-bound based solutions [2]. In addition, once the GNN models are trained *a priori*, they can enable efficient online GED computation. Although the existing GNN-based solutions have shown early promise for GED learning and computation, the following technical limitations directly impair their usability and performance especially for large graphs:

(1) **Low Graph Embedding Quality.** Existing approaches embed each node of a graph into a vector *w.r.t.* node features (*e.g.*, node labels), and further aggregate node embeddings to a global graph embedding vector, which, unfortunately, maintains rather limited structure and label information pertaining to GED. This general-purpose, *GED-agnostic* embedding mechanism is typically not optimized for graph similarity computation, thus inevitably causing inaccurate GED estimation results;
(2) **Type-oblivious GED Learning.** Existing approaches model GED as a global graph similarity score for graphs. The fact is that, GED is collectively determined by graph edit operations of six different types, including *node insertion/deletion*, *edge insertion/deletion*, and *node/edge relabeling*, each of which has disparate transformative impacts on graphs (*e.g.*, changing node/edge information, or altering graph connectivity), and thus should be modeled and learned individually. The existing *type-oblivious* approaches often lead to coarse-grained, inaccurate GED estimation;

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489513

(3) **No Edge Label Support.** Existing GNN-based approaches are confined to graphs without edge labels [2, 30], and thus cannot support edge relabeling operations, a critical constituent in the GED definition. Additionally, their graph embedding mechanisms fail to incorporate edge labels as well, rendering incomplete GED modelling and inaccurate GED approximation in practice;
(4) **High Training Cost.** Training GNN-based models requires a large number of graph pairs with their ground-truth GEDs as the training data, which are extremely expensive to obtain due to the NP-hardness of the exact GED computation. For the large graph datasets, this expensive training cost is simply unacceptable.

To systemically address these limitations, we propose in this paper a new concept of *type-aware* GED learning and computation for node/edge-labeled graphs: Instead of modelling GED as a global graph similarity score, we investigate for each type of graph edit operations its unique transformative impacts on graphs, and design a fine-grained, type-aware embedding and GED learning framework, TaGSim (**T**ype-**a**ware **G**raph **Sim**ilarity Learning and Computation) for accurate and efficient GED computation. In the first stage of TaGSim, we extract within the localized neighborhood of nodes/edges of graphs the salient information pertaining to the transformative impacts incurred by each graph edit type, including the edge relabeling operations that are omitted in existing GNN-based solutions, and encode them into *type-aware* graph embeddings. In the second stage of TaGSim, we feed the type-aware graph embeddings to the corresponding neural tensor networks (NTNs) and fully connected neural networks (NNs) in order to learn and estimate individually the number of operations for each graph edit type. After the number of each constituent edit type in GED is estimated, we aggregate them as the final estimated GED score. In sum, TaGSim enables type-aware GED learning and computation on node- and edge-labeled graphs, and supports the complete set of graph edit types in a fine-grained, type-aware approach, thereby resulting in highly accurate and efficient GED estimation for real-world, large graphs.

To train TaGSim cost-effectively, we design a graph pair generator that can produce a series of graph pairs with pre-determined, type-aware GED vectors without recourse to costly, exact GED computation. The graph pair generator can efficiently produce as much training data as possible that help train TaGSim in sufficient configurations for type-aware GED learning. In contrast to existing GNN methods that require costly pair-wise GED computation for the ground-truth GEDs, our graph-pair generator can significantly reduce the training cost for TaGSim.

To the best of our knowledge, TaGSim is the first end-to-end, type-aware GED learning and computation framework that supports all different types of graph edit operations for GED. TaGSim is more efficient and accurate for GED computation than state-of-the-art solutions, including the up-to-date GNN-based method, *SimGNN* [2]. The contributions of TaGSim are summarized below,

(1) We propose the idea of type-aware GED learning and computation, where each type of graph edit operations is modeled and learned individually based on high-quality, type-aware graph embeddings (Section 4.1);
(2) We develop an end-to-end GED learning and computation framework, TaGSim, which supports the complete set of graph edit operations allowed in GED, and enables type-aware GED learning and computation (Section 4.2);
(3) We design an efficient and cost-effective graph-pair generator that can produce training samples without recourse to costly GED computation. This can significantly lower the training barrier for GNN-based models in GED learning and computation (Section 4.3);
(4) We perform experimental studies on five real-world datasets, and the results demonstrate significant advantages of TaGSim, in terms of both GED estimation effectiveness and efficiency, compared with eight existing GED computation methods (Section 5).

The remainder of the paper is organized as follows: In Section 2 we brief related work for GED computation. In Section 3, we introduce key definitions and preliminary concepts for TaGSim. In Section 4, we discuss the technical details of TaGSim. We report our experimental results and key findings in Section 5, followed by concluding remarks in Section 6.

2 RELATED WORKS

GED-based Graph Similarity Computation. There exist numerous graph proximity measures that quantify the similarity between graphs, including graph edit distance (GED) [7, 37] and maximum common subgraph (MCS) [13], while GED has arguably been the most general and widely used one. Given a pair of graphs $\mathcal{G}$ and $\mathcal{G}'$, GED is the minimum number of graph edit operations that transform $\mathcal{G}$ into $\mathcal{G}'$, or vice versa. GED is a metric, and the exact GED computation is NP-hard [8]. As a result, many techniques have been proposed for GED approximation. Following the *filter-and-verification* paradigm, some algorithms exploit count-based GED lower bounds, $\underline{\text{GED}}(\mathcal{G}, \mathcal{G}')$, to identify and filter dissimilar graph-pairs [22, 26, 43, 45, 51]: If $\underline{\text{GED}}(\mathcal{G}, \mathcal{G}')$ is sufficiently large, $\mathcal{G}$ and $\mathcal{G}'$ are guaranteed dissimilar because $\underline{\text{GED}}(\mathcal{G}, \mathcal{G}') \leq \text{GED}(\mathcal{G}, \mathcal{G}')$. The combinatorial or heuristic search-based methods take advantage of A* strategies for search space exploration and unpromising subspace pruning [10, 14, 18, 33, 34, 38, 41]. These methods are typically efficient for small graphs. However, existing GED lower bounds are often loose, and the performance of search-based methods tends to deteriorate significantly on large graphs. Recently, deep learning techniques have been considered for GED approximation [2, 11, 30, 44]. For instance, graph neural networks (GNN) are used to extract embeddings from graphs, which are further fed into deep learning models for GED estimation. Compared with the search-based methods, GNN-based solutions achieve higher GED estimation accuracy, and their efficacy is not severely affected by graph sizes.

Graph Neural Networks. Recent years have seen great success of deep learning in numerous high-impact areas, including graphs and networks. Among various deep learning models, graph neural networks are primarily designed to learn graph representations by transforming, propagating, and aggregating node features, and have achieved outstanding performance in graph classification [24, 27, 42, 46], graph representation learning [31, 47], and link

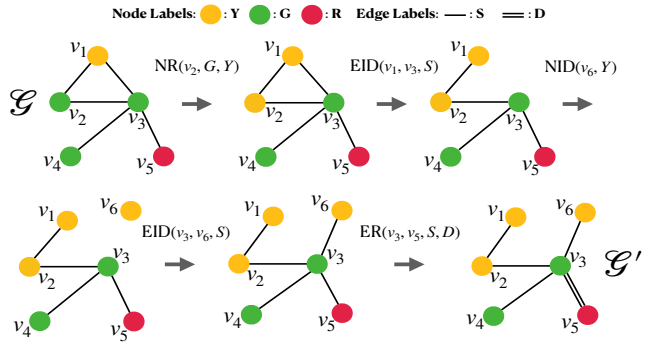


Figure 1: Given a graph $\mathcal{G}$ with the node label set $\Sigma_{\mathcal{V}} = \{$Y (yellow), G(green), R (red)$\}$ and the edge label set $\Sigma_{\mathcal{E}} = \{$S (single-bond), D (double-bond)$\}$, we transform $\mathcal{G}$ to another graph $\mathcal{G}'$ based on the following graph edit operations: (1) relabel node v_2 from label G to label Y; (2) delete edge (v_1, v_3); (3) insert node v_6 with label Y; (4) insert edge (v_3, v_6) with label S; (5) relabel edge (v_3, v_5) from label S to label D. As a result, $GED(\mathcal{G}, \mathcal{G}') = 5$.

prediction [9, 50]. The core operation of GNN is information aggregation among graph nodes [1, 19, 46]. Concretely, given a node u, in each layer of GNN, u's own feature is aggregated with those of its neighboring nodes for an updated new feature of u. A linear or non-linear transformation is usually applied on the aggregated features. After several layers of computation, the resultant node features can be used for downstream applications. There are various aggregation mechanisms in GNN models: Spectral methods consider information aggregation as spectral convolutions on graphs, and the aggregation is defined on graph spectral filters [6, 11, 24]; Spatial methods perform aggregation on groups of spatially close neighbors [42]. GNNs have also been used for graph similarity computation. SimGNN [2] uses graph convolutional network (GCN) for node feature aggregation, and considers the graph attention mechanism to readout graph embeddings from the aggregated node features; GHashing [30] constructs indices based on graph embeddings for similarity search. Note that the hyper-parameters of GNN that are related to information aggregation and linear/non-linear transformation need to be learned in the model training process, which is typically costly. In this paper, we propose *graph aggregation layer* (GAL) to extract type-aware graph embeddings, which are parameter-free.

3 PRELIMINARIES

In this paper, we consider simple, undirected, and labeled graphs for similarity computation. A graph $\mathcal{G}$ is a 4-tuple $(\mathcal{V}, \mathcal{E}, l, \Sigma)$, where $\mathcal{V}$ is a node set; $E \subseteq \mathcal{V} \times \mathcal{V}$ is an edge set; the labeling function $l : \mathcal{V} \cup \mathcal{E} \rightarrow \Sigma$ assigns nodes/edges with labels from a node/edge label set $\Sigma_{\mathcal{V}}/\Sigma_{\mathcal{E}}$, where $\Sigma_{\mathcal{V}} \cup \Sigma_{\mathcal{E}} = \Sigma$. We can edit or transform $\mathcal{G}$ based on the following types of *graph edit operations*:

- **Node Relabeling (NR)**: replace the label of node $u \in \mathcal{V}$ from $l(u)$ to $l'(u)$, denoted as $NR(u, l(u), l'(u))$, where $l(u), l'(u) \in \Sigma_{\mathcal{V}}$;
- **Node Insertion/deletion (NID)**: insert or delete an isolated node u with label $l(u)$ in or from $\mathcal{G}$, denoted as $NID(u, l(u))$;
- **Edge Relabeling (ER)**: replace the label of an edge $(u, v) \in \mathcal{E}$ from $l(u, v)$ to $l'(u, v)$, denoted as $ER(u, v, l(u, v), l'(u, v))$, where $l(u, v), l'(u, v) \in \Sigma_{\mathcal{E}}$;
- **Edge Insertion/deletion(EID)**: insert or delete an edge (u, v) with label $l(u, v)$ in or from $\mathcal{G}$, denoted as $EID(u, v, l(u, v))$.

Given a pair of graphs $\mathcal{G}$ and $\mathcal{G}'$, we can transform $\mathcal{G}$ step-by-step to $\mathcal{G}'$, or vice versa, by a *shortest* sequence P of graph edit operations, termed as *graph edit sequence*, $P(\mathcal{G}, \mathcal{G}')$. The *graph edit distance* between $\mathcal{G}$ and $\mathcal{G}'$, $GED(\mathcal{G}, \mathcal{G}')$, is defined as the number of graph edit operations in $P(\mathcal{G}, \mathcal{G}')$: $GED(\mathcal{G}, \mathcal{G}') = |P(\mathcal{G}, \mathcal{G}')|$. Note that $GED(\mathcal{G}, \mathcal{G}')$ is a metric [16], and computing $GED(\mathcal{G}, \mathcal{G}')$ is NP-hard [17]. It is worth mentioning that the order of graph edit operations in $P(\mathcal{G}, \mathcal{G}')$ is irrelevant to $GED(\mathcal{G}, \mathcal{G}')$. To this end, we define the type-aware *graph edit vector* as follows,

Definition 1. **[Graph Edit Vector]** *Given graphs $\mathcal{G}$ and $\mathcal{G}'$, where $\mathcal{G}$ can be transformed to $\mathcal{G}'$ following the graph edit sequence $P(\mathcal{G}, \mathcal{G}')$, the graph edit vector (GEV) of $\mathcal{G}$ and $\mathcal{G}'$* w.r.t. *P is a four-dimensional vector whose dimensions represent the number of graph edit operations in $P(\mathcal{G}, \mathcal{G}')$ corresponding to the four graph edit types, NR, NID, ER, or EID, respectively:*

$$GEV_P(\mathcal{G}, \mathcal{G}') = [\#NR, \#NID, \#ER, \#EID]^\top \quad (1)$$

Example 1. *Consider $\mathcal{G}$ and $\mathcal{G}'$ shown in Figure 1. We transform $\mathcal{G}$ to $\mathcal{G}'$ based on the graph edit sequence $P(\mathcal{G}, \mathcal{G}') = (NR(v_2, G, Y), EID(v_1, v_3, S), NID(v_6, Y), EID(v_3, v_6, S), ER(v_3, v_5, S, D))$. As a result, $GED(\mathcal{G}, \mathcal{G}') = 5$, and the graph edit vector of $\mathcal{G}$ and $\mathcal{G}'$* w.r.t. *P is $GEV_P(\mathcal{G}, \mathcal{G}') = [1, 1, 1, 2]^\top$.* □

It suffices to compute $GED(\mathcal{G}, \mathcal{G}')$ by accumulating the four-dimensional values of $GEV_P(\mathcal{G}, \mathcal{G}')$. Consequently, the NP-hard GED problem translates directly to the computation of type-aware GEV. As the identification of graph edit sequence, $P(\mathcal{G}, \mathcal{G}')$, is theoretically intractable, we consider in this paper an estimation of type-aware GEV for $\mathcal{G}$ and $\mathcal{G}'$.

4 TAGSIM

Existing GED solutions, including lower-bound based methods [10, 22, 26, 51] and GNN-based deep learning techniques [2, 30], model and quantify GED as a single, global similarity score between graphs. In fact, GED is collectively determined by the number of graph edit operations of four different types: *NR*, *NID*, *ER*, and *EID*, each of which has unique, fine-grained transformative impacts on the localized neighborhood of graphs. For instance, by inserting or deleting edges, *EID* may change graph connectivity, but not the node information. In contrast, *NR* can change node labels, while not altering the edge information nor graph connectivity. State-of-the-art, *type-oblivious* solutions that treat GED as a global graph similarity score ignore such salient impacts of different graph edit types, thus resulting in coarse-grained, inaccurate GED estimations [2, 30]. To address this weakness, we propose TaGSim (**T**ype-**a**ware **G**raph **Sim**ilarity), a *type-aware* GED learning and computation framework that refines the GED modelling, learning, and computation respecting different graph edit types. Specifically, GED is decoupled to four distinct components, as modeled in the four GEV dimensions, each of which corresponds to one particular graph edit type to be learned and estimated individually. TaGSim can significantly enhance the effectiveness and efficiency in GED computation, compared with existing type-oblivious methods.

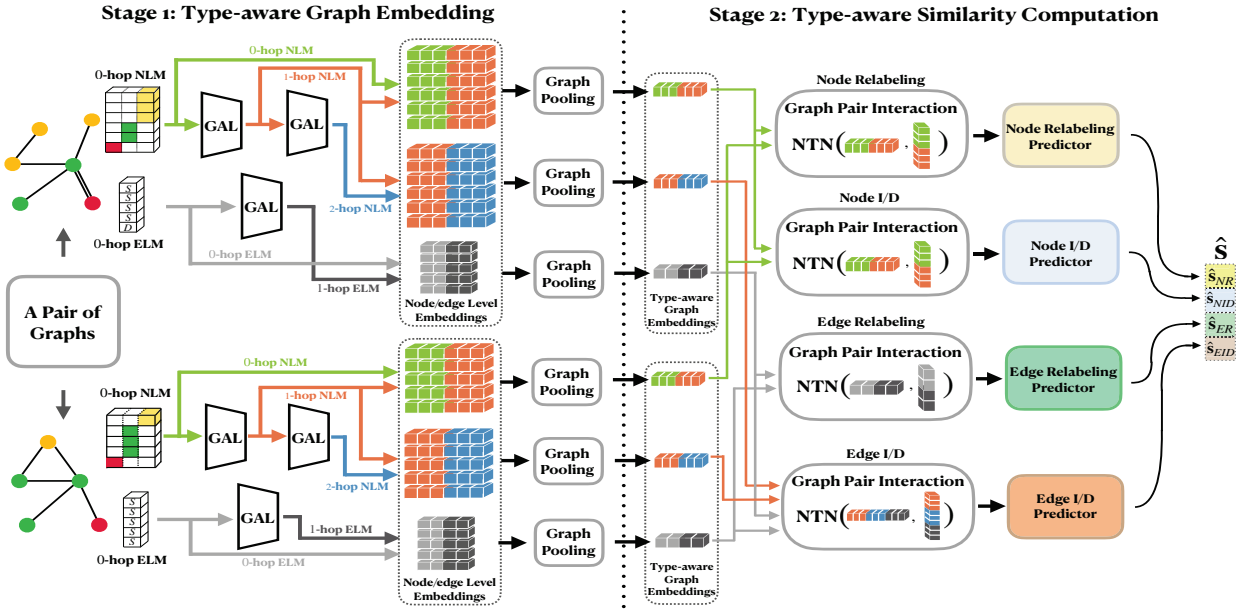


Figure 2: The TaGSim Framework: Stage 1 (left) is type-aware graph embedding; Stage 2 (right) is type-aware similarity computation, where we estimate the number of graph edit operations for each edit type; that is, the dimensions of GEV.

Algorithm 1 The TaGSim Framework

Input: A graph pair $(\mathcal{G}, \mathcal{G}')$, where $\mathcal{G} = (\mathcal{V}, \mathcal{E}, l, \Sigma)$, $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', l', \Sigma)$
Output: The estimated type-aware GEV, $\hat{\mathbf{S}}$, for $\mathcal{G}$ and $\mathcal{G}'$

1: $\mathbf{H}_0^n \leftarrow l(\mathcal{V}), \mathbf{H}_0^e \leftarrow l(\mathcal{E})$; ▷ 0-hop NLMs, ELMs of $\mathcal{G}$
2: $\mathbf{H}_0^{n'} \leftarrow l'(\mathcal{V}'), \mathbf{H}_0^{e'} \leftarrow l'(\mathcal{E}')$; ▷ 0-hop NLMs, ELMs of $\mathcal{G}'$
3: $\mathbf{H}_k^n \leftarrow \mathbf{GAL}(\mathbf{H}_{k-1}^n), k \geq 1$; ▷ k-hop NLMs of $\mathcal{G}$
4: $\mathbf{H}_k^e \leftarrow \mathbf{GAL}(\mathbf{H}_{k-1}^e), k \geq 1$; ▷ k-hop ELMs of $\mathcal{G}$
5: $\mathbf{H}_k^{n'} \leftarrow \mathbf{GAL}(\mathbf{H}_{k-1}^{n'}), k \geq 1$; ▷ k-hop NLMs of $\mathcal{G}'$
6: $\mathbf{H}_k^{e'} \leftarrow \mathbf{GAL}(\mathbf{H}_{k-1}^{e'}), k \geq 1$; ▷ k-hop ELMs of $\mathcal{G}'$
7: $\mathbf{h}_{cat(i,j)}^n \leftarrow \mathbf{Pooling}(\mathbf{H}_i^n || \mathbf{H}_j^n), i, j \geq 0$; ▷ Node-level embedding of $\mathcal{G}$
8: $\mathbf{h}_{cat(i,j)}^e \leftarrow \mathbf{Pooling}(\mathbf{H}_i^e || \mathbf{H}_j^e), i, j \geq 0$; ▷ Edge-level embeddings of $\mathcal{G}$
9: $\mathbf{h}_{cat(i,j)}^{n'} \leftarrow \mathbf{Pooling}(\mathbf{H}_i^{n'} || \mathbf{H}_j^{n'}), i, j \geq 0$; ▷ Node-level embedding of $\mathcal{G}'$
10: $\mathbf{h}_{cat(i,j)}^{e'} \leftarrow \mathbf{Pooling}(\mathbf{H}_i^{e'} || \mathbf{H}_j^{e'}), i, j \geq 0$; ▷ Edge-level embeddings of $\mathcal{G}'$
11: $\hat{s}_{NR} \leftarrow \mathbf{NN}_{NR}(\mathbf{NTN}_{NR}(\mathbf{h}_{cat(i,j)}^n, \mathbf{h}_{cat(i,j)}^{n'})), i, j \geq 0$; ▷ #NR
12: $\hat{s}_{NID} \leftarrow \mathbf{NN}_{NID}(\mathbf{NTN}_{NID}(\mathbf{h}_{cat(i,j)}^n, \mathbf{h}_{cat(i,j)}^{n'})), i, j = 0$; ▷ #NID
13: $\hat{s}_{ER} \leftarrow \mathbf{NN}_{ER}(\mathbf{NTN}_{ER}(\mathbf{h}_{cat(i,j)}^e, \mathbf{h}_{cat(i,j)}^{e'})), i, j \geq 0$; ▷ #ER
14: $\hat{s}_{EID} \leftarrow \mathbf{NN}_{EID}(\mathbf{NTN}_{EID}(\mathbf{h}_{cat(i,j)}^n || \mathbf{h}_{cat(p,q)}^e, \mathbf{h}_{cat(i,j)}^{n'} || \mathbf{h}_{cat(p,q)}^{e'})),$
$i, j \geq 1, p, q \geq 0$; ▷ #EID
15: **return** $\hat{\mathbf{S}} = [\hat{s}_{NR}, \hat{s}_{NID}, \hat{s}_{ER}, \hat{s}_{EID}]$

The TaGSim framework is sketched in Algorithm 1, which contains two main stages. The first one is *type-aware graph embedding* (Lines 1-10): For each type of graph edit operations, TaGSim extracts node/edge-level embeddings by *Graph Aggregation Layers* (GALs) based on the unique transformative impacts of that edit type (Lines 1-6; Details in Section 4.1). The node/edge embeddings are further aggregated into type-aware graph embeddings by graph pooling functions for the next-stage GED estimation (Lines 7-10). The second stage, *type-aware similarity computation*, uses neural tensor networks (NTNs) to establish interactions between type-aware graph embeddings, and the results are further fed into fully connected neural networks in order to estimate each dimension of GEVs (Lines 11-14; Details in Section 4.2). To facilitate the training of TaGSim without resort to costly exact GED computation, we design a type-aware graph-pair generator that can efficiently generate graph pairs, together with their GEVs, as the ground truth (Section 4.3). Once trained, TaGSim can be used to learn and estimate for any pair of graphs their GED following the two stages analogously as in the TaGSim training. The full TaGSim framework is illustrated in Figure 2.

4.1 Type-aware Graph Embedding

To estimate each dimension of GEV, or equivalently, the number of graph edit operations of each type, we first investigate the intrinsic transformative impacts of different graph edit operations on graphs. In particular, we examine the localized regions surrounding nodes and edges where graph edit operations arise. Given a node $u \in \mathcal{V}$, the *k-hop neighborhood* of u is $N^k(u) = \{v | dist(u, v) = k, v \in \mathcal{V}\}$, where $dist(u, v)$ is the shortest distance between u and v in $\mathcal{G}$. $N^k(u)$ comprises all the k-hop neighboring nodes of u, and $N^0(u) = \{u\}$ when $k = 0$. In addition, the *k-hop node-label multiset* of u, defined as $L^k(u) = \{l(v) | v \in N^k(u)\}$, comprises node-labels of all the k-hop neighboring nodes of u with repetitive labels maintained cumulatively (In what follows, we use the format of *element:multiplicity* to represent elements of a multiset for brevity). We further define the *k-hop node-label multiset* (k-hop NLM) of the graph $\mathcal{G}$ as follows,

Definition 2. **[k-hop Node-label Multiset (NLM)]** *The k-hop node-label multiset of the graph $\mathcal{G}$, denoted as $NLM^k(\mathcal{G})$, is a multiset of the k-hop node-label multisets for all the nodes of $\mathcal{G}$: $NLM^k(\mathcal{G}) = \{L^k(u) | u \in \mathcal{V}\}$ with repetitive elements, $L^k(\cdot)$, allowed in $NLM^k(\mathcal{G})$.* □

Example 2. *Consider the graph $\mathcal{G}$ in Figure 1. The 0-hop NLM of $\mathcal{G}$ is $NLM^0(\mathcal{G}) = \{\{Y:1\}, \{G:1\}, \{G:1\}, \{G:1\}, \{R:1\}\}$ (for brevity, it is represented as $\{\{G:1\}:3, \{Y:1\}:1, \{R:1\}:1\}$). The 1-hop NLM of $\mathcal{G}$ is $NLM^1(\mathcal{G}) = \{\{G:1\}:2, \{G:2\}:1, \{G:1, Y:1\}:1, \{G:2, Y:1, R:1\}:1\}$.* □

Likewise, we extend the definition of k-hop neighborhood of nodes to edges. Given an edge $e = (u, v) \in \mathcal{E}$, the *k-hop neighborhood* of e is $N^k(e) = \{e' | dist(e, e') = k, e' \in \mathcal{E}\}$, where $dist(e, e')$ is the shortest

distance between e and e' in $\mathcal{G}$ [1], and $N^k(e)$ comprises all the k-hop neighboring edges of e. $N^0(e) = \{e\}$ when $k = 0$. Furthermore, the *k-hop edge-label multiset* of e, $L^k(e) = \{l(e') | e \in N^k(e)\}$, comprises edge labels for all the k-hop neighboring edges of e with repetitive labels maintained cumulatively. We further define the *k-hop edge-label multiset* (k-hop ELM) of the graph $\mathcal{G}$ as follows,

Definition 3. **[k-hop Edge-label Multiset (ELM)]** *The k-hop edge-label multiset of the graph $\mathcal{G}$, denoted as $ELM^k(\mathcal{G})$, is a multiset of the k-hop edge-label multisets for all the edges of $\mathcal{G}$: $ELM^k(\mathcal{G}) = \{L^k(e) | e \in \mathcal{E}\}$, with repetitive elements, $L^k(\cdot)$, allowed in $ELM^k(\mathcal{G})$.* □

Example 3. *Consider the graph $\mathcal{G}'$ in Figure 1. The 0-hop ELM of $\mathcal{G}'$ is $ELM^0(\mathcal{G}') = \{\{S: 1\}: 4, \{D: 1\}: 1\}$, and the 1-hop ELM of $\mathcal{G}'$ is $ELM^1(\mathcal{G}') = \{\{S: 1\}: 1, \{S: 3\}: 1, \{S: 2, D: 1\}: 2, \{S: 3, D: 1\}: 1\}$.* □

The k-hop NLM and k-hop ELM are the key data structures that maintain the node-label and edge-label information within k-hop ($k \geq 0$) neighborhood surrounding nodes and edges of $\mathcal{G}$, respectively. When a graph edit operation arises, it will transform $\mathcal{G}$ by triggering different changes to $NLM^k(\mathcal{G})$ and $ELM^k(\mathcal{G})$ depending on the type of that graph edit operation. Below we examine for each graph edit type its unique transformative impacts upon $\mathcal{G}$ subject to $NLM^k(\mathcal{G})$ and $ELM^k(\mathcal{G})$.

1. Node Relabeling (*NR*). Consider t node relabeling operations, denoted as $NR_1, \ldots, NR_t (t \geq 1)$, which transform $\mathcal{G}$ to $\mathcal{G}'$. Each NR_i changes an existing node label l_i to a new label l_i', where $l_i, l_i' \in \Sigma_{\mathcal{V}} (1 \leq i \leq t)$. We denote the multiset of node-labels $\bigcup_{1 \leq i \leq t} \{l_i\}$ as the *old* label set, and the multiset of node-labels $\bigcup_{1 \leq i \leq t} \{l_i'\}$ as the *new* label set. In the case where there are no common labels between the old and the new label sets; that is,

$$(\cup_{1 \leq i \leq t} \{l_i\}) \cap (\cup_{1 \leq i \leq t} \{l_i'\}) = \emptyset, \tag{2}$$

the difference between $\mathcal{G}$ and $\mathcal{G}'$ incurred by node relabeling operations is precisely reflected by the node label collections of $\mathcal{G}$ and $\mathcal{G}'$; that is, the 0-hop NLMs of $\mathcal{G}$ and $\mathcal{G}'$: $NLM^0(\mathcal{G})$ and $NLM^0(\mathcal{G}')$. The reason is that, for each node-relabeling operation NR_i, the old label l_i is uniquely and unambiguously substituted by the new label l_i' due to the condition in Equation 2, and the number of node relabeling operations, *#NR*, can be precisely determined by the Hamming distance of $NLM^0(\mathcal{G})$ and $NLM^0(\mathcal{G}')$.

Theorem 1. *Given node relabeling operations $NR_1, \ldots, NR_t (t \geq 1)$ that transform $\mathcal{G}$ to $\mathcal{G}'$, for each $NR_i (1 \leq i \leq t)$, it leads to at most one node-label change between $NLM^0(\mathcal{G})$ and $NLM^0(\mathcal{G}')$.* □

Since one *NR* operation affects at most one node in $\mathcal{G}$, Theorem 1 guarantees that the influence of *NR*s on $NLM^0(\mathcal{G})$ is upper bounded. It thus suffices to use 0-hop NLMs to estimate *#NR* in this case. However, when the condition of Equation 2 fails, 0-hop NLMs may err in quantifying *#NR*. Consider for instance two node-relabeling operations, NR_1 and NR_2, that transform $\mathcal{G}$ to $\mathcal{G}'$, where NR_1 replaces the label of a node u from l_1 to l_2, while NR_2 replaces the label of another node v from l_2 to l_1. In this case, both the old and new label sets are $\{l_1, l_2\}$. Since $NLM^0(\mathcal{G}) = NLM^0(\mathcal{G}')$, no node relabeling operations can be identified.

As the number of node relabeling operations, *#NR*, is not solely determined by 0-hop NLMs, we consider the higher-order, structure-enriched label information encoded in k-hop NLMs ($k \geq 1$) of graphs. Note that when an *NR* operation arises on a node u, for every node v within the k-hop neighborhood of u, its k-hop node-label multiset, $L^k(v)$, needs an update to reflect the label change of u. Consider $NLM^1(\mathcal{G})(k = 1)$ as an example: similar to Theorem 1, one *NR* operation changes at most d_u elements in $NLM^1(\mathcal{G})$, where d_u is the degree of u. As a result, the k-hop ($k \geq 1$) NLM of $\mathcal{G}$, $NLM^k(\mathcal{G})$, maintains the information for node-label updates triggered by *NR*s within k-hop neighborhood surrounding nodes of $\mathcal{G}$. Even in the case where the condition of Equation 2 fails, the higher-order k-hop NLMs can still capture the occurrences of *NR*s, thus providing accurate estimations for *#NR*.

[1] In a graph, the shortest distance between an edge e and itself is 0; that is, $dist(e, e) = 0$. Consider two edges $e(u, v)$ and $e'(v, w)$ that are incident on a common node v, where $u, v, w \in \mathcal{V}$ and $u \neq w$, then e' is a neighboring edge of e, and $dist(e, e') = 1$. This way, the notion of shortest distance can be extended to edges analogously.

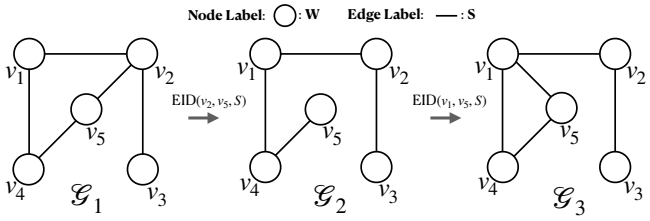


Figure 3: Given a graph $\mathcal{G}_1$ with the node-label set $\Sigma_{\mathcal{V}} = \{\mathbf{W}$ (white)$\}$ and the edge-label set $\Sigma_{\mathcal{E}} = \{\mathbf{S}$ (single-bond)$\}$, $\mathcal{G}_1$ is first transformed to $\mathcal{G}_2$ with the edge (v_2, v_5) deleted by an *EID* operation, EID(v_2, v_5, S). $\mathcal{G}_2$ is further transformed to $\mathcal{G}_3$ by inserting another edge (v_1, v_5) based on an *EID* operation, EID(v_1, v_5, S).

2. Node Insertion/deletion (*NID*). It is worth noting that a node-deletion operation can be avoided when transforming a graph $\mathcal{G}$ with fewer nodes to another graph $\mathcal{G}'$ with more nodes. Without loss of generality, we only consider node insertion operations for NID because of the symmetric property of GED. Since a node insertion operation leads to a new, isolated node, NID only changes the node count of $\mathcal{G}$, which can be well captured by the 0-hop NLM of $\mathcal{G}$. Therefore, $NLM^0(\mathcal{G})$ suffices for the $\#NID$ estimation.

3. Edge Relabeling (*ER*). Similar to *NR*, relabeling an edge $e = (u, v)$ may change not only the edge-label multiset of $\mathcal{G}$, $ELM^0(\mathcal{G})$, but also the k-hop edge-label multiset $L^k(e')$ of every edge e' that is within the k-hop neighborhood of e, where $k \geq 1$. Therefore, to estimate *#ER*, we consider both 0-hop and k-hop ($k \geq 1$) ELMs of $\mathcal{G}$.

4. Edge Insertion/deletion (*EID*). *EID* changes not only the edge label information, but graph connectivity as well. As a result, *EID* may affect k-hop data structures for both nodes and edges. Consider an *EID* operation related to either an existing or a new edge $e = (u, v)$ in $\mathcal{G}$. Node labels of the two incident nodes u and v stay unchanged, so there are no updates in $NLM^0(\mathcal{G})$. However, the k-hop node-label multisets for (1) u, (2) v, and (3) u's (and v's) $(k-1)$-hop neighbors are all changed due to the insertion or deletion of the edge (u, v). Therefore, the k-hop NLM of $\mathcal{G}$, $NLM^k(\mathcal{G})(k \geq 1)$, may change due to *EID*.

Example 4. *Consider graph $\mathcal{G}_1$ shown in Figure 3. The 1-hop NLM of $\mathcal{G}_1$ is $NLM^1(\mathcal{G}_1) = \{\{W: 2\}: 3, \{W: 3\}: 1, \{W: 1\}: 1\}$. When an* EID *arises that removes the edge (v_2, v_5), $\mathcal{G}_1$ is transformed to $\mathcal{G}_2$, and $\mathcal{G}_2$'s 1-hop NLM is $NLM^1(\mathcal{G}_2) = \{\{W: 2\}: 3, \{W: 1\}: 2\}$. Therefore, EIDs may trigger updates to $NLM^k(\mathcal{G}_1)$ when $k = 1$.* □

Theorem 2. *Consider an EID operation upon the edge (u, v) of $\mathcal{G}$. It may change at most two elements of $NLM^1(\mathcal{G})$ related to the incident nodes u and v.* □

According to Theorem 2, an *EID* upon edge $e = (u, v)$ changes the 1-hop NLMs of u and v, respectively; that is, the number of changes incurred by an *EID* upon 1-hop NLMs is upper bounded. We thus can estimate *#EID* by examining the differences between the 1-hop NLMs of two graphs.

When multiple *EID*s arise, however, the 1-hop NLM of $\mathcal{G}$ may fail to estimate *#EID* accurately. For example, consider the graph $\mathcal{G}_1$ in Figure 3, and there are two *EID*s upon $\mathcal{G}_1$, $EID(v_2, v_5, S)$ and $EID(v_1, v_5, S)$, that transform $\mathcal{G}_1$ to $\mathcal{G}_3$. Note that the 1-hop NLM of $\mathcal{G}_1$, $NLM^1(\mathcal{G}_1) = \{\{W: 2\}: 3, \{W: 3\}: 1, \{W: 1\}: 1\}$, and the 1-hop NLM of $\mathcal{G}_3$, $NLM^1(\mathcal{G}_3) = \{\{W: 2\}: 3, \{W: 3\}: 1, \{W: 1\}: 1\}$, are the same, so they cannot account for

Table 1: Transformative Impacts of Different Types of Graph Edit Operations on k-hop Data Structures: NLMs and ELMs

	0-hop NLM	0-hop ELM	k-hop NLM ($k \geq 1$)	k-hop ELM ($k \geq 1$)
NR	✔		✔	
NID	✔			
ER		✔		✔
EID		✔	✔	✔

the two *EID*s between $\mathcal{G}_1$ and $\mathcal{G}_3$ in this scenario. To this end, we further consider k-hop NLMs by increasing the values of k. Note that when $k = 2$, the 2-hop NLM of $\mathcal{G}_1$, $NLM^2(\mathcal{G}_1) = \{\{W: 1\}: 2, \{W: 2\}: 3\}$, and the 2-hop NLM of $\mathcal{G}_3$, $NLM^2(\mathcal{G}_3) = \{\{W: 1\}: 1, \{W: 2\}: 1, \{W: 3\}: 3\}$, are different due in particular to the two *EID* operations between $\mathcal{G}_1$ and $\mathcal{G}_3$.

Theorem 3. *Consider an EID operation upon the edge (u, v) of $\mathcal{G}$. It may change at most $2(d_{max})^{k-1}$ elements in $NLM^k(\mathcal{G})(k \geq 2)$, where d_{max} is the maximum node-degree of $\mathcal{G}$.* □

Proof. For any node u of $\mathcal{G}$, consider the $(k-1)$-hop neighborhood of u, $N^{k-1}(u)$. We have $|N^{k-1}(u)| \leq (d_{max})^{k-1}$. Then $\forall w \in N^{k-1}(u)$, we have $u \in N^{k-1}(w)$ by symmetry. For any node $w \in N^{k-1}(u) \wedge dist(w, v) > k$, after the insertion of edge (u, v), v is now within w's k-hop neighborhood; that is, $v \in N^k(w)$. From the perspective of u, there are at most $|N^{k-1}(u)|$ elements changed in $NLM^k(\mathcal{G})$ due to the *EID*. For node v, the situation is the same: there are at most $|N^{k-1}(v)|$ elements changed in $NLM^k(\mathcal{G})$. For edge deletion, $NLM^k(\mathcal{G})$ changes the same way as in the case of edge insertion. As a result, there are at most $|N^{k-1}(u)| + |N^{k-1}(v)| = 2(d_{max})^{k-1}$ elements changed in $NLM^k(\mathcal{G})$ because of the *EID*. □

Theorem 3 provides an upper bound for the number of possible changes upon k-hop NLMs ($k \geq 2$) incurred by an *EID*; Upper bounds on k'-hop ELMs can be derived in a similar way, where $k' \geq 0$. As a result, #*EID* can be estimated by a combination of k-hop NLMs ($k \geq 1$) and k'-hop ELMs ($k' \geq 0$) of graphs.

Based on the above discussion, we summarize the transformative impacts of each type of graph edit operations upon the k-hop data structures, as shown in Table 1. As such transformative impacts are type-variant, they have to be modeled individually in order to estimate the number of correponding graph edit types in GED. In the following, we propose Graph Aggregation Layer (GAL) to encode the k-hop NLMs and ELMs as low-dimensional, *type-aware* graph embeddings for accurate GED estimation.

4.1.1 ***Graph Aggregation Layer (GAL).*** For the core data structures k-hop NLMs and k-hop ELMs, they can be built from the initial node/edge labels and the adjacency matrix of $\mathcal{G}$. Inspired by graph representation learning methods [24, 42], we propose *graph aggregation layer* (GAL), which extracts graph embeddings to encode the salient information of k-hop NLMs and k-hop ELMs of $\mathcal{G}$. First of all, the *node-based GAL*, $\mathbf{H}_k^n (k \geq 0)$ can be computed as

$$\mathbf{H}_1^n = \mathbf{A} \cdot \mathbf{H}_0^n \tag{3}$$

where $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the adjacency matrix of $\mathcal{G}$; $\mathbf{H}_0^n \in \mathbb{R}^{|\mathcal{V}|\times d_n}$ is the 0-hop node-feature matrix with d_n the size of node feature vectors. At first, $\mathbf{H}_0^n$ is initialized as the one-hot encoding of the 0-hop NLM of $\mathcal{G}$, $NLM^0(\mathcal{G})$. Based on Equation 3, $\mathbf{H}_1^n \in \mathbb{R}^{|\mathcal{V}|\times d_n}$ approximates the 1-hop node-label multiset information for the nodes of $\mathcal{G}$. We thus use $\mathbf{H}_1^n$ as the embeddings of 1-hop NLM of $\mathcal{G}$, $NLM^1(\mathcal{G})$. Analogously, for k-hop NLMs of $\mathcal{G}$, $NLM^k(\mathcal{G})$, where $k \geq 2$, they can be derived based on a k-step iterative computation of GAL as,

$$\mathbf{H}_k^n = \mathbf{A} \cdot \mathbf{H}_{k-1}^n \tag{4}$$

where $\mathbf{H}_k^n$ summarizes the k-hop node-label multiset information for each node of $\mathcal{G}$. To this end, node-based GALs encode k-hop NLMs of $\mathcal{G}$ as node embeddings, which can be used to estimate the following dimensions of GEV: #*NR*, #*NID*, and #*EID*.

Different from existing GNN-based graph similarity solutions where edge labels are ignored in GED learning and computation [2, 11], TaGSim takes into consideration both edge labels and edge relabeling operations, *ER*. To handle the edge label information, we propose *edge-based GAL* analogously. First, we introduce *edge adjacency matrix* of $\mathcal{G}$ as follows,

Definition 4. **[Edge Adjacency Matrix]** *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the edge adjacency matrix $\mathbf{E}$ of $\mathcal{G}$ is a $|\mathcal{E}| \times |\mathcal{E}|$ matrix with its element $\mathbf{E}_{i,j} (1 \leq i, j \leq |\mathcal{E}|)$ defined as*

$$\mathbf{E}_{i,j} = \mathbb{1}_{\mathcal{E}_i \cap \mathcal{E}_j \neq \emptyset} \tag{5}$$

where $\mathbb{1}$ is the indicator function; $\mathcal{E}_i$ denotes the i-th edge of $\mathcal{E}$; $\mathcal{E}_i \cap \mathcal{E}_j \neq \emptyset$ means that edge $\mathcal{E}_i$ and edge $\mathcal{E}_j$ are incident on a common node. □

Similar to the adjacency matrix $\mathbf{A}$, the edge adjacency matrix $\mathbf{E}$ maintains the edge connectivity information of $\mathcal{G}$: for each edge of $\mathcal{G}$, its neighboring edges can be explicitly identified by $\mathbf{E}$. We further define the edge-based GAL as

$$\mathbf{H}_1^e = \mathbf{E} \cdot \mathbf{H}_0^e \tag{6}$$

where $\mathbf{H}_0^e \in \mathbb{R}^{|\mathcal{E}|\times d_e}$ is the 0-hop edge feature matrix with d_e the size of edge feature vectors, and it is initialized as the one-hot encoding of the edge-label information in $ELM^0(\mathcal{G})$. According to Equation 6, $\mathbf{H}_1^e$ summarizes the 1-hop edge-label multiset information for each edge of $\mathcal{G}$, so we use it as the edge embedding of the 1-hop ELM of $\mathcal{G}$, $ELM^1(\mathcal{G})$. Analogously, for the k-hop ELMs of $\mathcal{G}$, $ELM^k(\mathcal{G})$, where $k \geq 2$, we compute $\mathbf{H}_k^e$ as edge embeddings by a k-step iterative computation. As a result, edge-based GALs encode k-hop ELMs of $\mathcal{G}$ as edge embeddings, which can be used to estimate the following dimensions of GEV: #*ER* and #*EID*.

Note that the node-based GAL, $\mathbf{H}_k^n$, and the edge-based GAL, $\mathbf{H}_k^e$, where $k \geq 0$, are designed to encode the salient information in the core data structures of k-hop NLMs and k-hop ELMs in the forms of node/edge embeddings. In addition, different from existing GNN methods [24, 42] where linear/non-linear transformations, together with a large number of hyper-parameters, are required in different layers of GNNs, our GAL structures are parameter-free, thus leading to a significantly lower training cost than existing GNN-based solutions. As a result, GALs enable us to construct type-aware graph embeddings for each type of graph edit operations. We then introduce the techniques on how to combine them for GED learning and estimation.

4.1.2 ***Embedding Concatenation and Pooling.*** According to the analysis in Section 4.1, each type of graph edit operations has different transformative effects on the key data structures, k-hop NLMs and k-hop ELMs. In the meantime, the node-based GAL and the edge-based GAL encode the salient information of k-hop NLMs and ELMs into low-dimensional node and edge embeddings, which, as a result, can help estimate the number of graph edit operations in different types. For instance, for *NR* operations, they may affect $NLM^0(\mathcal{G})$ and $NLM^k(\mathcal{G})(k \geq 1)$. As a consequence, we need $\mathbf{H}_{k'}^n$, where $k' = 0$, and $\mathbf{H}_{k''}^n$, where $k'' \geq 1$, together to learn and estimate #*NR*. To this end, we combine and concatenate different embeddings from $\mathbf{H}_k^n$ and/or $\mathbf{H}_k^e$ for different types of graph edit operations, based on the results in Table 1. We denote, for instance, a concatenation of $\mathbf{H}_i^n$ and $\mathbf{H}_j^n$, where $i, j \geq 0$, as

$$\mathbf{H}_{cat(i,j)}^n = \mathbf{H}_i^n \,||\, \mathbf{H}_j^n \tag{7}$$

where $||$ is the concatenation operator; Both $\mathbf{H}_i^n$ and $\mathbf{H}_j^n$ are d_n-dimensional node embeddings that are concatenated toward a new $2d_n$-dimensional node embedding: $\mathbf{H}_{cat(i,j)}^n \in \mathbb{R}^{|\mathcal{V}|\times 2*d_n}$. For example, $\mathbf{H}_{cat(0,1)}^n$ combines $\mathbf{H}_0^n$ and $\mathbf{H}_1^n$, which can be used for learning and estimating #*NR*. This way, for each graph edit type, the node/edge embeddings are combined adaptively to enable type-aware GED learning and computation.

Algorithm 2 Graph Pair Generator

Input: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, l, \Sigma)$; A target GEV
Output: The graph pair $(\mathcal{G}, \mathcal{G}^s)$ with the ground truth GEV

1: $\mathcal{G}^s = (\mathcal{V}^s, \mathcal{E}^s, l^s, \Sigma) \leftarrow \mathcal{G}$;
2: **for** $i = 1, 2, \ldots, GEV[NR]$ **do** ▷ Node Relabeling
3: Randomly select $v \in \mathcal{V}^s$ where v is not relabeled;
4: $NR(v, l^s(v), l^{s'}(v))$ where $l^s(v), l^{s'}(v) \in \Sigma_{\mathcal{V}}$ and $l^s(v) \neq l^{s'}(v)$;
5: **end for**
6: **for** $i = 1, 2, \ldots, GEV[NID]$ **do** ▷ Node Insertion
7: $NID(v_{new}, l^s(v_{new}))$ where $l^s(v_{new}) \in \Sigma_{\mathcal{V}}$;
8: **end for**
9: Randomly select $ToDel \in \{0, 1, \ldots, GEV[EID]\}$;
10: Del_Edge $\leftarrow$ [ ]; ▷ Initialized as an empty list
11: **for** $i = 1, 2, \ldots, ToDel$ **do** ▷ Edge Deletion
12: Randomly select $e \in \mathcal{E}^s$;
13: Del_Edge.append(e);
14: $EID(e, l^s(e))$;
15: **end for**
16: **for** $i = 1, 2, \ldots, GEV[ER]$ **do** ▷ Edge Relabeling
17: Randomly select $e \in \mathcal{E}^s$ where e is not relabeled;
18: $ER(e, l^s(e), l^{s'}(e))$ where $l^s(e), l^{s'}(e) \in \Sigma_{\mathcal{E}}$ and $l^s(e) \neq l^{s'}(e)$;
19: **end for**
20: **for** $i = 1, 2, \ldots, GEV[EID] - ToDel$ **do** ▷ Edge Insertion
21: $EID(e_{new}, l^s(e_{new}))$ where $l^s(e_{new}) \in \Sigma_{\mathcal{E}}$, $e_{new} \notin$ Del_Edge;
22: **end for**
23: **return** $(\mathcal{G}, \mathcal{G}^s)$, GEV

To this end, the embeddings are solely represented at the node or edge level. To read out the ultimate embedding at the more general graph level, an extra pooling layer is required. For the example of $\mathbf{H}^n_{cat(0,1)}$, the pooling can be represented as

$$\mathbf{h}^n_{cat(0,1)} = \mathbf{Pooling}(\mathbf{H}^n_{cat(0,1)}) \tag{8}$$

where $\mathbf{h}^n_{cat(0,1)} \in \mathbb{R}^{2*d_n}$. In practice, there are many choices for the pooling function. To retain the information of k-hop NLMs and ELMs of $\mathcal{G}$, we simply choose $\mathsf{sum}(\cdot)$ as the default pooling function, and the output of graph pooling is the final type-aware graph embeddings we design for GED estimation in the next step.

4.2 Type-aware Similarity Computation

In the second-stage of TaGSim, we estimate the four GEV dimensions, equivalently the number of different graph edit types, based on separate type-aware GED computation modules.

4.2.1 Graph Pair Interaction. Given the type-aware graph embeddings for a pair of input graphs, which turn out to be the output of the first stage of TaGSim, they are fed into a graph interaction module that generates a vector as the GED similarity embedding for the graph pair. Here we adopt the neural tensor network (NTN) [39] for graph pair interaction: For each graph edit type t, a type-aware NTN_t is trained and deployed to compute type-aware similarity embeddings, as shown in Figure 2. Specifically, given the type-aware graph embeddings $\mathbf{h}_1, \mathbf{h}_2 \in \mathbb{R}^d$ for graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, the type-aware NTN *w.r.t.* graph edit type t is formulated as

$$\text{NTN}_t(\mathbf{h}_1, \mathbf{h}_2) = \sigma(\mathbf{h}_1 \mathbf{W}_1^{[1:L]} \mathbf{h}_2^\top + \mathbf{W}_2(\mathbf{h}_1 \,||\, \mathbf{h}_2)^\top + \mathbf{b}) \tag{9}$$

where $\mathbf{W}_1^{[1:L]} \in \mathbb{R}^{d \times d \times L}$, $\mathbf{W}_2 \in \mathbb{R}^{L \times 2d}$, and $b \in \mathbb{R}^L$ are learnable weights; L is a hyper-parameter controlling the output dimension; $\sigma(\cdot)$ is an activation function. The output of a type-aware NTN_t is a similarity embedding vector of size L.

Table 2: Statistics of Datasets

Dataset $\mathcal{D}$	$\lvert\mathcal{D}\rvert$	Avg. $\lvert\mathcal{V}\rvert$	Avg. $\lvert\mathcal{E}\rvert$	$\lvert\Sigma_{\mathcal{V}}\rvert$	$\lvert\Sigma_{\mathcal{E}}\rvert$
LINUX	1,000	7.6	6.9	1	1
IMDB	1,500	13.0	65.9	1	1
AIDS700	700	8.9	8.8	29	1
AIDS	42,689	25.6	27.5	66	3
PubChem	23,903	48.3	50.8	10	3

4.2.2 Graph Edit Vector (GEV) Estimation. After we get the type-aware similarity embeddings for the input graph pair, several fully connected neural networks (NNs) are constructed as predictors in order to estimate the GED between graphs. Each type-aware predictor targets on one type of graph edit operations corresponding to a dimension of GEV, and all the four predicted scores are aggregated as the final estimated GED score, $\widetilde{GED}$. During the training of TaGSim, the output of the predictors are compared against the type-wise ground-truth GED scores based on the *mean squared error* (mse) loss function as follows

$$\mathcal{L}_t = \frac{1}{|\mathcal{D}|} \sum_{(\mathcal{G}_i, \mathcal{G}_j) \in \mathcal{D}} (\hat{s}_t(\mathcal{G}_i, \mathcal{G}_j) - s_t(\mathcal{G}_i, \mathcal{G}_j))^2 \tag{10}$$

and the overall loss for all different types of graph edit operations is

$$\mathcal{L} = \sum_{t \in \mathcal{T}} \mathcal{L}_t \tag{11}$$

where $\mathcal{T} = \{NR, NID, EID, ER\}$ is the set of all graph edit types; $\mathcal{D}$ is the training set of graph pairs; $s_t(\mathcal{G}_i, \mathcal{G}_j)$ denotes the ground-truth GED score *w.r.t.* the graph edit type t between $\mathcal{G}_i$ and $\mathcal{G}_j$, which is typically transformed to $s_t(\mathcal{G}_i, \mathcal{G}_j) = \exp(-2 \cdot \text{GEV}(\mathcal{G}_i, \mathcal{G}_j)[t] / (|\mathcal{G}_i| + |\mathcal{G}_j|))$ for the ease of computation; $\hat{s}_t$ is the output of the predictor for the graph edit type t.

4.3 Efficient Training for TaGSim

4.3.1 Graph Pair Generator. Existing GNN-based GED computation models rely heavily on the availability of training graph pairs together with their ground truth GEDs. However, getting the exact GED between graphs is very time-consuming, or even intractable especially for large graphs [33, 49]. In addition, the state-of-the-art, exact GED methods [10, 33] return only an overall GED score for a given graph pair, while not bookkeeping the fine-grained values for each graph edit type in GEV. As a result, this global GED score cannot be readily used while training our type-aware model, TaGSim.

We thus propose a graph pair generator that constructs graph pairs with fine-grained, type-aware GED values to facilitate the training of TaGSim, as presented in Algorithm 2. Given as input a graph $\mathcal{G}$ and a target GEV, the algorithm generates a synthetic graph $\mathcal{G}^s$ such that $GEV(\mathcal{G}, \mathcal{G}^s)$ conforms to the target GEV. Specifically, starting from $\mathcal{G}$, we apply each type of graph edit operations upon $\mathcal{G}$ according to the targe GEV. For example, assume $GEV[NR] = 2$. We thus execute *NR* operations twice at random upon $\mathcal{G}$ (Lines 2 - 5). This way, the resultant $GEV(\mathcal{G}, \mathcal{G}^s)$ is the same as the target GEV. Our graph pair generator has the following clear advantages: (1) it can efficiently produce as much training data as possible that can train TaGSim with sufficient configurations for GEVs, thus solving the data scarcity issue for GNN training; (2) it totally avoids the costly process of exact GED computation, thus making the training of TaGSim efficient and cost-effective in practice.

4.3.2 Computational Complexity of TaGSim. For the first, type-aware graph embedding stage, according to [2, 24], the time complexity of computing Equation 3 and Equation 6 for graph embeddings within each iteration is $O(|\mathcal{E}|)$ and $O(|\mathcal{V}|)$, respectively. For the second, type-aware similarity computation stage, the time complexity of graph pair interaction and GEV prediction is $O((d_n^2 + d_e^2)L)$, where L is the output dimension in Equation 9, $d_n = O(|\Sigma_{\mathcal{V}}|)$, and $d_e = O(|\Sigma_{\mathcal{E}}|)$.

5 EXPERIMENTS

In this section, we report our experimental studies and main findings for GED-based graph similarity computation in real-world graphs. We implement our type-aware graph similarity learning and computation method, TaGSim, and compare it with state-of-the-art GED solutions toward answering the following questions: (1) Is the type-aware graph embedding and learning approach effective for GED estimation? (2) Is TaGSim more efficient than state-of-the-art solutions for GED computation? (3) Can our graph pair generator produce high-quality training data for deep learning based GED solutions? All our experiments have been carried out on a Linux machine running Ubuntu Server 20.04 with two Intel 2.3GHz ten-core CPUs and 256GB memory.

5.1 Datasets and Preprocessing

We consider in our experimental studies five real-world graph datasets, whose key statistics are reported in Table 2, including the dataset size $|\mathcal{D}|$, the graph sizes in terms of the average number of nodes (avg. $|\mathcal{V}|$) and the average numbers of edges (avg. $|\mathcal{E}|$), the size of the node label set $|\Sigma_{\mathcal{V}}|$ and the size of the edge label set $|\Sigma_{\mathcal{E}}|$:

- **LINUX** [2, 45] consists of 48,747 program dependence graphs generated from the Linux kernel. Each graph in the dataset represents a kernel function with nodes as code statements and edges illustrating statement dependencies. Following [2], we sample 1,000 graphs with ten or fewer nodes as the Linux dataset. Both nodes and edges of the graphs are unlabeled: $|\Sigma_{\mathcal{V}}| = |\Sigma_{\mathcal{E}}| = 1$;
- **IMDB** [2, 48] comprises 1,500 graphs, each of which represents an ego-network of movie actors and actresses. An edge in the graph indicates two individuals co-occur in the same movie. Both the nodes and edges of the graphs are unlabeled;
- **AIDS** [10, 18, 51] is an antivirus screen chemical dataset released by the Developmental Therapeutics Program at NCI/NIH[2]. It contains 42,689 graphs with labeled nodes and edges;
- **AIDS700** [2] is a subset of the AIDS dataset with 700 graphs, each of which has at most ten nodes. Different from AIDS where graphs are edge-labeled, the edges of the graphs in AIDS700 are unlabeled;
- **PubChem** [10, 18, 51] is a chemical compound dataset in the PubChem project[3]. It consists of 23,903 graphs, which are both node-labeled and edge-labeled.

Training Datasets. In the training phase for the GNN based methods, including TaGSim, the training data is generated by the graph pair generator (Section 4.3). Specifically, we select 80% of graphs from each dataset as the input to the graph pair generator. For each graph g in a dataset, the graph pair generator produces a certain number (ranging from 20 to 1,500) of counterpart graphs g', based on different GEVs, $GEV(g, g')$, as training data. In the LINUX dataset, the training dataset contains 800× 1,000 graph pairs; In IMDB, the training dataset size is 1,200 × 1,500; In AIDS700, the training dataset size is 560 × 1,120; In AIDS, the training dataset size is 34,151 × 20; In PubChem, the training dataset size is 19,122 × 20.

Test Datasets. In the test phase, we select test graph pairs from the remaining 20% data in each graph dataset. There are two different types of test graph pairs. The first type, termed *synthetic test set*, consists of graph pairs generated from the graph pair generator in a similar way as training graph-pair generation: In LINUX, the size of synthetic test set is 200 × 1,000 (each test graph spawns 1,000 synthetic graph pairs); In IMDB, the size is 300 × 1,500; In AIDS700, the size is 140 × 700; In AIDS, the size is 8,538 × 1,000; In PubChem, the size is 4,781 × 500. The second type of test set, termed *real test set*, consists of graph pairs directly from the graph dataset. In this category, the exact GED between test graph pairs has to be computed based on the best-known exact GED computation method, AStar$^+$-LSa [10]. In LINUX, there are 200 × 1,000 real graph pairs; In IMDB, the real test set size is 300 × 1,500; In AIDS700, the size is 140 × 700; In AIDS, we select twenty test graphs, each of which is paired with 800 graphs from AIDS, and the real test size is 20 × 800. Note that we do not consider real test sets for PubChem because exact GED computation based on AStar$^+$-LSa is simply intractable for the large graphs in PubChem.

[2]https://cactus.nci.nih.gov/download/nci/AID2DA99.sdz. (Last visited: Oct. 2021)
[3]http://pubchem.ncbi.nlm.nih.gov (Last visited: Oct. 2021)

5.2 Experimental Settings

Baseline Methods. We consider eight existing GED computation methods as the baselines to be compared with TaGSim in our experimental studies. Such baselines are broadly classified into two categories:

(1) **Combinatorial Search-based Methods.** Methods in this category aim at exploiting some combinatorial structures or GED lower-bounds for GED approximation. Given a pair of graphs, the results returned by these methods are estimated GEDs *without* exact GED verification. The representative methods in this category include (1) *A*-Beam* [29], an algorithm following the beam search strategy, which takes sub-exponential time for GED computation; (2) *Hungarian* [25, 32], a cubic-time algorithm based on the Hungarian method for weighted graph matching; (3) *VJ* [12, 20], a cubic-time algorithm based on bipartite graph matching; (4) *Inves* [22], a GED lower-bound based algorithm that leverages a partition-based lower bound based on the number of disjoint mismatching substructures between graphs for false-positive graph identification and pruning.

(2) **GNN-based Learning Methods.** Methods in this category take advantage of graph neural networks for GED estimation. These GNN based models are required to be trained before test graph pairs are investigated for GED learning and computation: *SimGNN* [2] is the most up-to-date method in this category, and therefore the main competitor of our proposed method, TaGSim. SimGNN derives node embeddings from the input graphs, and leverages the attention mechanism to readout graph embeddings for GED computation; *SimpleSum* [2] is a variant of *SimGNN*, which generates graph embeddings based on the summation of node embeddings; *AttDegree* [2] is another variant of *SimGNN*, which applies the attention mechanism based on node degrees as the attention weights. *HierarchicalMax* [11] adopts the max pooling functions to generate graph embeddings. Specifically, for each dimension, *HierarchicalMax* selects maximum values among all the node embedding vectors in order to construct graph embeddings.

Evaluation Methods. For GNN-based learning methods, including our solution, TaGSim, we first use the training datasets to train the corresponding GNN models, which can be done offline. For each GED computation method, we examine every test graph pair $(\mathcal{G}, \mathcal{G}')$ from the test datasets, including both the synthetic test sets and the real test sets, to estimate their GED value, $\widetilde{GED}(\mathcal{G}, \mathcal{G}')$.

We consider the following experimental metrics to evaluate the effectiveness of different GED computation techniques: (1) *mean squared error* (***mse***), which measures the average discrepancy between the exact and the estimated GEDs for the test graph pairs in the test sets; (2) *Spearman's Rank Correlation Coefficient* (ρ) [40], and (3) *Kendall's Rank Correlation Coefficient* (τ) [21], both of which evaluate how well the estimated GED ranking results match the true ranking results. Specifically, for each test graph $\mathcal{G}$, we estimate GEDs for the test graph pairs $(\mathcal{G}, \mathcal{G}_{cand})$ where $\mathcal{G}_{cand}$ is a graph from the synthetic/real test sets. We then rank all such graphs, $\mathcal{G}_{cand}$, based on a nondecreasing order of their GED similarity scores *w.r.t.* $\mathcal{G}$; (4) *Precision at k* (p@**k**), which computes the ratio of estimated top-k similar graphs within the ground truth top-k similar results ($k = 20$ by default). For the GED estimation efficiency, we primarily compare the runtime cost of TaGSim against that of *SimGNN*, the state-of-the-art GNN-based solution, and that of *AStar$^+$-LSa* [10], the currently best-known exact GED computation algorithm.

Table 3: GED Estimation Effectiveness on Synthetic Test Sets.

Datasets		A*-Beam	Hungarian	VJ	Inves	SimpleSum	Hier-Max	AttDegree	SimGNN	**TaGSim**
LINUX	mse(10^{-3})	29.097	18.172	26.830	26.348	3.642	8.124	3.252	3.553	**1.249**
	ρ	0.878	0.646	0.540	0.350	0.867	0.787	0.889	0.864	**0.950**
	τ	0.776	0.495	0.405	0.233	0.682	0.615	0.758	0.678	**0.844**
	p@20	0.809	0.439	0.090	0.843	0.203	0.302	0.412	0.593	**0.887**
IMDB	mse(10^{-3})	24.056	29.229	99.695	21.023	2.008	2.488	1.730	5.747	**0.911**
	ρ	0.916	0.602	0.674	0.415	0.863	0.835	0.921	0.652	**0.959**
	τ	0.870	0.576	0.601	0.267	0.652	0.673	0.783	0.481	**0.870**
	p@20	**0.790**	0.769	0.575	0.011	0.341	0.344	0.407	0.508	0.745
AIDS700	mse(10^{-3})	39.593	21.830	10.806	76.475	1.041	1.087	1.025	1.159	**0.945**
	ρ	0.770	0.672	0.599	0.555	0.881	0.876	0.881	0.867	**0.885**
	τ	0.661	0.530	0.451	0.420	0.720	0.714	0.729	0.702	**0.734**
	p@20	0.577	0.538	0.317	0.314	0.641	0.632	0.631	0.604	**0.655**
AIDS	mse(10^{-3})	21.427	24.363	110.882	48.782	3.776	5.013	5.499	3.650	**1.472**
	ρ	0.614	0.453	0.535	0.270	0.652	0.613	0.656	0.645	**0.812**
	τ	0.468	0.340	0.384	0.175	0.473	0.441	0.477	0.468	**0.692**
	p@20	0.545	0.485	0.085	0.110	0.251	0.223	0.274	0.240	**0.558**
PubChem	mse(10^{-3})	20.502	13.586	148.474	28.288	8.305	4.817	3.140	8.537	**0.822**
	ρ	0.636	0.641	0.425	0.517	0.401	0.595	0.645	0.411	**0.868**
	τ	0.527	0.515	0.306	0.496	0.377	0.423	0.465	0.379	**0.702**
	p@20	0.468	0.542	0.089	0.175	0.050	0.224	0.275	0.051	**0.586**

Table 4: GED Estimation Effectiveness on Real Test Sets.

Datasets		A*-Beam	Hungarian	VJ	Inves	SimpleSum	Hier-Max	AttDegree	SimGNN	**TaGSim**
LINUX	mse(10^{-3})	9.268	29.805	63.863	37.894	12.701	20.729	12.836	12.840	**5.278**
	ρ	0.827	0.638	0.581	0.536	0.889	0.689	0.840	0.884	**0.941**
	τ	0.714	0.517	0.450	0.454	0.757	0.536	0.723	0.753	**0.834**
	p@20	**0.924**	0.836	0.251	0.848	0.275	0.072	0.053	0.193	0.867
IMDB	mse(10^{-3})	**13.412**	15469	15.760	45.280	98.600	105.419	88.993	77.871	35.690
	ρ	0.893	0.863	0.879	0.870	0.836	0.116	0.671	0.715	**0.958**
	τ	0.853	0.796	0.810	0.713	0.759	0.109	0.661	0.626	**0.926**
	p@20	0.933	0.936	0.937	0.533	0.907	0.344	0.289	0.886	**0.986**
AIDS700	mse(10^{-3})	12.090	25.296	29.157	38.594	21.983	26.526	17.311	14.769	**5.890**
	ρ	0.609	0.510	0.517	0.129	0.175	0.475	0.396	0.534	**0.679**
	τ	0.463	0.378	0.383	0.112	0.125	0.378	0.285	0.397	**0.520**
	p@20	**0.493**	0.392	0.345	0.079	0.015	0.086	0.075	0.024	0.266
AIDS	mse(10^{-3})	456.203	436.020	456.500	22.196	96.824	75.495	105.457	84.331	**9.827**
	ρ	0.220	0.233	0.229	0.563	0.294	0.359	0.350	0.374	**0.688**
	τ	0.201	0.214	0.211	0.499	0.206	0.253	0.246	0.266	**0.527**
	p@20	0.049	0.059	0.022	0.188	0.067	0.008	0.083	0.022	**0.322**

Model Settings in TaGSim. In TaGSim, we use type-aware graph embeddings to estimate each dimension of GEV. In our experimental studies, if not specified otherwise, we consider $\mathbf{h}^{n}_{cat(0,1)}$ (Equation 8) as the type-aware graph embedding for the graph edit type *NR*; For the graph edit type *NID*, we consider $\mathbf{h}^{n}_{cat(0,1)}$; For the graph edit type *EID*, we consider $\mathbf{h}^{n}_{cat(1,2)}$; For the graph edit type *ER*, we consider $\mathbf{h}^{e}_{cat(0,1)}$ if the graphs are edge-labeled. In addition, we also conduct experiments to explore the effects of different combinations of type-aware graph embeddings, and the results are reported in Section 5.3.3.

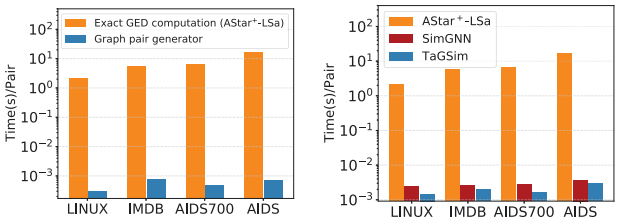
For the pooling function (Equation 8), we select $\mathsf{sum}(\cdot)$ as the default function in TaGSim. We also consider other pooling functions and discuss their impacts in Section 5.3.4. For the setting of the NTN network (Equation 9), we consider $L = 16$ and the ReLU function [28] for σ by default. The GEV predictors consist of three layers of neural networks with 16, 8, 4 neurons in each layer, respectively, and the output is a scalar value. During the training of TaGSim, we set the batch size to be 128, and Adam [23] is chosen as the optimizer for back propagation with the learning rate of 0.001. The parameter studies of TaGSim are reported in Section 5.3.5.

5.3 Experimental Results

5.3.1 GED Estimation Effectiveness. The results for GED estimation effectiveness of nine different methods are tabulated in Table 3 and Table 4 for synthetic test sets and real test sets, respectively. On all the different datasets, TaGSim consistently achieves the best performance under virtually every evaluation metric. In particular, when compared with the most up-to-date GNN-based method, *SimGNN*, TaGSim achieves notable performance gains

Table 5: GED Estimation Effectiveness: Type-oblivious vs. Type-aware

Methods	LINUX				IMDB				AIDS700				AIDS			
	mse	ρ	τ	p@20	mse	ρ	τ	p@20	mse	ρ	τ	p@20	mse	ρ	τ	p@20
Type-oblivious	13.771	0.888	0.760	0.355	93.565	0.803	0.739	0.754	13.728	0.567	0.422	0.151	75.877	0.413	0.322	0.049
Type-aware	**5.278**	**0.941**	**0.834**	**0.867**	**35.690**	**0.958**	**0.926**	**0.986**	**5.890**	**0.679**	**0.520**	**0.266**	**9.827**	**0.688**	**0.527**	**0.322**



(a) Training Data Acquisition Time (b) Graph Similarity Computation Time

Figure 4: Training and GED Estimation Efficiency.

for GED learning and computation: on the LINUX and IMDB datasets, the mean squared errors (mse) of TaGSim are only about 15.85% to 35.15% of those in *SimGNN*; On AIDS and PubChem, TaGSim's mse results are less than 10% of the results of *SimGNN*. Regarding other rank-based evaluation metrics (ρ, τ, and p@20), TaGSim achieves 30% to 12X effectiveness improvement in comparison with *SimGNN*, especially on AIDS and PubChem datasets where graphs are large and structurally complex. Such significant performance gains are primarily attributed to the type-aware GED learning idea and the computation framework, TaGSim, in which each type of graph edit operations can be modeled, learned, and estimated in a fine-grained approach, thus resulting in more accurate GED estimation results than all existing type-oblivious approaches.

When compared with the combinatorial search-based solutions, TaGSim achieves more significant performance gains especially in AIDS and PubChem datasets where graphs are structurally more complicated and of larger sizes. Specifically, TaGSim outperforms the state-of-the-art, partition-based GED lower-bound method, *Inves*, in terms of all the evaluation metrics. The performance gains *w.r.t.* mse can be one to two orders of magnitude in AIDS and PubChem datasets. We also recognize that existing GED lower bounds in *Inves* are still loose: In many experimental scenarios, *Inves* fails to deliver accurate GED estimations for real-world graphs.

As a consequence, TaGSim can accurately estimate GED for real-world graphs. It has demonstrated competitive performance gains and clear advantages in GED learning and computation, compared with the state-of-the-art GED solutions.

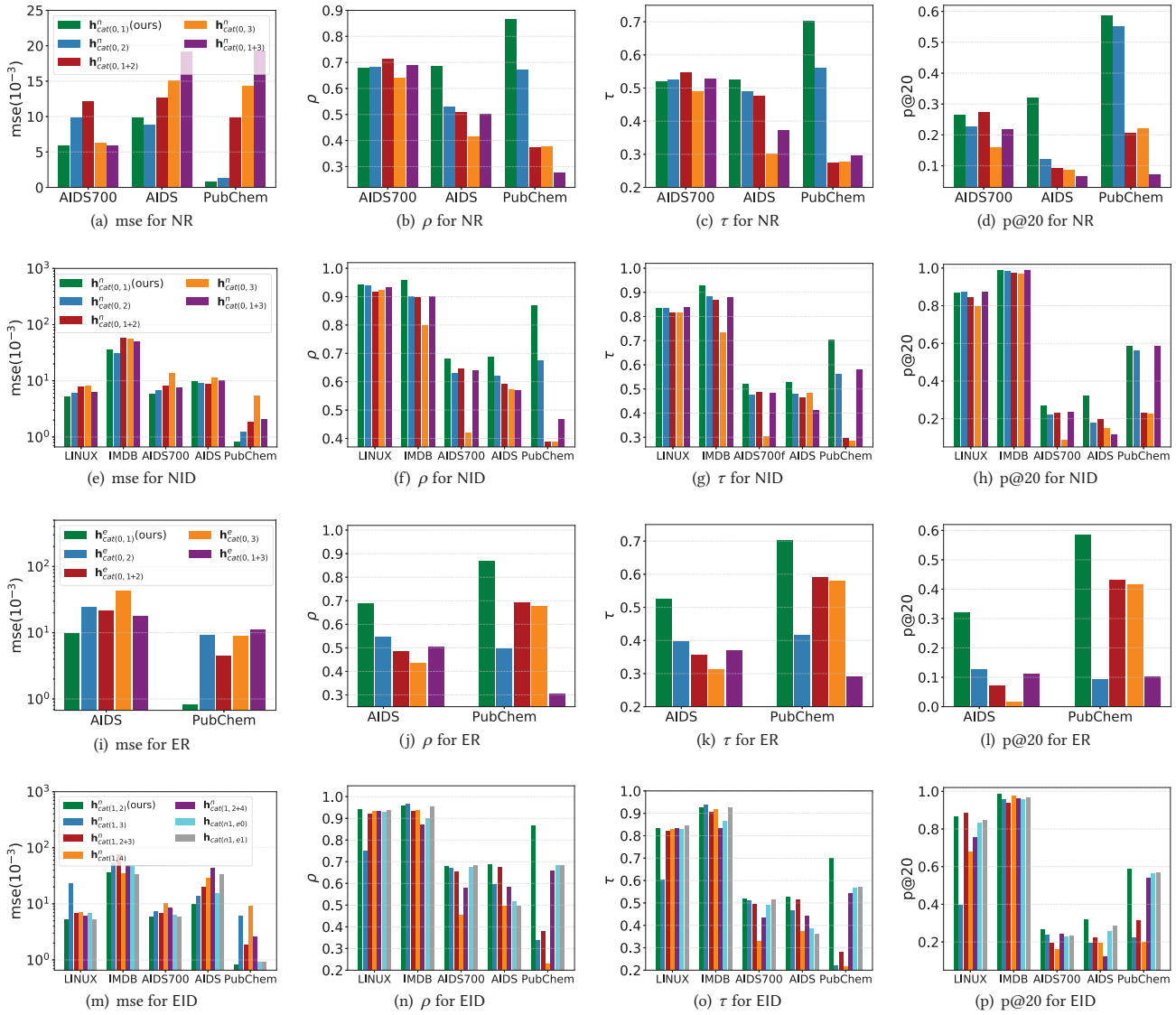
5.3.2 Model Training and GED Estimation Efficiency. We further examine the runtime cost of training data acquisition and GED learning and computation for TaGSim, and the results on the real test sets are presented in Figure 4. In Figure 4(a), we report the average runtime cost for graph pair generation, as proposed in Section 4.3, and the average runtime cost of *AStar*$^+$*-LSa* for exact GED computation. It is worth noting that, the runtime of the graph pair generator is three orders of magnitude less than that of *AStar*$^+$*-LSa*, which is very time-consuming in the experiments. Specifically, in the AIDS dataset, it typically takes at least 10 seconds for *AStar*$^+$*-LSa* to compute the exact GED for a given graph pair as the ground-truth GED. In order to train the GNN-based models with thousands of graph pairs together with their exact GED scores, the runtime cost for the training data acquisition can be exceptionally high. In contrast, our graph pair generator (Algorithm 2) is very efficient, which can help train and deploy TaGSim fast in practice.

We further report the average runtime cost for GED computation by different methods, and the results are presented in Figure 4(b). We recognize that TaGSim is 20% up to 50% faster than *SimGNN* for GED learning and estimation in different datasets. The primary reason is that, the type-aware graph embeddings of TaGSim do not need linear and non-linear transformations for node and graph embeddings, which, however, are required steps in *SimGNN* and other GNN-based methods. In other words, TaGSim has a simplified graph embedding framework, thus leading to an accelerated GED learning process. Furthermore, TaGSim is three orders-of-magnitude faster than the exact GED method, *AStar*$^+$*-LSa*, in all the datasets. When computing GEDs between large graphs in the AIDS dataset, we recognize a significant slowdown in *AStar*$^+$*-LSa*, whereas TaGSim can work efficiently especially for real-world, large graphs.

In summary, TaGSim is significantly more efficient than the exact GED computation methods. Furthermore, it outperforms the state-of-the-art, GNN-based solution, *SimGNN*, in terms of GED estimation efficacy, in real-world datasets due in particular to its simplified, type-aware graph embedding framework.

5.3.3 Effectiveness of Type-aware Graph Embedding and Learning. In this experiment, we substitute the type-aware graph embedding and learning components in TaGSim with the traditional, type-oblivious GNN structures employed by *SimGNN*, which consist of three graph convolutional network (GCN) layers and an extra attention layer. We denote this method Type-oblivious, as different types of graph edit operations are modeled and learned indiscriminately in this framework. We compare Type-oblivious with TaGSim (denoted as Type-aware in this setting), and the effectiveness results for GED estimation on real test sets are shown in Table 5. It is interesting to note that, Type-aware achieves consistently better effectiveness results for GED estimation than Type-oblivious on all datasets. The primary reason is that our type-aware graph embedding and learning modules are designed to identify and encode different transformative impacts incurred by different types of graph edit operations. Consequently, Type-aware provides fine-grained, and more accurate prediction models than Type-oblivious. In addition, the type-aware graph embeddings of TaGSim involve no extra hyper-parameters to be learned, which, however, are required in *SimGNN*. As a result, the type-aware solution, TaGSim, outperforms the type-oblivious method, *SimGNN*, in terms of both the GED estimation effectiveness and efficiency.

To systematically study the performance of type-aware graph embeddings, we consider different k-hop NLMs and ELMs (by varying the values of k), together with their possible combinations in TaGSim, and examine the estimation effectiveness for different graph edit types. The effectiveness results are illustrated in Figure 5. For instance, for the graph edit type *NR* with the default setting $\mathbf{h}^n_{cat(0,1)}$, we consider graph embeddings leveraging further neighborhood information with $k = 2, 3$; that is, $\mathbf{h}^n_{cat(0,2)}$, $\mathbf{h}^n_{cat(0,3)}$, $\mathbf{h}^n_{cat(0,1+2)}$, and $\mathbf{h}^n_{cat(0,1+3)}$ (here $\mathbf{h}^n_{cat(0,1+2)} = \text{Pooling}(\mathbf{H}^n_{cat(0,1+2)})$, where $\mathbf{H}^n_{cat(0,1+2)} = \mathbf{H}^n_0 || (\mathbf{H}^n_1 + \mathbf{H}^n_2)$). For *EID*, according to Table 1, we note that only this graph edit type can affect both $\mathbf{H}^n_k (k \geq 1)$ and $\mathbf{H}^e_k (k \geq 0)$. We thus explore different combinations of $\mathbf{H}^n_k$ and $\mathbf{H}^e_k$ for the estimation of *EID*. While extending the graph embeddings for one graph edit type, we keep the embeddings for other types of graph edit operations with the default settings. From the experimental results as reported in Figure 5, we note



(a) mse for NR (b) ρ for NR (c) τ for NR (d) p@20 for NR

(e) mse for NID (f) ρ for NID (g) τ for NID (h) p@20 for NID

(i) mse for ER (j) ρ for ER (k) τ for ER (l) p@20 for ER

(m) mse for EID (n) ρ for EID (o) τ for EID (p) p@20 for EID

Figure 5: TaGSim is configured with different k-hop NLMs and ELMs for type-aware graph embedding. The "ours" in each figure (colored in green) denotes the default settings for TaGSim. In each figure, the symbol $\mathbf{h}^n_{cat(0,1+2)} = \text{Pooling}(\mathbf{H}^n_{cat(0,1+2)})$, where $\mathbf{H}^n_{cat(0,1+2)} = \mathbf{H}^n_0 || (\mathbf{H}^n_1 + \mathbf{H}^n_2)$. Other terms, such as $\mathbf{h}^n_{cat(0,1+3)}$, $\mathbf{h}^e_{cat(0,1+2)}$, are defined and computed analogously. Note that the four figures of each row share the same legends, which are only shown in the first figure of that row, for the sake of clarity.

that in almost all different datasets, our default settings for type-aware graph embeddings can achieve the most satisfactory effectiveness results. For instance, in Figure 5(i), the GED estimation effectiveness of $\mathbf{h}^n_{cat(0,1)}$ for the edit type *ER* is better than those of $\mathbf{h}^n_{cat(0,1+2)}$ and $\mathbf{h}^n_{cat(0,1+3)}$ (in terms of mse, the lower the better). In practice, with at most two layers ($k \leq 1$) of GAL computation, we can obtain sufficiently good type-aware graph embeddings in TaGSim for GED computation.

5.3.4 Performance of Different Pooling Functions. In this experiment, we consider different pooling functions to study their potential effects on TaGSim. As mentioned in Section 4.1.2, besides the default setting of the pooling function, sum$(\cdot)$, we also consider mean$(\cdot)$ and max$(\cdot)$ (select the maximum value for each dimension of node embedding vectors as the resultant dimension of the graph embedding), and the effectiveness results on real test sets are presented in Figure 6. Note that TaGSim with the sum pooling function consistently achieves the best performance in terms of all the evaluation metrics. The reason is that sum can easily maintain the graph size information (the numbers of nodes and edges), while mean and max cannot. Additionally, in type-aware graph embedding, sum helps preserve the neighborhood information within k-hop NLMs and ELMs, thus making it an ideal pooling function in TaGSim.

5.3.5 Parameter Sensitivity. In TaGSim, the neural tensor networks have the hyper-parameter L that controls the output dimensions (Section 4.2.1).

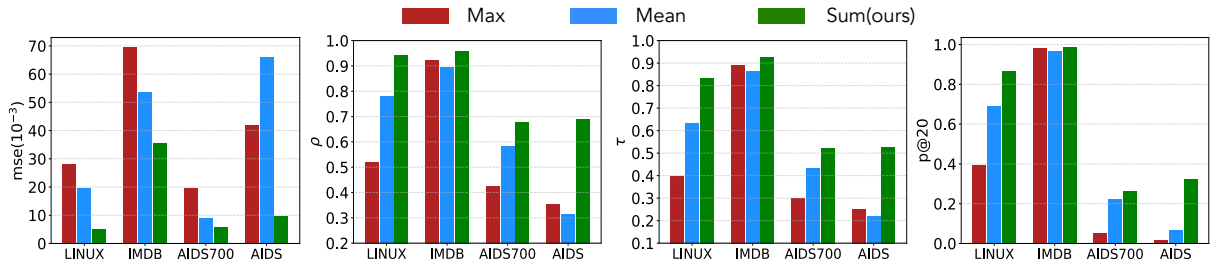


Figure 6: GED estimation effectiveness for TaGSim with different pooling functions. "Ours" in the figures denotes the default setting, Sum(·).

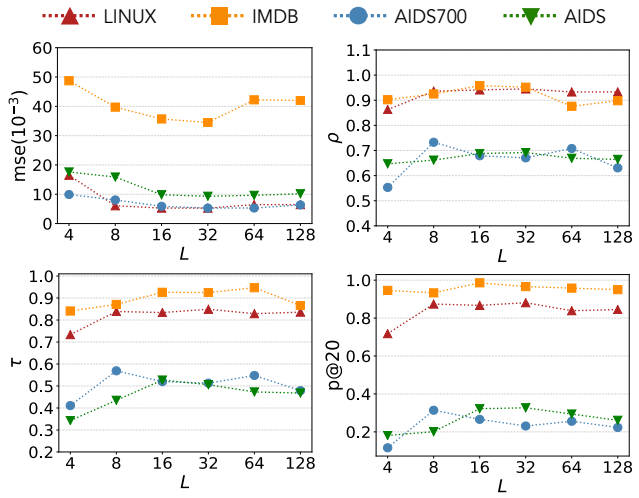


Figure 7: TaGSim with different output dimensions (L) in neural tensor networks (NTU). The default setting is $L = 16$.

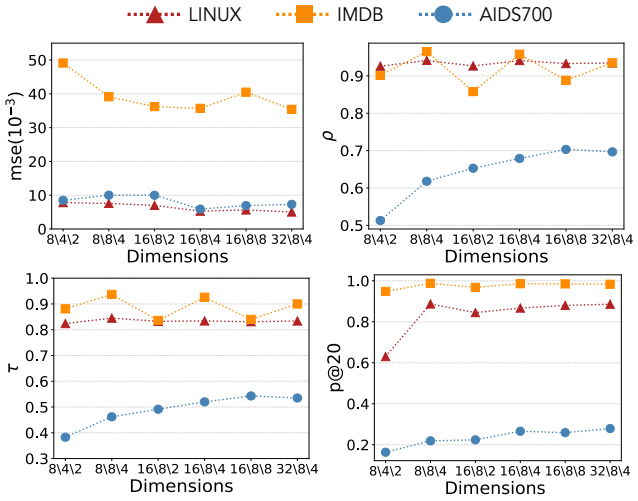


Figure 8: TaGSim with different dimensions for three hidden layers. The default setting of TaGSim is "16\8\4".

We examine the performance of TaGSim by setting different values of L, and the effectiveness results are presented in Figure 7. We note that, by increasing the values of L starting from 4, the performance of TaGSim keeps improving and converges to a high level when $L = 16$ or 32. When the values of L keep increasing (*e.g.*, $L = 128$), the performance of TaGSim deteriorates slightly because there are more learnable weights or parameters in TaGSim, resulting in a higher probability of overfitting. It turns out that TaGSim with $L = 16$ or 32 typically achieves the best performance under different evaluation metrics.

In TaGSim, the neural network based GEV predictors consist of three fully connected layers. In this experiment, we fine tune the dimension for each of the three hidden layers, represented in the form of $dim(L_1) \setminus dim(L_2) \setminus dim(L_3)$, and the results for GED estimation effectiveness are presented in Figure 8. Note that the estimation effectiveness of TaGSim improves when the dimensions of hidden layers increase, and it converges fast under different evaluation metrics in different datasets. Empirically, the dimension of the last hidden layer cannot be set too low (*e.g.*, smaller than or equal to 2). Typically $dim(L_3) = 4$ is sufficiently good for GED estimation in different graph datasets.

6 CONCLUSION

The graph edit distance (GED) based similarity computation has been critical and fundamental in a proliferation of real-world, graph-based applications. In this paper, we proposed a new concept of type-aware graph similarity learning and computation, where each type of graph edit operations was modeled, learned, and estimated in a type-aware, fine-grained approach. Following this concept, we proposed a novel graph similarity learning and computation framework, TaGSim, consisting of two deep learning based functional modules: (1) type-aware graph embedding and (2) type-aware similarity learning and estimation. To facilitate the training of TaGSim, we also designed a cost-effective graph pair generator that can provide sufficient graph pairs with fine-grained, ground truth GEDs without recourse to the costly exact GED computation. Our experimental studies have validated the efficiency and effectiveness of TaGSim on five real-world datasets, compared with eight existing GED computation solutions.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF Grant No. 1743142), the Air Force Office of Scientific Research (AFOSR Award No. FA95501810106), and the Army Research Office (ARO Award No. W911NF1810395). Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the funding agencies.

REFERENCES

[1] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. 2019. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *International Conference on Machine Learning (ICML'19)*. 21–29.

[2] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A neural network approach to fast graph similarity computation. In *the Twelfth ACM International Conference on Web Search and Data Mining (WSDM'19)*. 384–392.

[3] David B. Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. 2020. Comparing heuristics for graph edit distance computation. *VLDB J.* 29, 1 (2020), 419–458.

[4] David B. Blumenthal and Johann Gamper. 2018. Improved Lower Bounds for Graph Edit Distance. *IEEE Trans. Knowl. Data Eng.* 30, 3 (2018), 503–516.

[5] David B Blumenthal and Johann Gamper. 2020. On the exact computation of the graph edit distance. *Pattern Recognition Letters* 134 (2020), 46–57.

[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *2nd International Conference on Learning Representations (ICLR'14)*.

[7] Horst Bunke. 1983. What is the distance between graphs. *Bulletin of the European Association for Theoretical Computer SCience* 20 (1983), 35–39.

[8] Horst Bunke. 1997. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters* 18, 8 (1997), 689–694.

[9] Lei Cai, Jundong Li, Jie Wang, and Shuiwang Ji. 2021. Line graph neural networks for link prediction. *IEEE Transactions on Pattern Analysis & Machine Intelligence* (2021), 1–1.

[10] Lijun Chang, Xing Feng, Xuemin Lin, Lu Qin, Wenjie Zhang, and Dian Ouyang. 2020. Speeding Up GED Verification for Graph Similarity Search. In *IEEE 36th International Conference on Data Engineering (ICDE'20)*. 793–804.

[11] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 3844–3852.

[12] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. 2011. Speeding up graph edit distance computation through fast bipartite matching. In *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, 102–111.

[13] Mirtha-Lina Fernández and Gabriel Valiente. 2001. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters* 22, 6-7 (2001), 753–758.

[14] Andreas Fischer, Ching Y. Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. 2015. Approximation of Graph Edit Distance Based on Hausdorff Matching. *Pattern Recogn.* 48, 2 (2015), 331–343.

[15] George Fletcher, Jan Hidders, and Josep Llus Larriba-Pey. 2018. *Graph Data Management: Fundamental Issues and Recent Developments*. Springer Inc.

[16] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A Survey of Graph Edit Distance. *Pattern Anal. Appl.* 13, 1 (2010), 113–129.

[17] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[18] Karam Gouda and Mosab Hassaan. 2016. CSI_GED: An efficient approach for graph edit similarity computation. In *IEEE 32nd International Conference on Data Engineering (ICDE'16)*. 265–276.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *the 31th International Conference on Neural Information Processing Systems (NIPS'17)*. 1025–1035.

[20] Roy Jonker and Anton Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* 38, 4 (1987), 325–340.

[21] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[22] Jongik Kim, Dong-Hoon Choi, and Chen Li. 2019. Inves: Incremental Partitioning-Based Verification for Graph Similarity Search. In *22nd International Conference on Extending Database Technology (EDBT'19)*. 229–240.

[23] Diederik P. Kingma and Jimmy Lei Ba. 2015. ADAM: A method for stochastic optimization. In *3th International Conference on Learning Representations (ICLR'15)*.

[24] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *the 5th International Conference on Learning Representations (ICLR'17)*.

[25] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[26] Yongjiang Liang and Peixiang Zhao. 2017. Similarity search in graph databases: A multi-layered indexing approach. In *IEEE 33rd International Conference on Data Engineering (ICDE)*. 783–794.

[27] Yao Ma, Suhang Wang, Charu C. Aggarwal, and Jiliang Tang. 2019. Graph Convolutional Networks with EigenPooling. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'19)*. 723–731.

[28] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning (ICML'10)*. 807—-814.

[29] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. 2006. Fast suboptimal algorithms for the computation of graph edit distance. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 163–172.

[30] Zongyue Qin, Yunsheng Bai, and Yizhou Sun. 2020. GHashing: Semantic Graph Hashing for Approximate Similarity Search in Graph Databases. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'20)*. 2062–2072.

[31] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. Gcc: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'20)*. 1150–1160.

[32] Kaspar Riesen and Horst Bunke. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing* 27, 7 (2009), 950–959.

[33] Kaspar Riesen, Sandro Emmenegger, and Horst Bunke. 2013. A novel software toolkit for graph edit distance computation. In *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, 142–151.

[34] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. 2007. Speeding Up Graph Edit Distance Computation with a Bipartite Heuristic. In *Mining and Learning with Graphs (MLG'07)*. 21–24.

[35] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. 2020. Approximate Graph Edit Distance in Quadratic Time. *IEEE ACM Trans. Comput. Biol. Bioinform.* 17, 2 (2020), 483–494.

[36] Kaspar Riesen, Andreas Fischer, and Horst Bunke. 2015. Estimating Graph Edit Distance Using Lower and Upper Bounds of Bipartite Approximations. *Int. J. Pattern Recognit. Artif. Intell.* 29, 2 (2015).

[37] Alberto Sanfeliu and King-Sun Fu. 1983. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics* 3 (1983), 353–362.

[38] Francesc Serratosa and Xavier Cortés Llosa. 2015. Graph Edit Distance: Moving from global to local structure to solve the graph-matching problem. *Pattern Recognition Letters* 65 (2015), 204–210.

[39] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. 2013. Reasoning with neural tensor networks for knowledge base completion. *Advances in neural information processing systems (NIPS'13)* 26 (2013), 926–934.

[40] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).

[41] Sousuke Takami and Akihiro Inokuchi. 2018. Accurate and Fast Computation of Approximate Graph Edit Distance based on Graph Relabeling. In *Proceedings of the 7th International Conference on Pattern Recognition Applications and Methods*. 17–26.

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *the 6th International Conference on Learning Representations (ICLR'18)*.

[43] Guoren Wang, Bin Wang, Xiaochun Yang, and Ge Yu. 2012. Efficiently Indexing Large Sparse Graphs for Similarity Search. *IEEE Trans. Knowl. Data Eng.* 24, 3 (2012), 440–451.

[44] Runzhong Wang, Tianqi Zhang, Tianshu Yu, Junchi Yan, and Xiaokang Yang. 2021. Combinatorial Learning of Graph Edit Distance via Dynamic Embedding. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'21)*. 5241–5250.

[45] Xiaoli Wang, Xiaofeng Ding, Anthony KH Tung, Shanshan Ying, and Hai Jin. 2012. An efficient graph indexing method. In *IEEE 28th International Conference on Data Engineering (ICDE'12)*. 210–221.

[46] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *the 7th International Conference on Learning Representations (ICLR'19)*.

[47] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *International Conference on Machine Learning (ICML'18)*, Vol. 80. 5449–5458.

[48] Pinar Yanardag and SVN Vishwanathan. 2015. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery & data mining (KDD'15)*. 1365–1374.

[49] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment* (2009), 25–36.

[50] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. 5171–5181.

[51] Xiang Zhao, Chuan Xiao, Xuemin Lin, Wenjie Zhang, and Yang Wang. 2018. Efficient structure similarity searches: a partition-based approach. *VLDB J.* 27, 1 (2018), 53–78.