GHashing: Semantic Graph Hashing for Approximate Similarity Search in Graph Databases

Zongyue Qin
Peking University
Beijing, China
qinzongyue@pku.edu.cn

Yunsheng Bai
UCLA
Los Angeles, CA, US
yba@cs.ucla.edu

Yizhou Sun
UCLA
Los Angeles, CA, US
yzsun@cs.ucla.edu

ABSTRACT

Graph similarity search aims to find the most similar graphs to a query in a graph database in terms of a given proximity measure, say Graph Edit Distance (GED). It is a widely studied yet still challenging problem. Most of the studies are based on the pruning-verification framework, which first prunes non-promising graphs and then conducts verification on the small candidate set. Existing methods are capable of managing databases with thousands or tens of thousands of graphs, but fail to scale to even larger databases, due to their exact pruning strategy. Inspired by the recent success of deep-learning-based semantic hashing in image and document retrieval, we propose a novel graph neural network (GNN) based semantic hashing, i.e. **GHashing**, for approximate pruning. We first train a GNN with ground-truth GED results so that it learns to generate embeddings and hash codes that preserve GED between graphs. Then a hash index is built to enable graph lookup in constant time. To answer a query, we use the hash codes and the continuous embeddings as two-level pruning to retrieve the most promising candidates, which are sent to the exact solver for final verification. Due to the approximate pruning strategy leveraged by our graph hashing technique, our approach achieves significantly faster query time compared to state-of-the-art methods while maintaining a high recall. Experiments show that our approach is on average 20× faster than the only baseline that works on million-scale databases, which demonstrates GHashing successfully provides a new direction in addressing graph search problem for large-scale graph databases.

CCS CONCEPTS

• **Information systems** → **Nearest-neighbor search**; • **Computing methodologies** → *Supervised learning by regression*; **Neural networks**.

KEYWORDS

Graph Similarity Search, Semantic Hashing, Graph Neural Network

ACM Reference Format:
Zongyue Qin, Yunsheng Bai, and Yizhou Sun. 2020. GHashing: Semantic Graph Hashing for Approximate Similarity Search in Graph Databases. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403257

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403257

1 INTRODUCTION

Graph similarity search is an important problem in graph databases, which is critical to various real-world problems including drug design, program analysis, and business process management [11, 16, 22, 26, 27]. For example, one might want to search a drug database for a query chemical compound, with the hope to find drugs with similar structures and thus similar properties as desired.

In this paper, we consider range query, which aims to retrieve graphs from the database similar enough to a query. Different proximity measures can be used for such search task. Without loss of generality, we consider Graph Edit Distance (GED) [5] as the proximity measure, which is among the most popular ones, as many graph similarity measures are just its special cases [16]. Note that the exact GED computation is NP-hard.

The major bottleneck for graph similarity search lies in the large size of the graph databases, as the naive solution requires the scan of the whole database and GED computation is expensive. For example, the Alchemy molecule database provided in [7] contains 119, 487 graphs. Most of the existing graph search algorithms adopt a pruning-verification framework [22] to reduce unnecessary GED computations by pruning non-promising graphs. In the pruning stage, a lower bound for GED between the query graph and every data graph is computed. Graphs with GED lower bounds larger than the given threshold are pruned. In the verification stage, exact GED computation is performed between the query graph and the remaining candidates, and the graphs with the distance below the specified range will be returned.

Most of the existing pruning strategies are exact, ensuring all positive results are included, which becomes the bottleneck to handle large-scale graph databases. In one category, such as **Inves** [11] and **BSS_GED** [8], GED lower bound is computed for every graph, which is much cheaper than the exact GED computation but the whole database needs to be scanned. In contrast, methods such as **ML-Index** [16], **Pars** [26] and *k*-**AT** [22] build a database index to avoid the need of going through every data graph. However, these methods in practice suffer from long query latency and cannot scale well for two reasons: First, it can be slow to check if a particular substructure in the index exists in the query graph, which requires subgraph isomorphism check (NP-hard); Second, their GED lower bounds are often too loose, which leads to too many candidates being included and makes verification stage very slow.

In order to further scale graph similarity search to even larger databases, we propose an approximate pruning strategy which allows false negative. Specifically, inspired by the recent success of

deep-learning-based semantic hashing in image retrieval [13, 17] and information retrieval [23, 25], we adopt and extend the idea of supervised hashing to graph data by learning the hash function from graph pairs labeled with true GEDs. The GNN-based hash function and index are both fast and adaptive to different graph similarity metrics. Unlike traditional index which involves subgraph isomorphism checking as described previously, the hash index avoids graph comparison in real graph space and is much faster to be built. Additionally, since GNN is agnostic to graph similarity metrics, the whole model can be trained on other metrics such as Maximum Common Subgraph (MCS) [6]. For the online search stage, we prune graphs that are not close enough to the query in terms of their hash codes and embeddings, and verify the remaining candidates using exact GED solver. The pruning procedure is super fast as the hash code-based retrieval is constant time, and the embedding-based GED approximation is easy to compute, at the cost of introducing false negative error, due to the approximate nature of these GED preserving representations. Experimental results on both real-world and synthetic graph databases have shown that our GHashing-based pruning strategy is very effective, with a significant speedup in terms of query processing time while maintaining a relative high recall.

To summarize, we make the following contributions.

(1) We provide a first attempt to use a neural-network-based approach to address the similarity search problem for graph databases via graph hashing.
(2) We propose a novel method **GHashing**, a GNN-based graph hashing approach, which can automatically learn a hash function that maps graphs into binary vectors, to enhance the pruning-verification framework by providing a fast and accurate pruner.
(3) We conduct extensive experiments to show that our method not only has an average F1-score of 0.80 on databases with 5 million graphs but also on average 20× faster than the only state-of-the-art baseline on million-scale datasets.

2 PRELIMINARIES AND PROBLEM DEFINITION

In this work, we consider undirected graphs with node labels. Such a graph can be denoted as a triplet (V, E, L), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges and L is a labeling function that maps nodes to labels.

Graph Edit Distance. The graph edit distance (GED) between two graphs g_1 and g_2, denoted as $ged(g_1, g_2)$, is defined as the minimum number of edit operations to make g_1 isomorphic to g_2. An edit operation can be one of the following: (1) insert an isolated vertex with any label; (2) delete an isolated vertex; (3) change the label of a vertex; (4) insert an edge; (5) delete an edge.

Hamming Distance. Given two binary codes with length B, $\mathbf{c}_1, \mathbf{c}_2 \in \{-1, 1\}^B$, the hamming distance between $\mathbf{c}_1$ and $\mathbf{c}_2$, denoted as $\|\mathbf{c}_1 - \mathbf{c}_2\|_H$, is defined as the number of dimensions they differ, i.e. $\sum_{i=1}^{B} I_{\{c_1(i) \neq c_2(i)\}}$, where $c_1(i)$ and $c_2(i)$ are the ith dimension of $\mathbf{c}_1$ and $\mathbf{c}_2$ respectively, and $I_{\{\cdot\}}$ is the indicator function.

Graph Similarity Search under GED. Given a graph database with N graphs, $D = \{g_1, g_2, \cdots, g_N\}$, and a query graph q and a threshold $\tau \in \mathbb{N}$, the graph similarity search under GED measure aims to find $\{g | g \in D \wedge ged(g, q) \leq \tau\}$.

3 RELATED WORK

3.1 Graph Similarity Search

Since the exact computation for GED is NP-hard, existing solutions for graph similarity search typically adopt a pruning-verification framework. In the pruning phase, a lower bound for GED between q and g is computed. If it is already larger than τ, then g can be filtered out. Then in the verification phase, GED solvers are used for verifying the remaining graphs.

3.1.1 Pruning. Most of the pruning algorithms are inspired by the q-gram concept for string edit distance computation [21]. These methods decompose each data graph g into a multiset of q-grams and compute a lower bound for GED by counting the mismatched q-grams between g and q. To speed up the pruning, they build an inverted index with q-grams being the keys so that they only need to go through every key of the index, instead of every data graph, to find mismatched q-grams for the entire database. The earliest methods use special sub-structures of graphs as q-grams. K-**AT** [22] extracts sub-trees of g encompassing the k-hop neighborhood of a given vertex as q-grams. In contrast, **GSimSearch** [27] proposes path-based q-grams. However, the GED lower bounds of these methods are loose due to the fact that decomposed q-grams might overlap and their inability to handle large-degree nodes [16]. To overcome their limitations, **Pars** [26] proposes using non-overlapping graph partitions as q-grams. Inspired by **Pars**, **ML-Index** [16] proposes partitioning data graphs multiple times with different granularity in a selectivity-aware way, which makes **Pars** a special, degraded case of **ML-Index**. So far, **ML-Index** is the state-of-the-art index-based pruning algorithm for graph similarity search.

There are also pruning algorithms without index. Instead, they compare the query graph with every data graph so that they have tighter bounds for GED than index-based algorithms. For example, **BSS_GED** [8] and **Inves** [11] propose both pruning algorithms and verification algorithms. **BSS_GED** proposes a lower bound based on the degrees of vertices of g and q. And **Inves** incrementally partitions a data graph based on the query graph and uses the results to compute a lower bound of their distance. In this paper, we call this type of pruning algorithm pair-wise pruning since the pruning phase is conducted pair by pair.

However, none of the existing algorithms are scalable enough to deal with databases with millions of graphs. Although **ML-Index** is better than the earlier methods, its pruning ability is still too weak to handle massive databases since its bounds for GED tend to be loose. As for pair-wise pruning, they have to scan the entire database and compute a lower bound for every data graph. Thus they are inefficient when the number of data graphs is large. Also, existing pruning algorithms always require complicated computation. For example, **GSimSearch** needs to compute an exponential number of paths. **ML-Index** and **Pars** need to do sub-graph matching, which is NP-hard. As a result, their pruning phases, although faster than exact GED computation, are still slow.

3.1.2 Verification. In the verification phase, exact algorithms are used to verify if the GED between the query graph and a candidate

graph is indeed within the threshold. The most classical algorithm for exact GED computation is A* search algorithm [19]. But recently faster algorithms have been proposed [8, 11]. In our experiments, the verification algorithm in **BSS_GED** is more efficient than A* and the verification algorithm in **Inves**, so we adopt it to be the verification algorithm in our implementation.

In addition, since getting approximate answers within a shorter time is far more ideal than waiting for a long time to get exact ones in many real-life applications, many methods have been proposed to compute GED approximately. [18, 20] treats the GED computation as a linear sum assignment problem and get an estimation of GED in polynomial time. **GBDA** [15] introduces a probabilistic graphic model to predict GED and has better effectiveness than [18, 20]. There have also been studies trying to use graph neural network to estimate the similarity between a pair of graphs. **UGraphEmb** [3] learns a function that generates a continuous embedding for each input graph, and the L2 distance of two embeddings approximates the similarity between two graphs. **SimGNN** [1] generates an embedding for a graph and input two graphs' embeddings into a neural tensor network to get the similarity score. It also designs a pairwise vertex comparison method to enhance performance. **GMN** [14] jointly reasons on the pair through a cross-graph attention-based matching mechanism to compute the similarity score. **GraphSim** [2] produces a similarity matrix for the graph pair which is fed into CNNs for similarity score computation.

Although these approximate verification algorithms are faster, it is not wise to use them alone to handle massive graph databases, as they have to scan the entire database. Instead, combining them with index-based pruning algorithms is more efficient, which will be considered as future work.

3.2 Semantic Hashing for Similarity Search

Hashing methods for approximate nearest neighbor has been popular due to its promising performance in both efficiency and effectiveness [23]. These methods use hash functions to map input data to a discrete code space called Hamming Space, and hamming distance between hash codes reflects the similarity between input items. Then hash codes are stored in a hash index so that retrieving items with a specific hash code only takes constant time.

Semantic hashing uses deep neural networks to learn the embeddings or embedding functions [13, 17]. However, so far existing semantic hashing methods only focus on extending the hashing methods to images and documents and applying semantic hashing to graph similarity search is challenging for several reasons.

First, semantic hashing typically treats similarity search as a classification task, so they only need the hamming distance between hash codes of two similar items smaller than a constant threshold and vice versa. But as the threshold for graph similarity search may vary from query to query, it is more appropriate to treat it as a regression problem, which requires estimating GED by hamming distance. As a result, a more powerful hash function is needed.

Second, semantic hashing predicts the similarity between items by the hamming distance between hash codes. But, as hamming distance is upper-bounded while the GED is not, how to establish a relationship between hamming distance and GED is a problem.

Third, the goal of graph similarity search is to identify those similar graph pairs, but in real datasets, similar graph pairs are extremely rare. For example, in AIDS dataset only 0.5‰ graph pairs have GED smaller than 7. The lack of positive examples makes it difficult to train the hash function. Besides, while the similarity between images and documents can be easily computed by counting the number of common tags, computing GED is extremely slow, which makes it even harder to sample positive examples from datasets. Therefore, how to efficiently handle the scarcity of positive examples, i.e. similar graph pairs, is another important problem.

4 APPROACH

As illustrated in Figure 1, our approach is divided into two stages: the offline stage and the online stage. In the *offline stage*, we train a GNN based hash function and then use it to construct a hash index. The value of the hash index is a list of graph IDs. The algorithm to construct the hash index with learned hash function is described in appendix A. For the *online stage*, we follow the pruning-verification framework. When a query graph comes in, we first compute its hash code and embedding. Then based on the query threshold and the hash code, we retrieve all data graphs stored in a hamming ball. Next, an optional second-level pruning can be done by computing the L2 distance between the query and every retrieved graph via their embeddings. Finally the remained graphs are sent to final verification.

4.1 Offline Stage

4.1.1 GNN Architecture for Hash Function. Our hash function relies on GNN to extract features from graphs, which has proven to be effective in previous studies. As illustrated in Figure 1, our hash function $H(\cdot)$ can be divided into three components: graph feature extraction, embedding and, binarization, and the embedding function $F(\cdot)$ is composed of the first two components.

In the graph feature extraction stage, we use three graph convolution layers [12] and a graph attention pooling layer [3] to extract features from graphs. The details of the two types of layers are described in appendix D. In the embedding stage, fully connected layers generate continuous embeddings such that the L2 distance between embeddings approximates GED. Finally, binarization stage converts embeddings into hash codes (binary vectors) with several fully connected layers. The output layer uses tanh as the activation function so that the outputs are within $(-1, 1)$, which is further binarized by a threshold 0. We also design a special loss (Section 4.1.2) for binarization to force the binary constraint to the outputs.

As mentioned earlier, the first two parts compose a graph embedding function $F(\cdot)$. It is trained so that $\|F(g_1) - F(g_2)\|_2^2 \approx ged(g_1, g_2)$. There are two benefits of learning such an embedding function compared to learning a hash function directly. First, it helps the training of hash function. The intuition is that a graph's hash code and continuous embedding should be generated based on the same features because both of them aim to preserve GED between graphs. F and H are jointly learned so that the GCN layers and pooling layers can learn to extract those common features. Besides, training continuous embedding is easier than training binary codes because of the discrete nature of the latter. Therefore, it is likely that the neural network can generate continuous embedding with good quality after a short time of training and then learns to compress and binarize the embedding into binary code gradually.

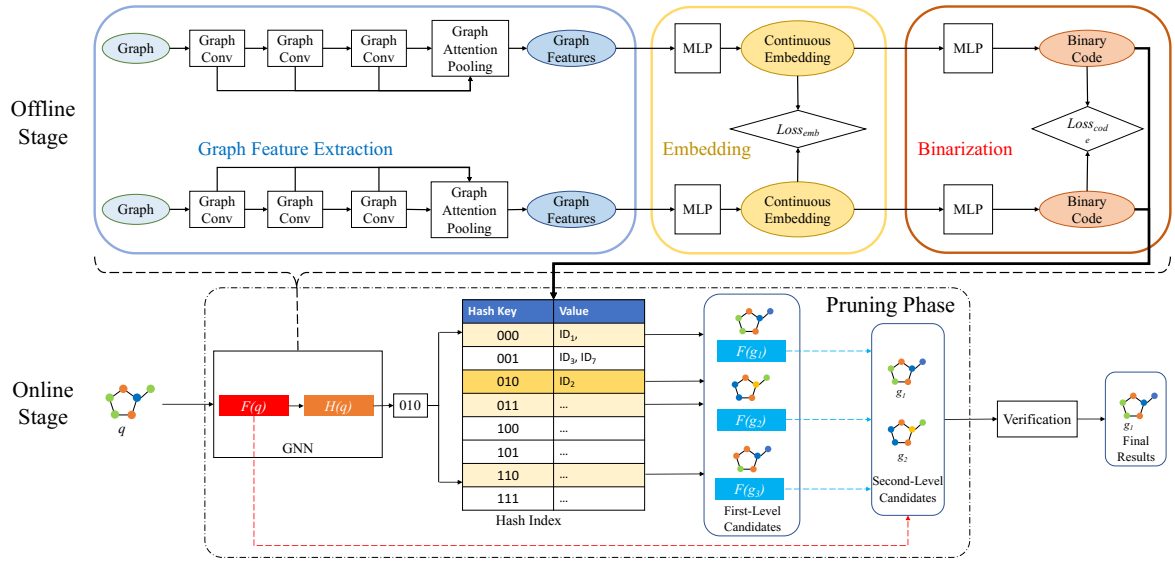


Figure 1: Overview illustration of GHashing. GHashing consists of an offline stage and an online stage. In the offline stage we learn the hash function and build a hash index where graph IDs are stored. The hash function (GNN) $H(\cdot)$ can be divided into three components: graph feature extraction, embedding, and binarization. The first two components compose the embedding function $F(\cdot)$. The training loss consists of two components, one for continuous embeddings ($Loss_{emb}$) and the other for hash codes ($Loss_{code}$). After training, the output hash codes for all data graphs are used to construct the hash index. Then in the online stage, when a query graph q comes in, we first compute its hash code $H(q) = 010$ and continuous embedding $F(q)$ with GNN. After retrieving graphs whose hash codes are within a hamming ball of small radius around 010 (000, 010, 011, 110), we use their embeddings and $F(q)$ to estimate their GED for a second-level pruning. The graphs remained are sent to the verification phase where exact GED are computed.

Second, the continuous embedding can be exploited to explicitly improve the efficiency and pruning ability for our pruning method. Specifically, after we use hash code and hash index to retrieve all candidate graphs, we propose using the continuous embedding of them for a second-level filtering, where we compute $\|F(g) - F(q)\|_2^2$ for each g in the candidates and remove ones whose corresponding results are larger than a threshold. Since computing L2-distance between vectors is far faster than computing GED between graphs, this second-level filtering could save lots of unnecessary verification time, at the cost that a few more correct results may be falsely filtered. So whether to use it is, in fact, a trade-off and is optional according to a user's demand.

4.1.2 Loss Function. As illustrated in Figure 1, the loss function of **GHashing** has two components: the loss for continuous embeddings (L_{emb}) and the loss for hash codes (L_{code}), and the integrated loss can be expressed as

$$\lambda L_{code}(g_1, g_2) + (1 - \lambda) L_{emb}(g_1, g_2) \tag{1}$$

where $\lambda \in (0, 1)$ is a hyperparameter to control the trade-off between the two tasks.

Design of L_{code}. A natural option for $L_{code}(g_1, g_2)$ is to measure the difference between the predicted distance $\hat{y}$ and the ground truth GED y:

$$L_{code}(g_1, g_2) = L(\hat{y}, y), \tag{2}$$

where $\hat{y} = |H(g_1) - H(g_2)\|_H$, $y = ged(g_1, g_2)$, and $L(\cdot, \cdot)$ can be any loss function for regression, *e.g.* $L2$ loss. However, a problem is that while GED can be arbitrarily large, the hamming distance between two B-bit hash codes can not be larger than B. Therefore, dissimilar graph pairs with GED far more larger than B will have more influence on the training than the similar graph pairs which we actually care about. Besides, as B cannot be too large for semantic hashing to work efficiently, such dissimilar graph pairs are common in real datasets. Therefore, a simple loss such as $L2$ loss will not serve the purpose, and a more sophisticated loss design is needed.

Specifically, we argue that a good loss function $L(\hat{y}, y)$ in L_{code} should satisfy the following properties:

(1) **Small punishment when both values are large.** In real world applications, the threshold τ in the range query is often small. In this case, if both predicted distance $\hat{y}$ and ground truth distance y are very big, we should consider the loss as small, as we can correctly prune those graphs based on $\hat{y}$. Therefore, we define a truncated loss with the form $L(\hat{y}, y) = L'(\min\{\gamma, \hat{y}\}, \min\{\gamma, y\})$, where γ is the hyperparameter to control the maximum GED we are interested in.

(2) **Asymmetric, i.e.,** $L(\hat{y}, y) \neq L(y, \hat{y})$**.** Once a correct result is falsely pruned, there is no way to recover it. That is to say, the case that the hamming distance is larger than the GED (over-estimation) is worse than the case that the hamming distance is smaller (under-estimation), and the former should be punished more severely, e.g., $L(3, 5) < L(5, 3)$.

(3) **Minimized when** $\hat{y} = y$**.** When the predicted GED distance is the same as the ground truth, the loss should give us the minimum value.

(4) **Convex.** The loss function should be convex to reduce the difficulty of optimization.

To that end, we propose the following exponential-weighted *L2* loss, which satisfies all the properties mentioned above:

$$L(\hat{y}, y) = L'(\hat{y}', y') = e^{a(\hat{y}'-y')}(\hat{y}' - y')^2 \tag{3}$$

where $a > 0$ is a hyperparameter related to convexity property, which will be discussed in Theorem 1; and $\hat{y}' = \min\{\gamma, \hat{y}\}$ and $y' = \min\{\gamma, y\}$ are called *clipped hamming distance* and *clipped GED*, which are designed for Property (1).

The additional benefit for clipping GED and hamming distance by an upper bound γ is for efficiency concern. Computing GED is extremely slow, so it will take too long for the training to complete if we compute the exact GED for every training pair. Computing clipped GED, i.e. $\min\{\gamma, ged(g_1, g_2)\}$, is much faster due to the search-based nature for GED solvers. We can either compute a lower bound for GED in a short time and see if it is already larger than γ to avoid the exact GED computation, or we can ignore all the search paths with lengths longer than γ, to accelerate the search algorithm for exact GED computation.

For the exponential-weighted *L2* loss, we have the following theorem. (The details of proof is in appendix E.)

Theorem 1. *The exponential-weighted L2 loss defined in (3) is minimized when $\hat{y}' = y'$. Besides, assume $|\hat{y}' - y'| \leq \gamma$, then the loss is convex if $a \in (0, \frac{2-\sqrt{2}}{\gamma})$.*

Based on (3), Theorem 1 and the fact that $|\hat{y}' - y'| \leq \gamma$, it is easy to verify that our loss satisfies requirement (2), (3), and (4), if $a \in (0, \frac{2-\sqrt{2}}{\gamma})$. Besides, when the hamming distance and GED are both larger than γ, the loss becomes 0, which satisfies requirement (1). The loss being zero is reasonable because, as mentioned before, if both the hamming distance and GED are larger than γ, our approach can always correctly prune it.

Handling discrete constraints. So far, we still face another challenge regarding optimization, as it is infeasible to directly optimize the loss because each dimension of $H(g)$ is binary and the function is non-differentiable. A common trick is to drop the binary constraint and replace the hamming distance with the L2 norm, since under the constraint that $H(\cdot) \in \{-1, 1\}^B$, $\|H(g_1) - H(g_2)\|_{\mathcal{H}} = \frac{1}{4}\|H(g_1) - H(g_2)\|_2^2$ [13].

It is, however, inappropriate to completely ignore the binary constraint due to the discrepancy between the Euclidean space and the Hamming space. Thus we still need a way to encourage the output of GNN close to ± 1. To this end, we add a binary regularization proposed in [17] to the loss, which is $\||H(g_1)|-\mathbf{1}\|_1+\||H(g_2)|-\mathbf{1}\|_1$, where $\|\cdot\|_1$ is the L1-norm, $\mathbf{1}$ is a vector of all ones and $|\cdot|$ is the absolute value operation.

To sum up, the loss function of the output hash codes is

$$L_{code}(g_1, g_2) = e^{a(\hat{y}'-y')}(\hat{y}'-y')^2+\beta\||H(g_1)|-\mathbf{1}\|_1+\beta\||H(g_2)|-\mathbf{1}\|_1 \tag{4}$$

where $\hat{y}' = \min\{\gamma, (\frac{1}{4}\|H(g_1) - H(g_2)\|_2^2\}$, $y' = \min\{\gamma, ged(g1, g2)\}$, $a \in (0, \frac{2-\sqrt{2}}{\gamma})$ and β is a hyper-parameter.

Design of L_{emb}. As for L_{emb}, we use MSE as the loss function. Besides, as computing exact GED is too slow, we also use clipped GED as the training label. Since graph similarity search does not care about graph pairs with large GED, it should not affect the effectiveness of our approach. So the loss for embeddings is

$$L_{emb}(g_1, g_2) = (\min\{\gamma, \|F(g_1) - F(g_2)\|_2^2\} - y')^2 \tag{5}$$

where $y' = \min\{\gamma, ged(g1, g2)\}$.

In addition, to minimize the binary regularization's influence on learning F, we suggest that the weight for L_{emb} should be far larger than the weight for L_{code}, which means λ in Eq. 1 should be small.

4.1.3 Training Data Preparation. Another challenge for training the hash function is that similar graph pairs are scarce in most of the real-world datasets. For example, AIDS is a commonly used dataset for graph similarity search. On average, only 0.5‰ graph pairs in AIDS have GED smaller than 7. However, as our goal is to identify similar graphs, similar graph pairs are important during training, and the scarcity of similar graph pairs during training will result in sub-optimal performance of the graph neural network. A naive solution is to first compute the GED between every pair of graphs and then split them into similar graph pairs and dissimilar graph pairs for sampling. However, unlike the similarity between images and documents, which can be easily computed by counting the number of common tags, computation for GED is extremely slow. Therefore, such a method can only handle small databases.

Instead, we propose generating synthetic similar graph pairs to augment data during training, since it helps GNN to identify which graphs are similar, and thus increases the performance of the hash function. In addition, generating synthetic graph pairs is much faster than computing GED. Therefore it is more efficient when training on massive datasets.

Algorithm 2 in appendix B illustrates how to generate synthetic similar graph pairs. Given Algorithm 2, we obtain our training samples for each iteration following the steps below. First, we randomly sample m graphs from the datasets and compute the GED between each two of them. Then for each sampled graph, we use Algorithm 2 to generate k similar graphs and corresponding labels, where k is a hyper-parameter. Therefore, for each iteration we have $\frac{m(m-1)}{2} + mk$ training graph pairs in total.

4.2 Online Stage

The online stage consists of pruning phase and verification phase. In the *pruning phase*, we first compute the query graph q's hash code $H(q)$ and continuous embedding $F(q)$ with the trained GNN, i.e. GHashing. Then we search the keys of the index to find hash codes whose hamming distance to $H(q)$ is smaller than $\tau + t$. The value of t provides a trade-off between efficiency and effectiveness. In this work we set $t = 1$, and it will be further studied in future work. Based on the hash codes obtained in last step, we retrieve all data graphs mapped to these codes as well as their embedding with the hash index. Recall that the square of the L2-distance between continuous embeddings approximates the GED between graphs. So we can do an optional pruning by excluding every graph whose embedding, say $F(g)$, satisfies that $\|F(g) - F(q)\|_2^2 > \tau + 0.5$. The threshold $\tau+0.5$ is set heuristically and will be studied further. Since computing L2-norm of a vector is far more faster than computing exact GED, this further filtering will reduce some more unnecessary GED computation and shorten the time to respond a query.

Table 1: Properties of baselines. "FP" means if it has false positives. "FN" means if it has false negatives. "I" means if it builds index. "M" means if it can complete tests on datasets with 5 million graphs in two days.

Name	FP	FN	I	M
ML-Index[16]	N	N	Y	N
Inves[11]	N	N	N	N
BSS_GED[8]	N	N	N	Y
Naive	N	Y	Y	Y
GH	N	Y	Y	Y

In the *verification phase*, after getting the final candidates, we verify every graph remained with exact verification algorithm. In our implementation, we adopt **BSS_GED** as the verification algorithm. More details about the algorithm for the query are described in appendix C.

5 EXPERIMENTS

The details of experiment settings are introduced in appendix G.

5.1 Datasets and Baselines

We use three real-world datasets, AIDS[1], LINUX [24], ALCHEMY [7], and two synthetic datasets in the experiments. The graphs of synthetic datasets are generated using the ER [9] and BA model [4] respectively, and the node labels are sampled from multinomial distributions. The details of datasets are described in appendix F. From each dataset, we randomly select 100 graphs to be the query graphs.

When evaluating effectiveness, as other pruning methods are all exact, we compare **GH** with a naive version of our approach, which is simply learning continuous embedding and then binarizing it by threshold 0. We denote it as **Naive**. When evaluating efficiency, we choose **ML-Index**, **Inves** and **BSS_GED** as baselines. **ML-Index** is the state-of-the-art index-based pruning algorithm, so we do not compare **GH** with other index-based algorithms. For independent pruning algorithm (**GH**, **Naive**, **ML-Index**), we use the verification algorithm in **BSS_GED** in verification phase due to its efficiency. The detailed properties of each method are listed in Table 1.

Table 2: Recall/$|C|$ ratio ($\times 10^{-5}$) for GHashing and Naive

τ	AIDS			LINUX			ALCHEMY		
	GH	GH-	Naive	GH	GH-	Naive	GH	GH-	Naive
1	**690**	30	7.0	**23**	15	16	**11**	3.5	4.2
2	**270**	19	5.5	**20**	12	12	**4.9**	2.4	2.7
3	**150**	15	4.8	**16**	10	10	**2.8**	1.7	1.9
4	**98**	11	4.3	**14**	8.8	8.6	**1.9**	1.4	1.4
5	**69**	7.3	3.8	**13**	7.9	7.7	**1.5**	1.0	1.2
6	**52**	6.2	3.6	**11**	6.7	6.6	**1.2**	1.0	1.0

5.2 Effectiveness Analysis

We evaluate the effectiveness of **GH** with $\tau = \{1, 2, 3, 4, 5, 6\}$, which are commonly used in [11, 16, 27].

[1]https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data

Table 3: Recall/$|C|$ ratio ($\times 10^{-5}$) for different code lengths on AIDS dataset

τ	24 bit	32 bit	48 bit
1	28	30	70
2	18	19	43
3	13	15	30
4	9.6	11	21
5	7.5	8.8	15
6	6.1	7.1	12

First we evaluate the average candidate set size ($|C|$), precision, recall and F1-score of **GH**, **GH** without the second-level pruning (denoted as **GH-**) and **Naive** on three real datasets. The average candidate set size is an important metric for effectiveness because if a pruning method lets every data graph pass, the precision and recall will both be 1 but the pruning is not effective at all. Besides, the verification time, which takes up most of the response time, is linear to the candidate set size, so it is also an indicator of efficiency.

The results in Figure 2 (a)-(d) show that **GH** achieves F1-score of at least 0.85 under different τ on AIDS. Its average candidate number is 42× smaller than **Naive**'s, while its F1-score is only 0.05 less. Also, results in Figure 2 (e)-(h) show that on LINUX, **GH** is not only able to return candidate sets on average 40% smaller than **Naive**'s, it also achieves better recall and F1-score. In fact, **GH**'s recall is at least 0.92 and its precision is at least 0.99 on LINUX. In addition, Figure 2 (k), (l) show that on ALCHEMY, while **Naive**'s average recall and F1-score are only 0.60 and 0.73, **GH**'s average recall and F1-score are 0.77 and 0.86, which are significantly larger. Besides, Figure 2 (i) show that the average candidate set sizes between **GH** and **Naive** are close on ALCHEMY.

What is more, we emphasize that the recall and candidate set size are the most important metrics about effectiveness. This is because, after verification, all false positives will be discarded so the precision is always 1 (unless all correct results are falsely pruned, in which case the precision is 0). So the recall reveals the possibility of a correct result being included in returned results and the ratio between recall and candidate set size reveals the cost to retrieve one correct result. Higher ratio suggests better trade-off between efficiency and effectiveness. So in Table 2, we report the recall/$|C|$ ratio of **GH** and **Naive**. The ratio of **GH** is significantly higher than **Naive**'s, proving that **GH** provides a better pruning power than **Naive**.

In addition, comparing **GH** and **GH-** in Figure 2 (a), (e), (i), we observe that $|C|$ of **GH** are an order of magnitude smaller than **GH-**'s, which helps shorten the response time. On the other hand, results in Figure (c), (d), (g), (h), (k), (l) show that second-level pruning slightly decreases the recall and F1-score of our approach. In Table 2, the recall/$|C|$ ratio of **GH** is always larger than **GH-**'s, suggesting better effectiveness. But the second level pruning will also introduce extra query latency. Therefore, we conclude that the second level pruning provides a trade-off between efficiency and effectiveness.

We also show the violin plot of the predicted GED by the clipped hamming distance between hash codes under different real GEDs on the AIDS dataset in Figure 3. The results in Figure 3 show that the hamming distance is usually smaller than the real GED. It matches

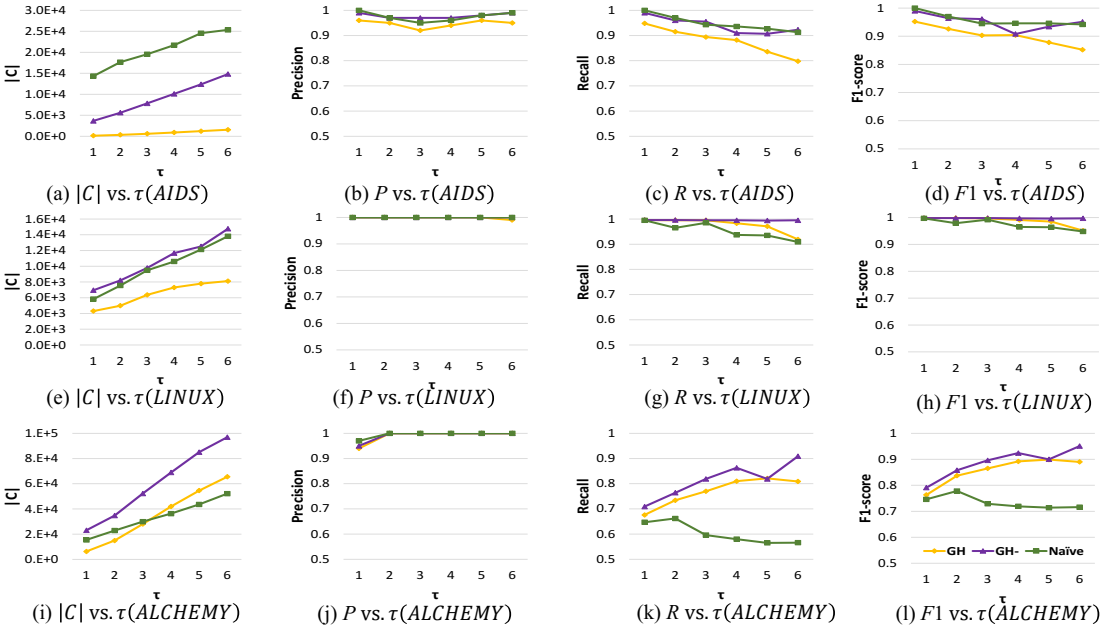


Figure 2: Candidate set size ($|C|$), precision (P), recall (R) and F1-score ($F1$) of our approach (GH), our approach without optional second-level pruning (GH-) and naive semantic graph hashing (Naive) on three real datasets.

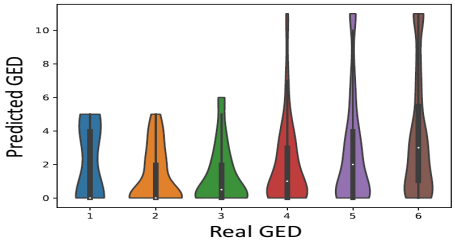


Figure 3: Violin plots of predicted GED given real GED on AIDS dataset. A violin plot is an extension of the box plot, where each "violin" is a plot of the density function of a histogram produced from the data.

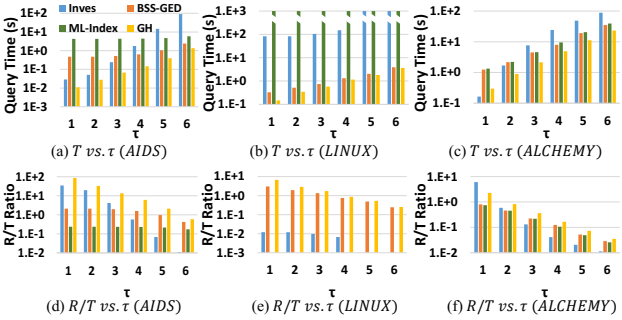


Figure 4: Average query time (T), recall/time ratio for GH and three baselines on three real datasets. The truncated bar means the total running time exceeds 1000 minutes.

our goal to design the exponential-weighted L2 loss and is ideal for a non-exact pruning algorithm, as it suggests that only a small number of correct results will be falsely pruned.

Finally, we evaluate how the length of the hash codes affect the effectiveness of **GH**. We run the experiments on AIDS with the length of hash codes being $\{24, 32, 48\}$ bits. We exclude the second-level pruning so that the returned results are solely determined by hash codes. We report the recall/$|C|$ ratio in Table 3. As the length of hash codes increases from 24 to 48, the recall/$|C|$ ratio increases steadily by 2.2×. It means that longer hash codes provide a better trade-off between efficiency and effectiveness, which fits intuition.

5.3 Efficiency Analysis

In this subsection, we evaluate the efficiency and scalability of **GH** and three competitors, **ML-Index**, **Inves** and **BSS_GED**.

Table 4: Costs of offline stage for GHashing and ML-Index

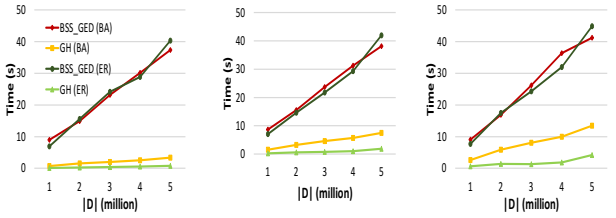
Dataset		AIDS	ALCHEMY	LINUX	ER	BA
GH	Time	759s	443s	1143s	3172s	3544s
	Space	56M	136M	52M	5.1G	5.1G
ML-I	Time	696s	248s	N/A	N/A	N/A
ndex	Space	678M	960M	N/A	N/A	N/A

5.3.1 Evaluating Offline Stage. We first compare the time and space costs of the offline stage in **GH** and **ML-Index**. The results are shown in Table 4. The offline stage of **GH** includes training GNN and building index. The reported space costs include the costs for neural networks and the hash index. As for **ML-Index**, since it has to build different indices for different query thresholds, we report the sum of the time and space costs of it for $\tau = \{1, 2, 3, 4, 5, 6\}$.

In our experiments, **ML-Index** runs out of memory when building indices for the LINUX dataset and does not finish building after 24 hours for ER and BA datasets, which proves it is not scalable enough to handle massive databases. As for AIDS and ALCHEMY datasets, the space costs of our approach is significantly less than **ML-Index**. It is because our approach only needs one index for different thresholds while **ML-Index** builds different indices for different thresholds. Besides, our index only stores hash code, continuous embedding, and ID of each graph, while **ML-Index** has to store the unique partitions of all the graphs. In addition, the time costs of our approach are within a reasonable range for all datasets.

5.3.2 Evaluating Online Stage. Then we compare the efficiency of the online stage of **GH** with three baselines. We conduct experiments on three real-world datasets with $\tau = \{1, 2, 3, 4, 5, 6\}$. We consider the average query time as the principal indicator of search efficiency. We also report the recall/time ratio for each method. Similar to the recall/$|C|$ ratio, recall/time ratio reveals the time cost to retrieve a correct results. During experiments, we terminate the process if the total query time exceeds 1000 minutes.

Figure 4 (a)-(c) shows the average query time *w.r.t* τ, for different methods on three real datasets. The results show that our approach is the fastest in all datasets where $\tau = \{2, 3, 4, 5, 6\}$ and is the fastest in two out of three datasets where $\tau = 1$. On average, our approach is 9.07× faster than the fastest baseline **BSS_GED** where $\tau = \{1, 2, 3\}$ and 5.45× faster where $\tau = \{1, 2, 3, 4, 5, 6\}$. Figure 4 (d)-(e) shows the recall/time ratio for different methods. **GH** achieves the highest R/T ratio in all datasets where $\tau = \{2, 3, 4, 5, 6\}$ and in two out of three datasets where $\tau = 1$. Therefore, **GH** provides the best trade-off between efficiency and effectiveness compared to all the baselines.

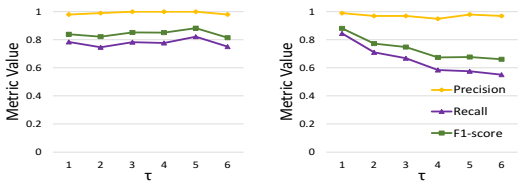


(a) T_{total} vs. $|D|$ ($\tau = 1$) (b) T_{total} vs. $|D|$ ($\tau = 2$) (c) T_{total} vs. $|D|$ ($\tau = 3$)

Figure 5: Average query time of GH and BSS_GED on two million-scale synthetic datasets, ER and BA

To demonstrate the scalability **GH**, we test it on synthetic datasets with 100M, 200M, 300M, 400M and 500M graphs and $\tau = \{1, 2, 3\}$. As neither **ML-Index** nor **Inves** can complete the test in two days, we only compare **GH** with **BSS_GED**. Figure 5 gives the average query time of two methods. Compared to **BSS_GED**, our approach is on average 20× faster, showing that **GH** is an efficient tool for graph similarity search on million-scale databases.

We also report the precision and recall of **GH** on two synthetic datasets with 5 million graphs where $\tau = \{1, 2, 3, 4, 5, 6\}$ in Figure 6. We observe that the average F1-score of our approach is 0.746 on the ER dataset and 0.854 on the BA dataset. In addition, Table 5 shows the recall/time ratio of **GH** and **BSS_GED** on two synthetic datasets where $\tau = \{1, 2, 3\}$. The ratio of **GH** is significantly higher than the ratio of **BSS_GED**, proving that our approach is not only efficient but also achieves high effectiveness on million-scale databases.

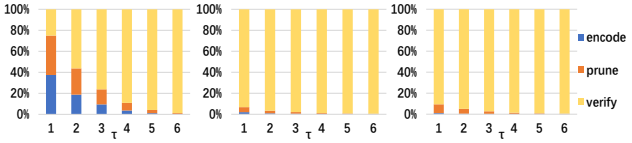


(a) (BA) (b) (ER)

Figure 6: Precion, recall and F1-score of GHashing on two synthetic datasets, ER and BA

Table 5: Recall/time ratio ($\times 10^{-2}$) for GHashing and BSS_GED on synthetic datasets with 5 million graphs

τ	BA		ER	
	GH	BSS_GED	GH	BSS_GED
1	23.3	2.7	110.6	2.5
2	9.9	2.6	37.6	2.4
3	5.8	2.4	15.9	2.2



(a) AIDS (b) LINUX (c) ALCHEMY

Figure 7: Running time of different stages for GHashing

Figure 7 shows the percentage of time of different stages to the total time for **GH**. The encoding time is the time to compute the query graph's embedding and hash code. It's constant *w.r.t* τ and only takes up a small fraction of query time. The pruning time is the time for the first level and second level pruning. The absolute value of pruning time slightly increases with τ, while its percentage keeps dropping. This is because the verification time increases rapidly with τ, and takes up most of the query time, as shown in Figure 7. So the bottleneck of **GH** is still the verification stage.

5.4 Ablation Studies of GHashing

Table 6: Recall/$|C|$ ratio ($\times 10^{-5}$) for GH- against 3 alternatives of our approach on AIDS dataset

τ	GH-	NoExp	NoEmb	NoAug
1	**30**	27	8.8	23
2	**19**	16	6.7	15
3	**15**	11	5.5	12
4	**11**	8.5	4.8	9.1
5	**8.8**	6.8	4.1	7.4
6	**7.1**	5.6	3.7	6.2

To investigate if the techniques proposed in Section 4 help to learn better hash codes, we conduct several ablation studies on AIDS dataset. In this series of experiments, we exclude second-level pruning so that the results are only determined by hash codes.

There are three alternatives.

(1) **NoExp**. We replace the exponential-weighted L2 loss with ordinary L2 loss to evaluate the effect of our proposed loss function;
(2) **NoEmb**. We train a hash function with λ in (1) equal to 1 to test if learning extra continuous embeddings helps to learn better hash codes;
(3) **NoAug**. We train the hash function without data augmentation to test if generating similar graph pairs helps the learning process.

We report the recall/$|C|$ ratio of each method in Table 6. We observe that **GH-** achieves the highest ratio where $\tau = \{1, 2, 3, 4, 5, 6\}$. It suggests that **GH-** has the best effectiveness and thus proves our techniques useful. Besides, the gap between **GH-** and **NoEmb** is most significant, suggesting the importance of learning continuous embeddings that preserves proximity between graphs.

6 CONCLUSION AND FUTURE WORK

In this work, we propose **GHashing**, a scalable GNN-based supervised hashing for graph similarity search, which exploits both the learning ability of DNN and the efficiency of hashing methods for approximate nearest neighbors. We extend the power of semantic hashing so that it can predict distances between graphs instead of only binary labels. We describe an algorithm to use it as a pruning method for graph similarity search, which is more efficient and scalable than previous approaches at the cost of missing a small number of correct results. The experimental results show that **GHashing** is on average 20× faster than the fastest baseline on million-scale databases while maintaining a high recall. In practice, **GHashing** strikes excellent trade-off between speed and accuracy and thus can be the ideal choice when approximate retrieval is acceptable.

One topic for future work is to revise it for different similarity metrics. Given the powerful ability of GNN and the generality of GED, **GHashing** has the potentials to retrieve similar graphs under different metrics and even the ideal similar graphs. Another topic is to explore how to use **GHashing** for similarity search with large graphs. As computing GED between large graphs is slow, how to efficiently compute the training labels becomes a problem. One possible solution is to compute approximate GED instead, but it will affect the quality of training data. Last but not least, it is important to improve the accuracy of predicting GED by hash codes and embeddings. So far, the candidate sets of **GHashing** usually contain many false positive results, which makes verification necessary. But as verification is slow, reducing the false positives in candidate sets so that we could exclude the verification stage would be ideal.

Acknowledgements This work is partially supported by NSF III-1705169, NSF CAREER Award 1741634, NSF 1937599, DARPA HR00112090027, Okawa Foundation Grant, and Amazon Research Award.

REFERENCES

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. In *WSDM*.
[2] Yunsheng Bai, Hao Ding, Ken Gu, , Yizhou Sun, and Wei Wang. 2020. Learning-based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching. *AAAI* (2020).
[3] Yunsheng Bai, Hao Ding, Yang Qiao, Agustin Marinovic, Ken Gu, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Unsupervised Inductive Graph-Level Representation Learning via Graph-Graph Proximity. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. 1988–1994. https://doi.org/10.24963/ijcai.2019/275
[4] Albert-Laszlo Barabasi and Reka Albert. 1999. Albert, R.: Emergence of Scaling in Random Networks. Science 286, 509-512. *Science (New York, N.Y.)* 286 (11 1999), 509–12. https://doi.org/10.1126/science.286.5439.509
[5] Horst Bunke. 1983. What is the distance between graphs. *Bulletin of the EATCS* 20 (1983), 35–39.
[6] Horst Bunke and Kim Shearer. 1998. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters* 19, 3-4 (1998), 255–259.
[7] Guangyong Chen, Pengfei Chen, Chang-Yu Hsieh, Chee-Kong Lee, Benben Liao, Renjie Liao, Weiwen Liu, Jiezhong Qiu, Qiming Sun, Jie Tang, Richard Zemel, and Shengyu Zhang. 2019. Alchemy: A Quantum Chemistry Dataset for Benchmarking AI Models. *arXiv preprint arXiv:1906.09427* (2019).
[8] Xiaoyang Chen, Hongwei Huo, Jun Huan, and Jeffrey Scott Vitter. 2019. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems* 163 (2019), 762–775.
[9] Pàl Erdös and A. Rényi. 1959. On random graphs I. *Publ. Math. Debrecen* 6 (01 1959), 290–297.
[10] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 1263–1272. http://proceedings.mlr.press/v70/gilmer17a.html
[11] Jongik Kim, Dong-Hoon Choi, and Chen Li. 2019. Inves: Incremental Partitioning-Based Verification for Graph Similarity Search. In *EDBT*.
[12] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SJU4ayYgl
[13] H. Lai, Y. Pan, Ye Liu, and S. Yan. 2015. Simultaneous feature learning and hash coding with deep neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 3270–3278. https://doi.org/10.1109/CVPR.2015.7298947
[14] Yujia Li, Chengcun Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *ICML*.
[15] Zijian Li, Xun Jian, Xiang Lian, and Lei Chen. 2017. An Efficient Probabilistic Approach for Graph Similarity Search. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2017), 533–544.
[16] Yongjiang Liang and Peixiang Zhao. 2017. Similarity Search in Graph Databases: A Multi-Layered Indexing Approach. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 783–794. https://doi.org/10.1109/ICDE.2017.129
[17] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2019. Deep Supervised Hashing for Fast Image Retrieval. *International Journal of Computer Vision* (03 2019). https://doi.org/10.1007/s11263-019-01174-4
[18] Kaspar Riesen and Horst Bunke. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing* 27 (06 2009), 950–959. https://doi.org/10.1016/j.imavis.2008.04.004
[19] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. 2007. Speeding Up Graph Edit Distance Computation with a Bipartite Heuristic. In *Mining and Learning with Graphs, MLG 2007, Firence, Italy, August 1-3, 2007, Proceedings*.
[20] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. 2015. Approximate Graph Edit Distance in Quadratic Time. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* PP (09 2015), 1–1. https://doi.org/10.1109/TCBB.2015.2478463
[21] Esko Ukkonen. 1992. Approximate string-matching with Q-grams and maximal matches. *Theoretical Computer Science* 92 (01 1992), 191–211. https://doi.org/10.1016/0304-3975(92)90143-4
[22] Guangtao Wang, Baoezeng Wang, Xiaochun Yang, and Lei Yu. 2012. Efficiently Indexing Large Sparse Graphs for Similarity Search. *IEEE Trans. Knowl. Data Eng.* 24 (03 2012), 440–451. https://doi.org/10.1109/TKDE.2010.28
[23] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to Hash for Indexing Big Data - A Survey. *Proc. IEEE* 104, 1 (2016), 34–57. https://doi.org/10.1109/JPROC.2015.2487976
[24] Xiaoli Wang, Xiaofeng Ding, Anthony Tung, Shanshan Ying, and Hai Jin. 2012. An Efficient Graph Indexing Method. *Proceedings - International Conference on Data Engineering* (04 2012). https://doi.org/10.1109/ICDE.2012.28
[25] Ye Zhang, Md. Mustafizur Rahman, Alex Braylan, Brandon Dang, Heng-Lu Chang, Henna Kim, Quinten McNamara, Aaron Angert, Edward Banner, Vivek Khetan, Tyler McDonnell, An Thanh Nguyen, Dan Xu, Byron C. Wallace, and Matthew Lease. 2016. Neural Information Retrieval: A Literature Review. *CoRR* abs/1611.06792 (2016). arXiv:1611.06792 http://arxiv.org/abs/1611.06792
[26] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. 2013. A partition-based approach to structure similarity search. *Proceedings of the VLDB Endowment* 7 (11 2013), 169–180. https://doi.org/10.14778/2732232.2732236
[27] Xiang Zhao, Chuan Xiao, Xuemin Lin, Wei Wang, and Yoshiharu Ishikawa. 2013. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal* 22 (12 2013). https://doi.org/10.1007/s00778-013-0306-1

A INDEX CONSTRUCTION

Algorithm 1 Build Hash Index I

Input: GNN hash function H, GNN embedding function F, graph database D
Output: inverted index I

1: Initialize an empty index I
2: **for** $g \in D$ **do**
3: $emb \leftarrow F(g)$
4: $code \leftarrow H(g)$
5: **if** $code \notin I.keys$ **then**
6: Add $code$ into $I.keys$
7: $I[code] \leftarrow \emptyset$
8: $I[code] \leftarrow I[code] \bigcup \{(g, emb)\}$
9: **return** I

The pseudocode of building the index with trained GNN is illustrated in Algorithm 1. The index consists of key-value pairs, where the key is the hash code and the value is a set of graph-embedding pairs. At first, the index is empty, which means its key set is empty (line 1). For each data graph g, we first compute its embedding and hash code (line 3-4). Then, we initialize an empty set if the hash code wasn't in I's key set (line 5-7). Then we add the graph, as well as its embedding into the value set $I[code]$ (line 8). In our implementation, we store the values consecutively and we record the position where a key's value starts and ends in an array sorted by the lexicographical order of keys. In this way, given a hash code, it takes $O(\log |I.keys|)$ time to find where its value is stored and $O(1)$ time to retrieve the values. Notice that the number of different hash codes, i.e. $|I.keys|$, can't be larger than $|D|$. In fact in our experiments the number of different hash codes is far less than $|D|$.

B SIMILAR GRAPH PAIRS GENERATION

Algorithm 2 illustrates how to generate synthetic simiar graph pairs for training. There are two inputs, a graph g sampled from the database and a parameter M, which is the upper bound of GED between synthetic graph pairs. Initially, it makes a copy of g to be g'. Then it randomly samples an integer n, which is the number of edit operations it is going to take (line 2). Next it randomly chooses n valid operations step by step and performs them on g' (line 3-6). Finally, we return the synthetic similar graph pairs and n as an approximation of $ged(g, g')$. Although n is an upper bound for the real GED, they should be close when M is small. In our experiments, we found that the value of M had little impact on the algorithm as long as it is small (≤ 3). So we set $M = 1$ for efficiency.

Algorithm 2 Generate Synthetic Similar Graph Pairs

Input: graph g, M
Output: graph pair (g, g') and an upper bound $ged(g, g')$

1: $g' \leftarrow g$
2: $n \leftarrow$ **Random**$(1, M)$
3: **for** $i = 1 \rightarrow n$ **do**
4: Randomly select a valid edit operation
5: Perform the edit operation on g'
6: **return** (g, g'), n

C QUERY WITH GHASHING

Algorithm 3 gives the details about the online stage of **GHashing**.

Algorithm 3 Graph Similarity Search

Input: query graph q, threshold τ, embedding function F, hash function H, hash index I
Output: $\{g \in D | ged(g, q) \leq \tau\}$

1: // `Pruning Stage`
2: $c_q \leftarrow H(q)$
3: $e_q \leftarrow F(q)$
4: $C \leftarrow \emptyset$
5: $S \leftarrow \{code \in I.keys | \|code - c_q\|_{\mathcal{H}} \leq \tau + 1\}$
6: **for** $code \in S$ **do**
7: **for** $(g, e) \in I[code]$ **do**
8: **if** $\|e - e_q\|_2^2 \leq \tau + 0.5$ **then**
9: $C \leftarrow \{g\} \bigcup C$
10: // `Verification Stage`
11: $ret \leftarrow \emptyset$
12: **for** $g \in C$ **do**
13: **if** $ged(g, q) \leq \tau$ **then**
14: $ret \leftarrow \{g\} \bigcup ret$
15: **return** (ret)

Given a query graph q, we first compute the hash code and continuous embedding for q (line 2-3). Then we search the keys of the index to find hash codes whose hamming distance to c_q is smaller than $\tau + 1$ (line 5). Notice that for S in line 5 of Algorithm 3) we have

$$|S| \leq \min\left(|I.keys|, \sum_{i=0}^{\tau+1} \binom{B}{i}\right) \tag{6}$$

where B is the length of hash codes. The inequality above suggests that when $\tau+1$ is small, it is efficient to just enumerate all the binary codes whose hamming distance to $H(q)$ is within the threshold and check if they are in $I.keys$. However, if $\tau+1$ is large, $\sum_{i=0}^{\tau+1} \binom{B}{i}$ might be too large and it would be more efficient to scan every key of the hash index and compute its hamming distance to $H(q)$. Therefore, to obtain S, we first compute if $|I.keys| > \sum_{i=0}^{\tau+1} \binom{B}{i}$. If so, we would use breadth first search to enumerate all valid codes. Otherwise, we simply scan every hash keys of index. Once S is obtained, we retrieve all candidates graphs as well as their embedding from the index. Then we use the continuous embedding of the candidates to conduct a secondary pruning step (line 6-9). After getting the final candidates, we verify every graph remained with exact verification algorithm(line 10-14). In our implementation, we adopt **BSS_GED** as the verification algorithm.

D GRAPH CONVOLUTION LAYER AND GRAPH POOLING LAYER

According to [10], most of GNNs can be expressed in the following form.

$$m_v^{(l+1)} = \sum_{w \in N(v)} M_l(h_v^{(l)}, h_w^{(l)}, e_{vw}) \tag{7}$$

$$h_v^{(l+1)} = U_{(l)}(h_v^{(l)}, m_v^{(l+1)}) \tag{8}$$

$$h_g^{(l)} = R(\{h_v^{(l)} | v \in g\}) \tag{9}$$

Here $N(v)$ is v's neighbors, so $m_v^{(l+1)}$ is the message passing from v's neighbors to v at layer $l+1$. $h_v^{(l+1)}$ is the hidden representation of v at layer $l+1$ and it is determined by v's hidden representation at layer l as well as $m_v^{(l+1)}$. $h_g^{(l)}$ is the hidden representation of graph g at layer l which is computed by aggregating the vertex representations at layer l together via reduction function $R(\cdot)$. To sum up, (7), (8) define a convolution layer and (9) defines a pooling layer.

The convolution layer used in our implementation is based on graph convolutional network (GCN) [12]. Following the form given above, GCN can be expressed as $m_v^{(l+1)} = \sum_{w \in N(v)} W^{(l)} h_w^{(l)}$ and $h_v^{(l+1)} = \frac{1}{|N(v)|+1}(h_v^{(l)} + m_v^{(l+1)})$, where $W^{(l)}$ is the parameters of layer l.

And We adopt the attention mechanism proposed in [3] as the graph pooling layer, which is

$$h_g^{(l)} = \sum_{n=1}^{|V|} \sigma\left((h_v^{(l)})^T \mathrm{ReLU}(W_l^a(\frac{1}{|V|}\sum_{u \in V} h_u^{(l)}))\right) h_v^{(l)} \tag{10}$$

where $\sigma(\cdot)$ is any activation function, W_l^a is trainable parameters and $ReLU(X) = \max(X, 0)$.

E PROOF OF THEOREM 1

It is obvious that (3) is minimized when $\hat{y}' = y'$. To prove its convexity, we compute its Hessian matrix, which is

$$e^{a(\hat{y}'-y')}\{[a(\hat{y}'-y')+2]^2-2\}\begin{pmatrix}1 & -1\\ -1 & 1\end{pmatrix}$$

Since $\begin{pmatrix}1 & -1\\ -1 & 1\end{pmatrix}$ is positive semi-definite, the Hessian matrix is positive semi-definite if and only if $e^{a(\hat{y}'-y')}\{[a(\hat{y}'-y')+2]^2-2\} \geq 0$. Notice that

$$e^{a(\hat{y}'-y')}\{[a(\hat{y}'-y')+2]^2-2\} \geq 0 \Longleftrightarrow [a(\hat{y}'-y')+2]^2-2 \geq 0$$

And

$$[a(\hat{y}'-y')+2]^2-2 < 0 \Longleftrightarrow a(\hat{y}'-y')+2 \in (-\sqrt{2}, \sqrt{2})$$

which means

$$\hat{y}'-y' \in (\frac{-\sqrt{2}-2}{a}, \frac{\sqrt{2}-2}{a})$$

Since $|\hat{y}'-y'| \leq M$, if $-M > \frac{\sqrt{2}-2}{a}$, then $\hat{y}'-y'$ is always larger than $\frac{\sqrt{2}-2}{a}$, which means $e^{a(\hat{y}'-y')}\{[a(\hat{y}'-y')+2]^2-2\} \geq 0$ and the Hessian matrix is positive semi-definite. Therefore, the loss is convex, if $a < \frac{2-\sqrt{2}}{M}$.

F DATASETS

Some important statistics of the datasets are illustrated in Table 7. Besides, we illustrate our procedure to generate two synthetic datasets.

ER dataset. Generating a graph under ER model requires two parameters: the number of nodes (n) and the probability for edge creation (p). When generating a graph, we sample n from a poisson distribution with expectation equal to 10 and the constraint that $0 < n < 30$. The upper bound for n is to prevent that it takes too long to compute the ground truth, since exact GED computation is NP-hard. And p is sample from a uniform distribution $U(0.3, 0.1)$. After generating an unlabeled graph with ER model, we sample the label for each node under a multinomial distribution $(0.6 : 0.2 : 0.1 : 0.05 : 0.05)$. The multinomial distribution is intended to mimic the common skewness of label's distribution in real world.

BA dataset. Generating a graph under BA model requires two parameters: the number of nodes (n) and the number of edges to attach from a new node to existing nodes (m), with the constraint that $1 \leq m < n$. We sample n in the same way as generating graphs under ER model, and we sample m from 1 to $n-1$ with equal probability. After generating an unlabeled graph with BA model, we sample the label for each node under a multinomial distribution $(0.8 : 0.1 : 0.05 : 0.025 : 0.025)$, with the intention to mimic the common skewness of distributions of labels in real world.

Table 7: Statistics of datasets

Dataset	$\|D\|$	ave $\|V\|$	ave $\|E\|$	$\|L_v\|$	$\|L_e\|$
AIDS	42, 687	25	55	62	1
LINUX	38, 661	16	34	1	1
ALCHEMY	119, 487	21	44	7	1
ER	5, 000, 000	10	66	5	1
BA	5, 000, 000	10	40	5	1

G EXPERIMENT CONFIGURATION

We implement all the baselines and the majority of our algorithm in c++, except the neural network of our approach is implemented in python with tensorflow[2]. All our experiments are carried out on a linux server with eighty 2.20GHz Intel(R) Xeon(R) CPUs, 528 GB main memory and eight Nvidia Tesla v100 GPUs.

In our experiments, hyper-parameters are not carefully chosen for any particular datasets. We use one-hot encoding scheme to get the initial node representations. The output dimensions for the 1st, 2nd and 3rd layer of GCN are 256, 128, 64, respectively. After getting the graph representation from the graph attention pooling layer, we use two fully connected layers to get the continuous embeddings of the graphs as shown in Figure 1 and the output dimensions of both layers are 256. Then we use another two fully connected layers to get the hash codes. The output dimensions are 128 and 32 respectively. One thing to note is that the output dimension of the last layer is also the length of the hash codes (32 by default, unless otherwise noted). All the layers except the last one use ReLU as activation function and the last one uses tanh as activation function so that the output is within $[-1, 1]$.

We set the slope of exponential-weighted L2 loss, i.e. a in (3), to 0.05, the clipping threshold γ to 11, the weight for hash code's loss, i.e. λ in (1), to $\frac{1}{11}$ and the weight for binary regularizer, i.e. β in (4), to 0.2. Besides, $M = 1$, $k = 1$ when generating synthetic graphs.

For each dataset, we train our model with at most 15000 iterations and stop training when we detect the training loss stop decreasing. Specifically, for every 50 iterations, we compute the average training loss of last 50 iterations (l_{50}) and of last 500 iterations (l_{500}). If $l_{500} - l_{50} < 0.1$, then we stop the training.

[2]The code is available on github https://github.com/ZongyueQin/Graph-Hashing