GE^2: A General and Efficient Knowledge Graph Embedding Learning System (Technical Report)

Anonymous Author(s)

ABSTRACT

Graph embedding learning computes an embedding vector for each node in a graph and finds many applications in areas such as social networks, e-commerce, and medicine. We observe that existing graph embedding systems (e.g., PBG, DGL-KE, and Marius) have long CPU time and high CPU-GPU communication overhead, especially when using multiple GPUs. Moreover, it is cumbersome to implement negative sampling algorithms on them, which have many variants and are crucial for model quality. We propose a new system called GE^2, which achieves both generality and efficiency for graph embedding learning. In particular, we propose a general execution model that encompasses various negative sampling algorithms. Based on the execution model, we design a user-friendly API that allows users to easily express negative sampling algorithms. To support efficient training, we offload operations from CPU to GPU to enjoy high parallelism and reduce CPU time. We also design COVER, which, to our knowledge, is the first algorithm to manage data swap between CPU and multiple GPUs for small communication costs. Extensive experimental results show that, comparing with the state-of-the-art graph embedding systems, GE^2 trains consistently faster across different models and datasets, where the speedup is usually over 2x and can be up to 7.5x. GE^2 is open-source at https://anonymous.4open.science/r/gege-9030.

KEYWORDS

Graph processing, graph embedding, machine learning

ACM Reference Format:

Anonymous Author(s). 2018. GE^2: A General and Efficient Knowledge Graph Embedding Learning System (Technical Report). In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Graph data are ubiquitous in many areas such as social networks [55], e-commerce [43, 56], finance [3], and medicine [29, 64]. *Graph embedding learning* computes an embedding vector for each node in a data graph such that similar nodes (e.g., adjacent in the graph or of similar type/role) have similar embeddings, and it is one of the most popularly adopted graph machine learning techniques in industry, for example, in applications such as community detection [55], e-commerce recommendation [43, 50], fraud detection [60], and drug discovery [29, 64]. Graph embedding learning has been extensively studied and many algorithms have been proposed, e.g., DeepWalk [37], Node2Vec [11], LINE [44], and SDNE [48]. A number of systems have also been developed to train graph embedding models, e.g., DGL-KE from Amazon Web Service [68], PyTorch Big Graph (PBG) from Meta [24], and Marius [33], but these systems suffer from two crucial limitations.

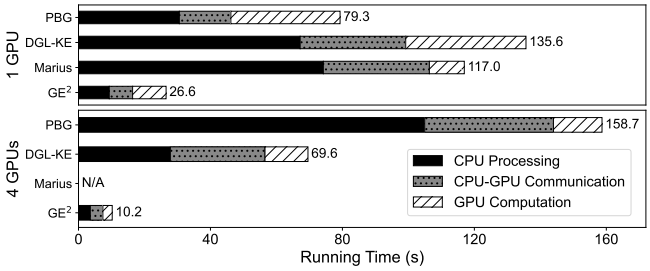


Figure 1: Running time decomposition for existing systems and GE^2 on the *Livejournal* graph and *Dot* model. Disk IO time is excluded for PBG, and Marius only allows 1 GPU.

Limited support for negative sampling algorithms. Graph embedding follows the contrastive learning paradigm and pairs real edges in a graph with fake edges to train embeddings. *Negative sampling* decides how to generate these fake edges and is crucial for the quality of node embeddings [61]. There are many negative sampling algorithms with diverse patterns. For example, RNS [40] conducts random sampling, DNS [40] uses node embeddings to calculate sampling probability, and KBGAN [2] trains a specialized model for sampling. Our experiments in Section 6 show that different negative sampling algorithms can yield significantly different embedding quality (usually measured by Mean Reciprocal Rank, MRR). However, the implementations of existing systems are tightly coupled with relatively simple negative sampling algorithms such as RNS, and writing new negative sampling algorithms takes substantial efforts. For instance, it takes us about 500 lines of code (LoC) to implement KBGAN on DGL-KE and 400 LoC to implement DNS on Marius. This hinders users from developing and using advanced negative sampling algorithms and thus limits the effectiveness of graph embedding models.

Poor efficiency for model training. The embeddings are usually large (e.g., millions of nodes and one high-dimension vector for each node) and do not fit in GPU memory. Thus, they are kept primarily on CPU memory and swapped between CPU and GPU to conduct training. In Figure 1, we decompose the running time of existing systems into 3 parts, i.e., *CPU processing*, *CPU-GPU communication*, and *GPU computation*. The results show that these systems

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

suffer from heavy CPU operations and high CPU-GPU communication overhead. In particular, both DGL-KE and Marius transfer each batch of training edges and the involved node embeddings to GPU for computation, and then write the gradients back to CPU for updates. As gradient computation is lightweight on GPU for graph embedding models, the edge batching and embedding update operations on CPU become dominant. PBG loads the partitions of node embeddings to GPU memory and reuses the embeddings to train multiple batches. Thus, PBG has smaller CPU and communication costs than DGL-KE and Marius when using 1 GPU. However, when using 4 GPUs, the CPU and communication costs of PBG increase rapidly because PBG conducts more fine-grained CPU-GPU communication to parallelize among multiple GPUs.

To tackle the generality and efficiency problems of existing graph embedding systems, we design a system called GE^2. To support better generality, we conduct an extensive survey of negative sampling algorithms and classify them into three categories, i.e., *static*, *dynamic*, and *adversarial*. On this basis, we propose a general execution model that encompasses all these algorithms. In particular, the execution model involves 3 steps, i.e., *select* to generate some initial candidates, *compute* to calculate the sampling bias of the candidates, and *sample* to choose the candidates according to their bias. Based on the execution model, we design a API to facilitate users to easily express different negative sampling algorithms (e.g., with around 10 lines of code). We also decouple system implementation from negative sampling algorithms for good generality.

To support higher efficiency, we adopt the partition-based training scheme of PBG, which loads the partitions of node embeddings to GPUs and reuses them across multiple batches. As multi-GPU servers become common, we consider scheduling node partition swapping between the CPU and multiple GPUs, which we call the partition scheduling problem (PSP). The problem is challenging because GPUs should exhibit no data conflicts for parallelization, have balanced workload to avoid stragglers, and pay small CPU-GPU communication costs for efficiency. To design a solution for PSP, we transform it into the *resolvable balanced incomplete block design (RBIBD) problem* [20]. Moreover, for a case that is special for the RBIBD problem but general enough for parallel training, we design a simple yet effective algorithm named *COVER* to solve PSP. Compared with the partition swapping strategies of existing systems, COVER has significantly smaller CPU-GPU communication cost, which is crucial for GE^2 to achieve a short training time.

We conduct comprehensive experiments to assess the performance of GE^2 comparing with the state-of-the-art graph embedding systems such as DGL-KE [68], Marius [33], and PBG [24]. The results demonstrate that GE^2 effortlessly supports various negative sampling algorithms and consistently outperforms the baseline systems across diverse datasets and graph embedding models. The speedup obtained by GE^2 over the baselines is up to 7.5x and over 2x in the majority of cases. GE^2 also achieves higher scalability than the baselines when using multiple GPUs. Our micro experiments also further affirm the effectiveness of the GE^2 designs.

To summarize, we make the following contributions.

- We conduct an extensive survey of negative sampling algorithms and design a general execution model along with a user-friendly API for their easy expression (Section 4).
- We define the partition scheduling problem for efficient parallel training with multiple GPUs and connect it with the classical RBIBD problem to design a solution (Section 5).
- We conduct comprehensive evaluations of GE^2 with on a single GPU and multiple GPUs, demonstrating its efficiency compared with existing embedding learning systems (Section 6).
- We open-source GE^2, which may benefit research in this community: https://anonymous.4open.science/r/gege-9030.

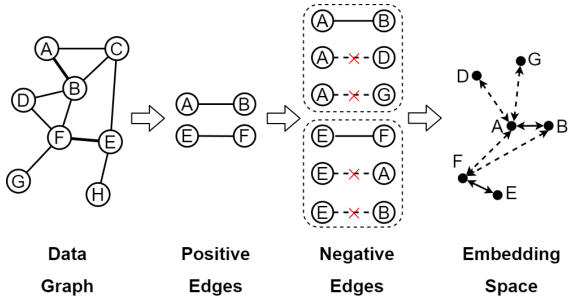


Figure 2: An illustration of graph embedding learning. The batch contains two positive edges, and $s=2$ negative edges are sampled for each positive edge. Training pushes the nodes connected by solid lines towards each other and the nodes connected by dotted lines apart from each other.

2 BACKGROUND

In this section, we introduce the background of graph embedding to provide a foundation for our subsequent discussions.

In general, graph embedding deals with a data graph in the form of $G = (V, R, E)$, where V, R, and E denote the sets of nodes, edge (i.e., relation) types, and edges, respectively. An edge $e \in E$ is a triplet (u, r, v), signifying that source node u and destination node v are linked by relation type r. For example, in an e-commerce user-product graph, each node represents either a user or a product, and an edge may indicate user actions such as viewing, clicking, or purchasing a product. Graph embedding learning aims to learn an embedding vector θ_v for every node $v \in V$, as well as θ_r (or matrix M_r) for each relation $r \in R$. Typically, the number of relationship types is significantly smaller than the number of nodes, and thus node embedding constitutes the major part of model parameters. Note that some graphs may lack explicit relations and can be treated as having only one relation type.

Graph embedding employs a *score function* denoted as $f(\theta_u, \theta_r, \theta_v)$ to quantify the likelihood of the existence of an edge $e = (u, r, v)$ in the data graph, and we also use $f(e)$ for conscienceless. The function takes various forms, such as $\theta_u^\top \theta_v$, $\theta_u^\top \mathrm{diag}(\theta_r)\theta_v$, or $\theta_u^\top M_r \theta_v$, depending on the specific graph embedding model [1, 26, 34, 41, 45, 58]. Based on their designs, these score functions can encode various similarity semantics, e.g., encouraging nodes adjacent in the graph topology or nodes with similar roles (e.g., company managers) to have similar embeddings. Model training aims to increase the score $f(e)$ for *positive edges* $e \in E$ and decrease the score $f(e')$

for *negative edges* $e' \notin E$. This is achieved by minimizing the following *contrastive loss function*:

$$\mathcal{L} = \sum_{e \in E} (-f(\theta_e) + \log(\sum_{e' \notin E} \exp(f(\theta_{e'})))). \quad (1)$$

Training is typically conducted in mini-batches, with each batch $\mathcal{B}$ containing b positive edges. *Negative sampling* is employed to generate s negative edges for each positive edge $e = (u, r, v)$. This process involves replacing the destination node v (or, alternatively, the source node u) with fictitious nodes, sampled according to specific probabilities. We assume that the destination node is replaced, and the discussions apply to source node replacement. In each batch, the relevant node embeddings are updated using stochastic gradient descent. Training is said to have finished an *epoch* when all edges in the train data graph are used once as positive edges.

Figure 2 provides an illustrative example of graph embedding learning. For this batch, two positive edges, (A, B) and (E, F), are used. Two negative edges are generated for each positive edge, e.g., (A, D) and (A, G) correspond to positive edge (A, B) and replace the destination node B. This batch involves a total of six node embeddings, i.e., $\{\theta_A, \theta_B, \theta_D, \theta_E, \theta_F, \theta_G\}$, and updates them.

By generating negative edges for training, negative sampling exerts a substantial influence on the quality of the learned embeddings [57], which is typically assessed using Mean Reciprocal Rank (MRR) [33, 57] and hit ratio [21, 24, 33]. A plethora of negative sampling algorithms have been proposed, such as RNS [40], DegreeNS [68], DNS [65] and KBGAN [2], and designing novel negative sampling algorithms remains an active area of research [5, 6, 62]. Negative sampling is also widely used in the industry. For example, Pinterest utilizes the PinSage algorithm to improve recommendation quality, which adopts Personalized PageRank scores to acquire more challenging negative samples [63]. Alibaba designs new negative sampling algorithms to improve the performance of models like DeepWalk and GraphSAGE for e-commerce [61]. However, the implementations of existing graph embedding systems are closely intertwined with relatively simple negative sampling algorithms. For instance, the negative sampler of Marius can only access edges while some advanced algorithms (e.g., DNS and KBGAN) need to utilize node embeddings. We address this generality challenge by devising a user-friendly API that expresses a wide array of negative sampling algorithms and providing a unified execution engine that accommodates these algorithms.

Relation to graph neural networks (GNNs). GNN models iteratively aggregate the neighboring nodes and employ neural network mapping to compute outputs for each node [12, 19, 46]. GNNs are also widely used for graph tasks but they are largely orthogonal to graph embedding. In particular, GNNs are usually trained in a supervised manner with labels for the nodes and edges while generating embedding is trained in an unsupervised manner. Moreover, graph embedding complements GNNs because GNNs require a feature vector for each node and graph embedding can be used to generate these features vectors for GNN training.

3 GE^2 SYSTEM OVERVIEW

In this section, we provide an overview of our GE^2 system and introduce its procedure for training graph embedding models.

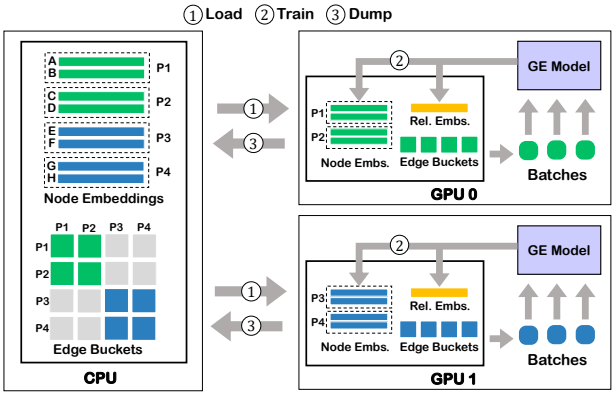


Figure 3: The workflow of the GE^2 system.

GE^2 considers a machine with sufficient CPU memory and one or multiple GPUs. The CPU memory holds all data, including graph topology, node embeddings, relation embeddings, and optimizer states (e.g., per embedding momentum for SGD variants), while the GPUs load their working sets from CPU memory to conduct training on demand. GE^2 allows users to customize their graph embedding models by specifying the score function and negative sampling algorithm using the API in Section 4. To conduct training, GE^2 adopts the partition-based scheme in PBG, which loads the node embedding to GPUs at partition granularity and reuses them for many batches. Different from PBG, GE^2 uses the COVER algorithm in Section 5 to schedule CPU-GPU data swap, which allows multiple GPUs to parallelize and achieves low CPU-GPU communication cost. In the following, we introduce the data layout and training workflow of GE^2.

Node partition and edge bucket. On CPU memory, GE^2 organizes the node embeddings into *node partitions* of equal sizes with a range partitioning on node ID. For instance, in Figure 3, the graph contains 8 nodes, which are organized into 4 partitions (i.e., in the left part of Figure 3). According to the node partitions, GE^2 organizes the graph topology into *edge buckets* with each edge bucket containing the edges from the source partition to the destination. In Figure 3, edge bucket (P_1, P_2) contains the edges from partition $P_1 = \{A, B\}$ to $P_2 = \{C, D\}$. This design allows the GPUs to process large graphs that do not fit in GPU memory and parallelize by handling different node partitions and edge buckets.

As illustrated in Figure 3, the training workflow of GE^2 consists of three steps, i.e., *load*, *train*, and *dump*.

Load. Each GPU loads some node partitions and the edge buckets formed by these partitions to prepare for training. For the example in Figure 3, GPU-0 loads partitions $\{P_1, P_2\}$ and the involved edge buckets are $(P_1, P_1), (P_1, P_2), (P_2, P_1), (P_2, P_2)$. In particular, GE^2 uses the COVER algorithm in Section 5 to assign a *buffer state* $\mathcal{S}_i = \{P_1^i, P_2^i, \cdots, P_q^i\}$ for each GPU, which contains q partitions and determines the data to load. The relation embeddings are loaded by all GPUs, and this is necessary because the edge buckets may contain all relation types. Such repetitive loading is not expensive because the relation embeddings are usually much smaller than the node embeddings. The optimizer states of the embeddings are also

Table 1: A summary of negative sampling algorithms, which sample a node v' to replace the destination node v for an edge $e = (u, r, v)$. *Candidate* prepares some nodes for bias computation, and *Bias* controls how these nodes are sampled.

Category	Candidate	Bias	Example Algorithm	Description
Static	No	No	RNS [40]	Sample nodes uniformly at random
		Degree	DegreeNS [68]	Sample nodes using their degrees as bias
		Frequency	UINS [35], Word2vec [32]	Sample nodes using an external bias
Dynamic	Neighbor	No	TUNS [42]	First select some nodes nearest to the source node in the embedding space and then sample uniformly
	Random	Score	DNS [65], MaxNS [38], SRNS [4]	Uniformly sample some nodes and then sample these candidates according to their scores for the source node
Adversarial	No	Model	GraphGAN [49], IRGAN [51], AdvIR [36]	Train another embedding (i.e., generator) for each node and sample based on generator scores for source node
	Random		KBGAN [2]	Uniformly sample nodes and then use the generator

loaded such that updates can be conducted on the GPUs. GE^2 uses separate threads to conduct parallel loading for different GPUs.

Train. Each GPU permutes the edges in its loaded edge buckets and goes over the edges with a batch size of b to conduct training. For each positive edge $e = (u, r, v)$, s negative edges are sampled, and we constrain that each GPU samples the fake destination nodes v' among the nodes in its buffer state $\mathcal{S}_i$. This ensures that the GPUs do not read node embeddings from CPU or peer GPU memory in the training step. Such a constraint limits the range of fake destinations but we observe that it does not affect model quality. This is because each GPU loads many node embeddings (e.g., one million), and sampling among these nodes already provides sufficient randomness. Similar constraints are also imposed by PBG and Marius. The COVER algorithm ensures that different GPUs handle separate node partitions, and thus the GPUs can update their node embeddings without synchronization. For the relation embeddings, the GPUs synchronize after every batch using an all-reduce operation because they may update the same relation embeddings.

Dump. After processing their respective edge buckets, the GPUs dump the node embeddings and optimizer states to CPU memory. As each GPU handles separate node partitions, there are no write-write conflicts, and thus locks are not required during dumping.

Multiple passes of the load, train, and dump steps may be required to go over all the edge buckets (i.e., an epoch). After the dump step of one pass, the load step of the next pass can start, with each GPU handling a buffer state (and thus edge buckets) that is different from the previous pass. The relation embeddings are only loaded in the first pass and dumped in the final pass because they are used in all passes. For diagonal edge buckets in the form of (P_j, P_j), GE^2 processes them the first time when partition P_j is loaded into GPU memory and ignores them when P_j is loaded later. To introduce more randomness, we reorganize the nodes into node partitions after each epoch. This is done by permuting the nodes to generate new node IDs, which is a very cheap operation.

We observe that the cost of synchronizing the relation embeddings in every mini-batch can be large, and thus we design GE^2 to allow users to configure a *delayed update* option to reduce this cost. Specifically, delayed update resembles the federated averaging algorithm [30], where each GPU updates its local relation embeddings without synchronization during the train step. After each pass of the load, train, and dump steps, the latest relation embeddings are computed as the average of the relation embeddings on all GPUs.

GE^2 has several advantages over existing systems. In particular, DGL-KE and Marius batch the edges and update the node embeddings on CPU, and conduct CPU-GPU communication for each batch, yielding long CPU processing time and high CPU-GPU communication cost. GE^2 reduces CPU time by offloading edge batching and negative sampling to the GPUs. The CPU-GPU communication cost is also reduced by only loading and dumping the node embeddings after processing some edge buckets, which are much larger than a batch of edges. PBG also adopts the partition-based training strategy but GE^2's COVER algorithm achieves lower CPU-GPU communication cost, especially when using multiple GPUs.

Currently, GE^2 assumes that all data fit in the CPU memory of a single machine, which can easily reach 512GB or 1TB nowadays and allows to handle reasonably large graphs. For example, assume that a machine has 512GB memory and each node embedding is a 128-dimensional float vector, single machine in-memory processing can handle a graph with 500 million nodes, which is sufficiently large for most applications. Moreover, cloud vendors provide machines with up to 4 TB of memory, which allow to handle even larger graphs (e.g., with 1 billion nodes). Similar to GE^2, systems such as DGL-KE [68], GraphVite [69], and HET [31] also consider graph embedding learning in the main memory. Extending GE^2 to disk-based and distributed training requires to consider disk-memory data swapping and data partitioning over the machines, which we leave for future work.

4 A GENERAL API FOR NEGATIVE SAMPLING

In this section, we first summarize existing negative sampling algorithms, then propose a general API for implementing these algorithms by identifying their common computation patterns, and finally show how to use the API to express some representative algorithms.

4.1 Negative Sampling Algorithms

For each positive edge $e = (u, r, v)$, negative sampling chooses a fake destination node v' to replace v according to a sampling

distribution $P(v'|u,r)$ (i.e., bias). There are many negative sampling algorithms, and we summarize 12 representatives in Table 1. In particular, we classify the algorithms into three categories based on how the sampling distribution is computed, i.e., *static*, *dynamic*, and *adversarial*.

Static algorithms. For these algorithms, the sampling bias does not change during training. For instance, RNS conducts unbiased sampling and chooses the nodes uniformly at random [40], and DegreeNS uses node degree as the sampling bias [68]. Some algorithms sample the nodes according to their external frequency outside the data graph. For example, UINS considers recommendation and uses the view count of user/item as bias [35], and Word2vec targets natural language and uses word appearance count as bias [32]. For static algorithms, *shared sampling* is a popular technique to improve efficiency, which organizes a batch of positive edges into g groups (with g much smaller than batch size b) and shares the negative destinations in each group. This reduces both the negative destinations to sample and the node embeddings to update.

Dynamic algorithms. These algorithms compute sampling bias using the node embeddings, which change dynamically during training. For example, TUNS first identifies K nodes that are the closest to the source node in the embedding space and then samples these nodes uniformly at random [42]. To avoid computing the distance/score for all nodes, some algorithms first sample a subset of the nodes uniformly at random and then compute the scores $f(\theta_u, \theta_r, \theta_{v'})$ of these candidates for the source node. They sample the candidates in different ways, e.g., MaxNS [38] chooses candidates with the largest scores while DNS [65] and SRNS [4] sample the candidates with probability proportional to their scores.

Adversarial algorithms. Inspired by generative adversarial networks (GANs) [10], these algorithms train another embedding $\tilde{\theta}_v$ for each node v to conduct sampling. This sampling-oriented model is called the *generator*, while the graph embedding model is called the *discriminator*. The idea is that a specialized generator can learn to produce high-quality negative edges for training. GraphGAN [49], IRGAN [51], and AdvIR [36] sample the destination node v' according to the scores $\tilde{f}(\tilde{\theta}_u, \tilde{\theta}_r, \tilde{\theta}_{v'})$ given by the generator. KBGAN [2] avoids computing the scores for all nodes by uniformly sampling some nodes before applying the generator model.

We observe that negative sampling algorithms are becoming increasingly diverse and complex. For instance, the static algorithms are proposed the earliest but later the dynamic and adversarial algorithms become more compute-intensive. There are also hybrid and more complex algorithms. For example, MCNS [61] selects both uniformly sampled nodes and nearest neighbors as candidates and then samples these candidates using the Metropolis-Hastings algorithm. Rather than sampling existing nodes, MixGCF [16] synthesizes hard negatives by mixing the embeddings of uniformly sampled candidates and their L-hop neighbors. Although simple negative sampling algorithms can improve accuracy by sampling more negatives for each positive edge, they usually yield lower model accuracy than complex algorithms [61], and using more negatives also increases training computation. As such, it is crucial to provide system support for the easy implementation of complex negative sampling algorithms. However, existing graph embedding systems focus on static negative sampling algorithms (e.g., RNS) and require considerable effort to implement other algorithms.

Algorithm 1: The SCS model for negative sampling algorithms

Input: edge batch $\mathcal{B}$, negative sample number s
Output: negative edges $\mathcal{M}$
for *each postive edge* $e \in \mathcal{B}$ **do**
  Get the candidate nodes $\mathcal{C}$ using SELECT(K);
  Calculate the sample bias $\mathcal{P}$ by COMPUTE($e, \mathcal{C}$);
  Sample s negative edges by SAMPLE($\mathcal{P}, s$) for $\mathcal{M}$;

```
// Core functions
Vector<Node> select(int K, int group_num = 0);
Bias compute(Edge e, Vector<Node> nodes);
Vector<Node> sample(Bias bias, int s);
// Auxiliary structures and functions
class Bias {
    Vector<Node> nodes, Vector<float> probs};
enum EmbeddingType {Generator, Discriminator};
Embedding getEmbeddings(Vector<Node> nodes,
                        EmbeddingType type);
Vector<float> score(Embedding emb,
                    Vector<Embedding> embs);
```

Figure 4: API for negative sampling.

4.2 Execution Model and API

We observe that negative sampling algorithms (e.g., those in Table 1) follow a general 3-step procedure with *select*, *compute*, and *sample*, which is summarized in Algorithm 1. In particular, for a positive edge e, the *select* step generates K candidate nodes, e.g., the random sampling in DNS [65] and KBGAN [2] or the nearest neighbors in TUNS [42]; the *compute* step calculates the sampling bias of the candidates for the positive edge, e.g., using the node embeddings in MaxNS [61] and SRNS [4] or another generator model in GraphGAN [49] and KBGAN [2]; the *sample* step chooses s destination nodes among the K candidates according to the bias, e.g., sampling those with the largest scores in MaxNS [61] or conducting probabilistic sampling in GraphGAN [49]. Some algorithms may skip certain steps, for instance, the static algorithms do not conduct the *select* step and *compute* step. Regarding Table 1, the select step corresponds to column *Candidate*, the compute step corresponds to column *Bias*, and the sample step finally decides the negative node.

Based on the above execution model for negative sampling, we design the C++ style API in Figure 4. The API has three core functions that correspond to the three steps of negative sampling. In particular, the `select` function accepts K as the number of candidates to generate and returns a list of node IDs. Users can control the number of groups in a batch with the `group_num` parameter for shared sampling (see Section 2), which is disabled if `group_num=0`. The `compute` function takes the positive edge e and the candidate nodes as input and returns `bias`, which is a list that contains the candidates and their probabilities to be sampled. The `sample` function accepts the bias and the number s of candidates to sample. We

provide two built-in options for `sample`, i.e., sampling the candidates with top scores (i.e., `TopSample`) and sampling the candidates by normalizing their bias into a distribution (i.e., `ProbSample`). Decomposing negative sampling into three steps yields clear algorithm logic and enables effective code reuses. For instance, several algorithms in Table 1 use uniform sampling for candidate generation and TopSample or ProbSample for node sampling, and we provide these methods in GE^2 to reduce user efforts. Although many algorithms use uniform sampling for candidate generation, we leave the select API for users to specify more sophisticated candidate generation methods, which is required for algorithms such as TUNS.

The API provides two auxiliary functions that are useful for dynamic and adversarial algorithms. `getEmbeddings` accepts a list of node IDs and returns the embeddings of these nodes produced by the generator model or the discriminator model. Users can also define multiple `score` functions that evaluate the score of an edge (e.g., for discriminator and generator), which will be used to compute the sampling bias. As GE^2 loads some node embedding partitions to each GPU to conduct training, the `select` and `getEmbeddings` functions only consider the local nodes on each GPU. Such a limit on the domain of negative sampling does not hinder model quality because each GPU holds many nodes and thus provides sufficient randomness for sampling [24]. Users can also utilize the operators and functions in LibTorch[1].

4.3 Use Cases of the API

Our execution model and API are relatively simple, in this part, we show that such simplicity leads to succinct expressions of negative sampling algorithms using RNS, DNS, and KBGAN as examples.

RNS randomly samples nodes and represents the static algorithms. As shown in Figure 5a, we bypass the `select` and `compute` functions and create a uniform bias (whose probabilities are all 1) for all nodes in the sub-graph (formed by the edge buckets loaded on one GPU). Then the `sample` function uniformly samples s nodes from the sub-graph.

DNS uses the scores of the candidate nodes as sampling bias and represents the dynamic algorithms. As shown in Figure 5b, the `select` function uniformly chooses K nodes from the sub-graph as candidates. The `compute` function uses the `getEmbeddings` to obtain the embedding of the source node of the positive edge and the embeddings of the candidates. Then, the `score` function is used to compute the scores of the candidates for the source node. Finally, the `sample` function calls the `TopSample` function to return the s nodes with highest scores among the candidates.

KBGAN uses the scores predicted by the generator model and represents the adversarial algorithms. In Figure 5c, the `compute` and `select` functions for KBGAN are similar to those for DNS in Figure 5b, except that KBGAN uses the embeddings produced by the generator model.

Our API can also implement hybrid algorithms. For instance, a hybrid of RNS and DegreeNS is widely used, which samples half of the negatives using RNS and half of the negatives using DegreeNS. We can modify lines 2 and 4 in Figure 5a to conduct DegreeNS and merge the sampling results of RNS and DegreeNS.

[1]https://pytorch.org/cppdocs

```
1 Vector<Node> sample(Bias bias, int s) {
2     return ProbSample(bias, s);
3 }
4 Bias bias = Uniform(subgraph.nodes);
5 Vector<Node> results = sample(bias, s);
```

(a) RNS

```
1 Vector<Node> select(int K) {
2     return ProbSample(Uniform(subgraph.nodes), K);
3 }
4 Bias compute(Edge e, Vector<Node> nodes) {
5     emb = getEmbeddings(e.src);
6     embs = getEmbeddings(nodes);
7     return Bias(nodes, score(emb, embs));
8 }
9 Vector<Node> sample(Bias bias, int s) {
10    return TopSample(bias, s);
11 }
12 Vector<Node> candidates = select(K);
13 Bias bias = compute(e, candidates);
14 Vector<Node> results = sample(bias, s);
```

(b) DNS

```
1 Bias compute(Edge e, Vector<Node> nodes) {
2     type = EmbeddingType::Generator;
3     emb = getEmbeddings(e.src, type);
4     embs = getEmbeddings(nodes, type);
5     return Bias(nodes, score(emb, embs));
6 }
7 Vector<Node> sample(Bias bias, int s) {
8     return ProbSample(bias, s);
9 }
```

(c) KBGAN

Figure 5: Use our API to write negative sampling algorithms. The `select` function of KBGAN is the same as DNS.

5 PARTITION SWAPPING SCHEDULING

In this section, we first define the *partition scheduling problem* (PSP), which determines how GPUs load and dump the node embedding partitions to conduct training in GE^2. Next, we transform the PSP problem into the *resolvable balanced incomplete block design* (RBIBD) problem [20], which is an extensively researched problem, to ensure that our solution inherits the favorable properties of RBIBD. Finally, we introduce a simple yet effective solution construction algorithm called COVER to solve the PSP problem.

5.1 Problem Definition and Requirements

Recall that in GE^2, each GPU loads a buffer state $\mathcal{S}_i = (P_1^i, P_2^i, \cdots, P_q^i)$ (referred to as state for conciseness), which contains a maximum of q node partitions, and processes the edge buckets formed by these partitions. Thus, we say that a buffer state covers the edge buckets it induces. To coordinate multiple GPUs to load and dump the buffer states, we need to specify all the involved buffer states.

Definition 1 (**partition scheduling problem, PSP**). *Given the total number of node partitions p and the number of node partition in a buffer state q, determine a set of buffer states $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2 \cdots \mathcal{S}_I\}$ that collectively cover all edge buckets.*

The buffer states in $\mathcal{S}$ must cover all edge buckets in order to carry out a complete epoch of training. While there exist numerous solutions to the PSP, a high-quality solution should meet the following three criteria.

Independent groups. In GE^2, multiple GPUs are employed to process distinct buffer states simultaneously. Buffer states processed concurrently should not share common node partitions; otherwise, different GPUs may update the same node embeddings, which necessitates synchronization in every batch. We say that a set of buffer states form an *independent group* if their node partitions have no overlap. For instance, in Figure 3, buffer states $\mathcal{S}_1 = \{P_1, P_2\}$ and $\mathcal{S}_2 = \{P_3, P_4\}$ constitute an independent group and can be processed in parallel. When there are G GPUs, the buffer states in $\mathcal{S}$ should be from independent groups whose size is a multiple of G. This enables the GPUs to process buffer states of an independent group in parallel without synchronizing node embedding.

Balanced workload. In GE^2, GPUs load buffer states, conduct training, and subsequently dump the buffer states to CPU in rounds. Node partitions processed by one GPU in the current round may be required by other GPUs in the subsequent round. In such cases, the GPUs must wait for the slowest GPU to complete its processing, which causes wasteful waiting. Therefore, to ensure balanced workload among the GPUs, all buffer states should cover the same number of edge buckets[2].

Non-overlapping buckets. To attain high efficiency, GE^2 should have a low CPU-GPU communication volume for loading and dumping node partitions. Communication cannot be further reduced when each edge bucket is covered by only one buffer state.[3] This is because, in this case, if we remove one node partition from a buffer state to reduce communication, the set of buffer states would no longer be able to cover all the edge buckets. Conversely, if two buffer states cover a common edge bucket, it may result in the waste of data IO and training computation.

5.2 Connection to The RBIBD Problem

It is challenging to design an algorithm that solves the PSP problem and satisfies the requirements above. However, we observe that our PSP problem can be converted into the RBIBD problem [20] in combinatorial mathematics.

Definition 2 (**RBIBD problem**). *The resolvable balanced incomplete block design (RBIBD) problem is to find an arrangement of w distinct objects into h blocks with the following properties:*

- ***Balanced**: each block contains exactly k distinct objects, and each object occurs in exactly t different blocks;*
- ***Concomitant**: every two distinct objects occur together in λ blocks;*
- ***Resolvable**: the blocks can be divided into t groups such that the blocks in each group is a complete replication of all the objects.*

An RBIBD problem is specified by its parameters (w, h, t, k, λ). If the problem has solutions, we have $wr = hk$ and $t(k-1) = \lambda(w-1)$. Thus, there are only three independent parameters, and we can specify an RBIBD problem by $RBIBD(w, k, \lambda)$. We transform our PSP problem into an RBIBD problem and show that the requirements in Section 5.1 are satisfied as follows:

- Our node partitions correspond to the objects, and our buffer states correspond to the blocks. Thus, we have $w = p$ and $k = q$, and can map a solution of RBIBD to our PSP problem.
- By setting $\lambda = 1$, the concomitant property of RBIBD ensures that every two distinct objects occur together in 1 block, and thus each edge bucket is covered by only one buffer state.
- The concomitant and balanced properties of RBIBD indicate that all buffer states cover the same number of edge buckets, i.e., $q(q-1)$. Thus, the workloads of the GPUs are balanced.
- The resolvable property of RBIBD ensures that each of the t groups of blocks do not contain common objects. Thus, each group of buffer states do not contain common node partitions and form an independent group for multiple GPU training.

Algorithm 2: The COVER Algorithm

```
Input: the number of node partitions p (p = 4^L), and the
       number of partitions in a buffer state q (q = 4)
Output: the set S of all buffer states
Bucket set E ← [(1, 1), ··· , (1, p), (2, 1), ··· , (p, p)]
Buffer state set S ← ∅
while Bucket set E larger than p do
    Node partition set P ← [1, 2, ··· , v]
    State group G ← ∅
    while Node partition set P not empty do
        Buffer state S_Δ ← ∅
        while Buffer state size |S_Δ| < q do
            Find the first partition P_j in P such that all edge
              buckets formed by P_j and S_Δ are in E
            Remove P_j from P and add P_j to S_Δ;
        Remove all edge bucket covered by S_Δ from E except
          the edge bucket on diagonal
        Add S_Δ to G
    Add G to S
return Buffer state set S
```

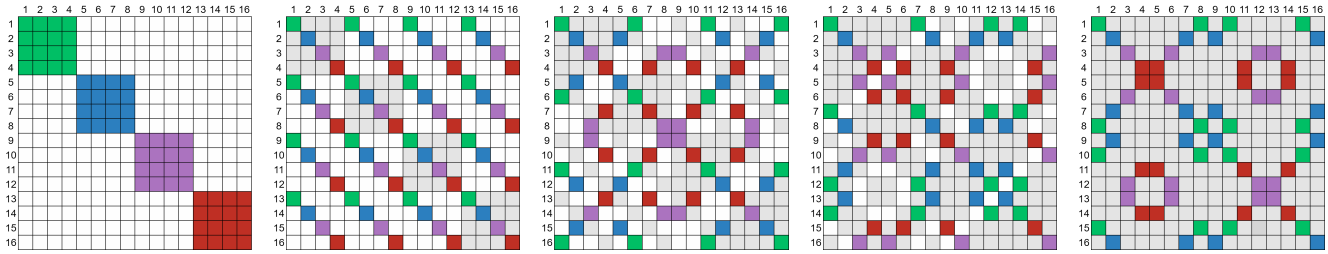
Theorem 1. *Given p node partitions and that each buffer state contains q node partitions, the partition scheduling problem, $PSP(p, q)$, can be solved via the RBIBD problem as $RBIBD(p, q, 1)$.*

Corollary 1. *Using the solution produced by $RBIBD(p, q, 1)$, the number of buffer states in each independent group is p/q, and there are $(p-1)/(q-1)$ such independent groups.*

Proof. With $wr = hk$ and $t(k-1) = \lambda(v-1)$ for the RBIBD problem and setting $\lambda = 1$, the number of block groups is $t = (w-1)/(k-1)$, and the number of blocks in each group is $h/t = (wt/k)/r = w/k$. Substituting $v = p$ and $k = q$, we obtain the results. □

The corollary demonstrates that, to ensure that the size of the independent buffer state group is a multiple of the number of GPUs, we need to set p and q properly.

[2]Different edge buckets contains a similar number of edges because we randomly assign the nodes to node partitions and each node partition contains many nodes. Thus, we use the number of edge buckets to quantify workload.
[3]We exclude the diagonal edge buckets here and in subsequent discussions.



(a) Group 1 (b) Group 2 (c) Group 3 (d) Group 4 (e) Group 5

Figure 6: The buffer states generated by the COVER algorithm when $v = 16$ and $k = 4$. In a group, we mark the edge buckets covered by different buffer states with different colors and the edge buckets covered in the previous groups with gray color. The buffer states can be obtained by reading squares with the same color along an axis of the matrix. For instance, in group 1, the 4 buffer states are $\{P_1, P_2, P_3, P_4\}$, $\{P_5, P_6, P_7, P_8\}$, $\{P_9, P_{10}, P_{11}, P_{12}\}$, and $\{P_{13}, P_{14}, P_{15}, P_{16}\}$. In group 2, the 4 buffer states are $\{P_1, P_5, P_9, P_{13}\}$, $\{P_2, P_6, P_{10}, P_{14}\}$, $\{P_3, P_7, P_{11}, P_{15}\}$, and $\{P_4, P_8, P_{12}, P_{16}\}$. After 5 groups, all the edge buckets are covered.

5.3 The COVER Algorithm

To solve RBIBD$(p, q, 1)$ to obtain all the buffer states, the values of p and q need to be specified. We advocate $p = 4^L$ (with L being a positive integer) and $q = 4$ for the following reasons.

- RBIBD$(p, q, 1)$ may not have solutions in general cases, and the sufficient conditions for solution existence are still unclear [13, 14, 39]. However, it is shown that RBIBD$(p, q, 1)$ can be solved with $p \equiv 4 (mod\ 12)$ and $q = 4$. Note that $p = 4^L$ and $q = 4$ belong to this case as $mod(4^L, 12) = 4$.
- In practice, the number of GPUs used for training (denote as G) is usually a power of two (e.g., 1, 2, 4, and 8), and the number of buffer states in an independent group (i.e., p/q) needs to be an integer multiple of G. We note that $p = 4^L$ and $q = 4$ meet this condition because the size of an independent group is 4^{L-1}.
- We can flexibly configure L such that the 4 node partitions in a buffer state fit in GPU memory. In particular, denote GPU memory capacity as C, the number of graph nodes as N, and embedding dimension as d, we have

$$4 \times Nd/p = 4 \times Nd/4^L \leq C, \tag{2}$$

which gives $L = \lceil \sqrt[4]{4Nd/C} \rceil$.

There exist solutions to our RBIBD problem [14] but they involve sophisticated techniques including abstract algebra, design theory, combinatorics, and group theory, which are difficult to understand and implement. As such, we propose a simple yet effective algorithm, called COVER, which adopts a greedy strategy to cover edge buckets, which is shown in Algorithm 2. In particular, Algorithm 2 first initializes the set $\mathcal{E}$ of all edge buckets in Line 3. It then uses three `while` loops to construct the buffer states. For the outermost loop (Lines 5-15), Algorithm 2 initializes the set of all node partitions $\mathcal{P}$ and tries to assign the partitions to buffer states. Once a partition is assigned to a buffer state, it is removed from the current partition set $\mathcal{P}$ (Line 12). This ensures that the buffer states in a state group $\mathcal{G}$ do not share common node partitions, and thus the buffer states in $\mathcal{G}$ form an independent group and can be processed in parallel.

The innermost loop (Lines 10-12) adds node partitions to an empty buffer state $\mathcal{S}_\Delta$ until $\mathcal{S}_\Delta$ contains q partitions. In each iteration, Algorithm 2 chooses the first partition P_Δ such that the edge buckets covered by P_Δ and $\mathcal{S}_\Delta$ are in the remaining edge bucket set $\mathcal{E}$ (Line 11). Once a buffer state $\mathcal{S}_\Delta$ is constructed, the edge buckets it covers are removed from the edge bucket set $\mathcal{E}$ (Line 13). This ensures that any two buffer states do not cover the same edge buckets. One subtlety is that we do not remove diagonal edge buckets of the form (P_j, P_j) from $\mathcal{E}$ in Line 13, and this allows Algorithm 2 to add the first edge bucket to an empty buffer state $\mathcal{S}_\Delta$ after generating the first group. As a result, in Line 5, the outermost loop can terminate when the number of remaining edge buckets in $\mathcal{E}$ is p because only the p diagonal edge buckets remain. Note that the diagonal edge buckets are already covered by the first group (i.e., when each partition is loaded to GPU memory for the first time). Thus, Algorithm 2 ensures that all edge buckets are covered. The correctness of Algorithm 2 is proved in Appendix A.

We provide an example of the buffer states generated by Algorithm 2 in Figure 6 with $p = 16$ and $q = 4$. We observe that each group contains 4 buffer states that collectively enumerate all node partitions, and the 5 groups cover all edge buckets. In particular, the buffer states in the first group are $\{P_1, P_2, P_3, P_4\}$, $\{P_5, P_6, P_7, P_8\}$, $\{P_9, P_{10}, P_{11}, P_{12}\}$, and $\{P_{13}, P_{14}, P_{15}, P_{16}\}$. Thus, after adding P_1 to $\mathcal{S}_\Delta = \emptyset$ in the second group, we cannot add P_2, P_3 and P_4 to $\mathcal{S}_\Delta$ as the induced edge buckets are already covered by the buffer state $\{P_1, P_2, P_3, P_4\}$ in the first group. P_5 is the first partition that satisfies Line 11 of Algorithm 2, and thus $\mathcal{S}_\Delta = \{P_1, P_5\}$. Considering P_1 and P_5, the first qualified partition is P_9; considering P_1, P_5 and P_9, the partition to add is P_{13}. Finally, the first buffer state of group 2 is $\mathcal{S}_\Delta = \{P_1, P_5, P_9, P_{13}\}$. Similarly, we can obtain the other 3 buffer states of group 2 as $\{P_2, P_6, P_{10}, P_{14}\}$, $\{P_3, P_7, P_{11}, P_{15}\}$, $\{P_4, P_8, P_{12}, P_{16}\}$. The other 3 groups can be obtained in a similar manner.

Discussions. Table 2 compares the communication volume of embedding swapping for different systems. N and E are the number of nodes and edges in the data graph, d is the embedding dimension, p is the number of node partitions, and G is the number of GPUs.

DGL-KE organizes each batch of edges for training on CPU and transfers them to GPU. Each positive edge needs s negative edges, one edge involves 2 node embeddings, and an epoch goes over E

Table 2: Communication volume in an epoch for different embedding swapping strategies, p is the number of partitions and G is the number of GPUs. The last column assumes $p = qG$ and serves as lower bound.

Strategy	Communication volume	Communication lower bound
CPU Batch (DGL-KE)	$2E(s+1)d$	$2E(s+1)d$
Hilbert (PBG)	$G \cdot p \cdot N \cdot d/2$	$G^2 \cdot N \cdot d$
BETA (Marius)	$G \cdot p \cdot N \cdot d/6$	$2 \cdot G^2 \cdot N \cdot d/3$
COVER (Ours)	$p \cdot N \cdot d/3$	$4 \cdot G \cdot N \cdot d/3$

positive edges. Thus, the communication volume is $2E(1+s)d$. PBG uses the classical Hilbert ordering [15], which keeps 2 partitions in GPU memory and swaps one partition each time to process one new edge bucket. The communication volume is $(p*N*d)/2$ when using a single GPU. To utilize G GPUs, PBG divides each partition into G smaller partitions and parallelizes different edge buckets among the GPUs, and thus the communication volume increases with the number of GPUs. Targeting at disk-based training, Marius designs the BETA algorithm to exchange partitions between CPU and disk but we also include it in the analysis by treating CPU and disk as GPU and CPU, respectively. In particular, BETA keeps q partitions in memory and swaps a single partition to process $q-1$ edge buckets each time. The communication volume is $(p*N*d)/6$ when using a single GPU and $q = 4$. However, BETA does not consider multiple GPUs, which is challenging because different GPUs should not access the same node partitions. If the method of PBG is used to parallelize multiple GPUs, the communication volume of BETA also increases with GPUs.

To express the communication volume as a function that only depends on G, the last column of Table 2 removes p by assuming $p = qG$. Take COVER for example, we substitute $p = qG$ and $q = 4$ into $p*N*d/3$ to obtain $4*G*N*d/3$. These values should be regarded as lower bounds rather than actual values for the communication volume. This is because besides the $p = qG$ requirement of the GPUs to hold non-overlapping node partitions and eliminate node embedding communication, p also depends on node embedding size, which in turn depends on the dataset and model. That is, q/p fraction of all the node embeddings must fit in the memory of each GPU, and when $q/(qG) = 1/G$ of the node embeddings does not fit in one GPU, larger p will be required.

6 EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments to evaluate our GE^2 and compare with the state-of-the-art graph embedding learning systems. The main findings include:

- *GE^2 is efficient, i.e., it speeds up existing systems by over 2X in most cases and scales much better when using multiple GPUs.*
- *GE^2 is general in supporting negative sampling algorithms.*
- *The designs of GE^2 are effective in improving efficiency.*

6.1 Experiment Settings

Datasets and algorithms. We conducted our experiments on the four graph datasets in Table 3, where *Dim.* is the dimension of the node embeddings, and *Emb. Size* is the total size of all node and relation embeddings. Among the four graphs, *Livejournal*[4] and *Twitter*[5] are social networks without relations, and *Freebase86m*[6] and *Wiki90m*[7] are knowledge graphs with relations. The four graphs are publicly available and widely used to evaluate graph embedding learning algorithms and systems. Following Marius [33], we split the edges of each graph into training, validation, and test subsets with proportions of 90%, 5%, and 5%, respectively. We used various score functions and negative sampling algorithms to evaluate the systems. As *Livejournal* and *Twitter* do not have relations, we used the popular Dot model [25], whose score function is $\theta_u^\top \theta_v$ and does not involve relation type. *Freebase86m* and *Wiki90m* have relations, and thus we used DistMult [58] and ComplEx [45], whose score functions are $\theta_u^\top \text{diag}(\theta_r)\theta_v$ and $\text{Real}(\theta_u^\top \text{diag}(\theta_r)\theta_v^*)$, respectively. Note that ComplEx learns embeddings with complex values, and θ_v^* means the element-wise conjugate of vector θ_v. For the negative sampling algorithm, we used a hybrid of RNS and DegreeNS [68] when comparing with existing systems because they only support static algorithms and this hybrid is shown to produce quality node embeddings. However, we also evaluated GE^2 on DNS [65] and KBGAN [2], which represent the dynamic and adversarial algorithms, respectively.

Table 3: Statistics of the datasets used in the experiments

G	$\|V\|$	$\|E\|$	$\|R\|$	*Dim.*	*Emb. Size*
Livejournal (LJ)	4.8M	68.9M	-	100	3.8G
Twitter (TW)	41.6M	1.46B	-	100	32G
Freebase86M (FB)	86.0M	304.7M	14824	100	66G
Wiki90M (WK)	91.2M	601.0M	1387	80	56G

For fair comparison, we used the same algorithm hyper-parameters for all systems and followed the configurations of Marius [33] unless specified otherwise. In particular, the batch size is 5×10^4 (i.e., number of positive edges in each batch), and the number of groups in a batch is 50 for shared sampling, which means that each group contains 10^3 positive edges; $s = 10^3$ negative edges were sampled and shared among the positive edges in one group. The optimizer was Adagrad [7], and the learning rate was 0.1. Training was conducted for 30 epochs on *Liverjournal* and 10 epochs on the other graphs. Marius used 25 epochs on *Liverjournal* but we observed that better embedding quality was achieved with 30 epochs.

Baseline systems. We compared GE^2 with three state-of-the-art graph embedding learning systems, i.e., DGL-KE from Amazon Web Service [68], PBG from Meta [24], and Marius [33]. We carefully configured the systems to ensure a fair comparison. In particular, PBG uses disk as the primary data storage, and we excluded all disk IO time when reporting its performance. For Marius, we stored all data in CPU memory (rather than disk by default) and turned on its optimizations to pipeline GPU training and CPU-GPU data exchange. We do include MariusGNN [47] in the experiments because it only differs from Marius in the strategy of swapping the

[4] https://snap.stanford.edu/data/soc-LiveJournal1.txt.gz
[5] https://snap.stanford.edu/data/twitter-2010.txt.gz
[6] https://data.dgl.ai/dataset/Freebase.zip
[7] https://dgl-data.s3-accelerate.amazonaws.com/dataset/OGB-LSC/wikikg90m-v2.zip

Table 4: Model quality (higher is better) and average epoch time of the systems when using a single GPU. Number in the brackets (e.g., 3.0x) is the speedup of GE^2 over the baseline.

Graph	Model	System	MRR	Hit@10	Time (s)
LJ	Dot	PBG	0.144	0.332	79.3 (3.0x)
		DGL-KE	0.150	0.325	135.6 (5.1x)
		Marius	0.151	0.331	90.5 (3.4x)
		GE^2	0.152	0.343	**26.6**
TW	Dot	PBG	0.014	0.033	940.0 (1.8x)
		DGL-KE	-	-	3209.9 (6.2x)
		Marius	0.015	0.032	1981.7 (3.8x)
		GE^2	0.015	0.034	**516.3**
FB	DistMult	PBG	0.365	0.503	688.2 (2.5x)
		DGL-KE	-	-	831.7 (3.0x)
		Marius	0.397	0.568	427.7 (1.6x)
		GE^2	0.404	0.604	**278.1**
	ComplEx	PBG	0.403	0.523	726.7 (2.5x)
		DGL-KE	-	-	823.8 (2.8x)
		Marius	0.436	0.578	420.3 (1.4x)
		GE^2	0.438	0.612	**292.6**
WK	DistMult	PBG	0.077	0.121	897.6 (2.3x)
		DGL-KE	-	-	1199.3 (3.1x)
		Marius	0.106	0.160	729.6 (1.9x)
		GE^2	0.103	0.161	**392.2**
	ComplEx	PBG	0.080	0.129	1202.1 (2.9x)
		DGL-KE	-	-	1273.5 (3.1x)
		Marius	0.108	0.162	757.5 (1.9x)
		GE^2	0.104	0.164	**408.9**

node partitions between disk and CPU memory. As such, Marius-GNN will perform the same as Marius when the node partitions are already in CPU memory. Unless stated otherwise, GE^2 used 16 node partitions and turned on the delayed update optimization for relation embeddings. All systems were compiled using NVCC-11.3 with O3 flag.

Experiment platform and performance metrics. We conducted the experiments on a server with 4 NVIDIA GeForce RTX 3090 GPUs, and each GPU has approximately 23.69 GB device memory. The server has 2 Intel Xeon Gold 6226R CPUs, and each GPU has 16 physical cores and supports hyper-threading. The main memory capacity was 378 GB. We evaluated the efficiency and model quality of the systems. For efficiency, we used the average running time to complete a training epoch, which is called epoch time for short. For model quality, we used *mean reciprocal rank* (MRR) and *hit rate at k* (Hit@k) [21, 24, 33, 68], which are standard quality metrics for graph embedding learning. Both MRR and Hit@k measure the rank of the score for a positive edge among all possible negative edges,

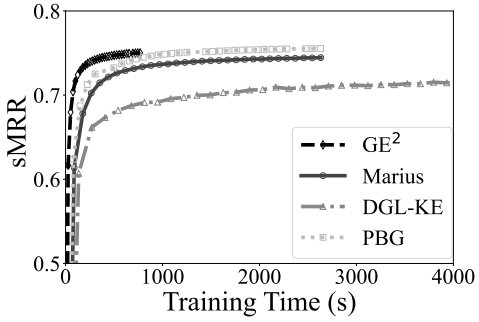


Figure 7: Test sMRR over the 30 training epochs for the *Dot* model on the *Livejournal* graph when using a single GPU.

and larger values indicate better embedding quality. By default, we use the exact MRR instead of the sampled MRR (i.e., sMRR) in Marius, which samples some negative edges for each positive edge to conduct evaluation. This is because KEM [21] observes that sampled MRR can be inaccurate, and following KEM [21], we use 10^4 test edges to compute MRR as using all the test edges will take a long time.

6.2 Main Results

Single GPU. Table 4 compares GE^2 with the baselines when using a single GPU to conduct training. DGL-KE misses model quality results for all graphs except for the smallest Livejournal since it runs OOM when evaluating the exact MRR. This is because DGL-KE materializes all possible edges between the target nodes for MRR evaluation and all nodes in the graph before checking edge existence, and OOM happens even if we run DGL-KE evaluation on the CPU with 378 GB memory. As such, we also evaluated the sMRR of the systems by sampling 10^4 negative edges for each positive edge following Marius [33] and observed that the sMRR scores of the other systems are comparable to DGL-KE.

The results in Table 4 show that the model quality of GE^2 is comparable to existing systems. In particular, considering MRR, GE^2 ranks first in four out of the six cases (i.e., a dataset plus a model) and second in the other two cases. Regrading training time, GE^2 consistently outperforms the baselines, and the speedup can be up to 6.2x (for DGL-KE on the *Twitter* graph) and is over 2x in most cases. As we will show in Section 6.3 with micro experiments, the high efficiency of GE^2 is attributed to higher GPU utilization and lower CPU-GPU communication cost than the baselines. Considering the baseline systems, both DGL-KE and Marius conduct CPU-GPU communication for each mini-batch but Marius is more efficient because it uses a pipeline to hide some of the communication costs and a more efficient GPU computation engine. Compared with the baselines, the speedup of GE^2 is larger for *Livejournal* and *Twitter* than *Freebase86m* and *Wiki90m* because the Dot model is simpler than DistMult and ComplEx, and thus the effect of GE^2's small CPU-GPU communication cost is more significant.

Figure 7 shows that the model quality of GE^2 improves smoothly with time for *Livejournal* (the results are similar for the other datasets), suggesting that training with GE^2 is stable. We use sMRR

Table 5: Model accuracy and training time of the systems for the Dot model on the *Twitter* graph.

	System	MRR	Hit@10	Time(s)
1 GPU	DGL-KE	-	-	3209.9 (6.2x)
	PBG	0.014	0.033	940.0 (1.8x)
	GE^2	0.015	0.034	**516.3**
2 GPUs	DGL-KE	-	-	2438.3 (7.5x)
	PBG	0.014	0.033	603.9 (1.9x)
	GE^2	0.015	0.033	**324.9**
4 GPUs	DGL-KE	-	-	1232.0 (6.8x)
	PBG	0.014	0.032	567.6 (3.1x)
	GE^2	0.015	0.033	**181.5**

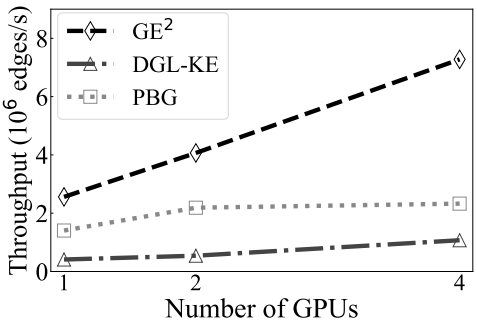


Figure 8: The relation between GPU count and training throughput for the Dot model on the *Twitter* graph.

in Figure 7 because it requires many MRR results, and computing each exact MRR is expensive. The sMRR scores are significantly higher than the exact MRR scores in Table 4 because sMRR considers fewer negative edges for each positive edge.

Multiple GPUs. Table 5 compares GE^2 with the baseline systems when using multiple GPUs to train the Dot model on the *Twitter* graph. The results on the other datasets are similar and thus omitted due to the page limit. This experiment does not include Marius because it can only utilize a single GPU. We also report the results with 1 GPU along side as a reference and transform the training time of the systems into *training throughput* (i.e., the number of edges processed per second) in Figure 8 to better understand the scalability of the systems. Ideally, training throughput should increase linearly with the number of GPUs.

The results show that GE^2 consistently outperforms both DGL-KE and PBG when using different number of GPUs. For instance, with 2 GPUs, GE^2 speeds up DGL-KE by 7.5X. Moreover, in Figure 8, the training throughput of GE^2 scales well when increasing the number of GPUs. In contrast, the scalability of PBG is poor in Figure 8, and the training throughput only increases marginally when increasing from 2 GPUs to 4 GPUs. This is because the CPU-GPU communication volume of PBG increases quickly with the number of GPUs while the CPU-GPU communication volume of GE^2 does not, which we have analyzed in Section 5.3 and will show later by experiments. As a result, the speedup of GE^2 over PBG is larger when using more GPUs. DGL-KE is slower than both PBG and GE^2 because it conducts CPU-GPU communication for every batch.

Table 6: Model quality and epoch time of GE^2 for Dot model on *Livejournal* with different negative sampling algorithms.

Graph	Method	1 GPU		2 GPUs		4 GPUs	
		MRR	Time	MRR	Time	MRR	Time
LJ	RNS	0.133	23.6	0.131	13.8	0.129	8.7
	DNS	**0.164**	50.3	**0.162**	27.8	**0.162**	18.3
	KBGAN	0.136	74.8	0.133	42.1	0.132	23.2

Advanced negative sampling algorithms. We use DNS [65] and KBGAN [2] as representatives of the dynamic and adversarial negative sampling algorithms, and evaluate the performance of GE^2 for them. We set the number of candidates (i.e., k) as 10^3 for the two algorithms and sampled 10^2 negative edges for each positive edge by computing the scores of these candidates. For a fair comparison of the algorithms, we also set the number of negative edges to sample for each positive edge as 10^2 for RNS [40] in this experiment.

As the three baseline systems (i.e., DGL-KE, PBG, and Marius) only support static negative sampling algorithms (e.g., RNS), we can only compare GE^2 with the implementations of DNS and KBGAN provided by PERec [54], an algorithm framework without system optimizations. PERec can only use a single GPU and failed to run DNS and KBGAN even on *Livejournal* (the smallest graph we used) due to timeout (set as 12 hours). Thus, we used a very small dataset *FB15k*, which was sampled from the *Freebase86m* graph and contained only 15k nodes. To align with PERec as much as possible, we disabled negative sample sharing for GE^2. When using a single GPU, PERec took 88.19s and 89.48s per epoch for DNS and KBGAN while GE^2 only took 15.41s and 17.64s. Thus, the speedup of GE^2 over PERec is close to 6x due to its efficient system designs.

In Table 6, we evaluated GE^2 on *Livejournal*. We also provide the results of RNS for reference. Table 6 shows that the advanced negative sampling algorithms (especially DNS) usually yield higher model quality than the simple RNS, although they take longer training time due to heavier computation in the sampling process. In particular, for *Livejournal*, the relative improvements in MRR of DNS over RNS are up to 23.3%. The training time of KBGAN the longest among the three algorithms because KBGAN uses two embeddings for each node and thus has higher CPU-GPU communication costs. The results also suggest that GE^2 scales well with the number of GPUs for the advanced negative sampling algorithms, which is in line with the results in Table 5.

Model quality. One may concern that the partitioning-based training of GE^2 can harm model quality because negative sampling is constrained to select the nodes loaded to one GPU. To examine this point, in Table 7, we compare with KEM [21] and HPO [22] by reusing their algorithm configurations for GE^2 (e.g., embedding dimension, number of negative samples, and learning rate). In particular, KEM is a system that supports different partitioning

Table 7: Comparing the model quality of GE^2 and baseline systems. The graph is FB, the model is ComplEx, d is embedding dimension, and s is the number of negative samples for each positive edge.

Configuration	System	MRR
d=100, s=1,000	GE^2	0.438
	KEM	0.423
d=128, s=10,000	GE^2	0.586
	HPO	0.594

Table 8: The CPU-CPU communication volume when training the Dot model on the *Livejournal* graph.

GPU	PBG	DGL-KE	Marius	GE^2
1	115.15 GB	121.18 GB	280.98 GB	40.54 GB
2	184.36 GB	120.57 GB	-	40.47 GB
4	312.60 GB	119.41 GB	-	40.35 GB

strategies for graph embedding learning, and HPO is an algorithm work that searches the optimal hyper-parameter configurations for training to achieve high model quality. The results show that the model quality of GE^2 matches the baselines with only small differences, suggesting that partitioning-based training does not hinder model quality. We conjecture that GE^2 has lower MRR than HPO because the optimal hyper-parameter configurations for HPO may not be optimal for GE^2 due implementation differences (e.g., parameter initialization method). PBG [24] also observes that partition-based training does not hinder model quality.

6.3 Micro Results

In this part, we conducted detailed profiling to explain the superior performance of GE^2 over the baseline systems. In these experiments, we used the *Livejournal* graph by default and note that the observations are similar for other graphs.

Running time breakdown. Figure 1 dissects the epoch time of GE^2 and the baseline systems into three parts: *CPU processing*, *CPU-GPU communication*, and *GPU computation*. *CPU processing* refers to the operations conducted on the CPU to prepare for training and update embeddings, *CPU-GPU communication* denotes the data transfer between CPU and GPU, and *GPU computation* encompasses all operations on GPU. Note that we disabled Marius's pipeline optimization in this experiment for clearer time decomposition, and so its epoch time in Figure 1 (117.0s) is longer than reported in Table 4 (90.5s). Marius does not have the results for 4 GPUs because it can only run on a single GPU.

The results in Figure 1 suggest that the *CPU processing* time of the baseline systems is notably longer than GE^2. For DGL-KE and Marius, this is because they handle edge batching and embedding updates on the CPU. For PBG, this is because it processes one edge bucket each time by loading two node partitions to GPU, which

Table 9: Running time of GE^2 when using other scheduling.

Graph Strategy	LJ Time (s)	TW Time (s)	FB Time (s)	WK Time (s)
PGH-Hilbert	79.3	940.0	688.2	897.6
GE^2-Hilbert	55.2	763.1	721.8	763.1
GE^2-COVER	26.6	516.3	292.6	392.2

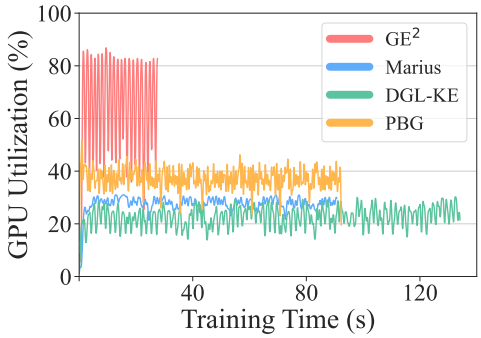


Figure 9: GPU utilization in an epoch for training the Dot model on the *Livejournal* graph with 1 GPU.

results in more CPU-GPU coordination rounds and the problem becomes more significant when using 4 GPUs. The short *CPU processing* time of GE^2 validates the benefits of offloading the edge batching and embedding update tasks to the GPUs. Moreover, the *CPU-GPU communication* time of GE^2 is also shorter than the baseline systems. This is attributed to the partition scheduling algorithm COVER as detailed in Table 8. The *Compute* time of GE^2 is similar to Marius but notably shorter than DGL-KE and PBG because GE^2 and Marius adopt the same efficient GPU computation engine.

In Table 8, we report the volume of CPU-GPU communication in one epoch for the systems. The results show that the communication volume of GE^2 is much smaller than the baseline systems and does not increase with the number of GPUs. In contrast, the communication volume of PBG increases quickly with the number of GPUs, which explains its poor scalability in Figure 8. These results are also consistent with our analysis in Table 2 of Section 5.3. One interesting phenomenon is that although PBG and DGL-KE have similar CPU-GPU communication volume when using 1 GPU, the communication time of DGL-KE is longer than PBG in Figure 1. This is because DGL-KE conducts many small communications for each batch while PBG conducts communication in the granularity of node partitions.

To better understand the performance gain of GE^2 over the baselines, we adapt GE^2 to use the Hilbert bucket ordering of PBG (denoted as GE^2-Hilbert). Table 9 compares the running time of GE^2-Hilbert with the original PBG (i.e., PBG-Hilbert) and GE^2 (i.e., GE^2-Cover). The results validate our previous explanations, i.e., GE^2 benefits from both the efficient GPU computation engine (comparing GE^2-Hilbert with PBG, which adopt the same bucket ordering) and the low CPU-GPU communication volume of COVER ordering

(comparing GE^2-Hilbert with GE^2-Cover, which adopt the same computation engine). For the FB graph, GE^2-Hilbert runs slightly slower than PBG-Hilbert because GE^2 re-indexes the node and edge IDs in the subgraph formed by each pair of node partitions, while PBG pre-processes all the indexes before training. FB is the sparsest among all the four graphs, and thus the re-index overhead of GE^2 becomes more significant w.r.t. the cost of training computation.

Figure 9 reports the GPU utilization of the systems when utilizing a single GPU. On average, GE^2 exhibits a GPU utilization of 63.28%, whereas PBG, DGL-KE, and Marius demonstrate a utilization of 37.14%, 22.69%, and 27.52%, respectively. DGL-KE and Marius have low GPU utilization because they batch edges and update embedding on CPU, and training runs fast on GPU, and thus the GPU is left idle waiting for CPU. Note that for Marius, the GPU utilization in Figure 9 is lower than reported in its paper because our GPU is faster and thus GPU waiting becomes more significant. PBG has higher GPU utilization than DGL-KE and Marius because it adopts the partition-based training paradigm, which loads the node partitions on GPU and reuses them for many batches.

7 RELATED WORK

Graph embedding learning systems. Graph embedding learns an embedding vector for each node in the data graph and finds many applications. For instance, Alibaba learns item embeddings from the user-item interaction graph and uses the embeddings to quantify item similarity for recommendation [50]. Apple learns embeddings for the entities (i.e., nodes) in knowledge graphs and links these entities with web contents for search and ranking [17]. Microsoft uses graph embeddings to capture the relation between search engine queries and recommend queries to users [23].

The rich applications of graph embeddings led to the development of training systems. AWS' DGL-KE [68] keeps a data graph and node embeddings in CPU memory, and transfers edges and related node embeddings to GPU for training at batch granularity. When using multiple GPUs, DGL-KE processes different relations on separate GPUs and allows asynchronous updates for node embeddings to reduce communication costs. Marius [33] treats disk as the primary data storage to handle very large graphs and employs an algorithm called BETA to swap one node partition between disk and CPU memory each time [33]. Similar to DGL-KE, Marius conducts CPU-GPU communication at batch granularity.

MariusGNN [47] observes that the BETA bucket ordering of Marius harms accuracy when training GNN models because the training samples produced by BETA are correlated and insufficient in randomness. As such, MariusGNN proposes the COMET ordering with two enhancements. First, to improve randomness, COMET uses many physical node partitions and randomly organizes multiple physical node partitions into one logical node partition for each epoch, and the logical node partitions are then managed with BETA. Second, to reduce sample correlation, COMET randomly assigns the task of processing each edge bucket to one of the buffer states that cover the edge bucket, while BETA eagerly processes each edge bucket in the first buffer state that covers the edge bucket. Both BETA and COMET target disk-based training with a single GPU and do not consider parallel training with multiple GPUs, which has distinct design requirements from disk-based training (e.g., no node embedding communication and balanced workload among the GPUs) and is tackled by our COVER bucket ordering.

PyTorch BigGraph (PBG) [24] mainly considers CPU training and utilizes disk as the primary data storage. To reduce sampling time and memory consumption, PBG introduces the shared sampling technique, which shares negative samples among a group of positive edges. GraphVite [69] tackles the special case where a node has two separate embeddings when serving as the source and destination of an edge. HET and PaGraph cache the embeddings of popular graph nodes on GPU to reduce CPU-GPU communication [27, 31].

KEM [21] conducts an extensive survey and evaluation of graph partitioning and negative sampling techniques for parallel graph embedding learning with multiple workers. For graph partitioning, *random partitioning* randomly assigns each edge to a worker, *relation partitioning* assigns different workers to keep different types of edges, *graph-cut partitioning* cuts a graph into patches with a small number of cross-patch edges and assigns each patch to one worker, and *stratification partitioning* organizes the node embeddings into non-overlapping partitions and assigns each worker to process the edge buckets covered by its local node embedding partitions (i.e., the 2D partition of PBG). KEM observes that stratification partitioning generally performs well because it eliminates cross-worker communication for node embeddings when combined with local sampling. Moreover, KEM improves the Hilbert ordering of PBG with *CAR*, which merges each pair of mirror edge buckets, removes the inactive node embeddings from partition loading, and reorganizes the node embeddings into partitions for each epoch. However, CAR is still similar to the Hilbert ordering of PBG by keeping two node partitions on each worker. In contrast, our COVER ordering is fundamentally different from the Hilbert ordering (i.e., by connecting to the RBIBD problem and keeping four node partitions on each worker) and achieves significantly smaller CPU-GPU communication volume.

Different from negative sampling algorithms, some algorithms improve embedding quality by pre-processing the data graph before training. For instance, UGE [53] constructs an unbiased graph from a potentially biased one and reduces the influence of some sensitive nodes on the graph embeddings. HEM [66] transforms a hypergraph into an uniform multigraph by integrating empty vertices, thereby allowing vertices to be included multiple times within a hyper-edge. While we agree that these pre-processing techniques are important for graph embedding learning, we do not include them as baselines in our experimental evaluation because our work focuses on system design issues and thus we believe it is more suitable to compare with graph embedding learning systems, e.g., Marius and PBG, which also focus on system issues such as data movement and computation scheduling. In contrast to system baselines such as Marius and PBG, [53] and [66] are important algorithmic works that are orthogonal to our work.

Graph neural network (GNN) systems. GNN models are also widely used for graph data [12], and many systems are designed to train GNNs efficiently, e.g., DGL [52], PyG [8], and AliGraph [67]. However, the computation pattern of GNNs is fundamentally different from graph embedding and so are the system optimizations. In particular, GNNs compute an output for each node in the data graph by aggregating its multi-hop neighbors, and the trainable

parameters are the neural network mappings. In contrast, graph embedding considers 1-hop neighbors directly connected by edges, and the trainable parameters are the node and relation embeddings. As such, systems for multi-GPU GNN training mainly consider reducing the communication caused by the multi-hop dependency [9, 18, 27, 28, 59]. For instance, P3 [9] uses model parallelism in the first layer of GNN models to place computation close to data and switches to data parallelism in the other layers.

8 CONCLUSIONS

We presented GE^2 as a general and efficient system for graph embedding learning. GE^2 offers a user-friendly API that allow users to easily express different negative sampling algorithms with diverse patterns. GE^2 also supports efficient training with multiple GPUs using the COVER algorithm to schedule data movements between CPU and GPU. Our experimental results show that GE^2 achieves much shorter training time than the state-of-the-art graph embedding systems and scales well when using multiple GPUs. For future work, we will extend GE^2 to distributed training on multiple machines.

REFERENCES

[1] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Annual Conference on Neural Information Processing Systems 2013. December 5-8, 2013, Lake Tahoe, Nevada, United States*. 2787–2795. https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html

[2] Liwei Cai and William Yang Wang. 2018. KBGAN: Adversarial Learning for Knowledge Graph Embeddings. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Association for Computational Linguistics, 1470–1480. https://doi.org/10.18653/v1/n18-1133

[3] Dawei Cheng, Fangzhou Yang, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. 2020. Knowledge Graph-based Event Embedding Framework for Financial Quantitative Investments. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*. ACM, 2221–2230. https://doi.org/10.1145/3397271.3401427

[4] Jingtao Ding, Yuhan Quan, Quanming Yao, Yong Li, and Depeng Jin. 2020. Simplify and Robustify Negative Sampling for Implicit Collaborative Filtering. In *Annual Conference on Neural Information Processing Systems 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/0c7119e3a6a2209da6a5b90e5b5b75bd-Abstract.html

[5] Bi'an Du, Xiang Gao, Wei Hu, and Xin Li. 2021. Self-Contrastive Learning with Hard Negative Sampling for Self-supervised Point Cloud Learning. In *MM '21: ACM Multimedia Conference, Virtual Event, China, October 20 - 24, 2021*. ACM, 3133–3142. https://doi.org/10.1145/3474085.3475458

[6] Wei Duan, Junyu Xuan, Maoying Qiao, and Jie Lu. 2022. Learning from the Dark: Boosting Graph Convolutional Neural Networks with Diverse Negative Samples. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 6550–6558. https://doi.org/10.1609/aaai.v36i6.20608

[7] John C. Duchi, Elad Hazan, and Yoram Singer. 2010. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In *COLT 2010 - The 23rd Conference on Learning Theory, Haifa, Israel, June 27-29, 2010*. Omnipress, 257–269. http://colt2010.haifa.il.ibm.com/papers/COLT2010proceedings.pdf#page=265

[8] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[9] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[10] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 2672–2680. https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html

[11] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 855–864. https://doi.org/10.1145/2939672.2939754

[12] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[13] Haim Hanani. 1961. The existence and construction of balanced incomplete block designs. *The Annals of Mathematical Statistics* 32, 2 (1961), 361–386.

[14] Haim Hanani, Dwijendra K Ray-Chaudhuri, and Richard M Wilson. 1972. On resolvable designs. *Discrete Mathematics* 3, 4 (1972), 343–357.

[15] David Hilbert and David Hilbert. 1935. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes: Nebst Einer Lebensgeschichte* (1935), 1–2.

[16] Tinglin Huang, Yuxiao Dong, Ming Ding, Zhen Yang, Wenzheng Feng, Xinyu Wang, and Jie Tang. 2021. MixGCF: An Improved Training Method for Graph Neural Network-based Recommender Systems. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*. ACM, 665–674. https://doi.org/10.1145/3447548.3467408

[17] Ihab F. Ilyas, JP Lacerda, Yunyao Li, Umar Farooq Minhas, Ali Mousavi, Jeffrey Pound, Theodoros Rekatsinas, and Chiraag Sumanth. 2023. Growing and Serving Large Open-domain Knowledge Graphs. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 253–259. https://doi.org/10.1145/3555041.3589672

[18] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org. https://proceedings.mlsys.org/book/300.pdf

[19] 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*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[20] Thomas P Kirkman. 1847. On a problem in combinations. *Cambridge and Dublin Mathematical Journal* 2 (1847), 191–204.

[21] Adrian Kochsiek and Rainer Gemulla. 2021. Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques. *Proc. VLDB Endow.* 15, 3 (2021), 633–645. https://doi.org/10.14778/3494124.3494144

[22] Adrian Kochsiek, Fritz Niesel, and Rainer Gemulla. 2022. Start small, think big: On hyperparameter optimization for large-scale knowledge graph embeddings. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 138–154.

[23] Jonathan Larson, Darren Edge, Nathan Evans, and Christopher M. White. 2020. Making Sense of Search: Using Graph Embedding and Visualization to Transform Query Understanding. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*. ACM, 1–8. https://doi.org/10.1145/3334480.3375233

[24] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. *CoRR* abs/1903.12287 (2019). arXiv:1903.12287 http://arxiv.org/abs/1903.12287

[25] Jure Leskovec. 2018. Tutorial: Representation Learning on Networks. http://snap.stanford.edu/proj/embeddings-www/

[26] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, Blai Bonet and Sven Koenig (Eds.). AAAI Press, 2181–2187. https://doi.org/10.1609/aaai.v29i1.9491

[27] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 401–415. https://doi.org/10.1145/3419111.3421281

[28] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 103–118. https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng

[29] Finlay MacLean. 2021. Knowledge graphs and their applications in drug discovery. *Expert opinion on drug discovery* 16, 9 (2021), 1057–1069.

[30] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*. PMLR, 1273–1282. http://proceedings.mlr.press/v54/mcmahan17a.html

[31] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *Proc. VLDB Endow.* 15, 2 (2021), 312–320. https://doi.org/10.14778/3489496.3489511

[32] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[33] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 533–549. https://www.usenix.org/conference/osdi21/presentation/mohoney

[34] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. 2011. A Three-Way Model for Collective Learning on Multi-Relational Data. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*. Omnipress, 809–816. https://icml.cc/2011/papers/438_icmlpaper.pdf

[35] Rong Pan, Yunhong Zhou, Bin Cao, Nathan Nan Liu, Rajan M. Lukose, Martin Scholz, and Qiang Yang. 2008. One-Class Collaborative Filtering. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*. IEEE Computer Society, 502–511. https://doi.org/10.1109/ICDM.2008.16

[36] Dae Hoon Park and Yi Chang. 2019. Adversarial Sampling and Training for Semi-Supervised Information Retrieval. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 1443–1453. https://doi.org/10.1145/3308558.3313416

[37] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 701–710. https://doi.org/10.1145/2623330.2623732

[38] Jinfeng Rao, Hua He, and Jimmy Lin. 2016. Noise-Contrastive Estimation for Answer Selection with Deep Neural Networks. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. ACM, 1913–1916. https://doi.org/10.1145/2983323.2983872

[39] Colin Reid and Alex Rosa. 2012. Steiner systems $S(2, 4, v)$-a survey. *The Electronic Journal of Combinatorics* (2012), DS18–Feb.

[40] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI 2009, Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, Montreal, QC, Canada, June 18-21, 2009*. AUAI Press, 452–461. https://www.auai.org/uai2009/papers/UAI2009_0139_48141db02b9f0b02bc7158819ebfa2c7.pdf

[41] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=HkgEQnRqYQ

[42] Zequn Sun, Wei Hu, Qingheng Zhang, and Yuzhong Qu. 2018. Bootstrapping Entity Alignment with Knowledge Graph Embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 4396–4402. https://doi.org/10.24963/ijcai.2018/611

[43] Zhu Sun, Jie Yang, Jie Zhang, Alessandro Bozzon, Long-Kai Huang, and Chi Xu. 2018. Recurrent knowledge graph embedding for effective recommendation. In *Proceedings of the 12th ACM Conference on Recommender Systems, RecSys 2018, Vancouver, BC, Canada, October 2-7, 2018*. ACM, 297–305. https://doi.org/10.1145/3240323.3240361

[44] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. ACM, 1067–1077. https://doi.org/10.1145/2736277.2741093

[45] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*. JMLR.org, 2071–2080. http://proceedings.mlr.press/v48/trouillon16.html

[46] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[47] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 144–161. https://doi.org/10.1145/3552326.3567501

[48] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural Deep Network Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 1225–1234. https://doi.org/10.1145/2939672.2939753

[49] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. GraphGAN: Graph Representation Learning With Generative Adversarial Nets. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 2508–2515. https://doi.org/10.1609/aaai.v32i1.11872

[50] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 839–848. https://doi.org/10.1145/3219819.3219869

[51] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. 2017. IRGAN: A Minimax Game for Unifying Generative and Discriminative Information Retrieval Models. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*. ACM, 515–524. https://doi.org/10.1145/3077136.3080786

[52] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[53] Nan Wang, Lu Lin, Jundong Li, and Hongning Wang. 2022. Unbiased Graph Embedding with Biased Graph Observations. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 1423–1433. https://doi.org/10.1145/3485447.3512189

[54] Xiang Wang, Yaokun Xu, Xiangnan He, Yixin Cao, Meng Wang, and Tat-Seng Chua. 2020. Reinforced Negative Sampling over Knowledge Graph for Recommendation. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 99–109. https://doi.org/10.1145/3366423.3380098

[55] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 1021–1028. https://doi.org/10.24963/ijcai.2018/142

[56] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Product Knowledge Graph Embedding for E-commerce. In *WSDM '20: The Thirteenth ACM International Conference on Web Search and Data Mining, Houston, TX, USA, February 3-7, 2020*. ACM, 672–680. https://doi.org/10.1145/3336191.3371778

[57] Lanling Xu, Jianxun Lian, Wayne Xin Zhao, Ming Gong, Linjun Shou, Daxin Jiang, Xing Xie, and Ji-Rong Wen. 2022. Negative Sampling for Contrastive Representation Learning: A Review. *CoRR* abs/2206.00212 (2022). https://doi.org/10.48550/arXiv.2206.00212 arXiv:2206.00212

[58] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6575

[59] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 417–434. https://doi.org/10.1145/3492321.3519557

[60] Ruichao Yang, Xiting Wang, Yiqiao Jin, Chaozhuo Li, Jianxun Lian, and Xing Xie. 2022. Reinforcement Subgraph Reasoning for Fake News Detection. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 2253–2262. https://doi.org/10.1145/3534678.3539277

[61] Zhen Yang, Ming Ding, Chang Zhou, Hongxia Yang, Jingren Zhou, and Jie Tang. 2020. Understanding Negative Sampling in Graph Representation Learning. In

KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020. ACM, 1666–1676. https://doi.org/10.1145/3394486.3403218

[62] Zhen Yang, Ming Ding, Xu Zou, Jie Tang, Bin Xu, Chang Zhou, and Hongxia Yang. 2023. Region or Global? A Principle for Negative Sampling in Graph-Based Recommendation. *IEEE Trans. Knowl. Data Eng.* 35, 6 (2023), 6264–6277. https://doi.org/10.1109/TKDE.2022.3155155

[63] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* ACM, 974–983. https://doi.org/10.1145/3219819.3219890

[64] Xiangxiang Zeng, Xinqi Tu, Yuansheng Liu, Xiangzheng Fu, and Yansen Su. 2022. Toward better drug discovery with knowledge graph. *Current opinion in structural biology* 72 (2022), 114–126.

[65] Weinan Zhang, Tianqi Chen, Jun Wang, and Yong Yu. 2013. Optimizing top-n collaborative filtering via dynamic negative item sampling. In *The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013.* ACM, 785–788. https://doi.org/10.1145/2484028.2484126

[66] Yaoming Zhen and Junhui Wang. 2023. Community detection in general hypergraph via graph embedding. *J. Amer. Statist. Assoc.* 118, 543 (2023), 1620–1629.

[67] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242. https://www.vldb.org/pvldb/vol15/p1228-zheng.pdf

[68] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020.* ACM, 739–748. https://doi.org/10.1145/3397271.3401172

[69] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019.* ACM, 2494–2504. https://doi.org/10.1145/3308558.3313508

A PROOF FOR THE COVER ALGORITHM

We show that COVER can always produce buffer states of size 4 that satisfy our design requirements.

Correctness of COVER. *When the total number of node partitions p is 4^L ($L \geq 1$) and each buffer state has $q = 4$ node partitions, COVER can terminate (i.e., cover all the edge buckets) and produce buffer states that all contain 4 node partitions.*

Proof. We will the correctness of COVER using mathematical induction.

- Base case: When $L = 1$, the number of node partitions 4^L is 4, and the COVER algorithm can directly put all node partitions into one buffer state. The output of COVER algorithm when $L = 2$ can be found in Figure 6. Thus, COVER is correct for both $L = 1$ and $L = 2$.
- Induction hypothesis: Assume that COVER is correct when the number of node partitions 4^L, where $L \geq 2$.
- Induction step: We need to prove COVER is correct when the number of node partitions 4^{L+1}.

First, we divide the 4^{L+1} node partitions into 4 non-overlapping groups, each containing 4^L node partitions. We then use the COVER algorithm to form buffer states for each group, which ensures that the buffer states for each group satisfy our requirements. Next, we consider the interaction between the 4 groups. Each group is further divided into 4 subgroups, each containing 4^{L-1} node partitions. There are now 16 subgroups. We can then combine these 16 subgroups according to the last four combinations in Figure 6 to form 4 new groups. For each combination, each new group contains 4 subgroups, and each new group contains 4^L node partitions. To prevent the repeated generation of buffer states in the node partition within a particular subgroup, the COVER algorithm can be used to build the buffer state for each new group. It follows the same steps as 4^L partitions in the interaction stage for its construction. In this way, all buffer states across 4 groups also meet the requirements.

Therefore, COVER is correct when the number of node partitions is 4^{L+1}, and by mathematical induction it holds for all $L \geq 1$.

□

This property guarantees that the size of buffer state is always 4, which enables the COVER algorithm to meet our target requirement in Section 5.