Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding

Yutong Ye
East China Normal University
Shanghai, China
52205902007@stu.ecnu.edu.cn

Xiang Lian
Kent State University
Kent, Ohio, USA
xlian@kent.edu

Mingsong Chen
East China Normal University
Shanghai, China
mschen@sei.ecnu.edu.cn

ABSTRACT

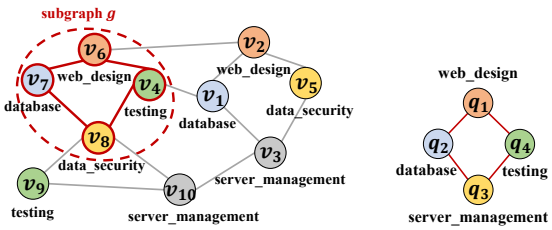
The classic problem of *exact subgraph matching* returns those subgraphs in a large-scale data graph that are isomorphic to a given query graph, which has gained increasing importance in many real-world applications such as social network analysis, knowledge graph discovery in the Semantic Web, bibliographical network mining, and so on. In this paper, we propose a novel and effective *graph neural network (GNN)-based path embedding framework* (GNN-PE), which allows efficient *exact subgraph matching* without introducing *false dismissals*. Unlike traditional GNN-based graph embeddings that only produce *approximate* subgraph matching results, in this paper, we carefully devise GNN-based embeddings for paths, such that: if two paths (and 1-hop neighbors of vertices on them) have the subgraph relationship, their corresponding GNN-based embedding vectors will strictly follow the dominance relationship. With such a newly designed property of path dominance embeddings, we are able to propose effective pruning strategies based on path label/dominance embeddings and guarantee no false dismissals for subgraph matching. We build multidimensional indexes over path embedding vectors, and develop an efficient subgraph matching algorithm by traversing indexes over graph partitions in parallel and applying our pruning methods. We also propose a cost-model-based query plan that obtains query paths from the query graph with low query cost. Through extensive experiments, we confirm the efficiency and effectiveness of our proposed GNN-PE approach for exact subgraph matching on both real and synthetic graph data.

PVLDB Reference Format:
Yutong Ye, Xiang Lian, and Mingsong Chen. Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding. PVLDB, 17(7): 1628 - 1641, 2024.
doi:10.14778/3654621.3654630

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/JamesWhiteSnow/GNN-PE.

1 INTRODUCTION

For the past decades, graph data management has received much attention from the database community, due to its wide spectrum of real applications such as the Semantic Web [47], social networks [61], biological networks (e.g., gene regulatory networks [32] and protein-to-protein interaction networks [56]), road networks [17, 69], and so on. In these graph-related applications, one of the most important and classic problems is the *subgraph matching* query, which retrieves subgraphs g from a large-scale data graph G that match with a given query graph pattern q.



(a) collaboration social network G (b) query graph q

Figure 1: An example of the subgraph matching in collaboration social networks.

Below, we give an example of the subgraph matching in real applications of skilled team formation in collaboration networks.

Example 1. **(Skilled Team Formation in Collaboration Social Networks [3])** *In order to successfully accomplish a task, a project manager is interested in finding an experienced team that consists of members with complementary skills and having previous collaboration histories. Figure 1(a) illustrates a collaboration social network, G, which contains user vertices, $v_1 \sim v_{10}$, with skill labels (e.g., v_7 with the skill "database") and edges (each connecting two user vertices, e.g., v_7 and v_8, indicating that they have collaborated in some project before). Figure 1(b) shows a query graph q, specified by the project manager, which involves the required team members, $q_1 \sim q_4$, with specific skills and their historical collaboration requirements (e.g., the edge between nodes q_1 and q_2 that indicates the front-end and back-end collaborations). In this case, the project manager can specify this query graph q and issue a subgraph matching query over the collaboration network G to obtain candidate teams matching with q (e.g., subgraph g isomorphic to q, circled in Figure 1(a)).* ■

The subgraph matching has many other real applications. For example, in the Semantic Web application like the knowledge graph search [41], a SPARQL query can be transformed to a query graph q, and thus the SPARQL query answering is equivalent to a subgraph matching query over an RDF knowledge graph, which retrieves RDF subgraphs isomorphic to the transformed query graph q.

Prior Works. The subgraph isomorphism problem is known to be NP-complete [18, 23, 38], which is thus not tractable. Prior works on the subgraph matching problem usually followed the filter-and-refine paradigm [30, 34, 60, 67, 70], which first filters out subgraph false alarms with no *false dismissals* and then returns actual matching subgraphs by refining the remaining candidate subgraphs.

Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654630

Due to the high computation cost of *exact* subgraph matching, an alternative direction is to quickly obtain *approximate* subgraph matching results, trading the accuracy for efficiency. Previous works on approximate subgraph matching [19, 21, 40, 72] usually searched k most similar subgraphs in the data graph by using various graph similarity measures (e.g., graph edit distance [40, 72], chi-square statistic [21], and Sylvester equation [19]).

Moreover, several recent works [6, 39] utilized deep-learning-based approaches such as *Graph Neural Networks* (GNNs) to conduct approximate subgraph matching. Specifically, GNNs can be used to transform entire (small) complex data graphs into vectors in an embedding space offline. Then, we can determine the subgraph relationship between data and query graphs by comparing their embedding vectors, via either neural networks [6] or similarity measures (e.g., Euclidean or Hamming distance [39]). Although GNN-based approaches can efficiently, but approximately, assert the subgraph relationship between any two graphs, there is no theoretical guarantee about the accuracy of such an assertion, which results in approximate (but not exact) subgraph answers. Worse still, these GNN-based approaches usually work for comparing two graphs only, which are not suitable for tasks like retrieving the locations of matching subgraphs in a large-scale data graph.

Our Contributions. In this paper, we focus on *exact subgraph matching* queries over a large-scale data graph, and present a novel *GNN-based path embedding* (GNN-PE) framework for exact and efficient subgraph matching. In contrast to traditional GNN-based embeddings without any evidence of accuracy guarantees, we design an effective *GNN-based path dominance embedding* technique, which trains our newly devised GNN models to obtain embeddings of nodes (and their neighborhood structures) on paths such that: **any two paths (including 1-hop neighbors of vertices on them) with the subgraph relationship will strictly yield embeddings with the *dominance* [14] relationship**. This way, we can transform our subgraph matching problem to a dominating region search problem in the embedding space and guarantee a 100% recall ratio so that we do not miss any query results. In other words, we can retrieve candidate paths (including their locations in the data graph) matching with those in the query graph with no false dismissals. Further, we propose an effective approach to enhance the pruning power by using multiple sets of GNN-based path embeddings over randomized vertex labels in the data graph.

To deal with large-scale data graphs, we divide the data graph into multiple subgraph partitions and train GNN models for different partitions to enable parallel processing over path embeddings in a scalable manner. We also build an index over path label/dominance embedding vectors for each partition to facilitate the pruning, and develop an efficient and exact subgraph matching algorithm for (parallel) candidate path retrieval and refinement via our proposed cost-model-based query plan.

In this paper, we make the following contributions:

(1) We propose a novel GNN-PE framework for exact subgraph matching via GNN-based path embeddings in Section 2.
(2) We design an effective *GNN-based path dominance embedding* approach for exact subgraph retrieval in Section 3.
(3) We develop an efficient parallel subgraph matching algorithm that traverses indexes over GNN-based path embeddings by using effective pruning methods in Section 4.

Table 1: Symbols and Descriptions

Symbol	Description
G	a data graph
q	a query graph
g	a subgraph of the data graph G
v_i (or q_i)	a vertex in graph G (or q)
e_{ij} (or $e_{q_iq_j}$)	an edge in graph G (or q)
m	the number of graph partitions G_j
M_j	a GNN model for G_j
g_{v_i} (or s_{v_i})	a unit star graph (or substructure) of center vertex v_i
$o(g_{v_i})$ (or $o(v_i)$)	an embedding vector of center vertex v_i from unit star graph g_{v_i}
$o(p_z)$ (or $o(p_q)$)	a path dominance embedding vector of path p_z (or p_q)
$o_0(p_z)$ (or $o_0(p_q)$)	a path label embedding vector of path p_z (or p_q)

(4) We devise a novel cost model for selecting the best query plan of the subgraph matching in Section 5.
(5) Through extensive experiments, we confirm the efficiency and effectiveness of our GNN-PE approach for exact subgraph matching over real/synthetic graphs in Section 6.

Section 7 reviews related works on exact/approximate subgraph matching and GNNs. Finally, Section 8 concludes this paper.

2 PROBLEM DEFINITION

Table 1 depicts the commonly used symbols and their descriptions.

2.1 Graph Data Model

We first give the model for an undirected, labeled graph, G, below.

Definition 1. ***(Graph,** G**)** *A graph, G, is represented by a quadruple $(V(G), E(G), \phi(G), L(G))$, where $V(G)$ is a set of vertices v_i, $E(G)$ is a set of edges e_{ij} $(= (v_i, v_j))$ between vertices v_i and v_j, $\phi(G)$ is a mapping function $V(G) \times V(G) \rightarrow E(G)$, and $L(G)$ is a labeling function that associates each vertex $v_i \in V(G)$ with a label $L(v_i)$.*

2.2 Graph Isomorphism

In this subsection, we give the definition of the classic graph isomorphism problem between undirected, labeled graphs.

Definition 2. ***(Graph Isomorphism [5, 23])*** *Given two graphs $G_A = (V_A, E_A, \phi_A, L_A)$ and $G_B = (V_B, E_B, \phi_B, L_B)$, we say that G_A is isomorphic to G_B (denoted as $G_A \equiv G_B$), if there exists an edge-preserving bijective function $f : V_A \rightarrow V_B$, such that: i) $\forall v_i \in V_A$, $L_A(v_i) = L_B(f(v_i))$, and ii) $\forall v_i, v_j \in V_A$, if $(v_i, v_j) \in E_A$ holds, we have $(f(v_i), f(v_j)) \in E_B$.*

In Definition 2, the graph isomorphism problem checks whether or not two graphs G_A and G_B exactly match each other.

Moreover, we say that G_A is *subgraph isomorphic* to G_B (denoted as $G_A \subseteq G_B$), if G_A is isomorhic to an induced subgraph, g_B, of graph G_B (i.e., $G_A \equiv g_B$). Note that, the subgraph isomorphism problem has been proven to be NP-complete [18, 38].

2.3 Subgraph Matching Queries

We now define a *subgraph matching query* over a large data graph G, which obtains subgraphs, g, that match with a query graph q.

Definition 3. ***(Subgraph Matching Query)*** *Given a data graph G and a query graph q, a subgraph matching query retrieves all the subgraphs g of the data graph G that are isomorphic to the query graph q (i.e., $g \equiv q$).*

Algorithm 1: The GNN-Based Path Embedding (GNN-PE) Framework for Exact Subgraph Matching

Input: a data graph G and a query graph q
Output: subgraphs g ($\subseteq G$) that are isomorphic to q
// **Offline Pre-Computation Phase**
1 divide graph G into m disjoint subgraphs G_1, G_2, ..., and G_m
2 **for** *each subgraph partition* G_j ($1 \leq j \leq m$) **do**
  // train GNN models for graph node/edge embeddings
3 train a GNN model M_j with node dominance embedding over vertices in G_j
4 generate embedding vectors $o(p_z)$ for paths p_z of lengths l in G_j via M_j
  // build an index over subgraph G_j
5 build aggregate R*-tree indexes, $\mathcal{I}_j$, over embedding vectors for paths of length l in G_j
// **Online Subgraph Matching Phase**
6 **for** *each query graph* q **do**
  // retrieve candidate paths
7 compute a cost-model-based query plan φ of multiple query paths p_q
8 obtain a query embedding vector $o(q_i)$ of each vertex q_i in q from GNNs M_j, and embeddings $o(p_q)$ of query paths p_q
9 find candidate path sets, $p_q.cand\_list$, that match with query paths, p_q, by traversing indexes $\mathcal{I}_j$
  // obtain and refine candidate subgraphs
10 assemble candidate subgraphs g from candidate paths in $p_q.cand\_list$ and refine subgraphs g via multi-way hash join
11 **return** subgraphs g ($\equiv q$)

The subgraph matching query (as given in Definition 3) has many real-world applications such as social network analysis. In this paper, we consider *exact* subgraph matching that obtains strictly isomorphic subgraphs, in contrast to *approximate* one [19, 39, 40] (i.e., finding subgraphs not exactly matching with query graph q).

2.4 GNN-Based Subgraph Matching Framework

Algorithm 1 presents a novel *GNN-based path embedding* (GNN-PE) framework for efficiently answering subgraph matching queries via path embeddings, which consists of two phases, offline pre-computation and online subgraph matching phases. That is, we first pre-process the data graph G offline by building indexes over path embedding vectors via GNNs (lines 1-5), and then answer online subgraph matching queries over indexes (lines 6-11). For the detailed descriptions, please refer to our technical report [64].

3 GNN-BASED DOMINANCE EMBEDDING

In this section, we discuss how to calculate GNN-based dominance embeddings for vertices/paths (lines 3-4 of Algorithm 1), which can enable subgraph relationships to be preserved in the embedding space and support efficient and accurate path candidate retrieval. ***Due to space limitations, we provide proofs of lemmas in our technical report [64] and omit them below.***

3.1 GNN Model for the Node Embedding

In this work, we use a GNN model (e.g., *Graph Attention Network* (GAT) [58]) to enable the node embedding in the data graph G. Specifically, the GNN takes a *unit star graph* g_{v_i} (i.e., a star subgraph containing a center vertex $v_i \in V(G)$ and its 1-hop neighbors) as input and an embedding vector, $o(v_i)$, of vertex v_i as output. Figure 2 illustrates an example of this GNN model (with unit star graph g_{v_1} as input), which consists of input, hidden, and output layers.

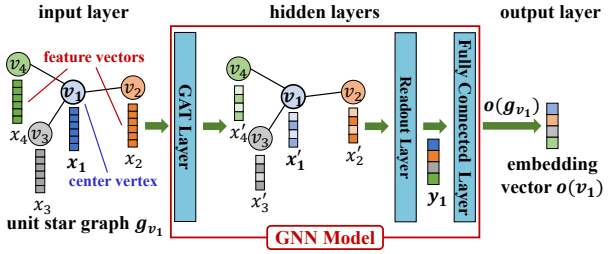


Figure 2: Illustration of our GNN model.

Input Layer. As mentioned earlier, the input of the GNN model is a unit star graph g_{v_i} (or its star substructure/subgraph, denoted as s_{v_i}). Each vertex v_j in g_{v_i} (or s_{v_i}) is associated with an initial feature vector x_j of size F, which is obtained via either vertex label encoding or one-hot encoding [12].

Figure 3 shows an example of unit star graph g_{v_1} in data graph G and one of its star substructures s_{v_1} ($\subseteq g_{v_1}$), which are centered at vertex v_1. In Figure 3(a), vertices $v_1 \sim v_4$ have their initial feature vectors $x_1 \sim x_4$, respectively. The case of the star substructure s_{v_1} in Figure 3(b) is similar.

Hidden Layers. As shown in Figure 2, our GNN model consists of GAT [58], readout, and fully connected layers.

GAT layer: In the first GAT layer, the feature vector of each vertex will go through a linear transformation parameterized by a weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$. Specifically, we compute an *attention coefficient*, $ac_{v_i v_j}$, between any vertices v_i and v_j as follows:

$$ac_{v_i v_j} = a(\mathbf{W}x_i, \mathbf{W}x_j), \tag{1}$$

which indicates the importance of vertex v_i to vertex v_j, where the shared attentional mechanism $a(\cdot,\cdot)$ is a function (e.g., a single-layer neural network with learnable parameters): $\mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$ that outputs the correlation between two feature vectors.

Denote $\mathcal{N}(v_i)$ as the neighborhood of a vertex v_i. For each vertex v_i, we aggregate feature vectors of its 1-hop neighbors $v_j \in \mathcal{N}(v_i)$. That is, we first use a softmax function to normalize attention coefficients $ac_{v_i v_j}$ as follows:

$$\alpha_{v_i v_j} = softmax(ac_{v_i v_j}) = \frac{exp(ac_{v_i v_j})}{\sum_{v_k \in \mathcal{N}(v_i)} exp(ac_{v_i v_k})}. \tag{2}$$

Then, the output of the GAT layer is computed by a linear combination of feature vectors:

$$x'_i = \sigma\left(\sum_{v_j \in \mathcal{N}_{v_i}} \alpha_{v_i v_j} \mathbf{W} x_j\right), \tag{3}$$

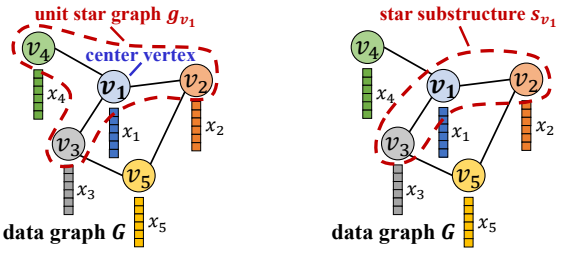
where $\sigma(Z)$ is a nonlinear activation function (e.g., rectified linear unit [46], Sigmoid [25], etc.) with input and output vectors, Z and $\sigma(Z)$, of length F', respectively.

To stabilize the learning process of self-attention, GAT uses a multi-head attention mechanism similar to [57], where each head is an independent attention function and K heads execute the transformation of Eq. (3) in parallel. Thus, an alternative GAT output x'_i can be a concatenation of feature vectors generated by K heads:

$$x'_i = \Big\|_{k=1}^{K} \sigma\left(\sum_{v_j \in \mathcal{N}_{v_i}} \alpha^{(k)}_{v_i v_j} \mathbf{W}^{(k)} x_j\right), \tag{4}$$

Algorithm 2: GNN Model Training

Input: i) a subgraph partition $G_j \subseteq G$; ii) a training data set D_j, and; iii) a learning rate η
Output: a trained GNN model M_j
// **generate a training data set** D_j
1 $D_j = \emptyset$
2 **for** *each vertex* $v_i \in V(G_j)$ **do**
3 obtain the unit star subgraph g_{v_i} and its star substructures s_{v_i}
4 add all pairs (g_{v_i}, s_{v_i}) to D_j
5 randomly shuffle pairs in D_j
// **train a GNN model** M_j **until the loss equals to** 0
6 **do**
 // the training epoch
7 **for** *each batch* $B \subseteq D_j$ **do**
8 obtain embedding vectors of pairs in B by M_j
9 compute the loss function $\mathcal{L}(B)$ of M_j by Eq. (7)
10 $M_j.update(\mathcal{L}(B), \eta)$
 // the testing epoch
11 $L_e = 0$
12 **for** *each batch* $B \subseteq D_j$ **do**
13 obtain embedding vectors of pairs in B by M_j
14 compute the loss function $\mathcal{L}(B)$ of M_j by Eq. (7)
15 $L_e \leftarrow L_e + \mathcal{L}(B)$
16 **while** $(L_e = 0)$;
17 repeat lines 6-16 to train b GNN models with random initial weight parameters to avoid local optimality
18 select the best model M_j (satisfying $L_e = 0$) with the smallest expected query cost $Cost_{M_j}$ (given in Eq. (8))
19 **return** the best trained GNN model M_j



(a) unit star graph $g_{v_1} \subseteq G$ (b) star substructure $s_{v_1} \subseteq g_{v_1}$

Figure 3: Illustration of the input for the GNN model.

where $\|_{k=1}^{K}$ is the concatenation operator of K vectors, $\alpha_{v_1 v_j}^{(k)}$ is the normalized attention coefficient computed by the k-th attention head, and $\mathbf{W}^{(k)}$ is the k-th parameterized weight matrix.

Readout layer: A readout layer [65, 68] generates an embedding vector, y_i, for the entire unit star graph g_{v_i}, by summing up feature vectors x'_j of all vertices v_j in g_{v_i}, which is permutation invariant. That is, we obtain:

$$y_i = \sum_{\forall v_j \in V(g_{v_i})} x'_j. \tag{5}$$

Fully Connected Layer: A fully connected layer performs a non-linear transformation of y_i (given in Eq. (5)) via an activation function $\sigma(\cdot)$ and obtains the embedding vector, $o(g_{v_i})$, of size d for vertex v_i. That is, we have:

$$o(g_{v_i}) = \sigma(\mathbb{W} y_i), \tag{6}$$

where $\sigma(\cdot)$ is an activation function and $\mathbb{W}$ is a $d \times (K \cdot F')$ weight matrix. In this paper, we used the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ ($\in (0, 1)$), where e is a mathematical constant. We will leave the study of using other activation functions as our future work.

Output Layer. In this layer, we output $o(g_{v_i})$ (given in Eq. (6)) as the embedding vector, $o(v_i)$, of center vertex v_i.

3.2 Node Dominance Embedding

In this subsection, we propose an effective *GNN-based node dominance embedding* approach to train the GNN model (discussed in Section 3.1), such that our node embedding via GNN can reflect the subgraph relationship between unit star graph g_{v_i} and its star substructures s_{v_i} in the embedding space. Such a node dominance embedding can enable exact subgraph matching.

Loss Function. Specifically, for the GNN model training, we use a training data set D_j that contains *all* pairs of unit star graphs g_{v_i} and their substructures s_{v_i} (for all vertices v_i in the subgraph partition G_j of data graph G). Then, we design a loss function $\mathcal{L}(D_j)$ over a training data set D_j as follows:

$$\mathcal{L}(D_j) = \sum_{\forall (g_{v_i}, s_{v_i}) \in D_j} \left\| max\{0, o(s_{v_i}) - o(g_{v_i})\} \right\|_2^2, \tag{7}$$

where $o(g_{v_i})$ and $o(s_{v_i})$ are the embeddings of a unit star graph g_{v_i} and its substructures s_{v_i}, respectively, and $||\cdot||_2$ is the L_2-norm.

GNN Model Training. To guarantee that we do not lose any candidate vertices for exact subgraph matching, we train (or even overfit) the GNN model until the loss function $\mathcal{L}(\cdot)$ (given in Eq. (7)) is equal to 0. Intuitively, from Eq. (7), when the loss function $\mathcal{L}(D_j) = 0$ holds, the embedding vector $o(s_{v_i})$ of any star substructure s_{v_i} is *dominating* [14] (or equal to) that, $o(g_{v_i})$, of its corresponding unit star graph g_{v_i}. In the sequel, we will simply say that $o(s_{v_i})$ *dominates* $o(g_{v_i})$ (denoted as $o(s_{v_i}) \preceq o(g_{v_i})$), if $o(s_{v_i})[t] \leq o(g_{v_i})[t]$ for all $1 \leq t \leq d$ (including the case where $o(s_{v_i}) = o(g_{v_i})$).

In other words, given the subgraph relationship between s_{v_i} and g_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), our *GNN-based node dominance embedding* approach can always guarantee that their embedding vectors $o(s_{v_i})$ and $o(g_{v_i})$ follow the dominance relationship (i.e., $o(s_{v_i}) \preceq o(g_{v_i})$).

Algorithm 2 illustrates the training process of a GNN model M_j over a subgraph parititon G_j ($1 \leq j \leq m$). For each vertex $v_i \in G_j$, we obtain all (shuffled) pairs of unit star subgraphs g_{v_i} and their star substructures s_{v_i}, which result in a training data set D_j (lines 1-5). Then, for each training iteration, we use a training epoch to update model parameters (lines 6-10), and a testing epoch to obtain the loss L_e (lines 11-15). The training loop of the GNN model M_j terminates until the loss L_e equals 0 (line 16).

To obtain GNN-based node embeddings with high pruning power, we train multiple GNN models with b sets of random initial weights (line 17), and select the one with zero loss and the highest quality of the generated node embeddings (i.e., the lowest expected query cost, $Cost_{M_j}$, as discussed in Eq. (8) below; line 18). Finally, we return the best trained GNN model M_j (line 19).

Complexity Analysis of the GNN Training. In Algorithm 2, we extract all star substructures s_{v_i} from each unit star subgraph g_{v_i} (line 3), which are used for GNN training. Thus, the number of pairs in the training data set is $\sum_{v_i \in V(G_j)} 2^{deg(v_i)}$, where $deg(v_i)$ is the degree of vertex v_i. On the other hand, we train b GNN models (as mentioned in Section 3.1) with different initial weights to achieve high pruning power (line 17). For each GNN model, the time complexity of the computation on the GAT layer is $O((|V(g_{v_i})| + |E(g_{v_i})|) \cdot F')$, and that for the fully connected layer

is $O(F' \cdot d)$. In this paper, for the input unit star subgraph g_{v_i} (or star substructure s_{v_i}), we have $|V(g_{v_i})| = deg(v_i)+1$ and $|E(g_{v_i})| = deg(v_i)$. Therefore, the total time complexity of the GNN training is given by $O\big(b \cdot \sum_{v_i \in V(G_j)} 2^{deg(v_i)} \cdot (2 \cdot deg(v_i) + d + 1) \cdot F' \cdot \mathbb{N}\big)$, where $\mathbb{N}$ is the number of training epochs until zero loss.

Since vertex degrees in real-world graphs usually follow the power-law distribution [7], only a small fraction of vertices have high degrees. For example, in US Patents graph data [53], the average vertex degree is around 9, which incurs about 512 ($= 2^9$) star substructures per vertex. Thus, this is usually acceptable for offline GNN training on a single machine. In practice, for vertices v_i with high degrees $deg(v_i)$ (e.g., greater than a threshold θ), instead of enumerating a large number of $2^{deg(v_i)}$ star substructures, we simply set their embeddings $o(v_i)$ to all-ones vectors $\mathbb{1}$. This is because embeddings of those high-degree vertices often have low pruning power, and it would be better to directly consider them as vertex candidates without costly star substructure enumeration/training. This way, our GNN training complexity is reduced to $O\big(b \cdot \sum_{v_i \in V(G_j), deg(v_i) \le \theta} 2^{deg(v_i)} \cdot (2 \cdot deg(v_i) + d + 1) \cdot F' \cdot \mathbb{N}\big)$.

Usage of the Node Dominance Embedding for Exact Subgraph Matching. Intuitively, with the node dominance embeddings, we can convert exact subgraph matching into the dominance search problem in the embedding space. Specifically, if a vertex q_i in the query graph q matches with a vertex v_i in some subgraph g of G, then it must hold that $o(g_{q_i}) \preceq o(g_{v_i})$, where $o(g_{q_i})$ is an embedding vector of vertex q_i (and its 1-hop neighbors) in query graph q via the trained GNN.

This way, we can always use the embedding vector $o(g_{q_i})$ of q_i to retrieve candidate vertices v_i in G (i.e., those vertices with embedding vectors $o(g_{v_i})$ dominated by $o(g_{q_i})$ in the embedding space). Our trained GNN with overfitting (i.e., the loss is 0) can guarantee that vertices v_i dominated by $o(g_{q_i})$ will not miss any truly matching vertices (i.e., with 100% recall ratio). This is because all possible query star structures g_{q_i} have already been offline enumerated and trained during the GNN training process (i.e., $g_{q_i} \equiv g_{s_i}$).

The Quality of the Generated Node Embeddings: Note that, b GNN models can produce at most b sets of node embeddings that can fully satisfy the dominance relationships (i.e., with zero loss). Thus, in line 18 of Algorithm 2, we need to select one GNN model with the best *node embedding quality*, which is defined as the expected query cost $Cost_{M_j}$ (or the expected number of embedding vectors, $o(g_{v_x})$, generated from unit star subgraphs dominated by that, $o(g_{q_i})$, of query star subgraphs) below:

$$Cost_{M_j} = \frac{\sum_{\forall g_{q_i}} \big|\{\forall v_x \in V(G) \mid o(g_{q_i}) \preceq o(g_{v_x})\}\big|}{\#\text{ of query unit star subgraphs } g_{q_i}}, \tag{8}$$

where g_{q_i} are all the possible query unit star subgraphs (i.e., all star substructures s_{v_x} extracted from g_{v_x}), and the # of possible query unit star subgraphs g_{q_i} is given by $|D_j|$.

Example 2. *Figure 4 illustrates an example of the node dominance embedding between data graph G and query graph q. Each vertex $v_i \in V(G)$ has a 2D embedding vector $o(v_i)$ via the GNN, whereas each query vertex $q_i \in V(q)$ is transformed to a 2D vector $o(q_i)$. For example, as shown in the tables, we have $o(v_1) = (0.78, 0.79)$ and $o(q_1) = (0.62, 0.61)$. We plot the embedding vectors of vertices in a 2D embedding space on the right side of the figure. We can see that $o(q_1)$ is dominating $o(v_1)$ and $o(v_2)$, which implies that g_{q_1} is potentially a subgraph of (i.e., matching with) g_{v_1} and g_{v_2}. On the other hand, since $o(q_1)$ is not dominating $o(v_3)$ in the 2D embedding space, query vertex q_1 cannot match with vertex v_3 in the data graph G.* ■

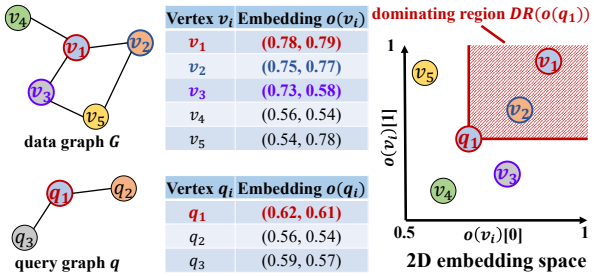


Figure 4: An example of node dominance embedding.

Multi-GNN Node Dominance Embedding. In order to further reduce the number of candidate vertices v_i that match with a query vertex $q_i \in V(q)$ (or enhance the pruning power), we use multiple independent GNNs to embed vertices. Specifically, for each subgraph partition G_j, we convert the label of each vertex to a new randomized label (e.g., via a hash function or using the vertex label as a seed to generate a pseudo-random number). This way, we can obtain a new subgraph G'_j with the same graph structure, but different vertex labels, and train/obtain a new GNN model M'_j with new embeddings $o'(v_i)$ of vertices $v_i \in V(G'_j)$.

With multiple versions of randomized vertex labels, we can obtain different vertex embeddings $o'(v_i)$, where the subgraph relationships between the unit star subgraph and its star substructures also follow the dominance relationships of their embedding vectors.

Therefore, for each vertex v_i, we can compute different versions (via different GNNs) of embedding vectors (e.g., $o(v_i)$ and $o'(v_i)$), which can be used together for retrieving candidate vertices v_i that match with a query vertex q_i with higher pruning power.

Convergence Analysis. To guarantee no false dismissals for online subgraph matching, we need to offline train/overfit a GNN model M_j, until the training loss equals zero (i.e., $\mathcal{L}(D_j) = 0$). Below, we give the convergence analysis of the GNN model training, including i) parameter settings for achieving sufficient GNN capacity, ii) the existence of GNN parameters to achieve the training goal, and iii) the target accessibility of the GNN training.

GNN Model Capacity for Overfitting: As mentioned in [22], the GNN model needs enough *capacity* to overfit the training data sets (e.g., achieving zero loss). In particular, the capacity of a GNN model M_j [43] can be defined as $M_j.cap = M_j.dep \times M_j.wid$, where $M_j.dep$ and $M_j.wid$ are the depth (i.e., # of layers) and width (i.e., the maximum dimension of intermediate node embeddings in all layers) of the GNN model M_j, respectively.

According to [43], the GNN model capacity to accomplish graph tasks with overfitting needs to satisfy the following condition:

$$M_j.cap \ge \tilde{\Omega}(|V(\cdot)|^{\delta}), \tag{9}$$

where $\delta \in [1/2, 2]$ is an exponent factor reflecting the complexity of the task (e.g., solving some NP-hard problems necessitates $\delta = 2$), $|V(\cdot)|$ is the size of the input graph, and $\tilde{\Omega}(\cdot)$ indicates that as the graph size increases, the GNN model capacity also increases with the same rate (up to a logarithmic factor).

In our subgraph matching task, the GNN model aims to learn a partial order between a unit star subgraph g_{v_i} and its star substructure s_{v_i} in each pair of the training data set D_j. Since this partial order exists in each individual pair, the GNN capacity is only relevant to the maximum input size of unit star subgraphs, i.e., $\max_{\forall g_{v_i} \in D_j}\{|V(g_{v_i})|\}^{\delta}$. Therefore, by overestimating the δ value (i.e., $\delta = 2$), we have the lower bound of the GNN capacity $M_j.cap$ to solve our partial-order learning problem below:

$$M_j.cap \geq \max_{\forall g \in D_j} \left\{|V(g)|^2\right\}. \tag{10}$$

Note that, for our task of learning partial order, counter-intuitively, the GNN model capacity is constrained theoretically by the input graph size, instead of the size of the training data set D_j [43].

Specifically, the GNN model M_j we use in this paper (as shown in Figure 2) has 3 hidden layers (i.e., GNN depth $M_j.dep = 3$). If we set $F = 1$, $K = 3$, $F' = 32$, and $d = 2$ by default, then we have the GNN width $M_j.wid = K \cdot F' = 96$. As a result, we have the model capacity $M_j.cap = M_j.dep \times M_j.wid = 3 \times 96 = 288$. On the other hand, since we set the degree threshold $\theta = 10$ for node embeddings (as discussed in the complexity analysis above), the maximum input size of unit star subgraph does not exceed 11 (i.e., $max_{\forall g \in D_j}(|V(g)|) = 11$). Thus, we can see that Inequality (10) holds (i.e., $288 = M_j.cap \geq \max_{\forall g \in D_j}\left\{|V(g)|^2\right\} = 11^2$ holds), which implies that our GNN model M_j has enough capacity to overfit the training data set D_j with zero loss.

The Existence of the GNN Model that Meets the Training Target: Next, we prove that there exists at least one set of GNN model parameters that make the loss equal to zero over the training data.

In the following lemma, we give a special case of GNN model parameters, which can ensure the dominance relationship between node embedding vectors $o(g_{v_i})$ and $o(s_{v_i})$ (satisfying $o(s_{v_i}) \preceq o(g_{v_i})$) of any two star subgraphs g_{v_i} and s_{v_i} (satisfying $s_{v_i} \subseteq g_{v_i}$).

Lemma 3.1. **(A Special Case of GNN Model Parameter Settings)** *For a unit star subgraph g_{v_i} and its star substructure s_{v_i} ($\subseteq g_{v_i}$), their GNN-based node embedding vectors satisfy the dominance condition that: $o(s_{v_i}) \preceq o(g_{v_i})$, if values of the weight matrix $\mathbb{W}$ (in Eq. (6)) in the fully connected layer are all zeros, i.e., $\mathbb{W} = \mathbf{0}$.*

In the special case of Lemma 3.1, the loss function $\mathcal{L}(D_j)$ given in Eq. (7) over all the training pairs in D_j is always equal to 0. Note that, although there is no pruning power in this special case (i.e., all the node embedding vectors are the same, which preserves the dominance relationships), it at least indicates that there exists a set of weight parameters (in Lemma 3.1) that can achieve zero loss.

In reality (e.g., from our experimental results), there are multiple possible sets of GNN parameters that can reach zero training loss (e.g., the special case of $o(s_{v_i}) = o(g_{v_i})$ given in Lemma 3.1). This is because we are looking for embedding vectors that preserve dominance relationships between individual pairs (g_{v_i}, s_{v_i}) in D_j (rather than seeking for a global dominance order for all star subgraphs).

The Target Accessibility of the GNN Training: Up to now, we have proved that we can guarantee enough GNN capacity for overfitting via parameter settings, and GNN parameters that can achieve zero loss exist. We now illustrate the accessibility of our GNN training that can meet the training target (i.e., the loss equals zero).

First, based on [43], if a GNN model M_j over connected attributed graphs has enough capacity, then the GNN model can approach the optimal solution infinitely. That is, we have:

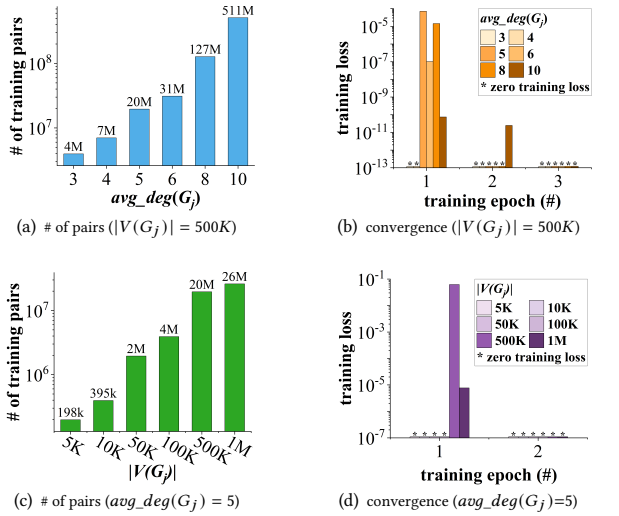
$$|f_j(\cdot) - f_{opt}(\cdot)| \to 0, \tag{11}$$

where $f_j(\cdot)$ is a non-linear function with input x_i and output $o(v_i)$ that learned by the GNN model M_j, and $f_{opt}(\cdot)$ is an optimal function for GNN that achieves zero loss.

Moreover, from [20], with randomized initial weights, first-order methods (e.g., *Stochastic Gradient Descent* (SGD) [35] with the Adam optimizer used in our work) can achieve zero training loss, at a linear convergence rate. That is, it can find a solution with $\mathcal{L}(\cdot) \leq \epsilon$ in $O(log(1/\epsilon))$ epochs, where ϵ is the desired accuracy.

Note that, to handle some exceptional cases that zero training loss cannot be achieved within a limited number of epochs, we may remove those relevant pairs (causing the loss to be non-zero) and train a new GNN on them, which we will leave as our future work.

In summary, we can train our GNN model for node dominance embeddings, and the training process can converge to zero loss.



(a) # of pairs ($|V(G_j)| = 500K$) (b) convergence ($|V(G_j)| = 500K$)

(c) # of pairs ($avg\_deg(G_j) = 5$) (d) convergence ($avg\_deg(G_j)$=5)

Figure 5: Illustration of the GNN training performance for node dominance embeddings.

The GNN Training Scalability w.r.t. Node Dominance Embedding. We train a GNN model (with $F = 1$, $K = 3$, $F' = 32$, and $d = 2$) over large training data sets D_j (containing pairs (g_{v_i}, s_{v_i})), where by default the vertex label domain size $|\Sigma| = 500$, the default average vertex degree, $avg\_deg(G_j) = 5$, $|V(G_j)| = 500K$, the learning rate of the Adam optimizer $\eta = 0.001$, and batch size $1K \sim 4K$.

Figure 5(a) illustrates the number of training pairs (g_{v_i}, s_{v_i}) that the GNN model (in Figure 2) can support (i.e., the loss function $\mathcal{L}(D_j)$ achieves zero), where we vary the average vertex degree, $avg\_deg(G_j)$ from 3 to 10. From our experimental results, our GNN model can learn as many as $\geq 511M$ pairs for a graph with $500K$ vertices and an average degree equal to 10.

Figure 5(b) reports the convergence performance of our GNN-PE approach, where $avg\_deg(G_j)$ varies from 3 to 10. As described in Algorithm 2, we train/update the parameters of the GNN model after each batch of a training epoch, and evaluate the loss of the GNN model after training all batches in D_j at the end of each training epoch. From the figure, the GNN training needs no more than three epochs (before the loss becomes 0), which confirms that

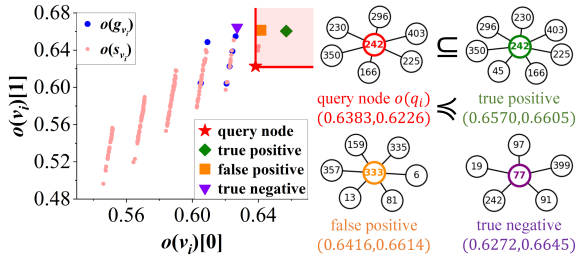


Figure 6: A visualization of node dominance embeddings.

we can train our designed GNN model M_j for node dominance embeddings within a small number of epochs, and the training process can converge fast to zero loss.

Figures 5(c) and 5(d) vary the graph size $|V(G_j)|$ from $5K$ to $1M$, and similar experimental results can be obtained, in terms of the # of training pairs and the convergence performance, respectively.

Visualization Analysis of Node Dominance Embeddings. We randomly sample 10 vertices v_i from synthetic graph G_j (used in Figure 5). For each vertex v_i, we obtain its unit star subgraph g_{v_i} and all star substructures s_{v_i}, and plot in Figure 6 their GNN-based node dominance embeddings $o(g_{v_i})$ (blue points) and $o(s_{v_i})$ (pink points), respectively, in a 2D embedding space.

As a case study shown in Figure 6, given a query node embedding $o(q_i)$ (red star point), its dominating region $DR(o(q_i))$ contains true positive (green diamond; matching vertex) and false positive (orange square; mismatching candidate vertex). The purple triangle point is not in $DR(o(q_i))$, which is true negative (i.e., not a matching vertex). From the visualization, our GNN-based embedding vectors are distributed on some piecewise curves, and their dominance relationships can be well-preserved.

3.3 Path Dominance Embedding

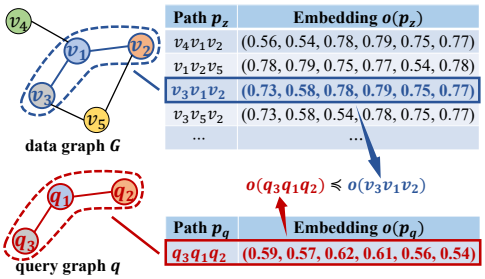
Next, we discuss how to obtain path dominance embedding from node embeddings (as discussed in Section 3.2). Specifically, given a path p_z in G starting from v_i and with length l, we concatenate embedding vectors $o(v_j)$ of all consecutive vertices v_j on path p_z and obtain a path embedding vector $o(p_z)$ of size $((l+1)\cdot d)$, where l is the length of path p_z and d is the dimensionality of node embedding vector $o(v_j)$. That is, we have:

$$o(p_z) = \Big\|_{\forall v_j \in p_z} o(v_j), \tag{12}$$

where $\|$ is the concatenation operator. Note that, node dominance embedding can be considered as a special case of path dominance embedding, where path p_z has a length equal to 0.

Property of the Path Dominance Embedding. Given two paths p_q and p_z, if path p_q (and 1-hop neighbors of vertices on p_q) is a subgraph of p_z (and 1-hop neighbors of vertices on p_z), then it must hold that $o(p_q) \preceq o(p_z)$.

Example 3. *Figure 7 shows an example of the path dominance embedding for paths with length 2. Consider a path $p_z = v_3v_1v_2$, and its 6D path embedding vector $o(p_z) = (0.73, 0.58; 0.78, 0.79; 0.75, 0.77)$, which is a concatenation of three 2D embedding vectors $o(v_3)||o(v_1)||o(v_2)$ (as given in Figure 4). Similarly, in the query graph q, we can obtain a 6D embedding vector $o(p_q)$ of a path $p_q = q_3q_1q_2$.*



Path p_z	Embedding $o(p_z)$
$v_4v_1v_2$	(0.56, 0.54, 0.78, 0.79, 0.75, 0.77)
$v_1v_2v_5$	(0.78, 0.79, 0.75, 0.77, 0.54, 0.78)
$v_3v_1v_2$	(0.73, 0.58, 0.78, 0.79, 0.75, 0.77)
$v_3v_5v_2$	(0.73, 0.58, 0.54, 0.78, 0.75, 0.77)
...	...

Path p_q	Embedding $o(p_q)$
$q_3q_1q_2$	(0.59, 0.57, 0.62, 0.61, 0.56, 0.54)

Figure 7: An example of path dominance embedding.

In the figure, we can see that $o(p_q)$ dominates $o(p_z)$, which indicates that path p_z may potentially match path p_q (according to the property of the path dominance embedding). ■

4 SUBGRAPH MATCHING WITH GNN-BASED PATH EMBEDDING

4.1 Pruning Strategies

In this subsection, we present effective pruning strategies, namely *path label* and *path dominance pruning*, to filter out false alarms of subgraphs g ($\subseteq G$) that cannot match with a given query graph q.

Path Label Pruning. Let $s_0(v_i)$ (or $s_0(q_i)$) be a special star substructure containing an isolated vertex v_i (or q_i; without any 1-hop neighbors). Assume that we can obtain an embedding vector, $o(s_0(v_i))$ (or $o(s_0(q_i))$), of the isolated vertex v_i (or q_i) via the GNN model (discussed in Section 3.2). For simplicity, we denote $o(s_0(v_i))$ (or $o(s_0(q_i))$) as $o_0(v_i)$ (or $o_0(q_i)$).

Similarly, we can concatenate embedding vectors $o_0(v_i)$ (or $o_0(q_i)$) of vertices v_i (or q_i) on path p_z (or p_q) and obtain a *path label embedding* vector $o_0(p_z) = \|_{\forall v_i \in p_z} o_0(v_i)$ (or $o_0(p_q) = \|_{\forall q_i \in p_q} o_0(q_i)$), which intuitively encodes labels of vertices on the path p_z (or p_q).

Lemma 4.1. **(Path Label Pruning)** *Given a path p_z in the subgraph g of data graph G and a query path p_q in query graph q, path p_z can be safely pruned, if it holds that $o_0(p_z) \neq o_0(p_q)$.*

Path Dominance Pruning. For paths $p_z \subseteq G$ (or $p_q \subseteq q$) of length l, their embedding vectors, $o(p_z)$ (or $o(p_q)$), follow the property of the path dominance embedding (as discussed in Section 3.3). Thus, we have the lemma of the *path dominance pruning* below.

Lemma 4.2. **(Path Dominance Pruning)** *Given a path p_z in the subgraph g of data graph G and a query path p_q in query graph q, path p_z can be safely pruned, if $o(p_q) \preceq o(p_z)$ does not hold (denoted as $o(p_q) \not\preceq o(p_z)$).*

4.2 Indexing Mechanism

In this subsection, we discuss how to obtain paths of length l in (expanded) subgraph partitions G_j and offline construct indexes, $\mathcal{I}_j$, over these paths to facilitate efficient processing of exact subgraph matching. Specifically, starting from each vertex $v_i \in V(G_j)$, we extract all paths p_z of length l (i.e., in an expanded subgraph partition that extends G_j outward by l-hop) and compute their path dominance embedding vectors $o(p_z)$ via the GNN model M_j. Then, we will build an aggregate R*-tree (or aR-tree) [8, 37] over these path embedding vectors $o(p_z)$, by using standard *insert* operator.

In addition to the *minimum bounding rectangles* (MBRs) of path embedding vectors $o(p_z)$ in index nodes, we store aggregate data such as those MBRs of path embedding vectors, $o'(p_z)$, via multi-GNNs (trained over randomized vertex labels, as discussed in Section 3.2), which is a concatenation of node embedding vectors $o'(v_t)$ for vertices $v_t \in p_z$ (Section 3.3). Moreover, in index nodes, we encode MBRs of path label embedding $o_0(p_z)$, which can be used for path label pruning (as given by Lemma 4.1 in Section 4.1).

Leaf Nodes. Each leaf node $N \in \mathcal{I}_j$ contains multiple paths p_z, where each path has an embedding vector $o(p_z)$ via a GNN in M_j.

Each path $p_z \in N$ is associated with aggregate data as follows:

- n path dominance embedding vectors $o'(p_z)$ obtained from n multi-GNNs, respectively, over randomized vertex labels in subgraph partition G_j, and;
- a path label embedding vector $o_0(p_z)$ via the GNN M_j.

Non-Leaf Nodes. Each non-leaf node $N \in \mathcal{I}_j$ contains multiple entries N_i, each of which is an MBR, $N_i.MBR$, of all path embedding vectors $o(p_z)$ for all paths p_z under entry N_i.

Each entry $N_i \in N$ is associated with aggregate data as follows:

- n MBRs, $N_i.MBR'$, on path dominance embedding vectors $o'(p_z)$ via n GNNs, resp., for all paths p_z in entry N_i, and;
- an MBR, $N_i.MBR_0$, over path label embedding vector $o_0(p_z)$ for all paths p_z under entry N_i.

4.3 Index-Level Pruning

We present effective pruning methods on the node level of indexes $\mathcal{I}_j$, which are used to prune (a group of) path false alarms in nodes.

Index-Level Path Label Pruning. We first discuss the *index-level path label pruning*, which prunes entries, N_i, in index nodes, containing path labels that do not match with the query path p_q.

Lemma 4.3. **(Index-Level Path Label Pruning)** *Given a query path p_q and an entry N_i of index node N, entry N_i can be safely pruned, if it holds that $o_0(p_q) \notin N_i.MBR_0$.*

Index-Level Path Dominance Pruning. Similarly, we can obtain the *index-level path dominance pruning*, which rules out those index node entries N_i, under which all path dominance embedding vectors are not dominated by that of the query path p_q.

Let $DR(o(p_q))$ be a *dominating region* that is dominated by an embedding vector $o(p_q)$ in the embedding space. For example, as shown in Figure 4, the embedding vector $o(q_1)$ of vertex q_1 (i.e., a special case of a path with length 0) has a dominating region, $DR(o(q_1))$. Then, we have the following lemma:

Lemma 4.4. **(Index-Level Path Dominance Pruning)** *Given a query path p_q and a node entry N_i, entry N_i can be safely pruned, if $DR(o(p_q)) \cap N_i.MBR = \emptyset$ or $DR(o'(p_q)) \cap N_i.MBR' = \emptyset$ holds.*

In Lemma 4.4, if embedding vector $o(p_q)$ (or $o'(p_q)$) does not fully or partially dominate $N_i.MBR$ (or $N_i.MBR'$), then the entire index entry N_i can be pruned. This is because any path p_z under entry N_i cannot be dominated by p_q in the embedding space, and thus cannot be a candidate path that matches with query path p_q.

4.4 GNN-Based Subgraph Matching Algorithm

In this subsection, we illustrate the exact subgraph matching algorithm by traversing the indexes over GNN-based path embeddings in Algorithm 3. Specifically, given a query graph q, we first obtain

Algorithm 3: Exact Subgraph Matching with GNN-Based Path Dominance Embedding

Input: i) a query graph q; ii) a trained GNN model M_j, and; iii) an aR-tree index $\mathcal{I}_j$ over subgraph partition G_j
Output: a set, $\mathcal{S}$, of matching subgraphs

1 obtain query path set Q with length l from a cost-model-based query plan φ
  for *each query path* $p_q \in Q$ **do**
2 $p_q.cand\_list = \emptyset$
3 obtain $o(p_q)$ and $o'(p_q)$ via multi-GNNs
4 obtain $o_0(p_q)$ via M_j
  `// traverse index` $\mathcal{I}_j$ `to find candidate paths`
5 initialize a *maximum heap* $\mathcal{H}$ accepting entries in the form $(N, key(N))$
6 $root(\mathcal{I}_j).list \leftarrow Q$
7 insert $(root(\mathcal{I}_j), 0)$ into $\mathcal{H}$
8 **while** $\mathcal{H}$ *is not empty* **do**
9 deheap an entry $(N, key(N)) = \mathcal{H}.pop()$;
10 **if** $key(N) < \min_{\forall p_q \in Q}\{||o(p_q)||_1\}$ **then**
11 terminate the loop;
12 **if** N *is a leaf node* **then**
13 **for** *each path* $p_z \in N$ **do**
14 **for** *each query path* $p_q \in N.list$ **do**
15 **if** $o_0(p_q) = o_0(p_z)$ `// Lemma 4.1`
16 **then**
17 **if** $o(p_q) \preceq o(p_z)$ *and* $o'(p_q) \preceq o'(p_z)$ **then**
18 $p_q.cand\_list \leftarrow p_q.cand\_list \cup \{p_z\}$
  `// Lemma 4.2`
19 **else**
20 **for** *each child node* $N_i \in N$ **do**
21 **for** *each query path* $p_q \in N.list$ **do**
22 **if** $o_0(p_q) \in N_i.MBR_0$ `// Lemma 4.3`
23 **then**
24 **if** $DR(o(p_q)) \cap N_i.MBR \neq \emptyset$ *and* $DR(o'(p_q)) \cap N_i.MBR' \neq \emptyset$ **then**
25 $N_i.list \leftarrow N_i.list \cup \{p_q\}$ `// Lemma 4.4`
26 **if** $N_i.list \neq \emptyset$ **then**
27 insert $(N_i, key(N_i))$ into heap $\mathcal{H}$
28 concatenate all candidate paths in $p_q.cand\_list$ for $p_q \in Q$ and refine/obtain matching subgraphs g in $\mathcal{S}$
29 **return** $\mathcal{S}$;

all query paths of length l in a set Q from the query plan φ (line 1). Then, for each query path $p_q \in Q$, we generate path embedding vectors $o(p_q)$, $o'(p_q)$ (via multi-GNNs), and $o_0(p_q)$ (via M_j) (lines 2-5). Next, we traverse the index $\mathcal{I}_j$ once to retrieve path candidate sets for each query path $p_q \in Q$ (lines 6-28). Finally, we refine candidate paths and join the matched paths to obtain/return subgraphs $g \in \mathcal{S}$ that are isomorphic to q (lines 29-30).

Refinement. After finding all candidate paths in $p_q.cand\_list$ for each query path $p_q \in Q$, we will assemble these paths (with overlapping vertex IDs) into candidate subgraphs to be refined and return the actual matching subgraph answers in $\mathcal{S}$ (lines 29-30).

Specifically, we consider the following two steps to obtain candidate subgraphs: 1) local join within each partition and 2) global join for partition boundaries. First, inside each partition, we perform the *multi-way hash join* by joining vertex IDs of candidate paths for different query paths. Then, for those boundary candidate paths across partitions, we also use the *multi-way hash join* to join them with candidate paths from all partitions globally. Finally, we can refine and return the resulting candidate subgraphs.

Complexity Analysis. Due to space limitations, please see the detailed descriptions for Algorithm 3 in our technical report [64].

Algorithm 4: Cost-Model-Based Query Plan Selection

Input: i) a query graph q; ii) path length l;
Output: a set, Q, of query paths in the query plan φ

1 $Q = \emptyset$; $Cost_Q(\varphi) = +\infty$;
2 select a starting vertex q_i with the highest degree
3 obtain a set, P, of initial paths of length l containing q_i
`// apply OIP, AIP, or εIP strategy in Section 5.2`
4 **for** *each possible initial path* $p_q \in P$ **do**
5 $local\_Q = \{p_q\}$; $local\_cost = 0$;
6 **while** *at least one vertex of* $V(q)$ *is not covered* **do**
7 select a path p of length l that connects with Q with minimum overlapping and minimum weight $w(p)$
8 $local\_Q \leftarrow local\_Q \cup \{p\}$
9 $local\_cost \leftarrow local\_cost + w(p)$
10 **if** $local\_cost < Cost_Q(\varphi)$ **then**
11 $Q \leftarrow local\_Q$
12 $Cost_Q(\varphi) \leftarrow local\_cost$
13 **return** Q

5 COST-MODEL-BASED QUERY PLAN

5.1 Cost Model

In this subsection, we provide a formal cost model to estimate the query cost of a query plan φ, which contains a set Q of query paths p_q from query graph q (used for retrieving matching paths from the index). Intuitively, fewer query paths with small overlapping would result in lower query cost, and fewer candidate paths that match with query paths will also lead to lower query cost.

Based on this observation, we define the query cost, $Cost_Q(\varphi)$, for query paths $p_q \in Q$ as follows:

$$Cost_Q(\varphi) = \sum_{p_q \in Q} w(p_q), \tag{13}$$

where $w(p_q)$ is the weight (or query cost) of a query path p_q.

Thus, our goal is to find a good query plan φ with query paths in Q that minimize the cost function $Cost_Q(\varphi)$ given in Eq. (13).

Discussions on the Calculation of Path Weights. We next discuss how to compute the path weight $w(p_q)$ in Eq. (13), which implies the search cost of query path p_q. Intuitively, when degrees of vertices in query path p_q are high, the number of candidate paths that may match with p_q is expected to be small, which incurs low query cost. We can thus set $w(p_q) = -\sum_{q_i \in p_q} deg(q_i)$, where $deg(q_i)$ is the degree of vertex q_i on query path p_q.

Alternatively, we can use other query cost metrics, such as the number of candidate paths (to be retrieved and refined) dominated by q_p in the embedding space. For example, we can set: $w(p_q) = |DR(o(p_q))|$, where $|DR(o(p_q))|$ is the number of candidate paths in the region, $DR(o(p_q))$, dominated by embedding vector $o(p_q)$.

5.2 Cost-Model-Based Query Plan Selection

Algorithm 4 illustrates how to select the query plan φ in light of the cost model (given in Eq. (13)), which returns a set, Q, of query paths from query graph q. Specifically, we first initialize an empty set Q and query cost $cost_Q(\varphi)$ (line 1). Then, we select a starting vertex q_i with the highest degree, whose node embedding vector expects to have high pruning power (line 2). Next, we obtain a set, P, of initial paths that pass through vertex q_i (line 3). We start from each initial path, p_q, in P, and each time expand the local path set $local\_Q$ by including one path p that minimally overlaps with $local\_Q$ and has minimum weight $w(p)$ (lines 4-9). For different initial paths in P, we always keep the best-so-far path set in Q and the smallest query cost in $Cost_Q(\varphi)$ (lines 10-12). Finally, we return the best query path set Q with the lowest query cost (line 13).

Discussions on the Initial Path Selection Strategy. In line 3 of Algorithm 4, we use one of the following three strategies to select initial query path(s) in P:

- **One-Initial-Path (OIP)**: select one path p_q with the minimum weight $w(p_q)$ that passes by the starting vertex q_i;
- **All-Initial-Path (AIP)**: select all paths that pass through the starting vertex q_i; and
- **ε-Initial-Path (εIP)**: randomly select ε paths passing through the starting vertex q_i.

Table 2: Statistics of real-world graph data sets.

Data Sets	$\|V(G)\|$	$\|E(G)\|$	$\|\sum\|$	$avg\_deg(G)$
Yeast (ye)	3,112	12,519	71	8.0
Human (hu)	4,674	86,282	44	36.9
HPRD (hp)	9,460	34,998	307	7.4
WordNet (wn)	76,853	120,399	5	3.1
DBLP (db)	317,080	1,049,866	15	6.6
Youtube (yt)	1,134,890	2,987,624	25	5.3
US Patents (up)	3,774,768	16,518,947	20	8.8

6 EXPERIMENTAL EVALUATION

6.1 Experimental Settings

To evaluate the effectiveness and efficiency of our GNN-PE approach, we conduct experiments on an Ubuntu server equipped with an Intel Core i9-12900K CPU, 128GB memory, and NVIDIA GeForce RTX 4090 GPU. The GNN training of our approach is implemented by PyTorch, where embedding vectors are offline computed on the GPU. The online subgraph matching is implemented in C++ with multi-threaded support on the CPU to enable parallel search on multiple subgraph partitions.

For the GNN model (as mentioned in Section 3.2), by default, we set the dimension of initial input node feature $F = 1$, attention heads $K = 3$, the dimension of hidden node feature $F' = 32$, and the dimension of the output node embedding $d = 2$. During the training process, we use the Adam optimizer to update parameters and set the learning rate $\eta = 0.001$. Due to different data sizes, we set different batch sizes for different data sets (from 128 to 1,024). Based on the statistics of real data graphs (e.g., only 7.24% vertices have a degree greater than 10 in Youtube), we set the degree threshold θ to 10 by default. Our source code and real/synthetic graph data sets are available at URL: *https://github.com/JamesWhiteSnow/GNN-PE*.

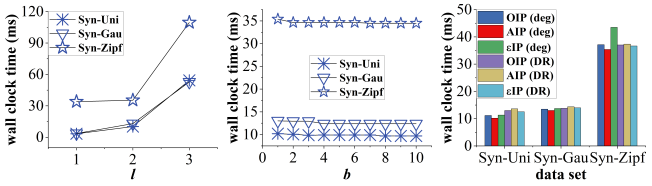
Baseline Methods. We compare the performance of our GNN-PE approach with that of eight representative subgraph matching baseline methods as follows: GraphQL (GQL) [28], QuickSI (QSI) [51], RI [13], CFLMatch (CFL) [11], VF2++ (VF) [31], DP-iso (DP) [26], CECI [10], and Hybrid [53] (mixed by GQL, RI, and QSI).

Real/Synthetic Graph Data Sets. We use both real and synthetic graphs to evaluate our GNN-PE approach, compared with baselines.

Real-world graphs. We used seven real-world graph data used by previous works [10, 11, 26–28, 49, 51, 53, 55, 71]. Based on graph sizes, we divide Yeast, Human, or HPRD into 5 partitions, WordNet into 7 subgraphs, DBLP into 30 partitions, Youtube into 346 partitions, and US Patents into 1,000 partitions. Statistics of these real graphs are summarized in Table 2.

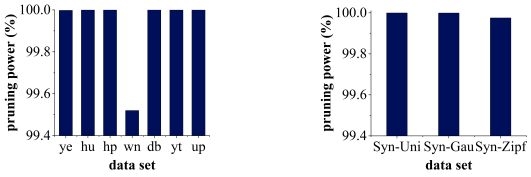
Table 3: Parameter settings.

Parameters	Values
the path length l	1, **2**, 3
the dimension, d, of the node embedding vector	**2**, 3, 4, 5
the number, n, of multi-GNNs	0, 1, **2**, 3, 4
the number, b, of GNNs with randomized initial weights	**1**, 2, 3, 4, 5, 6, 7, 8, 9, 10
the size, $\|V(q)\|$, of the query graph q	5, 6, **8**, 10, 12
the average degree, $avg\_deg(q)$, of the query graph q	2, **3**, 4
the size, $\|V(G)\|/m$, of subgraph partitions	5K, 6K, **10K**, 20K, 50K
the number, $\|\sum\|$, of distinct labels	100, 200, **500**, 800, 1K
the average degree, $avg\_deg(G)$, of the data graph G	3, 4, **5**, 6, 7
the size, $\|V(G)\|$, of the data graph G	10K, 30K, **50K**, 80K, 100K, 500K, 1M



(a) path length l (b) # of initial GNNs, b (c) query plan selection

Figure 8: GNN-PE efficiency evaluation w.r.t different parameters l, b, and query plan selection strategies.



(a) real-world graphs (b) synthetic graphs

Figure 9: GNN-PE pruning power on real/synthetic graphs.

Synthetic graphs. We generated synthetic graphs via NetworkX [24] which produces small-world graphs following the Newman-Watts-Strogatz model [62]. Parameter settings of synthetic graphs are depicted in Table 3. For each vertex v_i, we generate its label $L(v_i)$ by randomly picking up an integer in the range $[1, |\sum|]$, following the Uniform, Gaussian, or Zipf distribution. Accordingly, we obtain three types of data graphs, *Syn-Uni*, *Syn-Gau*, and *Syn-Zipf*.

Query Graphs. Similar to previous works [4, 10, 11, 26, 27, 33, 49, 55], for each graph data set G, we randomly extract/sample 100 connected subgraphs from G as query graphs, where parameters of query graphs q (e.g., $|V(q)|$ and $avg\_deg(q)$) are depicted in Table 3. Specifically, to generate a query graph q, we first perform a random walk in the data graph G until obtaining $|V(q)|$ vertices, and then check whether or not the average degree of the induced subgraph is larger than or equal to $avg\_deg(q)$. If yes, we randomly delete edges from the subgraph, until the average degree becomes $avg\_deg(q)$; otherwise, we start from a new vertex to perform the random walk.

Evaluation Metrics. In our experiments, we report the efficiency of our GNN-PE approach and baseline methods, in terms of the *wall clock time* (including both filtering and refinement time costs). We also evaluate the *pruning power* of our path label/dominance pruning strategies (as mentioned in Section 4.1), which is the percentage of candidate paths that can be ruled out by our pruning methods. For all the experiments, we take an average of each metric over 100 runs (w.r.t. 100 query graphs, resp.). We also test offline pre-computation costs of our GNN-PE approach, including the GNN training time, path embedding time, and index construction time.

Table 3 depicts parameter settings in our experiments, where default parameter values are in bold. For each set of subsequent experiments, we vary the value of one parameter while setting other parameters to their default values.

6.2 Parameter Tuning

The GNN-PE Efficiency Evaluation w.r.t. Path Length l. Figure 8(a) illustrates the GNN-PE performance, by varying the path length l from 1 to 3, where other parameters are by default. When l increases, more vertex labels and dominance embeddings on each path are used for pruning, which incurs higher pruning power. On the other hand, however, longer path length l will result in more data paths from graph G and higher dimensionality of the index, which may lead to higher costs to process more candidate paths. Thus, the GNN-PE efficiency is affected by the two factors above. From the figure, when $l = 1, 2$, the wall clock times are comparable for all the three synthetic graphs; when $l = 3$, the time cost suddenly increases due to much more candidate paths to process and the "dimensionality curse" [9]. Nonetheless, for all l values, the wall clock time remains low (i.e., 0.0033 $sec \sim$ 0.1096 sec).

The GNN-PE Efficiency Evaluation w.r.t. Number, b, of GNNs with Random Initial Weights. Figure 8(b) illustrates the performance of our GNN-PE approach, by varying the number, b, of the trained GNN models with random initial weights from 1 to 10, where other parameters are set by default. The experimental results show that the GNN-PE performance is not very sensitive to b. Since more GNNs trained will lead to higher training costs, in this paper, we set $b = 1$ by default. Nonetheless, for different b values, the query cost remains low (i.e., 0.01 $sec \sim$ 0.04 sec).

The GNN-PE Efficiency Evaluation w.r.t. Query Plan Selection Strategies. Figure 8(c) reports the performance of our GNN-PE approach with different query plan selection strategies, OIP, AIP, and εIP (as mentioned in Section 5), where the path weight $w(p_q)$ is estimated by vertex degrees or counts in dominating regions (denoted as (deg) and (DR), respectively), and default values are used for all parameters. From the figure, we can see that different strategies result in slightly different performances, and AIP (deg) consistently achieves the best performance.

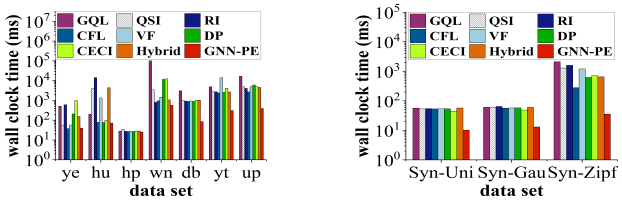
Due to space limitations, for the tuning of other parameters (e.g., d and n), please refer to our technical report [64]. In subsequent experiments, we will set parameters $l = 2$, $d = 2$, $n = 2$, and $b = 1$, and use AIP (deg) as our default query plan selection strategy.

6.3 Evaluation of the GNN-PE Effectiveness

The GNN-PE Pruning Power on Real/Synthetic Graphs. Figure 9 shows the *pruning power* of our proposed pruning strategies (i.e., path label/dominance pruning in Section 4.1) over both real and synthetic graphs, where the default values are used for all parameters. In subfigures, we can see that for all real/synthetic graphs, the pruning power can reach as high as 99.17% $\sim$ 99.99% (i.e., filtering out 99.17% $\sim$ 99.99% of candidate paths), which confirms the effectiveness of our pruning strategies and the efficiency of our proposed GNN-PE approach.

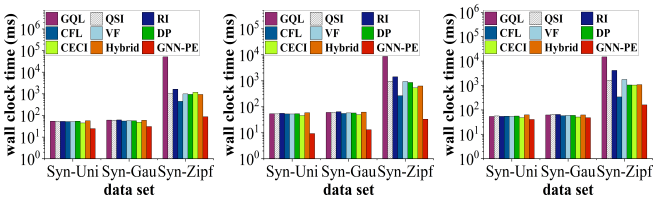
6.4 Evaluation of the GNN-PE Efficiency

The GNN-PE Efficiency on Real/Synthetic Graph Data Sets. Figure 10 compares the efficiency of our GNN-PE approach with 8



(a) real-world graphs (b) synthetic graphs

Figure 10: GNN-PE efficiency on real/synthetic graphs, compared with baseline methods.



(a) chain (b) star (c) tree

Figure 11: The GNN-PE efficiency w.r.t query graph patterns.

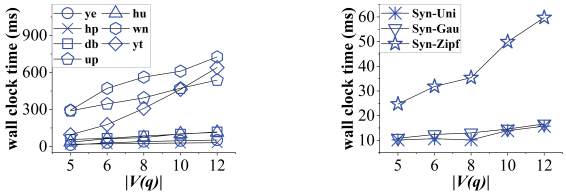
baseline methods over both real-world and synthetic graphs, where all parameters are set to default values. From the subfigures, we can see that our GNN-PE approach always outperforms baseline methods. Especially, for large-scale real (e.g., *yt* and *up*) and synthetic graphs (*Syn-Uni*, *Syn-Gau*, and *Syn-Zipf*), GNN-PE can achieve better performance than baselines by 1-2 orders of magnitude. For all real/synthetic graphs (even for *up* with 3.77M vertices), the time cost of our GNN-PE approach remains low (i.e., <0.56 *sec*).

To evaluate our GNN-PE query efficiency, in subsequent experiments, we vary different parameter values on synthetic graphs (e.g., query graph patterns, $|V(q)|$, $avg\_deg(G)$, $|V(G)|$, average degree of partitions, and # of edge cuts between partitions). To better illustrate the trends of curves, we omit the baseline results below.

The GNN-PE Efficiency w.r.t Query Graph Patterns. Figure 11 compares the performance of our GNN-PE approach with that of 8 baselines, for different query graph patterns, including chain, star, and tree, where the length of chain is 6, the degree of star is 6, the depth and fanout of tree are 3, respectively, and default values are used for other parameters. From the figure, we can see that, GNN-PE can achieve better performance than baselines by 1-2 orders of magnitude. For all synthetic graphs, the time cost of our GNN-PE approach remains low (i.e., <0.16 *sec*).

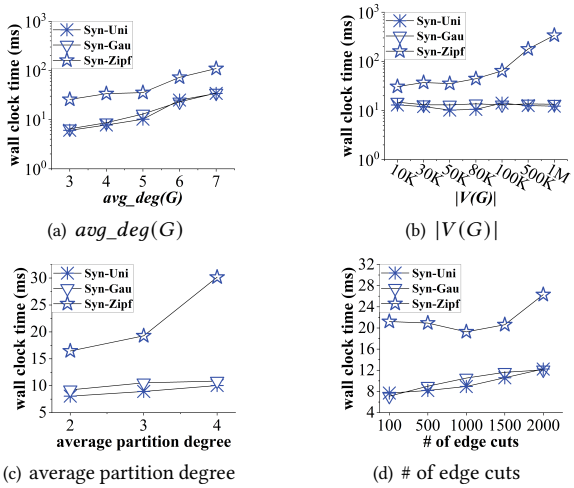
The GNN-PE Efficiency w.r.t. Query Graph Size $|V(q)|$. Figure 12 illustrates the performance of our GNN-PE approach by varying the query graph size, $|V(q)|$, from 5 to 12, where default values are used for other parameters. When the number, $|V(q)|$, of vertices in query graph q increases, more query paths from q are expected, which results in higher query costs for index traversal and refinement. Thus, larger $|V(q)|$ incurs higher wall clock time. For different query graph sizes $|V(q)|$, our GNN-PE approach can achieve low time costs (i.e., 0.01 *sec* ∼ 0.73 *sec*).

The GNN-PE Efficiency w.r.t. Average Degree, $avg\_deg(G)$, of the Data Graph G. Figure 13(a) presents the performance of our GNN-PE approach with different average degrees, $avg\_deg(G)$, of



(a) real-world graphs (b) synthetic graphs

Figure 12: GNN-PE efficiency w.r.t. query graph size $|V(q)|$.



(a) $avg\_deg(G)$ (b) $|V(G)|$

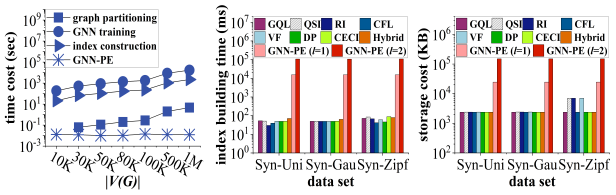
(c) average partition degree (d) # of edge cuts

Figure 13: GNN-PE efficiency evaluation on synthetic graphs.

the data graph G, where $avg\_deg(G)$ = 3, 4, ..., and 7, and default values are used for other parameters. Intuitively, higher degree $avg\_deg(G)$ in data graph G incurs lower pruning power and more candidate paths. Thus, when $avg\_deg(G)$ becomes higher, the wall clock time also increases. Nevertheless, the wall clock time remains small (i.e., less than 0.035 *sec* for *Syn-Uni* and *Syn-Gau*, and 0.109 *sec* for *Syn-Zipf*) for different $avg\_deg(G)$ values.

The GNN-PE Scalability Test w.r.t. Data Graph Size $|V(G)|$. Figure 13(b) tests the scalability of our GNN-PE approach with different data graph sizes, $|V(G)|$, from 10K to 1M, where default values are assigned to other parameters. Since graphs are divided into partitions of similar sizes and processed with multiple threads in parallel, the GNN-PE performance over *Syn-Uni* or *Syn-Gau* is not very sensitive to $|V(G)|$. Moreover, for *Syn-Zipf*, due to the skewed keyword distributions, the refinement step generates more intermediate results, which are more costly to join for larger $|V(G)|$. Nevertheless, for graph sizes from 10K to 1M, the wall clock time remains low (i.e., 0.010 *sec* ∼ 0.015 *sec* for *Syn-Uni* and *Syn-Gau*, and 0.031 *sec* ∼ 0.34 *sec* for *Syn-Zipf*), which confirms the scalability of our GNN-PE approach for large graph sizes.

The GNN-PE Efficiency w.r.t. Average Degree of Partitions. Figure 13(c) reports the performance of our GNN-PE approach by varying the average degree of partitions from 2 to 4, where # of edge cuts between any two partitions is 1,000 on average, and default values are used for other parameters. Higher degrees of partitions lead to more edge cuts and cross-partition paths, which incurs



(a) pre-computation (b) index building time (c) index storage cost

Figure 14: The GNN-PE offline pre-computation cost.

higher filtering/refinement costs. Thus, the time cost increases for a higher degree of partitions but remains small (i.e., <0.03 *sec*).

The GNN-PE Efficiency w.r.t. # of Edge Cuts Between Partitions. Figure 13(d) shows the GNN-PE performance by varying the average number of edge cuts between any two partitions from 100 to 2, 000, where the partition degree is 3, and default parameter values are used. In general, more edge cuts between partitions produce more candidate paths, which results in higher retrieval/refinement costs. There are some exceptions over *Syn-Zipf* (i.e., high query costs for fewer edge cuts), which is due to the skewed distribution of vertex labels in *Syn-Zipf*. Nonetheless, the query cost remains low (i.e., <0.03 *sec*) for different # of cross-partition edge cuts.

For other parameters (e.g., average degree, $avg\_deg(q)$, of the query graph q, subgraph partition size $|V(G)|/m$, and #, $|\sum|$, of distinct vertex labels), please refer to our technical report [64].

6.5 Offline Pre-Computation Performance

The GNN-PE Offline Pre-Computation Cost w.r.t. Data Graph Size $|V(G)|$. Figure 14(a) evaluates the offline pre-computation time of our GNN-PE approach, including time costs of the graph partitioning, GNN training, and index construction on path embeddings, compared with online GNN-PE query time, over synthetic graph *Syn-Uni* on a *single* machine, where we vary the graph size $|V(G)|$ from $10K$ to $1M$ and other parameters are set to default values. Specifically, for graph sizes from $10K$ to $1M$, the time costs of the graph partitioning, GNN training, and index construction are 0.06∼4.7 *sec*, 3.07 *min*∼4.79 *hours*, 21.07 *sec*∼36.63 *min*, respectively. The overall offline pre-computation time varies from 3.43 *min* to 5.4 *hours*, and the subgraph matching query cost is much smaller (i.e., 0.01∼0.014 *sec*). Please refer to our technical report [64] for similar experimental results over *Syn-Gau* and *Syn-Zipf*.

The GNN-PE Index Construction Time/Space Costs on Synthetic Graphs. Figures 14(b) and 14(c) compare the index construction time and storage cost, respectively, between our GNN-PE approach (path length $l = 1$ or 2) and 8 baselines, where other parameters to default values. From these figures, compared with baselines, in general, our GNN-PE approach needs higher time costs for offline index construction and more space costs for storing/indexing pre-computed GNN-based path embeddings. Note that, unlike baseline methods that construct an index for each query graph during online subgraph matching, our index construction is *offline and one-time only*, and the index construction time and storage cost are the summations over all partitions, which can be further optimized by using *multiple* servers in a distributed parallel environment (as our future work). Thus, our offline constructed index can be used to accelerate *numerous* online subgraph matching requests from users simultaneously with *high throughput*. Please refer to our technical report [64] for similar experimental results over real data sets.

7 RELATED WORK

Exact Subgraph Matching. Prior works on exact subgraph matching fall into two major categories, i.e., join-based [1, 2, 36, 45, 54] and backtracking-search-based algorithms [10, 11, 15, 26, 55]. In contrast, our proposed GNN-PE approach transforms graph paths to GNN-based embeddings in the embedding space (instead of directly over graph structures) and utilizes spatial indexes to conduct efficient searches at low query costs.

Several existing works [29, 52, 63] also considered subgraph matching by using feature vectors (e.g., frequency vectors based on path [29, 52] or non-path [63] patterns) to represent graphs. However, they retrieve small graphs in a graph database (e.g., chemical molecules) that contain the query graph, which differs from our problem that finds subgraphs in a single large data graph.

Approximate Subgraph Matching. An alternative problem is to develop methods to quickly return approximate subgraphs similar to a given query graph q. Existing works on approximate subgraph matching usually search for top-k most similar subgraphs from the data graph by using different measures [19, 21, 40, 72]. With AI techniques, recent works [6, 39, 42, 44, 59] proposed to use GNNs/DNNs to improve the efficiency of approximate subgraph matching. Although these methods can quickly check the subgraph isomorphism by avoiding the comparison of graph structures, they cannot guarantee the accuracy of query answers and have limited task scenarios (e.g., only applicable to approximately comparing two graphs, instead of finding subgraph locations in a data graph).

Learning-Based Graph Data Analytics. Recent works on learning-based graph data analytics used GNNs to generate graph embeddings for tasks like subgraph counting [16, 66], graph distance prediction [48], and subgraph matching [42, 50]). These works, however, either considered a different problem (e.g., count or graph distance estimation) or only reported approximate subgraph matching answers (instead of exact subgraph matching in our problem). Thus, we cannot apply their GNN embeddings to solve our problem.

8 CONCLUSIONS

In this paper, we propose a novel *GNN-based path embedding* (GNN-PE) framework for efficient processing of exact subgraph matching queries over a large-scale data graph. We carefully design GNN models to encode paths (and their surrounding 1-hop neighbors) in the data graph into embedding vectors, where subgraph relationships are strictly reflected by vector dominance constraints in the embedding space. The resulting embedding vectors can be used for efficient exact subgraph matching without false dismissals. Extensive experiments have been conducted to show the efficiency and effectiveness of our proposed GNN-PE approach over both real and synthetic graph data sets.

ACKNOWLEDGEMENTS

This work was supported by Natural Science Foundation of China (62272170), Natural Science Foundation (NSF CCF-2217104), and Shanghai International Joint Lab of Trustworthy Intelligent Software (22510750100). Mingsong Chen is the corresponding author.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems* 42, 4 (2017), 1–44.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 691—-704.

[3] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, Aristides Gionis, and Stefano Leonardi. 2012. Online team formation in social networks. In *Proceedings of the Web Conference (WWW)*. 839–848.

[4] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. 2019. Sequential and parallel solution-biased search for subgraph algorithms. In *Proceedings of the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. 20–38.

[5] László Babai. 2018. Group, graphs, algorithms: the graph isomorphism problem. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*. World Scientific, 3319–3336.

[6] 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 *Proceedings of the International Conference on Web Search and Data Mining (WSDM)*. 384–392.

[7] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 322–331.

[9] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 28–39.

[10] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1447–1462.

[11] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1199–1214.

[12] Ekaba Bisong and Ekaba Bisong. 2019. Introduction to Scikit-learn. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners* (2019), 215–229.

[13] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, 7 (2013), 1–13.

[14] S. Borzsony, D. Kossmann, and K. Stocker. 2001. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 421–430.

[15] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2017), 804–818.

[16] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. 2020. Can graph neural networks count substructures?. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 10383–10395.

[17] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. 2009. Monitoring path nearest neighbor in road networks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 591–602.

[18] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[19] Boxin Du, Si Zhang, Nan Cao, and Hanghang Tong. 2017. First: Fast interactive attributed subgraph matching. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1447–1456.

[20] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. 2019. Gradient descent provably optimizes over-parameterized neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 1–19.

[21] Sourav Dutta, Pratik Nayek, and Arnab Bhattacharya. 2017. Neighbor-aware search for approximate labeled graph matching using the chi-square statistics. In *Proceedings of the Web Conference (WWW)*. 1281–1290.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[23] Martin Grohe and Pascal Schweitzer. 2020. The graph isomorphism problem. *Commun. ACM* 63, 11 (2020), 128–134.

[24] Aric Hagberg and Drew Conway. 2020. Networkx: Network analysis with python. *URL: https://networkx. github. io* (2020).

[25] Jun Han and Claudio Moraga. 1995. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *Proceedings of the International Workshop on Artificial Neural Networks (IWANN)*. 195–201.

[26] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1429–1446.

[27] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 337–348.

[28] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 405–418.

[29] Craig A James. 2004. Daylight theory manual. *http://www. daylight. com/dayhtml/doc/theory/theory. toc. html* (2004).

[30] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. 2021. Fast: Fpga-based subgraph matching on massive graphs. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1452–1463.

[31] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81.

[32] Guy Karlebach and Ron Shamir. 2008. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology* 9, 10 (2008), 770–780.

[33] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 25–36.

[34] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 925–937.

[35] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 1–15.

[36] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 1099–1112.

[37] Iosif Lazaridis and Sharad Mehrotra. 2001. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 401–412.

[38] Harry R Lewis. 1983. Michael R. ΠGarey and David S. Johnson. Computers and intractability. A guide to the theory of NP-completeness. WH Freeman and Company, San Francisco1979, x+ 338 pp. *The Journal of Symbolic Logic* 48, 2 (1983), 498–500.

[39] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the International Conference on Machine Learning (ICML)*. 3835–3845.

[40] Zijian Li, Xun Jian, Xiang Lian, and Lei Chen. 2018. An efficient probabilistic approach for graph similarity search. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 533–544.

[41] Xiang Lian and Lei Chen. 2011. Efficient query answering in probabilistic RDF graphs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 157–168.

[42] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. 2020. Neural subgraph matching. *arXiv preprint arXiv:2007.03092* (2020).

[43] Andreas Loukas. 2020. What graph neural networks cannot learn: depth vs width. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 1–17.

[44] Brian McFee and Gert Lanckriet. 2009. Partial order embedding with multiple kernels. In *Proceedings of the International Conference on Machine Learning (ICML)*. 721–728.

[45] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 1692–1704.

[46] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*. 807–814.

[47] Abdelghny Orogat and Ahmed El-Roby. 2022. SmartBench: demonstrating automatic generation of comprehensive benchmarks for question answering over knowledge graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 3662–3665.

[48] Rishabh Ranjan, Siddharth Grover, Sourav Medya, Venkatesan Chakaravarthy, Yogish Sabharwal, and Sayan Ranu. 2022. Greed: A neural framework for learning graph distance functions. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 22518–22530.

[49] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 617–628.

[50] Indradyumna Roy, Venkata Sai Baba Reddy Velugoti, Soumen Chakrabarti, and Abir De. 2022. Interpretable neural subgraph matching for graph retrieval. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 8115–8123.

[51] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 364–375.

[52] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. 2002. Algorithmics and applications of tree and graph searching. In *Proceedings of the Principles of Database Systems (PODS)*. 39–52.

[53] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1083–1098.

[54] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A holistic approach to subgraph query processing. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 176–188.

[55] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 788–799.

[56] Damian Szklarczyk, Andrea Franceschini, Stefan Wyder, Kristoffer Forslund, Davide Heller, Jaime Huerta-Cepas, Milan Simonovic, Alexander Roth, Alberto Santos, Kalliopi P Tsafou, et al. 2015. STRING v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic Acids Research* 43, D1 (2015), D447–D452.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 1–11.

[58] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 1–12.

[59] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. 2016. Order-embeddings of images and language. In *Proceedings of the International Conference on Learning Representations (ICLR)*. 1–12.

[60] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement Learning Based Query Vertex Ordering Model for Subgraph Matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 245–258.

[61] Stanley Wasserman and Katherine Faust. 1994. Social network analysis: Methods and applications. (1994).

[62] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *Nature* 393, 6684 (1998), 440–442.

[63] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 335–346.

[64] Yutong Ye, Xiang Lian, and Mingsong Chen. 2023. Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding (Technical Report). arXiv:2309.15641 [cs.DB]

[65] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 4805–4815.

[66] Xingtong Yu, Zemin Liu, Yuan Fang, and Xinming Zhang. 2023. Learning to count isomorphisms with graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 4845–4853.

[67] Ye Yuan, Delong Ma, Zhenyu Wen, Zhiwei Zhang, and Guoren Wang. 2021. Subgraph matching over graph federation. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 437–450.

[68] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 4438–4445.

[69] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1992–2005.

[70] Yuejia Zhang, Weiguo Zheng, Zhijie Zhang, Peng Peng, and Xuecang Zhang. 2022. Hybrid Subgraph Matching Framework Powered by Sketch Tree for Distributed Systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1031–1043.

[71] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 340–351.

[72] Linhong Zhu, Wee Keong Ng, and James Cheng. 2011. Structure and attribute index for approximate graph matching in large graphs. *Information Systems* 36, 6 (2011), 958–972.