

# Combinatorial Learning of Graph Edit Distance via Dynamic Embedding

Runzhong Wang<sup>1,2</sup> Tianqi Zhang<sup>1,2</sup> Tianshu Yu<sup>3</sup> Junchi Yan<sup>1,2\*</sup> Xiaokang Yang<sup>2</sup>

<sup>1</sup> Department of Computer Science and Engineering, Shanghai Jiao Tong University

<sup>2</sup> MoE Key Lab of Artificial Intelligence, Shanghai Jiao Tong University <sup>3</sup> Arizona State University

{runzhong.wang, lygztq, yanjunchi, xkyang}@sjtu.edu.cn tianshuy@asu.edu

## Abstract

*Graph Edit Distance (GED) is a popular similarity measurement for pairwise graphs and it also refers to the recovery of the edit path from the source graph to the target graph. Traditional A\* algorithm suffers scalability issues due to its exhaustive nature, whose search heuristics heavily rely on human prior knowledge. This paper presents a hybrid approach by combining the interpretability of traditional search-based techniques for producing the edit path, as well as the efficiency and adaptivity of deep embedding models to achieve a cost-effective GED solver. Inspired by dynamic programming, node-level embedding is designated in a dynamic reuse fashion and suboptimal branches are encouraged to be pruned. To this end, our method can be readily integrated into A\* procedure in a dynamic fashion, as well as significantly reduce the computational burden with a learned heuristic. Experimental results on different graph datasets show that our approach can remarkably ease the search process of A\* without sacrificing much accuracy. To our best knowledge, this work is also the first deep learning-based GED method for recovering the edit path.*

## 1. Introduction

Graph edit distance (GED) is a popular similarity measurement between graphs, which lies in the core of many vision and pattern recognition tasks including image matching [10], signature verification [27], scene-graph edition [9], drug discovery [30], and case-based reasoning [46]. In general, GED algorithms aim to find an optimal edit path from source graph to target graph with minimum edit cost, which is inherently an NP-complete combinatorial problem [2]:

$$GED(\mathcal{G}_1, \mathcal{G}_2) = \min_{(e_1, \dots, e_l) \in \gamma(\mathcal{G}_1, \mathcal{G}_2)} \sum_{i=1}^l c(e_i) \quad (1)$$

where  $\gamma(\mathcal{G}_1, \mathcal{G}_2)$  denote the set of all possible “edit paths” transforming source graph  $\mathcal{G}_1$  to target graph  $\mathcal{G}_2$ .  $c(e_i)$  mea-

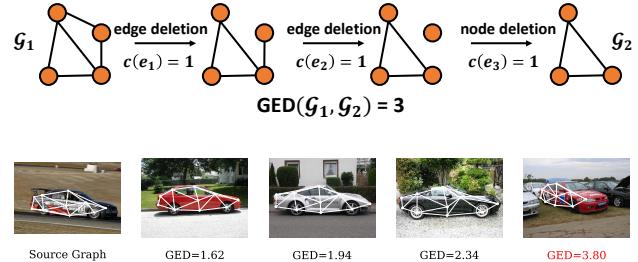


Figure 1. Top: an edit path between two simple graphs  $\mathcal{G}_1, \mathcal{G}_2$ . Bottom: an example of querying images via GED, where only geometric information is involved. The last image shows an “unsimilar” image based on GED measurement.

sures the cost of edit operation  $e_i$ .

Exact GED solvers [2, 32] guarantee to find the optimal solution under dynamic condition, at the cost of poor scalability on large graphs, and these exact solvers heavily rely on heuristics to estimate the corresponding graph similarity based on the current partial solution. Recent efforts in deep graph similarity learning [3, 4, 26] adopt graph neural networks [22, 34] to directly regress graph similarity scores, without explicitly incorporating the intrinsic combinatorial nature of GED, hence fail to recover the edit path. However, the edit path is often of the central interest in many applications [9, 10] and most GED works [2, 31, 13, 45, 32] still are more focused on finding the edit path itself.

As the growth of graph size, it calls for more scalable GED solvers which are meanwhile expected to recover the exact edit path. However, these two merits cannot both hold by existing methods. As discussed above, deep learning-based solvers have difficulty in recovering the edit path while the learning-free methods suffer scalability issue. In this paper, we are aimed to design a hybrid solver by combining the best of the two worlds.

Specifically, we resort to A\* algorithm [32] which is a popular solution among open source GED softwares [8, 20], and we adopt neural networks to predict similarity scores which are used to guide A\* search, in replacement of man-

\*Junchi Yan is the corresponding author.

ually designed heuristics in traditional A\*. We want to highlight our proposed Graph Edit Neural Network (GENN) in two aspects regarding the dynamic programming concepts: Firstly, we propose to reuse the previous embedding information given a graph modification (e.g. node deletion) where among the states of A\* search tree the graph nodes are deleted progressively<sup>1</sup>; Secondly, we propose to learn more effective heuristic to avoid unnecessary exploration over suboptimal branches to achieve significant speed-up.

The contributions made in this paper are:

1) We propose the first (to our best knowledge) deep network solver for GED, where a search tree state selection heuristic is learned by dynamic graph embedding. It outperforms traditional heuristics in efficacy.

2) Specifically, we devise a specific graph embedding method in the spirit of dynamic programming to reuse the previous computation to the utmost extent. In this sense, our method can be naturally integrated with the A\* procedure where a dynamical graph similarity prediction is involved after each graph modification, achieving much lower complexity compared to vanilla graph embeddings.

3) Experimental results on real-world graph data show that our learning-based approach achieves higher accuracy than state-of-the-art manually designed inexact solvers [13, 31]. It also runs much faster than A\* exact GED solvers [6, 32] that perform exhaustive search to ensure the global optimum, with comparable accuracy.

## 2. Related Work

### 2.1. Traditional GED Solvers

**Exact GED solvers.** For small-scale problems, an exhaustive search can be used to find the global optimum. Exact methods are mostly based on tree-search algorithms such as A\* algorithm [32], whereby a priority queue is maintained for all pending states to search, and the visiting order is controlled by the cost of the current partial edit path and a heuristic prediction on the edit distance between the remaining subgraphs [31, 45]. Other combinatorial optimization techniques, e.g. depth-first branch-and-bound [2] and linear programming lower bound [25] can also be adopted to prune unnecessary branches in the searching tree. However, exact GED methods are too time-consuming and they suffer from poor scalability on large graphs [1].

**Inexact GED solvers** aim to mitigate the scalability issue by predicting sub-optimal solutions in (usually) polynomial time. To our knowledge, bipartite matching based methods [13, 31, 45] so far show competitive trade-off between time and accuracy, where edge edition costs are encoded into node costs and the resulting bipartite matching problem can be solved in polynomial time by either Hungarian

<sup>1</sup>To distinguish the “nodes” in graphs and the “nodes” in the search tree, we name “state” for the ones in the search tree.

ian [23, 31] or Volgenant-Jonker [13, 19] algorithm. Beam search [20] is the greedy version of the exact A\* algorithm. Another line of works namely approximate graph matching [11, 18, 39, 41, 43, 48] are closely related to inexact GED, and there are efforts adopting graph matching methods e.g. IPFP [24] to solve GED problems [7]. Two drawbacks in inexact solvers are that they rely heavily on human knowledge and their solution qualities are relatively poor.

### 2.2. Deep Graph Similarity Learning

**Regression-based Similarity Learning.** The recent success in machine learning on non-euclidean data (i.e. graphs) via GNNs [14, 22, 34, 49] has encouraged researchers to design approximators for graph similarity measurements such as GED. SimGNN [3] first formulates graph similarity learning as a regression task, where its GCN [22] and attention [36] layers are supervised by GED scores solved by A\* [20]. Bai *et al.* [4] extends their previous work by processing a multi-scale node-wise similarity map using CNNs. Li *et al.* [26] propose a cross-graph module in feed-forward GNNs which elaborates similarity learning. Such a scheme is also adopted in information retrieval, where [12] adopts a convolutional net to predict the edit cost between texts. However, all these regression models can not predict an edit path, which is mandatory in the GED problem.

**Deep Graph Matching.** As another combinatorial problem closely related to GED, there is increasing attention in developing deep learning graph matching approaches [16, 17, 37] since the seminal work [44], and many researchers [33, 37, 38, 42] start to take a combinatorial view of graph matching learning rather than a regression task. Compared to graph similarity learning methods, deep graph matching can predict the edit path, but they are designated to match similarly structured graphs and lack particular mechanisms to handle node/edge insertion/deletions. Therefore, modification is needed to fit deep graph matching methods into GED, which is beyond the scope of this paper.

### 2.3. Dynamic Graph Embedding

The major line of graph embedding methods [14, 22, 34, 49] assumes that graphs are static which limit their application on real-world graphs that evolve over time. A line of works namely dynamic graph embedding [29, 28, 47] aims to solve such issue, whereby recurrent neural networks (RNNs) are typically combined with GNNs to capture the temporal information in graph evolution. The applications include graph sequence classification [28], dynamic link prediction [29], and anomaly detection [47]. Dynamic graph embedding is also encountered in our GED learning task, however, all these aforementioned works cannot be applied to our setting where the graph structure evolves at different states of the search tree, instead of time steps.

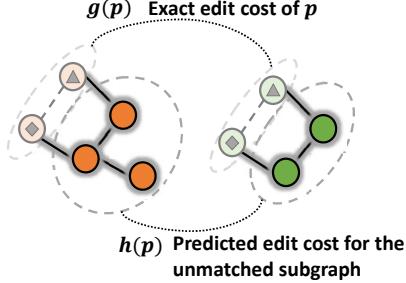


Figure 2. A partial edit path as one state of A\* search tree. Given the partial solution  $p = (u_{\blacklozenge} \rightarrow v_{\blacklozenge}, u_{\blacktriangle} \rightarrow v_{\blacktriangle})$ , the edge edition  $(u_{\blacklozenge}u_{\blacktriangle} \rightarrow v_{\blacklozenge}v_{\blacktriangle})$  can be induced from node editions.

### 3. Our Approach

In this section, we first introduce the A\* algorithm for GED in Sec. 3.1, then we present our efficient dynamic graph embedding approach GENN for A\* in Sec. 3.2.

#### 3.1. Preliminaries on A\* Algorithm for GED

To exactly solve the GED problem, researchers usually adopt tree-search based algorithms which traverse all possible combinations of edit operations. Among them, A\* algorithm is rather popular [31, 20, 32, 8] and we base our learning method on it. In this section, we introduce notations for GED and discuss the key components in A\* algorithm.

GED aims to find the optimal edit path with minimum edit cost, to transform the source graph  $\mathcal{G}_1 = (V_1, E_1)$  to the target graph  $\mathcal{G}_2 = (V_2, E_2)$ , where  $|V_1| = n_1, |V_2| = n_2$ . We denote  $V_1 = \{u_1, \dots, u_{n_1}\}$ ,  $V_2 = \{v_1, \dots, v_{n_2}\}$  as the nodes in the source graph and the target graph, respectively, and  $\epsilon$  as the “void node”. Possible node edit operations include node substitution  $u_i \rightarrow v_j$ , node insertion  $\epsilon \rightarrow v_j$  and node deletion  $u_i \rightarrow \epsilon$ , and the cost of each operation is defined by the problem. As shown in Fig. 2, the edge editions can be induced given node editions, therefore only node editions are explicitly considered in A\* algorithm.<sup>2</sup>

Alg. 1 illustrates a standard A\* algorithm in line with [31, 32]. A priority queue is maintained where each state of the search tree contains a partial solution to the GED problem. As shown in Fig. 2, the priority of each state is defined as the summation of two metrics:  $g(p)$  representing the cost of the current partial solution which can be computed exactly, and  $h(p)$  means the heuristic prediction of GED between the unmatched subgraphs. A\* always explores the state with minimum  $g(p) + h(p)$  at each iteration and the optimality is guaranteed if  $h(p) \leq h^{opt}(p)$  holds for all par-

<sup>2</sup>Node substitution can be viewed as node-to-node matching between two graphs, and node insertion/deletion can be viewed as matching nodes in source/target graph to the void node, respectively. The concepts “matching” and “edition” may interchange with each other through this paper.

---

#### Algorithm 1: A\* Algorithm for Exact GED

---

```

Input: Graphs  $\mathcal{G}_1 = (V_1, E_1), \mathcal{G}_2 = (V_2, E_2)$ , where
 $V_1 = \{u_1, \dots, u_{n_1}\}, V_2 = \{v_1, \dots, v_{n_2}\}$ 
1 Initialize OPEN as an empty priority queue;
2 Insert  $(u_1 \rightarrow w)$  to OPEN for all  $w \in V_2$ ;
3 Insert  $(u_1 \rightarrow \epsilon)$  to OPEN;
4 while no solution is found do
5   Select  $p$  with minimum  $(g(p) + h(p))$  in OPEN;
6   if  $p$  is a valid edit path then
7     return  $p$  as the solution;
8   else
9     Let  $p$  contains  $\{u_1, \dots, u_k\} \subseteq V_1$  and  $W \subseteq V_2$ ;
10    if  $k \leq n_1$  then
11      Insert  $p \cup (u_{k+1} \rightarrow v_i)$  to OPEN for all
12         $v_i \in V_2 \setminus W$ ;
13      Insert  $p \cup (u_{k+1} \rightarrow \epsilon)$  to OPEN;
14    else
15      Insert  $p \cup \bigcup_{v_i \in V_2 \setminus W} (\epsilon \rightarrow v_i)$  to OPEN;

```

---

**Output:** An optimal edit path from  $\mathcal{G}_1$  to  $\mathcal{G}_2$ .

---

tial solutions [31], where  $h^{opt}(p)$  means the optimal edit cost between the unmatched subgraphs.

A proper  $h(p)$  is rather important to speed up the algorithm, and we discuss three variants of A\* accordingly: **1)** If  $h(p) = h^{opt}(p)$ , one can directly find the optimal path greedily. However, computing  $h^{opt}(p)$  requires another exponential-time solver which is intractable. **2)** Heuristics can be utilized to predict  $h(p)$  where  $0 \leq h(p) \leq h^{opt}(p)$ . Hungarian bipartite heuristic [32] is among the best-performing heuristic where the time complexity is  $\mathcal{O}((n_1 + n_2)^3)$ . In our experiments, **Hungarian-A\*** [32] is adopted as the baseline traditional exact solver. **3)** Plain-A\* is the simplest, where it always holds  $h(p) = 0$  and such strategy introduces no overhead when computing  $h(p)$ . However, the search tree may become too large without any “look ahead” on the future cost.

The recent success of graph similarity learning [3, 4, 26] inspires us to predict high-quality  $h(p)$  which is close to  $h^{opt}(p)$  in a cost-efficient manner via learning. In this paper, we propose to mitigate the scalability issue of A\* by predicting  $h(p)$  via dynamic graph embedding networks, where  $h(p)$  is efficiently learned and predicted and the sub-optimal branches in A\* are pruned. It is worth noting that we break the optimality condition  $h(p) \leq h^{opt}(p)$ , but the loss of accuracy is acceptable, as shown in experiments.

#### 3.2. Graph Edit Neural Network

An overview of our proposed Graph Edit Neural Network-based A\* (GENN-A\*) learning algorithm is shown in Fig. 3. Our GENN-A\* can be split into node embedding module (Sec. 3.2.1), dynamic embedding technique (Sec. 3.2.2), graph similarity prediction module

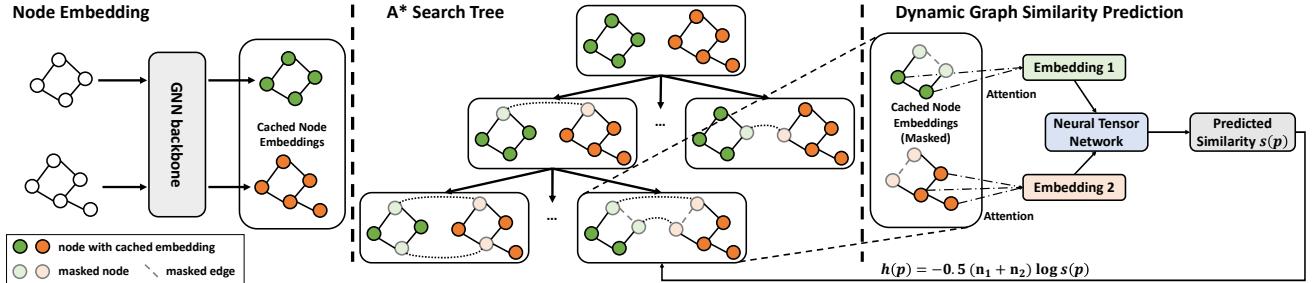


Figure 3. Our proposed GENN-A\*. *Left:* Node embedding. Input graphs are fed into GNN to extract node-level embeddings. These embeddings are cached to be reused in the following computation. *Middle:* A\* search tree. The state in the search tree is a matching of nodes between graphs. All matched nodes are masked (light color) and the unmatched subgraphs (dark color) will be involved to predict  $h(p)$ . *Right:* Dynamic graph similarity prediction. Cached embeddings are loaded for nodes in the unmatched subgraphs, and a graph-level embedding is obtained via attention. Finally the predicted graph similarity  $s(p) \in (0, 1)$  is obtained from graph-level embeddings by neural tensor network and transformed to the heuristic score  $h(p)$ .

(Sec. 3.2.3) and finally the training procedure (Sec. 3.2.4).

### 3.2.1 Node Embedding Module

The overall pipeline of our GENN is built in line with SimGNN [3], and we remove the redundant histogram module in SimGNN in consideration of efficiency. Given input graphs, node embeddings are computed via GNNs.

**Initialization.** Firstly, the node embeddings are initialized as the one-hot encoding of the node degree. For graphs with node labels (e.g. molecule graphs), we encode the node labels by one-hot vector and concatenate it to the degree embedding. The edges can be initialized as weighted or unweighted according to different definitions of graphs.

**GNN backbone.** Based on different types of graph data, Graph Convolutional Network (GCN) [22] is utilized for ordinary graph data (e.g. molecule graphs and program graphs) and SplineCNN [14] is adopted for graphs built from 2D images, considering the recent success of adopting spline kernels to learn geometric features [16, 33]. The node embeddings obtained by the GNN backbone are cached for further efficient dynamic graph embedding. We build three GNN layers for our GENN in line with [3].

### 3.2.2 Dynamic Embedding with A\* Search Tree

A\* is inherently a dynamic programming (DP) algorithm where matched nodes in partial solutions are progressively masked. When solving GED, each state of A\* contains a partial solution and in our method embedding networks are adopted to predict the edit distance between two unmatched subgraphs. At each state, one more node is masked out in the unmatched subgraph compared to its parent state. Such a DP setting differs from existing so-called dynamic graph embedding problems [29, 28, 47] and calls for efficient cues since the prediction of  $h(p)$  is encountered at every state

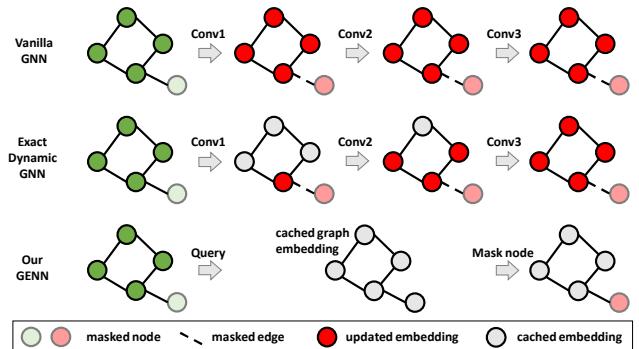


Figure 4. Comparison of three graph neural network variants for dynamic graph embedding in A\* algorithm. We assume three graph convolution layers in line with our implementation. In vanilla GNN, a complete forward pass is required for all nodes which contains redundant operations. The exact dynamic GNN caches all intermediate embeddings and only the 3-hop neighbors of the masked node are updated. Finally, our proposed GENN requires no convolution operation and is the most efficient.

of the search tree. In this section, we discuss and compare three possible dynamic embedding approaches, among which our proposed GENN is built based on DP concepts.

**Vanilla GNN.** The trivial way of handling the dynamic condition is that when the graph is modified, a complete feed-forward pass is called for all nodes in the new graph. However, such practice involves redundant computation, which is discussed as follows. We denote  $n$  as the number of nodes,  $F$  as embedding dimensions, and  $K$  as the number of GNN layers. Assuming fully-connected graph as the worst case, the time complexity of vanilla GNN is  $\mathcal{O}(n^2FK + nF^2K)$  and no caching is needed.

**Exact Dynamic GNN.** As shown in the second row of Fig. 4, when a node is masked, only the embeddings of neighboring nodes are affected. If we cache all intermediate embeddings of the forward pass, one can compute the

exact embedding at a minimum computational cost. Based on the message-passing nature of GNNs, at the  $k$ -th convolution layer, only the  $k$ -hop neighbors of the masked node are updated. However, the worst-case time complexity is still  $\mathcal{O}(n^2FK + nF^2K)$  (for fully-connected graphs), and it requires  $\mathcal{O}(nFK)$  memory cache for all convolution layers. If all possible subgraphs are cached for best time efficiency, the memory cost grows to  $\mathcal{O}(n^{2^n}FK)$  which is unacceptable. Experiment result shows that the speed-up of this strategy is negligible with our testbed.

**Our GENN.** As shown in the last row of Fig. 4, we firstly perform a forward convolution pass and cache the embeddings of the last convolution layer. During A\* algorithm, if some nodes are masked out, we simply delete their embeddings from the last convolution layer and feed the remaining embeddings into the similarity prediction module. Our GENN involves single forward pass which is negligible, and the time complexity of loading caches is simply  $\mathcal{O}(1)$  and the memory consumption of caching is  $\mathcal{O}(nF)$ .

Our design of the caching scheme of GENN is mainly inspired by DP: given modification on the input graph (node deletion in our A\* search case), the DP algorithm reuses the previous results for further computations in consideration of best efficiency. In our GENN, the node embeddings are cached for similarity computation on its subgraphs. In addition, DP algorithms tend to minimize the exploration space for best efficiency, and our learned  $h(p)$  prunes sub-optimal branches more aggressively than traditional heuristics which speeds up the A\* solver.

### 3.2.3 Graph Similarity Prediction

After obtaining the embedding vectors from cache, the attention module and neural tensor network are called to predict the similarity score. For notation simplicity, our discussions here are based on full-sized, original input graphs.

**Attention module for graph-level embedding.** Given node-level embeddings, the graph-level embedding is obtained through attention mechanism [36]. We denote  $\mathbf{X}_1 \in \mathbb{R}^{n_1 \times F}$ ,  $\mathbf{X}_2 \in \mathbb{R}^{n_2 \times F}$  as the node embeddings from GNN backbone. The global keys are obtained by mean aggregation followed with nonlinear transform:

$$\bar{\mathbf{X}}_1 = \text{mean}(\mathbf{X}_1), \bar{\mathbf{X}}_2 = \text{mean}(\mathbf{X}_2) \quad (2)$$

$$\mathbf{k}_1 = \tanh(\bar{\mathbf{X}}_1 \mathbf{W}_1), \mathbf{k}_2 = \tanh(\bar{\mathbf{X}}_2 \mathbf{W}_1) \quad (3)$$

where  $\text{mean}(\cdot)$  is performed on the first dimension (node dimension) and  $\mathbf{W}_1 \in \mathbb{R}^{F \times F}$  is learnable attention weights. Aggregation coefficients are computed from  $\mathbf{k}_1, \mathbf{k}_2 \in \mathbb{R}^{1 \times F}$  and  $\mathbf{X}_1, \mathbf{X}_2$ :

$$\mathbf{c}_1 = \delta(\mathbf{X}_1 \mathbf{k}_1^\top \cdot \alpha), \mathbf{c}_2 = \delta(\mathbf{X}_2 \mathbf{k}_2^\top \cdot \alpha) \quad (4)$$

where  $\alpha = 10$  is the scaling factor and  $\delta(\cdot)$  means sigmoid. The graph-level embedding is obtained by weighted summation of node embeddings based on aggregation coefficients  $\mathbf{c}_1 \in \mathbb{R}^{n_1 \times 1}$ ,  $\mathbf{c}_2 \in \mathbb{R}^{n_2 \times 1}$ :

$$\mathbf{g}_1 = \mathbf{c}_1^\top \mathbf{X}_1, \mathbf{g}_2 = \mathbf{c}_2^\top \mathbf{X}_2 \quad (5)$$

**Neural Tensor Network for similarity prediction.** Neural Tensor Network (NTN) [35] is adopted to measure the similarity between  $\mathbf{g}_1, \mathbf{g}_2 \in \mathbb{R}^{1 \times F}$ :

$$s(\mathcal{G}_1, \mathcal{G}_2) = f(\mathbf{g}_1 \mathbf{W}_2^{[1:t]} \mathbf{g}_2^\top + \mathbf{W}_3 \text{cat}(\mathbf{g}_1, \mathbf{g}_2) + \mathbf{b}) \quad (6)$$

where  $\mathbf{W}_2 \in \mathbb{R}^{F \times F \times t}$ ,  $\mathbf{W}_3 \in \mathbb{R}^{t \times 2F}$ ,  $\mathbf{b} \in \mathbb{R}^t$  are learnable, the first term means computing  $\mathbf{g}_1 \mathbf{W}_2[:, :, i] \mathbf{g}_2^\top$  for all  $i \in [1 \dots t]$  and then stacking them,  $f : \mathbb{R}^t \rightarrow (0, 1)$  denotes a fully-connected layer with sigmoid activation, and  $\text{cat}(\cdot)$  means to concat along the last dimension.  $t$  controls the number of channels in NTN and we empirically set  $t = 16$ .

In line with [3], the model prediction lies within  $(0, 1)$  which represents a normalized graph similarity score with the following connection to GED:

$$s(\mathcal{G}_1, \mathcal{G}_2) = \exp(-\text{GED}(\mathcal{G}_1, \mathcal{G}_2) \times 2/(n_1 + n_2)) \quad (7)$$

For partial edit path encountered in A\* algorithm, the predicted similarity score  $s(p)$  can be transformed to  $h(p)$  following Eq. 7:

$$h(p) = -0.5(n'_1 + n'_2) \log s(p) \quad (8)$$

where  $n'_1, n'_2$  means the number of nodes in the unmatched subgraph. The time complexities of attention and NTN are  $\mathcal{O}((n'_1 + n'_2)F^2)$  and  $\mathcal{O}(n'_1 n'_2 F t)$ , respectively. Since the convolution layers are called only once which is negligible, and the time complexity of loading cached GENN embedding is  $\mathcal{O}(1)$ , the overall time complexity of each prediction is  $\mathcal{O}((n'_1 + n'_2)F^2 + n'_1 n'_2 F t)$ . Our time complexity is comparable to the best-known learning-free prediction of  $h(p)$  [32] which is  $\mathcal{O}((n'_1 + n'_2)^3)$ .

### 3.2.4 Supervised Dynamic Graph Learning

The training of our GENN consists of two steps: Firstly, GENN weights are initialized with graph similarity score labels from the training dataset. Secondly, the model is fine-tuned with the optimal edit path solved by A\* algorithm. The detailed training procedure is listed in Alg. 2.

Following deep graph similarity learning peer methods [3, 4], our GENN weights are supervised by ground truth labels provided by the dataset. For datasets with relatively small graphs, optimal GED scores can be solved as ground truth labels. In cases where optimal GEDs are not available, we can build the training set based on other meaningful measurements, e.g. adopting semantic node matching ground truth to compute GED labels.

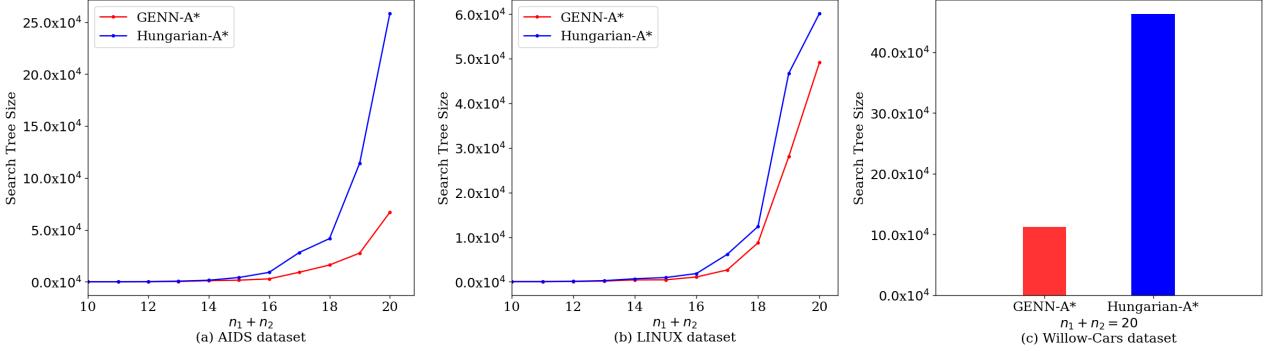


Figure 5. Average search tree size w.r.t. problem size ( $n_1 + n_2$ ). The search tree reduces significantly when the problem size grows, especially on more challenging AIDS and Willow-Cars where about  $\times 5$  and  $\times 4$  reductions of state are achieved respectively via GENN.

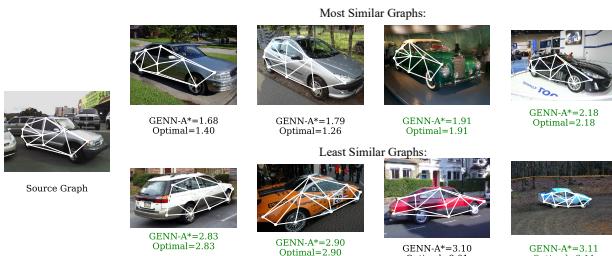


Figure 6. The visualization of a query on Willow-Cars dataset by GENN-A\*. All of the 4 most similar graphs are close to the source graph in terms of poses and graph structures, yet the 4 least similar ones vary greatly in their poses and appearances. Green letters mean our GENN-A\* solves the optimal GED.

#### Algorithm 2: The Training Procedure of GENN-A\*

**Input:** Training set of graphs pairs  $\{(\mathcal{G}_i, \mathcal{G}_j)\}$  with similarity score labels  $\{s^{gt}(\mathcal{G}_i, \mathcal{G}_j)\}$ .

1 **while** not converged **do** # training with GT labels  
 2   Randomly sample  $(\mathcal{G}_i, \mathcal{G}_j)$  from training set;  
 3   Compute  $s(\mathcal{G}_i, \mathcal{G}_j)$  by vanilla GENN;  
 4   Update parameters by  $MSE(s(\mathcal{G}_i, \mathcal{G}_j), s^{gt}(\mathcal{G}_i, \mathcal{G}_j))$ ;

5 **while** not converged **do** # finetune with optimal path  
 6   Randomly sample  $(\mathcal{G}_i, \mathcal{G}_j)$  from training set;  
 7   Solve the optimal edit path  $p^*$  and  $GED(p^*)$  by A\*;  
 8   Call GENN on  $(\mathcal{G}_i, \mathcal{G}_j)$  and cache the embeddings;  
 9   **for** partial edit path  $p \subseteq p^*$  **do**  
 10     compute  $g(p)$  and  $h^{opt}(p) = GED(p^*) - g(p)$ ;  
 11      $s^{opt}(p) = \exp(-2h^{opt}(p)/(n'_1 + n'_2))$ ;  
 12     compute  $s(p)$  from cached GENN embeddings;  
 13     Update parameters by  $MSE(s(p), s^{opt}(p))$ ;

**Output:** GENN with learned parameters.

We further propose a finetuning scheme of GENN to better suit the A\* setting. However, tuning GENN with the states of the search tree means we require labels of  $h^{opt}(p)$ , while solving the  $h^{opt}(p)$  for an arbitrary partial edit path is again NP-complete. Instead of solving as many  $h^{opt}(p)$  as needed, here we propose an efficient way of obtaining

multiple  $h^{opt}(p)$  labels by solving the GED only once.

**Theorem 1. (Optimal Partial Cost)** Given an optimal edit path  $p^*$  and the corresponding  $GED(p^*)$ , for any partial edit path  $p \subseteq p^*$ , there holds  $g(p) + h^{opt}(p) = GED(p^*)$ .

*Proof.* If  $g(p) + h^{opt}(p) > GED(p^*)$ , then the minimum edit cost following  $p$  is larger than  $GED(p^*)$ , therefore  $p$  is not a partial optimal edit path, which violates  $p \subseteq p^*$ . If  $g(p) + h^{opt}(p) < GED(p^*)$ , it means that there exists a better edit path whose cost is smaller than  $GED(p^*)$ , which violates the condition that  $p^*$  is the optimal edit path. Thus,  $g(p) + h^{opt}(p) = GED(p^*)$ .  $\square$

Based on Theorem 1, there holds  $h^{opt}(p) = GED(p^*) - g(p)$  for any partial optimal edit path. Therefore, if we solve an optimal  $p^*$  with  $m$  node editions,  $(2^m - 1)$  optimal partial edit paths can be used for finetuning. In experiments, we randomly select 200 graph pairs for finetuning since we find it adequate for convergence.

## 4. Experiment

### 4.1. Settings and Datasets

We evaluate our learning-based A\* method on three challenging real-world datasets: AIDS, LINUX [40], and Willow dataset [10].

**AIDS dataset** contains chemical compounds evaluated for the evidence of anti-HIV activity<sup>3</sup>. AIDS dataset is pre-processed by [3] who remove graphs more than 10 nodes and the optimal GED between any two graphs is provided. Following [3], we define the node edition cost  $c(u_i \rightarrow v_j) = 1$  if  $u_i, v_j$  are different atoms, else  $c(u_i \rightarrow v_j) = 0$ . The node insertion and deletion costs are both defined as 1. The edges are regraded as non-attributed, therefore edge substitution cost = 0 and edge insertion/deletion cost = 1. **LINUX dataset** is proposed by [40] which contains Program Dependency Graphs (PDG) from the LINUX kernel,

<sup>3</sup><https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data>

Method	Edit Path	AIDS			LINUX			Willow-Cars		
		mse ( $\times 10^{-3}$ )	$\rho$	p@10	mse ( $\times 10^{-3}$ )	$\rho$	p@10	mse ( $\times 10^{-3}$ )	$\rho$	p@10
SimGNN [3]	$\times$	1.189	0.843	0.421	1.509	0.939	0.942	-	-	-
GMN [26]	$\times$	1.886	0.751	0.401	1.027	0.933	0.833	-	-	-
GraphSim [4]	$\times$	<b>0.787</b>	0.874	0.534	<b>0.058</b>	<b>0.981</b>	<b>0.992</b>	-	-	-
GENN (ours)	$\times$	1.618	<b>0.901</b>	<b>0.880</b>	0.438	0.955	0.527	-	-	-
Beam Search [20]	$\checkmark$	12.090	0.609	0.481	9.268	0.827	<b>0.973</b>	1.820	0.815	0.725
Hungarian [31]	$\checkmark$	25.296	0.510	0.360	29.805	0.638	0.913	29.936	0.553	0.650
VJ [13]	$\checkmark$	29.157	0.517	0.310	63.863	0.581	0.287	45.781	0.438	0.512
GENN-A* (ours)	$\checkmark$	<b>0.635</b>	<b>0.959</b>	<b>0.871</b>	<b>0.324</b>	<b>0.991</b>	0.962	<b>0.599</b>	<b>0.928</b>	<b>0.938</b>

Table 1. Evaluation on benchmarks AIDS, LINUX and Willow-Cars. Our method can work either in a way involving explicit edit path generation as traditional GED solvers [31, 13, 32], or based on direct similarity computing without deriving the edit distance [3, 26, 4]. The evaluation metrics are defined and used by [3, 4]: **mse** stands for mean square error between predicted similarity score and ground truth similarity score.  $\rho$  means the Spearman’s correlation between prediction and ground truth. **p@10** means the precision of finding the closest graph among the predicted top 10 most similar ones. Willow-Cars is not compared with deep learning methods because optimal GED labels are not available for the training set. The AIDS and LINUX peer method results are quoted from [4].

and the authors of [3] also provides a pre-processed version where graphs are with maximum 10 nodes and optimal GED values are provided as ground truth. All nodes and edges are unattributed therefore the substitution cost is 0, and the insertion/deletion cost is 1.

**Willow dataset** is originally proposed by [10] for semantic image keypoint matching problem, and we validate the performance of our GENN-A\* on computer vision problems with the Willow dataset. All images from the same category share 10 common semantic keypoints. “Cars” dataset is selected in our experiment. With Willow-Cars dataset, graphs are built with 2D keypoint positions by Delaunay triangulation, and the edge edition cost is defined as  $c(\mathcal{E}_i \rightarrow \mathcal{E}_j) = |\mathcal{E}_i - \mathcal{E}_j|$  where  $\mathcal{E}_i, \mathcal{E}_j$  are the length of two edges. Edge insertion/deletion costs of  $\mathcal{E}_i$  are defined as  $|\mathcal{E}_i|$ . All edge lengths are normalized by 300 for numerical concerns. The node substitution has 0 cost, and  $c(u_i \rightarrow \epsilon) = c(\epsilon \rightarrow v_j) = \infty$  therefore node insertion/deletion are prohibited. We build the training set labels by computing the GED based on semantic keypoint matching relationship, and it is worth noting such GEDs are different from the optimal ones. However, experiment results show that such supervision is adequate to initialize the model weights of GENN.

Among all three datasets, LINUX has the simplest definition of edit costs. In comparison, AIDS has attributed nodes and Willow dataset has attributed edges, making these two datasets more challenging than LINUX dataset. In line with [3], we split all datasets by 60% for training, 20% for validation, and 20% for testing.

Our GENN-A\* is implemented with Pytorch-Geometric [15] and the A\* algorithm is implemented with Cython [5] in consideration of performance. We adopt GCN [22] for AIDS and LINUX datasets and SplineCNN [14] for 2D Euclidean data from Willow-Cars (#kernels=16). The number of feature channels are defined

Method	AIDS	LINUX	Willow-Cars
Hungarian-A* [32]	29.915	2.332	188.234
GENN-A* (ours)	<b>13.323</b>	<b>2.177</b>	<b>78.481</b>

Table 2. Averaged time (sec) for solving GED problems.

	Vanilla GNN	Exact Dynamic GNN	GENN (ours)	Hungarian [32]
time	2.329	3.145	0.417	0.358

Table 3. Averaged time (msec) of different methods to predict  $h(p)$ . Statistics are collected on LINUX dataset.

as 64, 32, 16 for three GNN layers. Adam optimizer [21] is used with 0.001 learning rate and  $5 \times 10^{-5}$  weight decay. We set batch size=128 for LINUX and AIDS, and 16 for Willow. All experiments are run on our workstation with Intel i7-7820X@3.60GHz and 64GB memory. Parallelization techniques e.g. multi-threading and GPU parallelism are not considered in our experiment.

## 4.2. Peer Methods

**Hungarian-A\*** [32] is selected as the exact solver baseline, where Hungarian bipartite matching is used to predict  $h(p)$ . We reimplement Hungarian-A\* based on our Cython implementation for fair comparison. We also select **Hungarian solver** [31] as the traditional inexact solver baseline in our experiments. It is worth noting that Hungarian bipartite matching can be either adopted as heuristic in A\* algorithm (Hungarian heuristic for A\*), or to provide a fast sub-optimal solution to GED (Hungarian solver), and readers should distinguish between these two methods. Other inexact solvers are also considered including **Beam search** [20] which is the greedy version of A\* and **VJ** [13] which is an variant from Hungarian solver.

For regression-based deep graph similarity learning methods, we compare **SimGNN** [3], **GMN** [26] and **GraphSim** [4]. Our GENN backbone can be viewed as a simplified version from these methods, because the time ef-

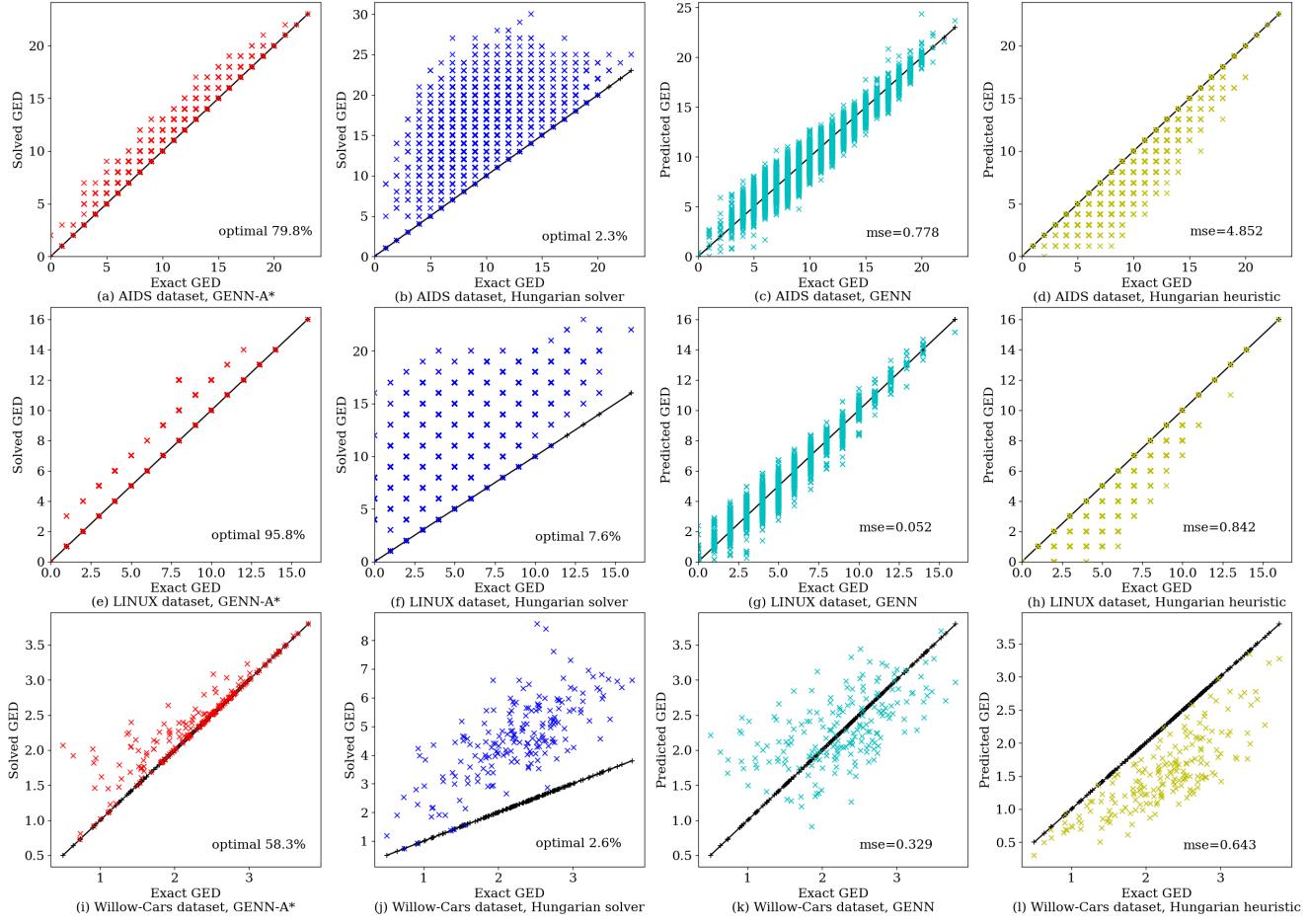


Figure 7. The scatter plots of our proposed GENN-A\* (red), inexact Hungarian solver [31] (blue, upper bound), our GENN network (cyan) and Hungarian heuristic for A\* [32] (yellow, lower bound) on AIDS, LINUX and Willow-Cars datasets. The left two columns are GED solvers and the right two columns are methods used to predict  $h(p)$  in A\* algorithm. Every dot is plotted with optimal GED value on x-axis and the solved (or predicted) GED value on y-axis. Optimal black dots are plotted as references. Our GENN-A\* (red) achieves tighter upper bounds than inexact Hungarian solver [31] (blue), where a significant amount of problems are solved to optimal. Our regression model GENN (cyan) also predicts more accurate  $h(p)$  than Hungarian heuristic [32] (yellow), resulting in reduced search tree size of GENN-A\* compared to Hungarian-A\*.

ficiency with dynamic graphs is our main concern.

### 4.3. Results and Discussions

The evaluation of AIDS, LINUX, and Willow-Cars dataset in line with [4] is presented in Tab. 1, where the problem is defined as querying a graph in the test dataset from all graphs in the training set. The similarity score is defined as Eq. 7. Our regression model GENN has comparable performance against state-of-the-art with a simplified pipeline, and our GENN-A\* best performs among all inexact GED solvers. We would like to point out that **mse** may not be a fair measurement when comparing GED solvers with regression-based models: Firstly, GED solvers can predict edit paths while such a feature is not supported by regression-based models. Secondly, the solutions of GED solvers are upper bounds of the optimal values, but

regression-based graph similarity models [3, 4, 26] predicts GED values on both sides of the optimums. Actually, one can reduce the **mse** of GED solvers by adding a bias to the predicted GED values, which is exactly what the regression models are doing.

The number of states which have been added to OPEN in Alg. 1 is plotted in Fig. 5, where our GENN-A\* significantly reduces the search tree size compared to Hungarian-A\*. Such search-tree reduction results in the speed-up of A\* algorithm, as shown in Tab. 2. Both evidences show that our GENN learns stronger  $h(p)$  than Hungarian heuristic [32] whereby redundant explorations on suboptimal solutions are pruned. We further compare the inference time of three discussed dynamic graph embedding method in Tab. 3, where our GENN runs comparatively fast against Hungarian heuristic, despite the overhead of calling Py-

Torch functions from Cython. Exact Dynamic GNN is even slower than the vanilla version, since its frequent caching and loading operations may consume additional time. It is worth noting that further speedup can be achieved by implementing all algorithms in C++ and adopting parallelism techniques, but these may be beyond the scope of this paper.

In Fig. 7 we show the scatter plot of GENN-A\* and inexact Hungarian solver [31] as GED solvers, as well as GENN and Hungarian heuristic as the prediction methods on  $h(p)$ . Our GENN-A\* benefits from the more accurate prediction of  $h(p)$  by GENN, solving the majority of problem instances to optimal. We also visualize a query example on Willow-Car images in Fig. 6 done by our GENN-A\*.

## 5. Conclusion

This paper has presented a hybrid approach for solving the classic graph edit distance (GED) problem by integrating a dynamic graph embedding network for similarity score prediction into the edit path search procedure. Our approach inherits the good interpretability of classic GED solvers as it can recover the explicit edit path between two graphs while it achieves better cost-efficiency by replacing the manual heuristics with the fast embedding module. Our learning-based A\* algorithm can reduce the search tree size and save running time, at the cost of little accuracy lost.

## Acknowledgments

This research was supported by China Major State Research Development Program (2020AAA0107600), NSFC (61972250, U19B2035).

## References

- [1] Zeina Abu-Aisheh, Benoit Gaüzère, Sébastien Bougleux, Jean-Yves Ramel, Luc Brun, Romain Raveaux, Pierre Héroux, and Sébastien Adam. Graph edit distance contest: Results and future challenges. *Pattern Recognition Letters*, 100:96–103, 2017. [2](#)
- [2] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods*, 2015. [1, 2](#)
- [3] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 384–392, 2019. [1, 2, 3, 4, 5, 6, 7, 8](#)
- [4] Yunsheng Bai, Hao Ding, Ken Gu, Yizhou Sun, and Wei Wang. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. In *AAAI*, pages 3219–3226, 2020. [1, 2, 3, 5, 7, 8](#)
- [5] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, 2011. [7](#)
- [6] Ralph Bergmann and Yolanda Gil. Similarity assessment and efficient retrieval of semantic workflows. *Information Systems*, 40:115–127, 2014. [2](#)
- [7] Sébastien Bougleux, Benoit Gaüzère, and Luc Brun. Graph edit distance as a quadratic program. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 1701–1706. IEEE, 2016. [2](#)
- [8] Lijun Chang, Xing Feng, Xuemin Lin, Lu Qin, Wenjie Zhang, and Dian Ouyang. Speeding up ged verification for graph similarity search. In *Proc. of ICDE’20*, 2020. [1, 3](#)
- [9] Lichang Chen, Guosheng Lin, Shijie Wang, and Qingyao Wu. Graph edit distance reward: Learning to edit scene graph. *European Conference on Computer Vision*, 2020. [1](#)
- [10] Minsu Cho, Kartek Alahari, and Jean Ponce. Learning graphs to match. In *ICCV*, 2013. [1, 6, 7](#)
- [11] Minsu Cho, Jungmin Lee, and Kyoung Mu Lee. Reweighted random walks for graph matching. In *Eur. Conf. Comput. Vis.*, 2010. [2](#)
- [12] Xinyan Dai, Xiao Yan, Kaiwen Zhou, Yuxuan Wang, Han Yang, and James Cheng. Edit distance embedding using convolutional neural networks. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020. [2](#)
- [13] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. Speeding up graph edit distance computation through fast bipartite matching. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 102–111. Springer, 2011. [1, 2, 7](#)
- [14] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. Splinecnn: Fast geometric deep learning with continuous b-spline kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 869–877, 2018. [2, 4, 7](#)
- [15] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. [7](#)
- [16] Matthias Fey, Jan E Lenssen, Christopher Morris, Jonathan Masci, and Nils M Kriege. Deep graph matching consensus. In *Int. Conf. Learn. Represent.*, 2020. [2, 4](#)
- [17] Bo Jiang, Pengfei Sun, Jin Tang, and Bin Luo. Glmnnet: Graph learning-matching networks for feature matching. *arXiv preprint arXiv:1911.07681*, 2019. [2](#)
- [18] Bo Jiang, Jin Tang, Chris Ding, Yihong Gong, and Bin Luo. Graph matching via multiplicative update algorithm. In *Advances in Neural Information Processing Systems*, pages 3187–3195, 2017. [2](#)
- [19] Roy Jonker and Anton Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. [2](#)
- [20] Riesen Kaspar. Graph matching toolkit. <https://github.com/dzambon/graph-matching-toolkit>, 2017. [1, 2, 3, 7](#)
- [21] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014. [7](#)
- [22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *Int. Conf. Learn. Represent.*, 2017. [1, 2, 4, 7](#)

- [23] Harold W. Kuhn. The hungarian method for the assignment problem. In *Export. Naval Research Logistics Quarterly*, pages 83–97, 1955. 2
- [24] Marius Leordeanu, Martial Hebert, and Rahul Sukthankar. An integer projected fixed point method for graph matching and map inference. In *Adv. Neural Inform. Process. Syst.*, 2009. 2
- [25] Julien Lerouge, Zeina Abu-Aisheh, Romain Raveaux, Pierre Héroux, and Sébastien Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition*, 72:254–265, 2017. 2
- [26] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning*, pages 3835–3845, 2019. 1, 2, 3, 7, 8
- [27] Paul Maergner, Vinaychandran Pondenkandath, Michele Alberti, Marcus Liwicki, Kaspar Riesen, Rolf Ingold, and Andreas Fischer. Combining graph edit distance and triplet networks for offline signature verification. *Pattern Recognition Letters*, 125:527–533, 2019. 1
- [28] Franco Manessi, Alessandro Rozza, and Mario Manzo. Dynamic graph convolutional networks. *Pattern Recognition*, 97:107000, 2020. 2, 4
- [29] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(4):5363–5370, 2020. 2, 4
- [30] Kaspar Riesen and Horst Bunke. Iam graph database repository for graph based pattern recognition and machine learning. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 287–297. Springer, 2008. 1
- [31] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 1, 2, 3, 7, 8, 9
- [32] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *Mining and Learning with Graphs*, pages 21–24, 2007. 1, 2, 3, 5, 7, 8
- [33] Michal Rolínek, Paul Swoboda, Dominik Zietlow, Anselm Paulus, Vít Musil, and Georg Martius. Deep graph matching via blackbox differentiation of combinatorial solvers. In *Eur. Conf. Comput. Vis.*, 2020. 2, 4
- [34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *Trans. on Neural Networks*, 2009. 1, 2
- [35] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Adv. Neural Inform. Process. Syst.*, NIPS’13, page 926–934, Red Hook, NY, USA, 2013. Curran Associates Inc. 5
- [36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 2, 5
- [37] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Learning combinatorial embedding networks for deep graph matching. In *Int. Conf. Comput. Vis.*, 2019. 2
- [38] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Neural graph matching network: Learning lawler’s quadratic assignment problem with extension to hypergraph and multiple-graph matching. *arXiv preprint arXiv:1911.11308*, 2019. 2
- [39] Tao Wang, Haibin Ling, Congyan Lang, and Songhe Feng. Graph matching with adaptive and branching path following. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2017. 2
- [40] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE ’12, page 210–221, USA, 2012. IEEE Computer Society. 6
- [41] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. A short survey of recent advances in graph matching. In *ICMR*, 2016. 2
- [42] Tianshu Yu, Runzhong Wang, Junchi Yan, and Baoxin Li. Learning deep graph matching with channel-independent embedding and hungarian attention. In *Int. Conf. Learn. Represent.*, 2020. 2
- [43] Tianshu Yu, Junchi Yan, Yilin Wang, Wei Liu, and Baoxin Li. Generalizing graph matching beyond quadratic assignment model. In *Advances in Neural Information Processing Systems 31*, pages 853–863, 2018. 2
- [44] Andrei Zanfir and Cristian Sminchisescu. Deep learning of graph matching. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2018. 2
- [45] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009. 1, 2
- [46] Christian Zeyen and Ralph Bergmann. A\*-based similarity assessment of semantic graphs. In *International Conference on Case-Based Reasoning*, pages 17–32. Springer, 2020. 1
- [47] Li Zheng, Zhenpeng Li, Jian Li, Zhao Li, and Jun Gao. Add-graph: Anomaly detection in dynamic graph using attention-based temporal gcn. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 4419–4425, 2019. 2, 4
- [48] Feng Zhou and Fernando De la Torre. Factorized graph matching. In *IEEE Conf. Comput. Vis. Pattern Recog.*, 2012. 2
- [49] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv:1812.08434*, 2018. 2