



Computing Graph Edit Distance via Neural Graph Matching

Chengzhi Piao
The Chinese University of Hong Kong
czpiao@se.cuhk.edu.hk

Tingyang Xu
Tencent AI Lab
tingyangxu@tencent.com

Xiangguo Sun
The Chinese University of Hong Kong
xgsun@se.cuhk.edu.hk

Yu Rong
Tencent AI Lab
yu.rong@hotmail.com

Kangfei Zhao
Tencent AI Lab
z kf1105@gmail.com

Hong Cheng
The Chinese University of Hong Kong
hcheng@se.cuhk.edu.hk

ABSTRACT

Graph edit distance (GED) computation is a fundamental NP-hard problem in graph theory. Given a graph pair (G_1, G_2) , GED is defined as the minimum number of primitive operations converting G_1 to G_2 . Early studies focus on search-based inexact algorithms such as A^* -beam search, and greedy algorithms using bipartite matching due to its NP-hardness. They can obtain a sub-optimal solution by constructing an edit path (the sequence of operations that converts G_1 to G_2). Recent studies convert the GED between a given graph pair (G_1, G_2) into a similarity score in the range $(0, 1)$ by a well designed function. Then machine learning models (mostly based on graph neural networks) are applied to predict the similarity score. They achieve a much higher numerical precision than the sub-optimal solutions found by classical algorithms. However, a major limitation is that these machine learning models cannot generate an edit path. They treat the GED computation as a pure regression task to bypass its intrinsic complexity, but ignore the essential task of converting G_1 to G_2 . This severely limits the interpretability and usability of the solution.

In this paper, we propose a novel deep learning framework that solves the GED problem in a two-step manner: 1) The proposed graph neural network GEDGNN is in charge of predicting the GED value and a matching matrix; and 2) A post-processing algorithm based on k -best matching is used to derive k possible node matchings from the matching matrix generated by GEDGNN. The best matching will finally lead to a high-quality edit path. Extensive experiments are conducted on three real graph data sets and synthetic power-law graphs to demonstrate the effectiveness of our framework. Compared to the best result of existing GNN-based models, the mean absolute error (MAE) on GED value prediction decreases by 4.9% ~ 74.3%. Compared to the state-of-the-art searching algorithm Noah, the MAE on GED value based on edit path reduces by 53.6% ~ 88.1%.

PVLDB Reference Format:

Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. Computing Graph Edit Distance via Neural Graph Matching. PVLDB, 16(8): 1817 - 1829, 2023.

doi:10.14778/3594512.3594514

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594514

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/ChengzhiPiao/GEDGNN>.

1 INTRODUCTION

Graph edit distance (GED) computation is a fundamental NP-hard problem in graph theory [5] and has attracted many researchers to design classical algorithms to solve it in history. Similar to the edit distance on strings, the graph edit distance on a pair of graphs is defined as the minimum number of atomic operations that can convert one graph to the other, and the sequence of atomic operations is called a graph edit path. Figure 1(a) depicts two graphs G_1 and G_2 , and Figure 1(b) lists one of the shortest graph edit paths that converts G_1 to G_2 with 3 operations. Since GED can capture both structural and attribute similarity between a pair of graphs, and the distance is interpretable given the edit path, GED is a perfect choice to measure graph similarity [11, 19] and is widely used in graph search queries [12, 19, 29, 36–38]. However, given the difficulty of GED computation, existing studies on exact algorithms such as A^* search [4] mostly suffer from the severe problem of scalability. They are usually intractable when the graph size is larger than 16 nodes [4]. Considering both the increasing size of graphs and the increasing volume of graph databases nowadays, recent studies have paid more attention to inexact GED computation. In this case, the solution quality of an algorithm is usually measured by the gap between its output and the ground-truth value.

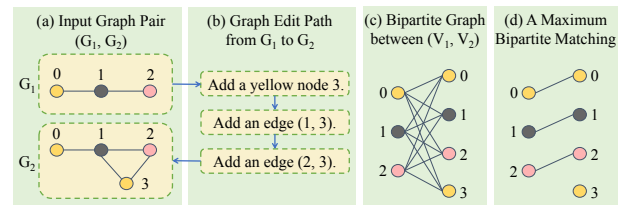


Figure 1: A Graph Edit Path for (G_1, G_2) and the Maximum Bipartite Matching used in GED Computation.

Existing approaches can be divided into three categories. 1) A^* -beam search [22] uses an extra parameter *beamsize* to control the maximum number of nodes in the queue compared to the standard A^* search algorithm and hence finds a near-optimal solution more efficiently. The effectiveness of A^* -beam search highly depends on *beamsize* and its time complexity is still exponential in theory. 2) Another line of work [10, 15, 17, 24, 25] replaces the edit distance metric by a similar greedy function, which can be easily optimized

by finding a minimum weight matching in $O(n^3)$ time (n is the number of nodes). They first invoke the greedy function to estimate the cost of matching one node in the source graph to a node in the target graph, and then simplify the calculation of GED by merely adding up the estimated cost of all matched node pairs. Compared to the exponential search space of A^* algorithm, such greedy algorithms are less expensive. Unfortunately, their effectiveness is quite low due to the limitation of hand-crafted greedy heuristics. 3) Recently, several end-to-end machine learning models [1, 2, 32] based on graph neural networks (GNN) have been proposed to predict GED. They first use a GNN to generate embeddings of the given graph pair, then combine them to capture the graph-to-graph information, e.g., generating an inter-graph embedding which is fed into a multi-layer perceptron (MLP) to obtain a real value as the predicted GED. After model training, these GNN based models can predict the GED value of a graph pair in $O(n + m)$ time (m is the number of edges) with extremely low errors. This approach treats GED computation as a regression problem, and a major limitation is that it cannot generate an edit path, which substantially reduces the interpretability and usability of the solution. In addition, the predicted GED can even be less than the true GED. In this case, the solution is called an *invisible solution* [32] and a feasible edit path for such predicted GED does not exist at all.

Different from existing approaches, we study the following problem: *Can a machine learning model predict the GED and the corresponding edit path and how?* This is much more challenging since it requires the model to deeply understand the essence of GED computation, i.e., the node matching relation between the given graph pair. Although both GED computation and graph alignment [20, 33] need to compute a node matching, these two problems are different in terms of ground truth definition, objective and matching scope. Please refer to Section 6 for a detailed discussion on the differences between GED computation and graph alignment.

In this paper, we propose a novel deep learning framework that solves the GED problem in a two-step manner. First, our proposed model, GEDGNN, not only predicts a GED value as existing machine learning models do, but also predicts a matching matrix by the cross matrix module. Specifically, each value in the matching matrix is a real number between 0 and 1, and reveals the extent to which a node pair should be matched. Second, a post-processing algorithm is used to derive multiple node matchings from the real matching matrix generated by GEDGNN, from which we generate a high-quality edit path. We creatively use the k -best matching algorithm [8] in a novel way to generate the edit path, thus make up the gap between what graph neural networks can produce (i.e., the matching matrix) and an actual solution to the GED problem. The merits of our GEDGNN model lie in the following three aspects: 1) The generated edit path clearly enhances the interpretability and usability of the solution by the machine learning approach. 2) The GED value prediction and node matching learning are seamlessly integrated into one end-to-end model. The extra information learned from node matching can in turn boost the accuracy of GED value prediction. 3) After generating an edit path by the post-processing algorithm, the path can be used to tighten the predicted GED value as the length of any edit path is an upper bound of the GED value. In summary, our main contributions are as follows.

1. We propose a novel deep neural learning framework that formulates the traditional GED computation problem as a prediction task. We introduce the mathematical constraint between bipartite node matching and the GED problem with two matrices. This mechanism enables us to obtain the predicted GED value and the corresponding node matching weights simultaneously.
2. To the best of our knowledge, we are the first to integrate the k -best matching algorithm into GED computation. We design an effective post-processing algorithm based on k -best matching to derive edit paths from the matching matrix generated by our model GEDGNN. The edit paths greatly improve the interpretability and usability of the predicted GED.
3. Extensive experiments are conducted on three real graph data sets and synthetic power-law graphs to verify the effectiveness of our algorithms. Our proposed algorithm has a far superior performance under all standards to existing methods. Compared to the best result of existing GNN-based models, the mean absolute error (MAE) on GED value prediction decreases by 4.9% ~ 74.3%. Compared to the state-of-the-art searching algorithm Noah [32], the MAE on GED value based on edit path reduces by 53.6% ~ 88.1%.

2 PROBLEM FORMULATION

2.1 Preliminary Concepts

In this paper, we compute graph edit distance between a pair of graphs (G_1, G_2) . Following the setting used in [2, 32], we use $G = (V, E, L)$ to denote an attributed undirected graph, where V , E and L denote the vertex set, edge set and node label set respectively.

DEFINITION 1. Graph Edit Distance. *Given a pair of graphs (G_1, G_2) , the minimum number of primitive operations converting G_1 to G_2 is called the graph edit distance and denoted by $GED(G_1, G_2)$. Specifically, there are three types of primitive operations: (1) adding or removing an edge; (2) adding or removing an isolated node; and (3) changing the label of a node.*

DEFINITION 2. Graph Edit Path. *Given a pair of graphs (G_1, G_2) , a sequence of primitive operations (o_1, o_2, \dots, o_k) that transforms G_1 to G_2 is called an edit path. Obviously, $k \geq GED(G_1, G_2)$ and the minimum length of an edit path is exactly the graph edit distance.*

2.2 Generating Edit Path from Node Matching

Given an edit path that converts G_1 to G_2 , there is essentially a correspondence between nodes in G_1 and G_2 , which can be modeled by bipartite matching. We introduce the concepts of bipartite graph and bipartite matching, and describe how to generate an edit path from a bipartite matching as in some previous studies [4, 22, 32].

DEFINITION 3. Bipartite Graph. *Given an undirected graph $G = (V, E)$ and a vertex partition $V = V_1 \cup V_2$ that $V_1 \cap V_2 = \emptyset$, $G = (V_1, V_2, E)$ is called a bipartite graph if $E \subseteq V_1 \times V_2$. When $E = V_1 \times V_2$, G is a complete bipartite graph which has $|V_1| \times |V_2|$ edges. $G = (V_1, V_2, E, w)$ is a weighted bipartite graph where $w : E \rightarrow \mathcal{R}$ is a weight function and $w(u, v)$ denotes the real value weight.*

DEFINITION 4. Bipartite Matching and Maximum Bipartite Matching. *Given a graph $G = (V, E)$, an edge subset $M \subseteq E$ is called a matching of G if no two edges in M share the same vertex. A matching of a bipartite graph is called a **bipartite matching**.*

Given a bipartite graph $G = (V_1, V_2, E)$, a matching M is called a **maximum bipartite matching** of G if $|M| = \min(|V_1|, |V_2|)$.

Given two graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$ for GED computation, we can construct a complete bipartite graph $G = (V_1, V_2, V_1 \times V_2)$. Then a maximum bipartite matching M on G establishes the node correspondence between G_1 and G_2 and can be used to generate an edit path as follows.

1. For any matched node pair $(u, v) \in M$, if $L(u) \neq L(v)$, change u 's label to $L(v)$. If $|V_1| < |V_2|$, there are $|V_2| - |V_1|$ nodes in G_2 unmatched and we need to add these extra nodes to G_1 ; if $|V_1| > |V_2|$, there are $|V_1| - |V_2|$ nodes in G_1 unmatched which are then removed. After this step, each node in G_1 matches a node in G_2 .

2. Assume nodes $u_1, u_2 \in V_1$ match $v_1, v_2 \in V_2$ respectively. We need to add an edge between u_1 and u_2 if $(u_1, u_2) \notin E_1$ and $(v_1, v_2) \in E_2$, or remove the edge (u_1, u_2) if $(u_1, u_2) \in E_1$ and $(v_1, v_2) \notin E_2$.

Figure 1(c) and 1(d) show the complete bipartite graph and a maximum bipartite matching respectively. The bipartite matching establishes the node correspondence between G_1 and G_2 , e.g., node 0 in G_1 corresponds to node 0 in G_2 . On top of this bipartite matching we apply the above procedure, then we can obtain the graph edit path shown in Figure 1(b).

The above procedure takes linear time, i.e., $O(n + m)$ where $n = \max(|V_1|, |V_2|)$ and $m = \max(|E_1|, |E_2|)$. In the following, we use $GED(G_1, G_2, M)$ to denote the number of edit operations to convert G_1 to G_2 based on the node matching M . Note that there are many possible matchings of G which lead to different edit paths.

With these definitions, our target problem is to design a graph neural network which predicts the graph edit distance given a pair of graphs and learns a good matching leading to a short edit path.

3 GRAPH NEURAL NETWORKS FOR GRAPH EDIT DISTANCE COMPUTATION

In this section, we design a graph neural network model, called **GEDGNN** for GED computation. As shown in Figure 2, for a given graph pair (G_1, G_2) , our model GEDGNN first generates their node embeddings H_1 and H_2 using a graph neural network. Then we use two separate cross matrix modules to capture the node-to-node correspondence between H_1 and H_2 . The cross matrix modules use trainable parameter matrices which connect H_1 and H_2 to predict the graph matching and the matching cost. One of the cross matrix modules outputs a matching matrix A_{match} of size $|V_1| \times |V_2|$ which is a prediction of the ground-truth matching matrix and reflects the extent of matching of different node pairs. The other outputs a cost matrix A_{cost} of size $|V_1| \times |V_2|$ in which an element $A_{cost}[u][v]$ denotes the cost of edit operations for matching $u \in V_1$ with $v \in V_2$. Finally, GEDGNN predicts the similarity score of G_1, G_2 by calculating the weighted sum of costs in A_{cost} with a bias value, i.e., $Softmax(A_{match}) \cdot A_{cost} + bias$.

3.1 GNN Backend

Graph neural network (GNN) [16] is widely used as a backend to generate node embeddings in existing graph similarity computation studies [1, 18]. For a graph $G = (V, E, L)$ fed into our model, the GNN generates the node embedding $H_{|V| \times d}$ through graph convolution operations, which iteratively conduct node feature

propagation and aggregation along graph edges. After a graph attention layer [2] or other simpler mechanisms, a weighted sum of all nodes' embedding becomes the graph embedding of G .

In GEDGNN, we use a three-layer GIN [30] as the backend with the same setting of Noah's graph embedding module [32], which is proved as powerful as the Weisfeiler-Lehman isomorphism test. For each node $u \in V$, the initial node embedding $H(u)$ is generated according to its feature. If G is a labeled graph, $H(u)$ is defined as the one-hot vector of its node label. If node u has k labels in an attributed graph, we can simply set these k dimensions as 1 in $H(u)$. In graphs without node label, we can set each $H(u)$ as a constant number, i.e., a 1-dimensional vector. Then in each layer of GIN, each node u updates its embedding by combining it with u 's neighbors' embedding using an injective function.

3.2 GED Prediction

3.2.1 Cross Matrix Module. The cross matrix module is designed to capture the node-to-node interaction between the graph pair (G_1, G_2) . After the GNN backend, their node embeddings H_1 and H_2 are inputted into the cross matrix module, the final output of which is a matrix $A \in \mathcal{R}^{n_1 \times n_2}$. Each element $A[u][v]$ is determined by $H_1[u]$, $H_2[v]$ and module parameters, and hence denotes the interaction between node $u \in V_1$ and node $v \in V_2$.

Multiplying the node embeddings by a trainable parameter matrix W is a direct idea to obtain the node-to-node level information:

$$A = H_1 W H_2^\top.$$

Recall that the sizes of H_1 and H_2 are $|V_1| \times d$ and $|V_2| \times d$ respectively, where d is the output feature dimension of the GNN backend. The size of W must be $d \times d$ and that of the result matrix A is $|V_1| \times |V_2|$, which only depends on the sizes of input graphs.

The above architecture has been proven to be effective in solving graph alignment [20]. With regard to the hardness of GED computation, c parameter matrices W_1, W_2, \dots, W_c are concurrently deployed to further enhance the expressive ability of GEDGNN:

$$A = [H_1 W_1 H_2^\top, H_1 W_2 H_2^\top, \dots, H_1 W_c H_2^\top],$$

where c is a hyper-parameter in this module.

Since A becomes a 3-dimensional vector of size $|V_1| \times |V_2| \times c$, a multi-layer perceptron (MLP) is adopted to decrease the last dimension of A from c to 1. Specifically, the MLP consists of three fully connected layers of size $(c, 2c)$, $(2c, c)$ and $(c, 1)$ respectively.

3.2.2 GED Value Prediction. In GEDGNN, a cost matrix $A_{cost} \in \mathcal{R}^{|V_1| \times |V_2|}$ is generated by the cross matrix module for the given graph pair (G_1, G_2) . $A_{cost}[u][v]$ denotes the cost of edit operations caused by the node pair (u, v) for $u \in V_1, v \in V_2$. Moreover, a matching matrix A_{match} is generated by another cross matrix module, which is supervised by the ground-truth matching matrix M^* . It is a real matrix where each value is in $(0, 1)$. During the training process, a binary cross entropy loss is used to minimize the difference between A_{match} and M^* , hence A_{match} can be close to M^* when GEDGNN is well trained. Since GED computation involves all node pairs, we smooth the values in A_{match} by a row softmax function so as to capture the importance of each node pair:

$$\forall \text{row } u, \text{Softmax}(A_{match})[u][v] = \frac{\exp(A_{match}[u][v])}{\sum_{v'} \exp(A_{match}[u][v'])}.$$

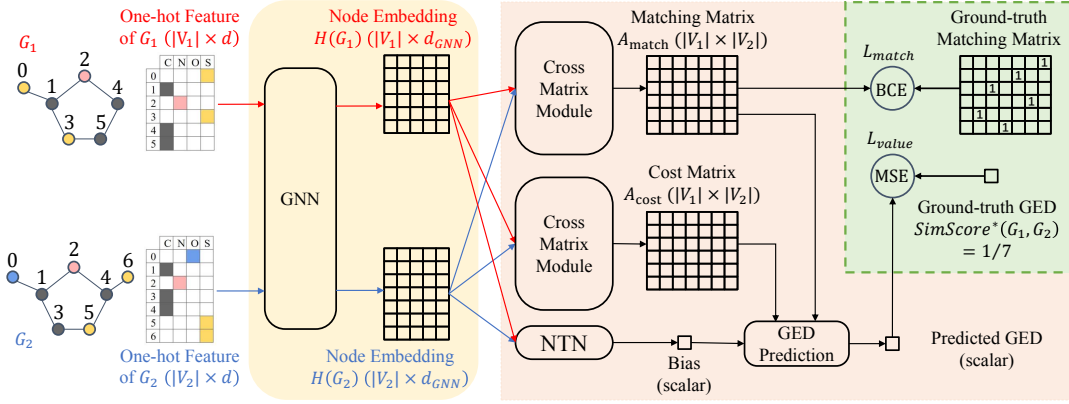


Figure 2: GEDGNN Architecture. Three colored areas denote the GNN backend, GED prediction and model training.

Combining these two matrices, we design the cost matrix model to predict the GED value as follows:

$$GED(G_1, G_2) = f(\text{Softmax}(A_{match}) \cdot A_{cost} + bias), \quad (1)$$

where f is a parameter-free function that adjusts the result's scale. Intuitively, $A_{cost}[u][v]$ denotes the cost of edit operations caused by the node pair (u, v) , and $\text{Softmax}(A_{match})[u][v]$ reveals the importance of this node pair. The cost model sums up all partial cost by weight and uses it to predict $GED(G_1, G_2)$.

The cost model is a generalization of existing methods which calculate GED by the partial cost of node pairs. Compared to our smooth matrix $\text{Softmax}(A_{match})$, classical greedy algorithms [10, 24, 25] use a hard 0-1 matrix where the number of 1s is exactly $\min(|V_1|, |V_2|)$ and each row and column contains at most one 1 instead, which corresponds to a maximum matching of G_1, G_2 . Therefore, the final result of greedy algorithms merely depends on several values in the cost matrix. At the cost of poor solution quality, this setting greatly simplifies the GED calculation and reduces the time complexity of greedy algorithms. Using the sigmoid function σ to get a similarity score in $(0, 1)$, the formula of cost matrix model (Eq. 1) can be rewritten as follows:

$$\text{SimScore}(G_1, G_2) = \sigma(\text{Softmax}(A_{match}) \cdot A_{cost} + bias).$$

Since both the matching and cost matrices come from the cross matrix module, which mainly captures the node-to-node information between G_1 and G_2 , a neural tensor network (NTN) is applied to calculate the bias value. The NTN module receives the graph embeddings of G_1 and G_2 as input, and outputs a scalar. It is used in previous studies [2, 32] to capture the graph-level information.

3.3 Model Training

Since we aim to predict the GED value and find a node matching (for generating graph edit path) at the same time, the loss function of GEDGNN should contain two parts, \mathcal{L}_{match} and \mathcal{L}_{value} , which denote the loss of matching and value prediction respectively. λ is a trade-off hyper-parameter:

$$\mathcal{L} = \mathcal{L}_{match} + \lambda \cdot \mathcal{L}_{value}. \quad (2)$$

A. Matching Loss. In GEDGNN, the ground-truth matching is used to supervise matching matrix training. Given (G_1, G_2) with a

ground-truth matching M^* (i.e., $GED(G_1, G_2, M^*) = GED(G_1, G_2)$), we first convert M^* to a 0-1 matching matrix of size $|V_1| \times |V_2|$:

$$M_{01}^*[u][v] = \begin{cases} 1 & (u, v) \in M^* \\ 0 & (u, v) \notin M^* \end{cases}$$

Recall that after the final sigmoid layer of the MLP in the cross matrix module, each value in the matching matrix is in the range $(0, 1)$. Hence, we adopt the binary cross entropy loss between the matching matrix A_{match} and the ground-truth matrix M_{01}^* as the matching loss:

$$\begin{aligned} \mathcal{L}_{match} &= \text{BCELoss}(A_{match}, M_{01}^*) \\ &= \frac{1}{|V_1| \cdot |V_2|} \sum_{u,v} (M_{01}^*[u][v] \cdot \log A_{match}[u][v] \\ &\quad + (1 - M_{01}^*[u][v]) \cdot \log(1 - A_{match}[u][v])). \end{aligned} \quad (3)$$

B. Value Loss. Since the machine learning model is inclined to predict the value in a small range, it is a good idea to convert the ground-truth GED value into a similarity score before training. There are several conversion functions in previous studies. In GEDGNN, we adopt the following linear function [32]:

$$\text{SimScore}(G_1, G_2) = \frac{GED(G_1, G_2)}{\max(|V_1|, |V_2|) + \max(|E_1|, |E_2|)}.$$

Since the denominator is an upper bound of GED, the similarity score must fall in $[0, 1]$. After this conversion, we can simply use the minimum squared error (MSE) to denote the value loss:

$$\mathcal{L}_{value} = (\text{SimScore}(G_1, G_2) - \text{SimScore}^*(G_1, G_2))^2, \quad (4)$$

where $\text{SimScore}^*(G_1, G_2)$ denotes the ground-truth similarity score.

4 POST-PROCESSING TO GENERATE GRAPH EDIT PATH

In this section, we introduce how to extract a high-quality node matching from the output matching matrix A_{match} via a post-processing algorithm in order to generate a graph edit path.

4.1 Extracting Ground-truth Matching

Since A_{match} is very close to the ground-truth matching matrix M^* when GEDGNN is well trained, their dot product, $M^* \cdot A_{match}$, should be the largest compared to other possible node matchings between G_1 and G_2 in the ideal case. Based on this observation, we can convert the problem of extracting M^* from A_{match} to a weighted bipartite matching problem. Specifically, we construct a weighted bipartite graph:

$$G = (V_1, V_2, V_1 \times V_2, w),$$

where $w = A_{match}$ denotes the weight of edges in G , and define the weight of a matching M as the sum weight of edges it contains:

$$w(M) = \sum_{(u,v) \in M} w(u,v). \quad (5)$$

For simplicity, we regard a matching M as a 0-1 matrix in the following formulae, hence Eq. 5 can be rewritten as:

$$w(M) = M \cdot A_{match}. \quad (6)$$

In the ideal case (i.e., A_{match} is very close to M^*), we can extract M^* from A_{match} by finding the maximum weight matching on G :

$$M^* = \arg \max_{M \in \mathcal{M}} w(M), \quad (7)$$

where \mathcal{M} denotes the set of all maximum bipartite matchings. Since each maximum matching contains the same number of edges (Definition 4), the one closest to A_{match} results in the maximum dot product. But in the real case, the output matching matrix A_{match} may have some difference from M^* as the training loss is not 0. Thus there is a chance that M^* is not the maximum weight matching on G . We have the following theorem to measure the gap between the maximum weight and $w(M^*)$:

THEOREM 1. Assume that $A_{match} = M^* + A_{error}$, we have:

$$\max_{M \in \mathcal{M}} w(M) - w(M^*) \leq \max_{M \in \mathcal{M}} (M - M^*) \cdot A_{error}. \quad (8)$$

PROOF. Replacing A_{match} by $M^* + A_{error}$, it is easy to derive:

$$\max_{M \in \mathcal{M}} M \cdot A_{match} \leq \max_{M \in \mathcal{M}} M \cdot M^* + \max_{M \in \mathcal{M}} M \cdot A_{error} \quad (9)$$

$$M^* \cdot A_{match} = M^* \cdot M^* + M^* \cdot A_{error} \quad (10)$$

Since $\max_{M \in \mathcal{M}} M \cdot M^* = M^* \cdot M^*$, (9) – (10) leads to (8) immediately. \square

A_{error} is the difference between A_{match} and the ground-truth matching M^* , which is related to the training loss \mathcal{L}_{match} . If the scale of A_{error} is small, then the gap between $w(M^*)$ and the maximum matching weight is small. Motivated by Theorem 1, we can increase the chance of finding M^* by enumerating several (instead of one) matchings on G with large weights. Thus we propose the k -best matching framework in Section 4.2 which computes top- k maximum weight matchings on G .

4.2 k-best Matching Framework

4.2.1 Nutshell Framework. Algorithm 1 shows a basic version of the k -best matching framework. The main idea is to calculate the top- k maximum weight matchings on G (k -best matchings in short), then convert them into graph edit paths by $GenPath(G_1, G_2, M_i)$ (i.e., the method described in Section 2.2) and finally return the shortest one among the k edit paths.

Algorithm 1: k -best Matching Framework

Input : a pair of graphs (G_1, G_2) , a matching matrix A_{match} and an integer k

Output: an edit path from G_1 to G_2

```

1 Construct  $G = (V_1, V_2, V_1 \times V_2, A_{match})$ ;
2  $M_1 \leftarrow$  the maximum weight matching on  $G$ ;
3  $BestPath \leftarrow GenPath(G_1, G_2, M_1)$ ;
4 for  $i \in [2, k]$  do
5    $M_i \leftarrow$  the  $i$ -th maximum weight matching on  $G$ ;
6    $path \leftarrow GenPath(G_1, G_2, M_i)$ ;
7   if  $len(path) < len(BestPath)$  then
8      $BestPath \leftarrow path$ ;
9 return  $BestPath$ ;
```

4.2.2 Improvement. Algorithm 1 simply regards the k -best matching algorithm as an encapsulated function, which can be further optimized if the two separate procedures, computing k -best matchings and generating edit paths, can be tightly coupled. In Algorithm 2, we propose a more powerful version of the k -best matching framework, in which GED computation is integrated into the k -best matching procedure. As a result, we can prune some bad matchings (whose weight is large but its corresponding edit path is long) by checking GED lower bound. Still using k iterations (i.e., the same time cost as Algorithm 1), we can explore more than k -best matchings by solution space pruning and find a shorter edit path.

The k -best matching algorithm works by iteratively splitting the solution space, and we conduct an extra GED lower bound checking during the splitting procedure. In the following, we first give an introduction to the idea of solution space splitting [8] and the concept of GED lower bound [7, 11], and then present how to combine them together in Algorithm 2 as the improved framework. **Solution Space Splitting.** Let $\Omega = \mathcal{M}_{max}(|V_1| \times |V_2|)$ denote the whole solution space, i.e., the set of all maximum node matchings between V_1 and V_2 . $\Omega_{I,O} \subseteq \Omega$ [8] denotes the subspace specified by two disjoint edge subsets I and O , in which each matching M must contain all edges of I but none of O , i.e.,

$$\Omega_{I,O} = \{M \mid M \in \Omega, I \subseteq M \text{ and } O \cap M = \emptyset\}.$$

Let M_1 and M_2 denote the best and second best matching in $\Omega_{I,O}$ respectively. By choosing an arbitrary edge e in M_1 but not in M_2 , $\Omega_{I,O}$ can be further split into two subspaces $\Omega_{I \cup \{e\}, O}$ and $\Omega_{I, O \cup \{e\}}$ with the following properties:

- $\Omega_{I \cup \{e\}, O} \cup \Omega_{I, O \cup \{e\}} = \Omega_{I,O}$
- $\Omega_{I \cup \{e\}, O} \cap \Omega_{I, O \cup \{e\}} = \emptyset$
- M_1 remains to be the best matching in $\mathcal{M}_{I \cup \{e\}, O}$.
- M_2 becomes the best matching in $\mathcal{M}_{I, O \cup \{e\}}$.

Based on [8], we have the following lemma to compute the next best matching by solution space splitting.

LEMMA 1. Let M_1, M_2, \dots, M_k denote the k -best matchings. Given a partition of the whole solution space, $\Omega = \Omega_{I_1, O_1} \cup \Omega_{I_2, O_2} \cup \dots \cup \Omega_{I_k, O_k}$, which satisfies that M_i is exactly a best matching in the subspace Ω_{I_i, O_i} for each $i \in \{1, 2, \dots, k\}$, one of the second best matchings of these k subspaces is the $(k+1)$ -th best matching.

Following Lemma 1, assume that the second best matching of Ω_{I_i, O_i} is the $(k+1)$ -th best matching, M_{k+1} , and e is an edge in M_i but not in M_{k+1} . We can further split Ω_{I_i, O_i} into $\Omega_{I_i \cup \{e\}, O_i}$ and $\Omega_{I_i, O_i \cup \{e\}}$. Use the former to replace Ω_{I_i, O_i} and regard the latter as the $(k+1)$ -th solution subspace $\Omega_{I_{k+1}, O_{k+1}}$, then the best matching of the i -th solution subspace remains unchanged and that of the new subspace is M_{k+1} . The condition of Lemma 1 still holds with $k+1$. In this way, the problem of finding k -best matchings is reduced to iteratively finding the second best matching of a subspace.

GED Lower Bound. A series of methods [7, 11] have been proposed to calculate a lower bound of $GED(G_1, G_2)$. We adopt the label set based method which can be calculated in linear time:

$$GEDLB(G_1, G_2) = |L(V_1) \oplus L(V_2)| + ||E_1| - |E_2||, \quad (11)$$

where $L(V_1)$ and $L(V_2)$ denote the multi-set of node labels of G_1 and G_2 respectively, and \oplus denotes a multi-set function that $A \oplus B = A \cup B - A \cap B$. Since we do not consider edge label in this paper, the lower bound on edge is simply set as the difference between $|E_1|$ and $|E_2|$. Eq. 11 can be easily extended to the cases with edge label or without node label. Furthermore, $GEDLB(G_1, G_2, I)$ denotes the lower bound of edit distance in a solution subspace specified by I :

$$GEDLB(G_1, G_2, I) \leq \min_{M \in \Omega_{I, O}} GED(G_1, G_2, M),$$

which can be calculated by adding up the number of edit operations caused by I and the standard GED lower bound on remaining parts of G_1 and G_2 . Here we do not consider the forbidden edge set O of a solution subspace since it is hard to integrate O into lower bound computation efficiently.

LEMMA 2. In the k -best matching framework, once the GED lower bound of a subspace sp is greater or equal to the length of the current best path, there is no need to further split sp .

Lemma 2 implies that we can prune some solution subspaces on-the-fly according to their GED lower bounds and the current best solution. By skipping these useless subspaces, the procedure of enumerating k -best matchings is accelerated and we actually explore more than k matchings within k iterations.

Combining solution space splitting with GED lower bound, the improved k -best matching framework is presented in Algorithm 2. Lines 3-9 initialize the first solution subspace sp_1 , where $sp_1.M_1$ and $sp_1.M_2$ denote the best and second-best matchings in sp_1 respectively, which can be found in $O(n^3)$ time by many classical algorithms. An introduction to implementing `GetBestMatching()` and `GetSecondBestMatching()` is given in [8]. Denote by $sp_1.lb$, the GED lower bound of sp_1 is calculated by the function `GEDLB()` according to Eq. 11. Moreover, `Update(BestPath, path)` means replacing the current best solution $BestPath$ by $path$ if $BestPath$ is None or $path$ is shorter. Lines 10-20 conduct k iterations of solution space splitting. In each iteration, it first enumerates existing

Algorithm 2: Improved k -best Matching Framework

Input : a pair of graphs (G_1, G_2) , a matching matrix A_{match} and an integer k

Output: an edit path from G_1 to G_2

```

1 Construct  $G = (V_1, V_2, V_1 \times V_2, A_{match})$ ;
2  $BestPath \leftarrow None$ ;
3 Initially,  $sp_1$  is the whole solution space:
4    $(sp_1.I, sp_1.O) \leftarrow (\emptyset, \emptyset)$ ;
5    $sp_1.M_1 \leftarrow GetBestMatching(G)$ ;
6    $sp_1.M_2 \leftarrow GetSecondBestMatching(G, sp_1)$ ;
7    $sp_1.lb \leftarrow GEDLB(G_1, G_2, sp_1.I)$ ;
8    $Update(BestPath, GenPath(G_1, G_2, sp_1.M_1))$ ;
9    $Update(BestPath, GenPath(G_1, G_2, sp_1.M_2))$ ;
10 for  $t \in [2, k]$  do
11    $(id, max\_weight) \leftarrow (None, -\infty)$ ;
12   for  $sp_i \in [sp_1, sp_2, \dots, sp_{t-1}]$  do
13     if  $sp_i.lb < len(BestPath)$  then
14        $weight \leftarrow sp_i.M_2 \cdot A_{match}$ ;
15       if  $weight > max\_weight$  then
16          $(id, max\_weight) \leftarrow (i, weight)$ ;
17    $(sp_{id}, sp_t) \leftarrow SpaceSplit(G, sp_{id})$ ;
18    $sp_{id}.lb \leftarrow GEDLB(G_1, G_2, sp_{id}.I)$ ;
19    $Update(BestPath, GenPath(G_1, G_2, sp_{id}.M_2))$ ;
20    $Update(BestPath, GenPath(G_1, G_2, sp_t.M_2))$ ;
21 return  $BestPath$ ;

22 Function SpaceSplit(G, sp)
23   Choose an arbitrary edge  $e \in sp.M_1$  but  $e \notin sp.M_2$ ;
24   Split  $sp$  into two subspaces by using  $e$  or not;
25   Set  $sp_e$  as the one using  $e$ :
26    $(sp_e.I, sp_e.O) \leftarrow (sp.I \cup \{e\}, sp.O)$ ;
27    $sp_e.M_1 \leftarrow sp.M_1$ ;
28    $sp_e.M_2 \leftarrow GetSecondBestMatching(G, sp_e)$ ;
29   Set  $sp_{-e}$  as the one not using  $e$ :
30    $(sp_{-e}.I, sp_{-e}.O) \leftarrow (sp.I, sp.O \cup \{e\})$ ;
31    $sp_{-e}.M_1 \leftarrow sp.M_2$ ;
32    $sp_{-e}.M_2 \leftarrow GetSecondBestMatching(G, sp_{-e})$ ;
33    $sp_{-e}.lb \leftarrow sp.lb$ ;
34   return  $(sp_e, sp_{-e})$ ;

```

subspaces to find whose second-best matching has the maximum weight, and then splits this subspace according to Lemma 1. Note that, by checking the GED lower bound with $BestPath$ in line 13, some useless subspaces are pruned. Lines 22-34 are the detailed implementation of solution space splitting as introduced above.

To summarize, there are two main improvements compared to the basic version (Algorithm 1). **1.** By Lemma 2 we can prune some useless solution spaces using GED lower bound. Using the same k iterations, we can explore more than k -best matchings by solution space pruning. **2.** The second-best matchings of all subspaces are also used to update the best graph edit path by lines 9, 19-20 in Algorithm 2. Recall that the time complexity of generating an edit path (`GenPath()`) is merely $O(n+m)$ whereas that of finding a

second-best matching is $O(n^3)$. Since the second-best matching of each subspace are a by-product of Algorithm 2 with large weights, we generate the corresponding edit paths for them at linear cost to see if they improve the solution. Thus we essentially generate $2k$ edit paths in k iterations and report the shortest one.

4.3 Improve GED Prediction using Edit Path

After extracting a high-quality solution $BestPath$ from the matching matrix by the k -best matching framework, a natural idea is using it to further improve the GED prediction.

LEMMA 3. *Given a graph pair (G_1, G_2) that $GED(G_1, G_2) = d^*$, the k -best matching framework finds a range bound of d^* :*

$$GEDLB(G_1, G_2) \leq d^* \leq len(BestPath)$$

Based on Lemma 3, the $BestPath$ can tighten the predicted GED value by GEDGNN if it is not in the range. We observe that the GED lower bound is usually too loose to use in real cases. If the predicted GED value is less than d^* , it may not be improved by the lower bound. If the predicted GED is larger than d^* , it is likely to be tightened by $len(BestPath)$. Towards this end, we can further fine-tune the model by adding an extra loss to Eq. 2:

$$\mathcal{L} = \mathcal{L}_{match} + \lambda \cdot \mathcal{L}_{value} - SimScore(G_1, G_2), \quad (12)$$

where a larger predicted similarity score leads to a lower loss, so that the fine-tuned model is inclined to predict large GED values.

4.4 Time Complexity Analysis

To find an edit path for a graph pair (G_1, G_2) , the whole pipeline of GEDGNN consists of three parts: model training, model testing (predicting GED and the matching matrix) and the post-processing algorithm. The cost of one forward passing on GEDGNN mainly consists of GNN backend and the cross matrix module. Here we use $n = \max(|V_1|, |V_2|)$ and $m = \max(|E_1|, |E_2|)$ to represent the input size. Denote the number of GNN layers by L and the maximum dimension of one layer by d , it takes $O(Lnd^2 + Lmd)$ time for GNN to generate node embedding by message passing. The number of hidden layers of the cross matrix module is c according to Section 3.2.1, thus it takes $O(cnd^2 + n^2d)$ time to compute $H_1W_iH_2^T$ for $i = 1, \dots, c$ and $O(c^2n^2)$ time to go through the MLP. In total, the time cost of one forward passing is $O(Lnd^2 + Lmd + cnd^2 + cn^2d + c^2n^2)$. Since the hyper-parameters L , d and c are fixed to constant values in our model, the time complexity of one forward passing can be simplified as $O(n^2)$ in terms of the input size. Then the time cost of training (testing) can be regarded as the product of training (testing) data size and the cost of one forward passing. As for the

post-processing algorithm, since it takes $O(n + m)$ time to generate a path from a node matching and $O(kn^3)$ time to find k -best matchings [8], the total post-processing cost is $O(kn^3)$.

5 EXPERIMENT

5.1 Experimental Settings

5.1.1 *Data Sets.* The proposed algorithms are evaluated on three real graph data sets: AIDS, Linux and IMDB. The statistics of these graph data sets are listed in Table 1, where $\#graphs$, $\overline{|V|}$, $\overline{|E|}$, $|V|_{max}$ and $|E|_{max}$ denote the number of graphs, the average and maximum number of vertices and edges respectively. Each graph in **AIDS** represents a chemical compound where nodes and edges denote atoms and covalent bonds respectively. Each node has a single label such as C, N, O, Cu, etc. **Linux** [1, 2] contains program dependence graphs generated from Linux kernel. Each graph is unlabelled and corresponds to a Linux kernel function, where nodes and edges denote statements and the dependency among them respectively. **IMDB** [1, 2, 32] is a larger data set which contains graphs with more than 10 nodes. Each graph in IMDB is an unlabeled ego-network where each node denotes a film actor/actress and each edge denotes a co-star relation. In addition, we generate some synthetic power-law graphs in larger scale for performance evaluation. We vary the number of nodes in the power-law graphs from 25 to 400.

Ground-truth Data Generation. Since all graphs in AIDS and Linux have no more than 10 nodes, we simply use exhaustive search to generate the ground-truth data on these graphs. But due to the NP-hardness of GED computation, the ground-truth data of IMDB is usually generated in a sub-optimal manner instead. An efficient training technique is proposed in TaGSim [1] that uses synthetic graph pairs instead of all graph pairs in a given data set. Specifically, for each graph G , we can randomly apply Δ graph edit operations on it and get a synthetic graph G' . It is clear that $GED(G, G') \leq \Delta$. If the size of G is large enough w.r.t. Δ and these operations edit distinct nodes/edges, the probability that $GED(G, G') < \Delta$ is quite low. In this case, Δ can be approximately regarded as the ground-truth GED of this synthetic graph pair (G, G') .

We generate ground-truth data in IMDB by combining the efficient training technique and the classical exact search method. All graphs in IMDB are partitioned into two sets, small graphs (with 10 or fewer nodes) and large graphs (with more than 10 nodes). For small graphs, we generate all pairwise ground-truth data by exact search. For each large graph, 100 synthetic graphs are generated using the above technique. The number of edit operations Δ is randomly distributed in $(0, 10]$. If the graph contains no more than 20 nodes, the range is further limited in $(0, 5]$. The ground-truth data for the power-law graphs is generated in a similar way.

Data Partitioning. Graphs in each data set are partitioned into training set, validation set and testing set in ratio of 6 : 2 : 2. In AIDS and Linux, all graph pairs in the training set are used for model training. For IMDB, assume that there are S small graphs and L large graphs in the training set, then $S \times S + L \times 100$ graph pairs are used for training in total. The training set can be regarded as a graph database and graphs in the validation/testing set are regarded as queries. For each query graph, we can search the most similar graph in the database by the predicted GED value. Following this classical setting [2, 32], for each graph G in the validation/testing

Table 1: Statistics of Graph Data Sets

	#graphs	$\overline{ V }$	$\overline{ E }$	$ V _{max}$	$ E _{max}$
AIDS	700	8.9	8.8	10	14
Linux	1000	7.6	6.9	10	13
IMDB	1500	13	65.9	89	1467
IMDB-small	148	8.1	25.2	10	45
IMDB-large	152	19.1	117.1	54	858

Table 2: Overall Performance on Real Graph Data Sets

AIDS	GED		Ranking				Graph Edit Path			Feasible Rate	Time (s/100p)
	MAE	Accuracy	ρ	τ	p@10	p@20	Precision	Recall	F1		
SimGNN	0.914	33.8%	0.832	0.693	62.4%	72.0%	-	-	-	67.6%	0.283
TaGSim	0.841	36.6%	0.850	0.715	64.6%	74.6%	-	-	-	66.2%	0.123
GPN	0.902	35.3%	0.822	0.684	58.6%	70.4%	-	-	-	66.8%	0.326
GEDGNN-value	0.773	39.7%	0.876	0.751	71.6%	77.9%	-	-	-	62.2%	0.408
Greedy	9.227	1.3%	0.464	0.362	50.2%	57.2%	32.5%	59.6%	41.2%	100.0%	110.000
Noah	3.078	6.3%	0.730	0.610	70.9%	75.1%	49.9%	62.0%	54.7%	100.0%	168.390
GEDGNN-matching	1.427	44.3%	0.806	0.704	85.5%	85.0%	68.5%	74.3%	71.0%	100.0%	122.900
Linux	GED		Ranking				Graph Edit Path			Feasible Rate	Time (s/100p)
	MAE	Accuracy	ρ	τ	p@10	p@20	Precision	Recall	F1		
SimGNN	0.456	59.6%	0.933	0.844	89.1%	92.0%	-	-	-	80.0%	0.244
TaGSim	0.391	66.8%	0.924	0.837	81.6%	87.8%	-	-	-	82.8%	0.117
GPN	0.135	88.4%	0.962	0.898	95.6%	96.9%	-	-	-	92.4%	0.321
GEDGNN-value	0.094	91.7%	0.963	0.903	96.2%	97.6%	-	-	-	96.7%	0.380
Greedy	5.252	7.6%	0.708	0.618	74.5%	79.5%	42.7%	77.9%	53.6%	100.0%	1.000
Noah	1.747	8.1%	0.874	0.802	90.9%	93.6%	79.8%	87.5%	82.8%	100.0%	77.237
GEDGNN-matching	0.208	91.8%	0.954	0.933	97.2%	97.0%	87.7%	89.6%	88.4%	100.0%	42.300
IMDB-small	GED		Ranking				Graph Edit Path			Feasible Rate	Time (s/100p)
	MAE	Accuracy	ρ	τ	p@10	p@20	Precision	Recall	F1		
SimGNN	0.979	62.4%	0.938	0.892	93.2%	96.1%	-	-	-	74.7%	0.243
TaGSim	2.387	8.5%	0.893	0.837	90.7%	92.5%	-	-	-	41.2%	0.104
GPN	0.968	26.4%	0.976	0.953	95.7%	98.7%	-	-	-	29.1%	0.308
GEDGNN-value	0.249	84.0%	0.972	0.952	97.5%	98.6%	-	-	-	90.3%	0.379
Greedy	1.534	84.1%	0.844	0.818	93.9%	92.6%	84.4%	89.8%	86.3%	100.0%	1.400
Noah	1.748	27.1%	0.893	0.856	91.2%	93.6%	87.7%	88.5%	88.0%	100.0%	270.536
GEDGNN-matching	0.226	96.3%	0.977	0.966	97.0%	97.8%	80.9%	82.0%	81.3%	100.0%	25.200
IMDB-large	GED		Ranking				Graph Edit Path			Feasible Rate	Time (s/100p)
	MAE	Accuracy	ρ	τ	p@10	p@20	Precision	Recall	F1		
SimGNN	1.470	23.2%	0.480	0.364	57.8%	63.0%	-	-	-	54.1%	0.258
TaGSim	3.752	7.4%	0.115	0.090	43.3%	50.3%	-	-	-	20.3%	0.113
GPN	1.591	27.7%	0.551	0.456	54.6%	59.0%	-	-	-	57.5%	0.331
GEDGNN-value	1.398	26.9%	0.629	0.500	69.6%	71.5%	-	-	-	68.5%	0.414
Greedy	24.961	40.4%	0.671	0.602	73.8%	72.4%	42.3%	75.8%	45.3%	100.0%	4.000
Noah	18.596	49.5%	0.547	0.525	60.3%	65.8%	45.5%	80.4%	48.9%	100.0%	10132.777
GEDGNN-matching	6.078	67.9%	0.795	0.751	86.6%	84.8%	74.6%	91.4%	77.3%	99.9%	264.800

set, 100 graphs are randomly chosen from the training set to form 100 graph pairs for validation/testing.

5.1.2 Competitors. Depending on whether an algorithm can produce an edit path or not, all competitors are divided into two groups: **1. End-to-end GED learning model.** Such models are mostly based on graph neural networks and only predict the GED value of a given graph pair, including (1) **SimGNN** [2], the first work that uses graph neural networks to predict GED value. (2) **TaGSim** [1], the state-of-the-art GED learning model. (3) **GPN**, the graph path network (GPN) proposed in **Noah** [32]. Although it is originally designed to supervise the A^* -beam search, its performance in GED value prediction is similar to that of **TaGSim**.

2. Node matching algorithms. This type of algorithm calculates $GED(G_1, G_2)$ by finding a node matching between G_1 and G_2 , from which an edit path can be generated, including (1) **Greedy**. We adopt the best setting of Hungarian [17] and VJ [15] as **Greedy**, which is the baseline graph edit path construction algorithm. (2) **Noah** [32]. It uses GPN to supervise the A^* -beam search, which is the state-of-the-art algorithm for finding graph edit paths.

We create two variants of our method for comparison with the above two groups of competitors respectively – **GEDGNN-value** which uses the end-to-end learning model GEDGNN to predict the GED value, and **GEDGNN-matching** which uses GEDGNN to predict a matching matrix, and then applies the post-processing algorithm to generate an edit path.

5.1.3 Detailed Setup.

Testbed. All experiments are conducted on a Windows PC using the i7-8750H CPU. All algorithms are implemented in Python and the machine learning models are implemented using PyTorch and PyTorch Geometric.

Metrics. To evaluate the performance of our model against other baselines, we consider the following metrics:

1. Precision of GED Prediction: we use Mean Absolute Error (MAE) and Accuracy as metrics. Given a predicted GED value d and the ground-truth d^* on each testing graph pair, MAE is defined as $|d - d^*|$ over all testing graph pairs, and Accuracy is defined as the ratio of the testing pairs with $round(d) = d^*$. $round(d)$ rounds d to the nearest integer and is used here since the predicted GED by machine learning models is a real value.

2. Ranking Metrics: for each testing graph G , 100 graphs are chosen to form graph pairs with G . These graphs can be ranked according to the graph edit distance between G and them. Therefore, we use Spearman’s Rank Correlation Coefficient ρ , Kendall’s Rank Correlation Coefficient τ , and precision at top 10 and top 20 ($p@10$, $p@20$) as the evaluation metrics.

3. Path Accuracy: we report the precision, recall and F1-score of the generated graph edit path compared with the ground-truth path. For a generated edit path P and a ground-truth path P^* , each of which can be regarded as a set of operations, we define $precision = \frac{|P \cap P^*|}{|P|}$, $recall = \frac{|P \cap P^*|}{|P^*|}$ and $F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$.

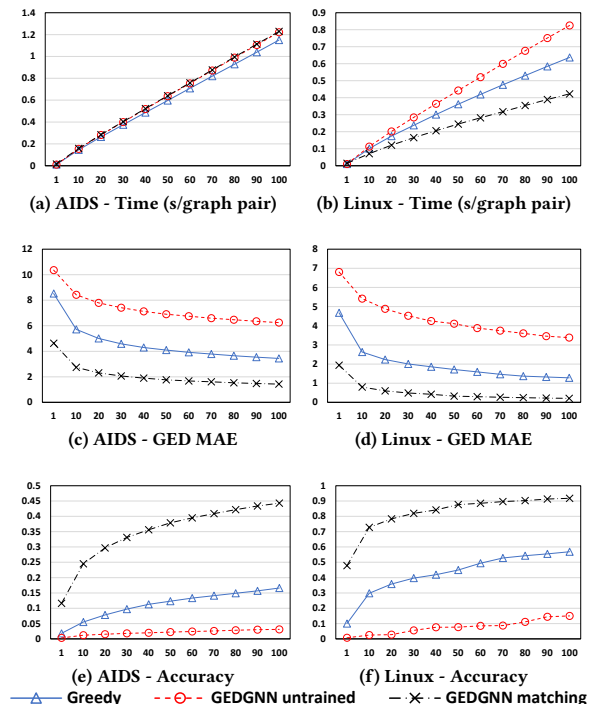


Figure 3: Evaluation of the k -best Matching Framework.

4. Feasible Rate: it denotes the ratio of the testing pairs with $\text{round}(d) \geq d^*$. It measures the percentage of predicted GED values that can lead to a feasible edit path.

5. Time Efficiency: we report the running time in seconds when computing GED for 100 graph pairs and denote it as Time (s/100p).

Hyper-parameter Settings. A 3-layer GIN is used as the GNN backend of GEDGNN, and the output feature size of each layer is 128, 64 and 32 respectively. We adopt 16 parameter matrices whose size is 32×32 in each cross matrix module. In the k -best matching framework, the number of iterations k is set as 100. In the loss function, the weight λ of value loss is set as 10 in AIDS and Linux, and set as 1 in other data sets. During training, the dropout rate, learning rate and weight decay are set as 0.5, 0.001 and 5×10^{-4} respectively. All machine learning models are fully trained for 10 to 20 epochs, until the decrease of training loss is less than 0.001.

5.2 Comparison with State-of-the-art Methods

Table 2 presents the performance of all methods on the real graphs. The best results in each group are highlighted in bold.

5.2.1 GED Precision and Ranking Metrics.

GED Learning Models. Most learning models can achieve a quite low MAE in GED value prediction, e.g., $\text{MAE} < 1.0$ for small graphs and $\text{MAE} < 1.6$ for large graphs. We observe that the state-of-the-art model **TaGSim** only has a reasonably good performance on AIDS where graph nodes have labels, but performs poorly on other data sets. This is because the type-aware GED proposed in **TaGSim** degenerates to the standard GED when graphs have no

labels. **GPN** outperforms **TaGSim** in unlabeled graphs since it uses graph isomorphism network (GIN) as backend, which is more powerful than the graph aggregation layer (GAL) of **TaGSim**. Our proposed model **GEDGNN-value** achieves the minimum MAE in all data sets. Compared to the best result of **SimGNN**, **TaGSim** and **GPN**, the GED MAE of our method **GEDGNN-value** decreases by 8.1%, 30.4%, 74.3% and 4.9% on AIDS, Linux, IMDB-small and IMDB-large respectively. It also achieves the highest accuracy and best performance in ranking metrics in most cases.

Node Matching Algorithms. Our proposed algorithm **GEDGNN-matching** greatly outperforms the other two node matching methods in all data sets. Compared to the best result of **Greedy** and **Noah**, the GED MAE decreases by 53.6%, 88.1%, 85.3% and 67.3% on AIDS, Linux, IMDB-small and IMDB-large respectively. It also achieves the highest accuracy and best performance in ranking metrics in all data sets.

We observe the MAE of node matching algorithms is generally larger than that of learning models, since finding an actual edit path is much harder than predicting GED value only, which is especially obvious for larger graphs in IMDB-large. However, the performance of our method **GEDGNN-matching** is competitive even compared with GED learning models. In IMDB-small, **GEDGNN-matching** achieves the smallest MAE among all methods.

We also observe in IMDB-large the accuracy of node matching algorithms is higher than that of learning models although the MAE of the former group is also larger. This is because the absolute error of GED for a few testing graph pairs is very large, e.g., $|d - d^*| = 130$ for a large graph with 40 nodes, which makes the MAE large. However, for most testing graph pairs, the predicted GED is accurate w.r.t. d^* , which explains the good performance in accuracy.

5.2.2 Path Accuracy. **GEDGNN-matching** achieves the highest precision, recall and F1-score of the generated graph edit path on AIDS, Linux and IMDB-large. Please note that multiple graph edit paths of the minimum length may exist, and each of such paths can be regarded as a ground-truth path according to Definition 2. Thus we generate 10 graph edit paths of the minimum length (if there exist) for each testing graph pair as the ground truth, and report the best measures after we compare our graph edit path with each ground-truth edit path. For the models which only predict the GED value but cannot generate edit paths, the path accuracy metrics do not apply to them and we use “-” to denote this.

5.2.3 Feasible Rate. As expected, the feasible rates of all GED learning models are less than 100% as they may predict a GED value $d \leq d^*$. In principle, the feasible rates of all node matching algorithms should be exactly 100% since they output an upper bound of GED as discussed before. However, we can see that the feasible rate of **GEDGNN-matching** on IMDB-large is 99.9%. Recall that the ground-truth GED value of IMDB-large is obtained by generating synthetic graph pairs by applying Δ graph edit operations. Such pseudo ground-truth may be larger than the true GED value with a very low probability. For such cases, **GEDGNN-matching** can find a better solution than the pseudo ground-truth, thus having a feasible rate of 99.9%. This shows a merit of **GEDGNN-matching**, i.e., it can find a high-quality solution which helps improve the pseudo ground-truth.

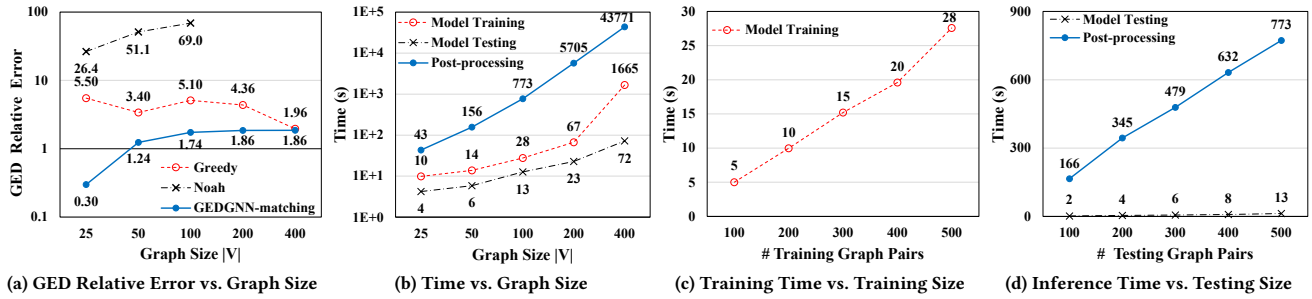


Figure 4: Accuracy and Efficiency on Large Power-law Graphs.

5.2.4 Time Efficiency. We report the testing and post-processing time in seconds per 100 testing graph pairs in Table 2. All GED learning models are very efficient, as their time complexity is $O(n+m)$ when the hyper-parameters are regarded as constant. As our model **GEDGNN-value** needs to compute the matching and cost matrices, its time complexity is $O(n^2)$. Therefore, **GEDGNN-value** runs a little bit slower than the other learning models.

Node matching algorithms are much slower than learning models as they compute GED by finding an edit path. Among the three, **Greedy** is the fastest and **Noah** is the slowest. This is because **Greedy** and **GEDGNN-matching** have polynomial time complexity $O(n^3)$ and $O(kn^3)$ respectively, but **Noah** is a special A^* -beam search algorithm with exponential time complexity. For some testing graphs pairs, **Noah** cannot finish in 6 hours.

5.3 Evaluation of k -best Matching Framework

In this experiment, we evaluate the effectiveness of our k -best matching framework given the matching matrix A_{match} produced by GEDGNN when we vary $k \in [1, 100]$. Besides our method **GEDGNN-matching**, two matching matrix generators are used as baseline for comparison: 1) **GEDGNN-untrained** generates the matching matrix using an untrained GEDGNN model. In this case, random initial parameters result in a random matching matrix. 2) **Greedy** generates the matching matrix using a heuristic greedy function as Hungarian [17] and VJ [15], which is then fed to our k -best matching framework for generating solutions. Figure 3 presents the performance of different methods on AIDS and Linux.

Evaluation of Time Efficiency. Figure 3(a) and (b) report the average running time in seconds for a testing graph pair. The running time increases linearly with k for all three methods. On Linux, **GEDGNN-matching** is the fastest among the three.

Evaluation of Solution Quality. We report the GED MAE in Figure 3(c) and (d), and the accuracy in Figure 3(e) and (f). Our method **GEDGNN-matching** achieves the best performance on both MAE and accuracy for all k values, and **Greedy** performs the second. These results prove that GEDGNN can learn a high-quality matching matrix, which finally produces a short graph edit path by our k -best matching framework.

5.4 Performance on Large Power-law Graphs

In this experiment, we generate synthetic power-law graphs of various sizes (from 25 to 400 nodes) as extreme data and evaluate the

GED accuracy and efficiency. For each graph size, 500 graph pairs are used for training and testing respectively. Figure 4(a) depicts the GED relative error, i.e., $relative_error(d, d^*) = (d - d^*)/d^*$ by **GEDGNN-matching**, **Greedy** and **Noah**. The relative error of our method is 0.299 when $|V| = 25$ and 1.860 when $|V| = 400$. It is much lower than that of **Greedy** and **Noah**. The A^* search-based algorithm **Noah** has a quite poor performance, e.g., its relative error is 26.448 when $|V| = 25$. We do not report its result when $|V| > 100$ since it cannot find an edit path of a graph pair within an hour.

In Figure 4(b)-(d) we report the time efficiency when we vary the graph size, training and testing set size on the power-law graphs, respectively. First we vary the graph size from 25 to 400 nodes, and set the training and testing set sizes as 500 graph pairs. We report the average training time per epoch, testing time and post-processing time in seconds in Figure 4(b). We can see the post-processing is the most time consuming as it has $O(kn^3)$ time complexity. The testing time grows the slowest since it only has forward passing in $O(n^2)$ time, whereas the training process conducts both forward and backward propagation. As the model training can be done offline, this training time cost is acceptable. Then we fix the graph size to 100 nodes and vary the training set size from 100 to 500 graph pairs. In Figure 4(c) we can see the training time grows linearly with the training set size. In Figure 4(d), we fix the graph size to 100 nodes and vary the test set size from 100 to 500 graph pairs. The testing and post-processing time grows linearly with the test set size. These results are consistent with our theoretical time complexity analysis in Section 4.4.

5.5 Ablation Studies

In this subsection, we conduct several ablation studies to verify the effectiveness of the key components in our model.

5.5.1 Cross Matrix Module and GED Lower Bound Pruning. In this experiment, we evaluate the effectiveness of two techniques: the cross matrix module and GED lower bound pruning. In the cross matrix module, a weight matrix W is learned in the training phase. In this ablation study, we simplify the cross matrix module by omitting the weight matrix W .

In Section 4.2, we improve the k -best matching algorithm by pruning some useless solution subspaces using the GED lower bound. In this ablation study, we evaluate the nutshell framework (Algorithm 1) which does not apply the pruning technique.

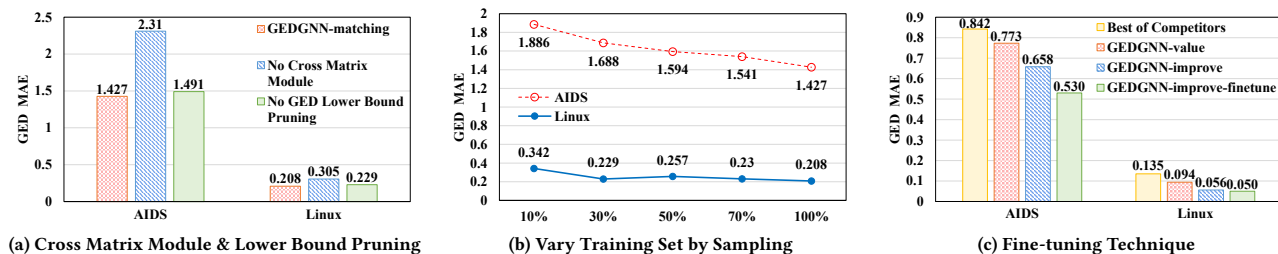


Figure 5: GED MAE in Ablation Studies.

Figure 5(a) depicts the GED MAE on AIDS and Linux by our original **GEDGNN-matching**, the one without the cross matrix module and the one without GED lower bound pruning. If we disable either technique, the GED MAE becomes larger. The performance decrease is especially notable without the cross matrix module. It confirms the importance of the cross matrix module.

5.5.2 Varying The Training Set Size. In our previous experiment, we use all pairs of training graphs for model training for AIDS and Linux, and the training set consists of both similar and dissimilar graph pairs, i.e., having diversified GED inputs. In this ablation study, we vary the training set size by randomly sampling 10%–70% graph pairs and evaluate its influence on GED computation. Figure 5(b) depicts the GED MAE as we vary the training set size on AIDS and Linux. We can observe that the MAE decreases as we sample more graph pairs for training. Sufficient training data that contains diversified graph pairs does improve the performance.

5.5.3 Fine-tuning Technique. In Section 4.3 we describe how to further improve GED prediction using an edit path and introduce an extra loss for fine-tuning in Eq. 12. In this ablation study, we evaluate the effectiveness of this technique. We use **GEDGNN-improve** to denote the GED prediction improved by an edit path, and **GEDGNN-improve-finetune** to denote the improved GED prediction with fine-tuning. For comparison, we also report the best result from the competitors, denoted as **Best of Competitors**.

Figure 5(c) reports the GED MAE by different GED prediction methods on AIDS and Linux. We can observe that **GEDGNN-improve** can reduce the MAE from **GEDGNN-value**, and the fine-tuning technique can further improve the solution quality.

5.6 Case Study

We conduct a case study of calculating GED between a six-node graph G_1 and a seven-node graph G_2 from AIDS in Figure 6 and finding an edit path that transforms G_1 to G_2 by our method **GEDGNN-matching**. The color of a node represents its label.

On the top part of Figure 6, we depict the matching matrix A_{match} produced by GEDGNN in a heat map. Each value x in A_{match} is a real number between 0 and 1. For the ease of presentation, we plot $int(1000x)$, e.g., 897 means 0.897 in A_{match} .

On the right, we depict the top-4 best node matchings generated by our k -best matching framework. In each matching, a cell (i, j) with value 1 means node i in G_1 matches node j in G_2 . Under each matching, we also list the weight w of this matching and the GED.

We can observe that the k -best matching framework generates the top-4 matchings with a decreasing matching weight. Matching M_2 can compute the true GED value of 2. In the bottom part of Figure 6, we visualize the edit path generated from M_2 . The node correspondence is shown by the dashed red lines in Figure 6.

6 RELATED WORK

Classical GED Computation. Classical solutions to GED computation try to design some algorithms [4, 7] to find exact GED values given a graph pair. However, due to the NP-hardness [5] of GED computation, the exact methods suffer from huge computation costs when the graph size increases. To this end, some heuristics have been proposed in recent years to provide an accurate estimation to the real GED. The core idea of early algorithms is simplifying some procedures of an exact algorithm. For example, in A^* -beam search algorithm [22], the size of the queue is bounded by a *beamsize*, so that the new coming nodes are ignored when the queue is full.

GNN-based GED Computation. GNN is a powerful tool to generate graph embedding, and is widely used in node classification [13, 16, 27] and graph classification [26, 31], link prediction and solving classical graph problems such as subgraph counting [28, 34, 35] and GED computation [1, 2, 32]. SimGNN [2] is an early work that uses GNN to predict GED. It proposes a neural tensor network (NTN) module which simply uses the *graph level embedding* of a given graph pair as input and outputs a similarity score as the predicted GED. Please note that in our model GEDGNN, the cross matrix module receives the *node level embedding* as input and hence can explicitly construct a node matching matrix. This essential difference determines that SimGNN cannot predict the node matching relation or generate an edit path. The recent work TaGSim [1] creatively splits GED prediction into predicting the number of each type of graph edit operations, the sum of which is the predicted GED. However, it proposes a more concise network architecture using graph aggregation layer (GAL), which cannot generate an edit path due to the same reason as SimGNN. In essence, Noah [32] is an A^* -beam search algorithm [22] equipped with a GNN model called graph path network (GPN). Although GPN has a sophisticated loss function that uses both ground-truth GED and edit path for training, it can merely predict a GED value like SimGNN and TaGSim. Noah improves A^* -beam search by using a pre-trained GPN as an advanced estimation function and can generate an edit path, but it still costs exponential time to search for the best node matching as A^* -beam search.

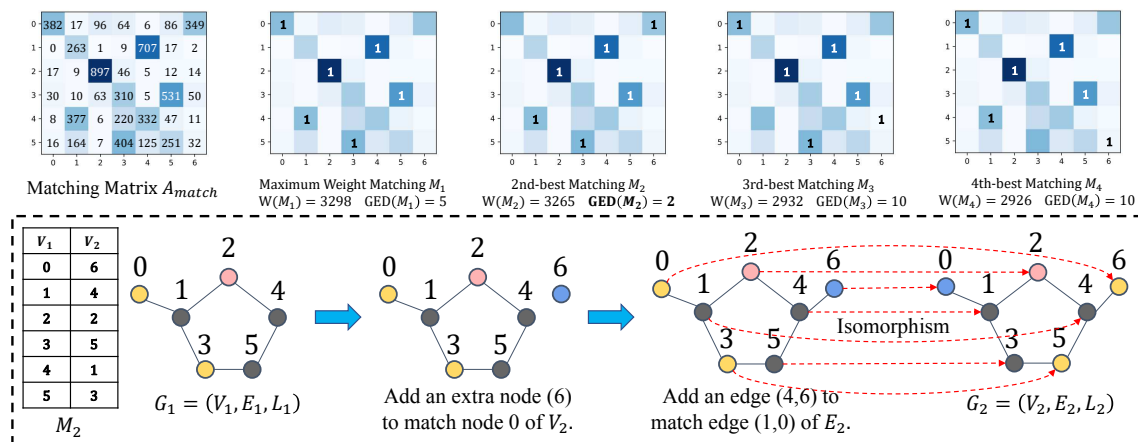


Figure 6: A Case Study.

Paassen et al. [23] consider a time series of graphs G_1, G_2, \dots, G_t and propose a graph edit network which can predict the graph snapshot at the next time G_{t+1} from G_t . It uses adjacent historical graph snapshots (G_i, G_{i+1}) and the corresponding graph edit path between them to train the model in order to capture the temporal edit pattern between a graph snapshot and its successor. Then given G_t , the trained model can predict an edit path and generate G_{t+1} by applying the edit path on G_t . It has a different goal from our model: it predicts G_{t+1} from G_t based on the learned temporal edit pattern, whereas our model computes the edit path as short as possible given a graph pair (G_1, G_2) with no temporal relation.

Graph/Node Matching. Many graph problems involve computing a node matching between two graphs. But node matching is generated and used for different purposes in different problems such as graph alignment [20, 33], GED computation [7, 10, 22, 25, 32], sub-graph matching/counting [24], and image feature matching [9, 14]. Graph alignment is widely used for solving entity resolution, social network alignment, protein-protein interaction network alignment, and so on. It is related to GED computation, but these two problems still have differences in terms of ground truth definition, objective and matching scope. The ground truth of graph alignment is usually unique and application specific. In contrast, GED computation seeks the shortest graph edit path. There may exist multiple graph edit paths of the same minimum length, each of which can be regarded as the ground truth. Accordingly, the objective of graph alignment is to maximize the node alignment accuracy with respect to the ground truth, whereas the objective of GED computation is to minimize the length of the graph edit path. In addition, in terms of the matching scope, graph alignment may only align a subset of the nodes between two graphs, while the remaining nodes may not align with each other. For example, among the users in two social networks, only a subset of them appear in both networks and can be aligned. The remaining users only appear in one of the two social networks and thus cannot be aligned. Such users that only appear in one social network are not included in the ground truth, nor considered in performance evaluation. In contrast, GED computation requires transforming one graph until it is isomorphic to the other. Thus the GED computation problem cannot be solved well

by existing graph alignment models or graph matching methods designed for other purposes.

Graph Similarity Computation. Graph similarity computation is a more general topic, which aims to compute or learn a metric to measure the similarity between two graphs. GED [5], maximal common subgraph [6] and other learning-based similarity measures [3, 18] are all reasonable metrics and can be used for downstream tasks such as graph similarity search and clustering. A recent survey [21] gives a comprehensive review of graph similarity learning.

7 CONCLUSION

In this paper, we study how to compute graph edit distance (GED) with edit path via machine learning models. Unlike existing models which treat GED computation as a regression task and can only predict the GED value, we focus on the essential target of GED, i.e., how to convert one graph to the other. Specifically, we propose a novel deep learning framework that solves the GED problem in a two-step manner: 1) The graph neural network GEDGNN is in charge of predicting the GED value and a matching matrix; and 2) A post-processing algorithm based on k -best matching is used to extract multiple node matchings from the matching matrix generated by GEDGNN. The best of them will finally lead to a high-quality edit path. The post-processing algorithm is a key innovation that makes up the gap between what graph neural networks can produce (i.e., the matching matrix) and an actual solution to the GED problem. Extensive experiments confirm the effectiveness of our framework. With regard to the accuracy of GED value prediction and the quality of graph edit path, our proposed framework outperforms the state-of-the-art algorithms significantly.

ACKNOWLEDGMENTS

The work was supported by grants from the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14217622], NSFC Grant No. U1936205, and CUHK Direct Grant No. 4055159. Tingyang Xu and Hong Cheng are the corresponding authors of this paper.

REFERENCES

- [1] Jiyang Bai and Peixiang Zhao. 2021. TaGSim: Type-Aware Graph Similarity Learning and Computation. *PVLDB* 15, 2 (2021), 335–347.
- [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 *WSDM*. 384–392.
- [3] Yunsheng Bai, Hao Ding, Ken Gu, Yizhou Sun, and Wei Wang. 2020. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. In *AAAI*, Vol. 34. 3219–3226.
- [4] David B. Blumenthal and Johann Gamper. 2020. On the exact computation of the graph edit distance. *Pattern Recognition Letters* 134 (2020), 46–57.
- [5] H. Bunke. 1997. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters* 18, 8 (1997), 689–694.
- [6] Horst Bunke and Kim Shearer. 1998. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters* 19, 3-4 (1998), 255–259.
- [7] Lijun Chang, Xing Feng, Xuemin Lin, Lu Qin, Wenjie Zhang, and Dian Ouyang. 2020. Speeding Up GED Verification for Graph Similarity Search. In *ICDE*. 793–804.
- [8] Chandra R. Chegireddy and Horst W. Hamacher. 1987. Algorithms for finding K-best perfect matchings. *Discrete Applied Mathematics* 18, 2 (1987), 155–165.
- [9] Chaoqi Chen, Weiping Xie, Wenbing Huang, Yu Rong, Xinghao Ding, Yue Huang, Tingyang Xu, and Junzhou Huang. 2019. Progressive Feature Alignment for Unsupervised Domain Adaptation. In *CVPR*. 627–636.
- [10] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. 2011. Speeding up graph edit distance computation through fast bipartite matching. In *GbRPR*. Springer, 102–111.
- [11] Karam Gouda and Mona Arafa. 2015. An improved global lower bound for graph edit similarity search. *Pattern Recognition Letters* 58 (2015), 8–14.
- [12] Karam Gouda and Mosab Hassaan. 2016. CSI_GED: An efficient approach for graph edit similarity computation. In *ICDE*. 265–276.
- [13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*. 1024–1034.
- [14] Bo Jiang, Pengfei Sun, Jin Tang, and Bin Luo. 2022. GLMNet: Graph Learning-Matching Networks for Feature Matching. *Pattern Recognition* 121 (2022), 108167.
- [15] Roy Jonker and Anton Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* 38, 4 (1987), 325–340.
- [16] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.
- [17] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97.
- [18] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *ICML*. 3835–3845.
- [19] Yongjiang Liang and Peixiang Zhao. 2017. Similarity Search in Graph Databases: A Multi-Layered Indexing Approach. In *ICDE*. 783–794.
- [20] Zhehan Liang, Yu Rong, Chenxin Li, Yunlong Zhang, Yue Huang, Tingyang Xu, Xinghao Ding, and Junzhou Huang. 2021. Unsupervised Large-Scale Social Network Alignment via Cross Network Embedding. In *CIKM*. 1008–1017.
- [21] Guixiang Ma, Nesreen K Ahmed, Theodore L Willke, and Philip S Yu. 2021. Deep graph similarity learning: A survey. *Data Mining and Knowledge Discovery* 35, 3 (2021), 688–725.
- [22] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. 2006. Fast suboptimal algorithms for the computation of graph edit distance. In *SPR-SSPR*. 163–172.
- [23] Benjamin Paassen, Daniele Grattarola, Daniele Zambon, Cesare Alippi, and Barbara Hammer. 2021. Graph Edit Networks. In *ICLR*.
- [24] Romain Raveaux, Jean-Christophe Burie, and Jean-Marc Ogier. 2010. A graph matching method and a graph matching distance based on subgraph assignments. *Pattern Recognition Letters* 31, 5 (2010), 394–406.
- [25] 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.
- [26] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying WEI, Wenbing Huang, and Junzhou Huang. 2020. Self-Supervised Graph Transformer on Large-Scale Molecular Data. In *NeurIPS*, Vol. 33. 12559–12571.
- [27] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2019. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *ICLR*.
- [28] Hanchen Wang, Rong Hu, Ying Zhang, Lu Qin, Wei Wang, and Wenjie Zhang. 2022. Neural Subgraph Counting with Wasserstein Estimator. In *SIGMOD*. 160–175.
- [29] Xiaoli Wang, Xiaofeng Ding, Anthony K.H. Tung, Shanshan Ying, and Hai Jin. 2012. An Efficient Graph Indexing Method. In *ICDE*. 210–221.
- [30] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *ICLR*.
- [31] Jinyu Yang, Peilin Zhao, Yu Rong, Chaochao Yan, Chunyuan Li, Hehuan Ma, and Junzhou Huang. 2021. Hierarchical Graph Capsule Network. In *AAAI*, Vol. 35. 10603–10611.
- [32] Lei Yang and Lei Zou. 2021. Noah: Neural-optimized A* Search Algorithm for Graph Edit Distance Computation. In *ICDE*. 576–587.
- [33] Dezhong Yao, Yuhong Gu, Gao Cong, Hai Jin, and Xinqiao Lv. 2022. Entity Resolution with Hierarchical Graph Attention Networks. In *SIGMOD*. 429–442.
- [34] Kangfei Zhao, Jeffrey Xu Yu, Qiyan Li, Hao Zhang, and Yu Rong. 2023. Learned sketch for subgraph counting: a holistic approach. *The VLDB Journal* (2023), 1–26.
- [35] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *SIGMOD*. 2142–2155.
- [36] Xiang Zhao, Chuan Xiao, Xuemin Lin, Wei Wang, and Yoshiharu Ishikawa. 2013. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal* 22, 6 (2013), 727–752.
- [37] Xiang Zhao, Chuan Xiao, Xuemin Lin, Wenjie Zhang, and Yang Wang. 2018. Efficient structure similarity searches: a partition-based approach. *The VLDB Journal* 27, 1 (2018), 53–78.
- [38] Weiguo Zheng, Lei Zou, Xiang Lian, Dong Wang, and Dongyan Zhao. 2014. Efficient graph similarity search over large graph databases. *TKDE* 27, 4 (2014), 964–978.