Senevirathne, Damitha; Wijesiri, Isuru; Dehigaspitiya, Suchitha; Dayarathna, Miyuru; Jayasena, Sanath; Suzumura, Toyotaro;, "Memory Efficient Graph Convolutional Network based Distributed Link Prediction", Seventh International Workshop on High Performance Big Graph Data Management, Analysis, and Mining, pp. 2977- 2986, 12 Dec. 2020 isbn: 978-1-7281-6251-5/20,


# Memory Efficient Graph Convolutional Network based Distributed Link Prediction

Damitha Senevirathne*, Isuru Wijesiri*, Suchitha Dehigaspitiya*,
Miyuru Dayarathna‡*, Sanath Jayasena*, and Toyotaro Suzumura†¶§
*Department of Computer Science & Engineering,
University of Moratuwa, Moratuwa, Sri Lanka
Email: {damithasenevirathne.16,isurumaduranga.16,suchitha.16,sanath}@cse.mrt.ac.lk
‡WSO2, Inc., USA
Email: miyurud@wso2.com
†IBM T.J. Watson Research Center, New York, USA / ¶MIT-IBM Watson AI Lab, Cambridge, MA, USA/
§Barcelona Supercomputing Center, Barcelona, Spain
Email: suzumura@acm.org

Abstract—Graph Convolutional Networks (GCN) have found multiple applications of graph-based machine learning. However, training GCNs on large graphs of billions of nodes and edges with rich node attributes consume significant amount of time and memory resources. This makes it impossible to train such GCNs on general purpose commodity hardware. Such use cases demand high-end servers with accelerators and ample amounts of memory. In this paper we implement a memory efficient GCN based link prediction on top of a distributed graph database server called JasmineGraph[1]. Our approach is based on federated training on partitioned graphs with multiple parallel workers. We conduct experiments with three real world graph datasets called DBLP-V11, Reddit, and Twitter. We demonstrate that our approach produces optimal performance for a given hardware setting. JasmineGraph was able to train a GCN on the largest dataset DBLP-V11(>10GB) in 20 hours and 24 minutes for 5 training rounds and 3 epochs by partitioning it into 16 partitions with 2 workers on a single server while the conventional training method could not process it at all due to lack of memory. The second largest dataset Reddit took 9 hours 8 minutes to train with conventional training while JasmineGraph took only 3 hours and 11 minutes with 8 partitions-4 workers in the same hardware giving 3 times improved performance. In case of Twitter dataset JasmineGraph was able to give 5 times improved performance. (10 hours 31 minutes vs 2 hours 6 minutes;16 partitions-16 workers).

Index Terms—Machine Learning; Graph Databases; Distributed Databases; Graph Theory; Graph Convolutional Neural Networks; GraphSAGE; Deep Learning; Distributed Learning; Link Prediction;

## I. Introduction

Recently, an increasing number of applications are found which require processing large scale graphs of millions or even billions of vertices [11]. Some of the notable examples include social networks, web graphs, transaction networks, citation graphs, etc [9] [27].

Traditional analytics on large scale graphs has been focusing around the analysis of graph properties. Examples include calculation of the graph structural properties such as Degree Distribution, Betweenness Centrality, Page Rank, k-core, triangles, etc [9] [13]. Graph-based machine learning field has emerged with significant potential to expand the horizons of graph processing [5].

Graph Convolutional Networks (GCN), which is one of the graph-based machine learning approaches, has gained significant attention from the research community [19]. The expectations of these GCN applications are expanding rapidly and we can observe the demand for applications to handle large scale graphs. These applications offer significantly higher accuracy compared to the previous approaches for tasks such as node classification, graph clustering, link prediction, etc.

Link prediction predicts the probability of having a link between two nodes in a graph [22]. There are multiple different ways for implementing GCN based link prediction applications. For example node embedding construction can be used along with similarity metrics and locality sensitive hashing to conduct link prediction [16]. However, most of these approaches are resource heavy and they require the use of high-end servers and clusters. Graphs take more space in memory and keeping the whole graph in memory is mandatory with traditional approaches. These techniques are impossible to run on commodity hardware on large scale graphs. Hence it has become a significant issue to find techniques to run such link prediction on top of large scale graphs. Even in the current settings running such graph-based machine learning on million scale graphs is a great challenge.

In this paper we propose a resource efficient approach for link prediction using distributed learning which intelligently utilizes the limited memory resources available in a computer system. Our approach is based on graph partitioning and it enables us to train any type of machine learning model for tasks such as link prediction, node classification on large scale graphs on top of which otherwise, we could not perform such training.

In our approach we first partition a large graph using Metis graph partitioner and then use the partitions for training the machine learning model [17]. Use of Metis enables us

[1]https://github.com/miyurud/jasminegraph

to partition the graph in a manner of minimizing the edge cuts and minimizing the information loss. The edge cuts are added into the graph partitions before training to reduce the information loss. These partitions are scheduled to train on distributed clients using our memory estimation algorithm and intelligent scheduling algorithm to maximize resource utilization. The distributed clients fetch global model weights and train for a given number of epochs and send back to the aggregator to aggregate. Then clients fetch the global model again and repeat the procedure for a given number of rounds.

Using experiments conducted on a server, we demonstrate how our approach enables us to train a single distributed ML model which we could not achieve with naive scheduling. We implement our solution on top of JasmineGraph, which is a distributed graph database engine [16]. JasmineGraph is a C/C++ based graph database borrowing several architectural elements from Acacia [7] [10] distributed graph database server. JasmineGraph was able to train a GCN on the largest dataset DBLP-V11(>10GB) in 20 hours and 24 minutes for 5 training rounds and 3 epochs by partitioning it into 16 partitions with 2 workers on a single server while the conventional (i.e., naive) training method could not process it at all due to lack of memory. The second largest dataset Reddit took 9 hours 8 minutes to train with the baseline conventional training while JasmineGraph took only 3 hours and 11 minutes with 8 partitions-4 workers in the same hardware giving 3 times improved performance. In case of Twitter dataset JasmineGraph was able to give 5 times improved performance. (10 hours 31 minutes vs 2 hours 6 minutes;16 partitions-16 workers).

The contributions of this paper can be listed as follows,

- *Memory Efficient Graph Neural Network Training* - We conduct machine learning model training concurrently in multiple workers of JasmineGraph. In order to do this we developed a memory efficient scheduling algorithm. We conduct this in a distributed manner and obtain better performance compared to the standard approach of standalone training.
- *Generic Machine Learning within a Graph Database* - We present a graph database server system architecture where the users can run their custom-developed machine learning models on top of the datasets stored in a distributed graph database server. These models can be used for any graph machine learning task such as link prediction, node classification, etc.
- *Performance optimizations* - We implement and evaluate the approach for link prediction based on GCNs. We also describe multiple different performance optimizations which we conduct to do this process in resource efficient manner.

The rest of the paper has been organized as follows. In the next section we describe the related work. Section III describes the system design. Section IV provides the implementation details. Section V provides the evaluation of the proposed approach. We provide a discussion of the results obtained in Section VI. Section VII concludes the paper.

## II. Related Work

Link prediction is a technique to predict the future connections of a network using the existing connections. Link prediction uses a scoring function such as graph distance, common neighbors, Katz index, Jaccard's coefficient, etc. to measure the likelihood of links.

Network embedding which is the process of representing the network topology, node and edge information of a network in a low-dimensional vector space is the first and the foremost step of any machine learning based graph analytics task. Network embeddings can be mainly discussed under three categories as matrix factorization [29], random walk [25], and deep learning approaches. Matrix factorization based algorithms obtain the node embedding by factoring a matrix such as adjacency matrix which represents the network connection. Random walk based approaches like DeepWalk [25] and Node2Vec [14] perform a random walk through the neighbourhood of a selected node to compute the embedding. However, the factorization and random walk embeddings are unable to include node feature information. Moreover, the number of model parameters of their learning models linearly grow with the graph size.

Gori *et al.* [12] presented Graph Neural Networks (GNN) for the first time, and it got further extended by Scarselli *et al.* [26]. Under this approach, the neighbour information is propagated through a recurrent architecture iteratively until reaching a stable position. This approach is computationally expensive due to having more iterations in the learning process. Several alternatives to GNN are emerging in the last few years overcoming the limitations of GNN. These approaches are Graph Convolutional Networks (GCN), Graph Attention Networks (GAN), Graph Generative Networks, Graph Autoencoders, and Graph Spatial-temporal Networks.

Bruna *et al.* (2013) introduced a graph convolution variant developed based on spectral graph theory [4] has done a revolution by generalizing the convolution theory to graph data. The methodology of GCN is learning a function to generate the target node's representation by aggregating its own features and neighbour features. Then graph attention model, graph generative model, graph autoencoders and graph spatial-temporal models have been implemented centering the GCN. Furthermore, GCN is categorized as spectral based, spatial based, and pooling module approaches. GCN has marked a significant success in learning graph data rather than GNN, some challenges exist. Most of them are scalability, graph complexity and computation efficiency. Karunarathna *et al.* presented a data partitioning based solution to fix these issues [16].

PyTorch-BigGraph (PBG) is a graph embedding system that includes several modifications to the traditional multi-relation embedding systems which allows for scaling to graphs of billions of nodes and trillions of edges [21]. PBG uses a block decomposition of adjacency matrix into $N$ buckets. This allows for training the edges from one bucket at a time. Then

PBG either swaps embeddings from each partition to disk to reduce memory usage or if enough memory is available it conducts distributed execution across multiple computers.

However, the node embedding methods in PBG do not include the utilization of GCNs. Therefore, PBG is unable to incorporate rich node feature information given available in certain graph data to construct higher quality node embeddings. Furthermore, PBG depends on a shared file system whereas our solution implemented on top of JasmineGraph can scale to multiple independent host file systems that work together to share the workload.

Euler is a distributed graph learning framework developed by the Alibaba group which allows for node embedding construction [1]. It allows for models developed in Tensorflow to be trained on heterogeneous graphs. However, it is untested for large graph datasets such as DBLP-V11. Similar to PBG, it depends on a shared file system that runs on top of Hadoop Distributed File System. We implement our solution to take advantage of multiple host file systems that do not require HDFS.

JanusGraph is a graph database capable of handling very large graphs [3]. However, JasnusGraph is dependent on Hadoop. Similar graph database systems include Acacia [10] [8], Trinity [28], etc. are some other examples for distributed graph databases. More importantly, neither system supports node embedding construction, link prediction or machine learning use cases on their graph data.

## III. System Design

An overview of the design of JasmineGraph is shown in Figure 1. The system has been designed with two main component categories: Master and Worker. Two types of communication protocols have been designed for communication between external clients and the JasmineGraph system, and another between the Master and the Worker as well as between the Workers themselves.

The MetaDB of JasmineGraph is a standalone SQLite database which is used for storing metadata such as vertex (i.e., node) count, edge count, graph path, and other details of the graphs that are being uploaded to JasmineGraph. The API for accessing the MetaDB is called MetaDB Interface.

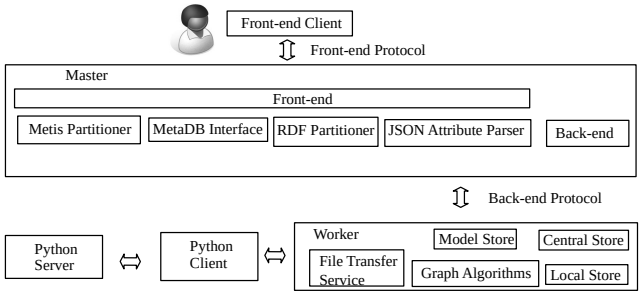


Fig. 1: Overview of JasmineGraph.

The modules RDF partitioner and JSON attribute parser are used for processing the knowledge graphs (especially the attributes). Once the graph structure has been extracted by these two components, they get partitioned by using the Metis Partitioner. The plain edge lists are directly partitioned using the Metis Partitioner.

Each worker consists of three data stores: Local, Central, and Model stores. The Local Store maintains the partitioned sub graphs. Central store maintains the edges where the starting and ending vertices stay in two different local stores. The attributes of each local/central partition too gets stored in the corresponding data store. During the machine learning (ML) model training process, the ML models are created for each partition. This means for a pair of local store and central store partitions, a corresponding ML model is trained. Such models get stored in the model store. The File Transfer Service module conducts data transfers between the Master and the Workers.

## IV. Implementation

The system architecture of the JasmineGraph is shown in Figure 2. We have extended the system with a set of Python workers (this is a set of Python clients and a server commonly denoted as $P_i$ in Figure 2) which sits along with the C/C++ workers. All the machine learning related processing are conducted by the Python workers. Weight/gradient exchange during the ML model training process happens directly between the Python workers.

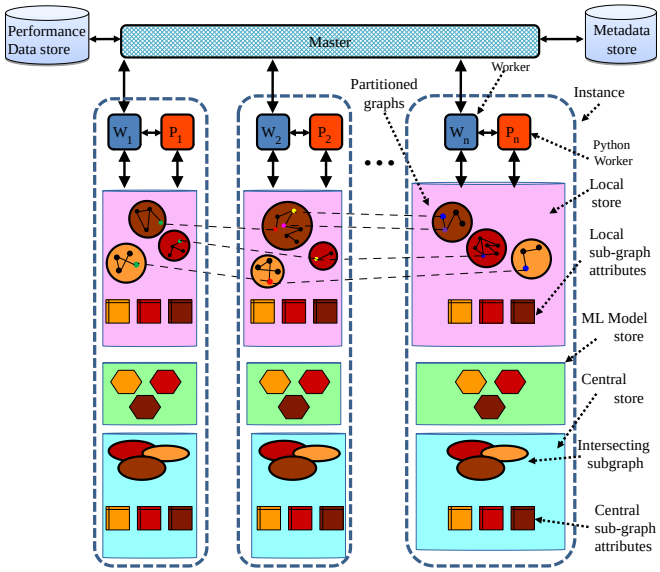


Fig. 2: Architecture of JasmineGraph.

One of the key improvements made to JasmineGraph system compared to the previous version (described in [16]) is the introduction of the ability to exchange the model parameters and gradients across the Python workers. This increases the accuracy of the model trained but simultaneously increases the communication overhead during the system's operation. Furthermore, depending on the level of partitioning that can be performed on a graph dataset, an approach which exchanges model parameters allows us to train a single machine learning

model from all the graph partitions which provides more room for extended tasks such as collaborative learning between JasmineGraph clusters.

Furthermore, we have changed the machine learning system architecture significantly in such a way that users of JasmineGraph can upload offline developed and verified ML models to the system. These model building code can be directly exported from notebooks and upload them to the system as Python script.

### A. Federated Training

In our approach we use distributed (federated) learning techniques since our data has been partitioned across workers.

**Algorithm 1** Federated training - Workers/Hosts

**Input:** Model - $M$, List of graph partitions in the host - $\mathcal{G}$, List of clients in the host - $\mathcal{C}$, Total memory capacity of host - $h$, Number of local epochs - $\mathcal{E}$, Learning rate - $\eta$, Sample percentage - $\mathcal{P}$, Batch size - $B$
**Output:** Trained Local model for every graph partition
$\mathcal{G} \leftarrow$ (reconstruct partitioned graphs of $\mathcal{G}$ with edge cuts)
$\mathcal{L} \leftarrow$ MEMESTIMATE($\mathcal{G}$); // Memory estimation per partition using the memory estimation algorithm
$\mathcal{S} \leftarrow$ SCHEDULE($\mathcal{L}, \mathcal{C}, h$); // Using the scheduling algorithm
$\mathcal{M} \leftarrow \{\}$;
**for** *each graph* $g \in \mathcal{G}$ **do**
  $\mathcal{M}[g] \leftarrow$ INITIALIZE($M$);
**end**
**Function** CLIENTTRAIN*($c, w$)*:
  $\mathcal{W} \leftarrow []$;
  **for** *each graph partition* $g \in \mathcal{S}[c]$ **do**
    $\mathcal{M}[g]$.weights $\leftarrow w$;
    $graph \leftarrow$ LOADTOMEMORY($g$);
    $\mathcal{B} \leftarrow$ (sample the $graph$ and split batches with sample percentage $\mathcal{P}$ and batch size $B$);
    **for** *each epoch* $e = 1, 2, ..., \mathcal{E}$ **do**
      **for** *each batch* $b \in \mathcal{B}$ **do**
        TRAIN($\mathcal{M}, \eta$); // $w \leftarrow w - \eta \nabla l(w; b)$
      **end**
    **end**
    Insert $\mathcal{M}[g]$.weights into $\mathcal{W}$;
    Delete $graph$ from memory;
  **end**
  return $\mathcal{W}$;

Python clients and Python server are the modules that conduct the distributed (i.e., federated) graph learning. There is one Python server for the distributed JasmineGraph setup and it usually located in the master node. Python clients are located in the worker nodes and there can be any number of clients per worker node. This number is decided based on the computation power and the number of partitioned sub graphs stored in the particular worker node.

When the training process starts, Python server loads the particular deep learning model ($\mathcal{M}$) from the local model store that to be trained and initialize model weights accordingly. The model loaded to the Python server is called the *Global model*. Then the initialized model weights are extracted from the global model and sent to every Python client related to JasmineGraph workers in the JasmineGraph cluster using the defined communication protocols.

Then the clients in JasmineGraph worker nodes load the same deep learning model ($\mathcal{M}$) from the local model store to the memory. A model ($\mathcal{M}[g]$) is initialized per every graph partition ($g$) to be trained upon. Before the training process is started graph partitions are scheduled and assigned for each client using our intelligent scheduling algorithm to maximize the resource utilization, to increase parallel processing, and to minimize the overall time taken for the training. The partition scheduling algorithm is explained in Section IV-B. Moreover, edge cuts stored in the central store are added to the partitioned sub graph for minimizing the information loss before the training.

**Algorithm 2** Federated model aggregation - Server

**Input:** Model - $M$, Set of workers/hosts - $\mathcal{H}$, Number of Rounds - $\mathcal{R}$
**Output:** Trained Global model
$\mathcal{M} \leftarrow$ INITIALIZE($M$);
$w_0 \leftarrow \mathcal{M}$.weights;
**for** *each round* $r = 1, 2..., R$ **do**
  $\mathcal{W} \leftarrow []$;
  **for** *each worker* $h \in \mathcal{H}$ ***in parallel*** **do**
    **for** *each client* $c \in h$ ***in parallel*** **do**
      $w \leftarrow$ CLIENTTRAIN($c, w_r$);
      $\mathcal{W}$.extend($w$);
    **end**
  **end**
  $w_{r+1} \leftarrow \sum_{k=1}^{\text{W.length}} \frac{n_k}{n} \mathcal{W}[k]$;
**end**
$\mathcal{M}$.weights $\leftarrow w_R$;

Then each Python client loads assigned partitioned sub graphs into the memory one by one and train its deep learning model ($\mathcal{M}[g]$) for given number of epochs ($\mathcal{E}$) with given hyper parameters (learning rate $\eta$ etc.). The training can be done on either CPU or GPU depending on the resources available in each client node.

Before starting the training weights of each model ($\mathcal{M}[g]$) are set to the weights that were sent by the Python server. After the training of each round partitioned graph models are saved to the disk. In this manner we maintain a local model per each graph partition. Here, a training round refers to the period of training that Python clients do asynchronously before synchronising their models by communicating with the Python server. After completion of training of every partition assigned, Python clients send local model weights related to each partitioned graph model ($\mathcal{M}[g]$) to the Python server using defined communication protocols. See Algorithm 1 for more details.

After receiving local model weights related to each and every sub graph used for training, the Python server aggre-

gates them using the federated aggregation algorithm to get weighted average of the model weights and updates the global model. After the aggregation, the global model is saved to the disk and weights ($w$) are sent back to Python clients to repeat the same training process for given number of training rounds ($r$) or until the global model converges. After each training round we evaluate the global model with the testing (or evaluation) data thus training can be early stopped if over-fitting happens. Because of we save each and every intermediate local and global models to the model store, the training process can re-start where it stopped in case of system failure or out of memory issue.

The model aggregation algorithm (Algorithm 2) is a customized version of the Federated Averaging algorithm developed by McMahan *et al.* [24]. In each training round ($r$) the aggregation algorithm takes the weighted average of the weights of each graph partition ($k$) stored in array $\mathcal{W}$; by number of training examples used ($n_k$) and applies the update $w_{r+1} \leftarrow \sum_{k=1}^{\text{W.length}} \frac{n_k}{n} W[k]$. $n$ is equal to the total number of training examples used for training ( $\sum_{k=1}^{\text{W.length}} n_k$ ).

### B. Graph Partition Scheduling

Training the model ($\mathcal{M}$) on a graph partition ($g$) requires the entire graph partition to be accessible in memory to be trained efficiently. However, once graph partitions are loaded into the Python clients, they will consume much more space in memory than what they would in storage. This leads to the problem of being limited in the number of partitions that can be held in memory at a given time.

We implement a scheduling algorithm that takes into consideration the space consumption of graph partitions in memory as well as the available resources to come up with a schedule for the training process. An order of execution will be output for each partition indicating when it should be taken into memory in the training sequence. The scheduler will consider the number of clients sharing a host machine as well as the number of hosts in the JasmineGraph cluster, therein generating a schedule for each client in each host. Therefore, our scheduling algorithm (See Algorithm 3) is applicable to both a horizontally scaled as well as a vertically scaled system.

The first step of the scheduling process is the estimation of memory consumption for each of the graph partitions. The memory estimation process is explained in Section IV-C. Next, we use the memory estimation for each partition as well as the number of clients in each host to get a schedule for loading partitions into memory in such a way that training will be done quickly while ensuring as much memory is utilized at a given time. We make the assumption that the time taken to train smaller partitions is less than the time taken for large partitions. We use a best-first approach to ensure that the largest partitions get assigned to the available memory first so that a single large partition will not be left training in the end which reduces overall memory utilization and increases the time taken. This works well even for the cases of skewed graph partitions where the size have significant variance. In our partition scheduler shown in Algorithm 3, first we sort the partition memory estimation list in the descending order and then iterate through the sorted estimation list to assign partitions to a free client in memory. Then in the simulation part of the algorithm, we take the smallest partition that has finished training and we remove it from the memory, mark the corresponding client as free to train another partition. We also update the training progress of the other partitions that are already in memory (With the assumption that clients train at the same rate).

### C. Memory Estimation

Graphs consume more space when loaded into memory with data science libraries than in the storage. This is due to the memory usage of the data types used. Usually large graphs contain billions of nodes.

Therefore, numerical data types that can handle up to billions of integers has to be used. Moreover, training GCNs on graphs using machine learning frameworks add more memory overhead. Table I contains memory usage of few common graph datasets when loaded using Numpy or Pandas using `int64` data type and trained with Tensorflow 2 and StellarGraph [6] in our Python client implementation. Note that we used GraphSAGE implementation of StellarGraph to build the GCN [6].

In our memory estimation algorithm (see Algorithm 4), first we calculate the graph size in memory using the partition metadata; number of nodes ($\mathcal{N}_N$), number of edges ($\mathcal{N}_E$), and number of node features ($\mathcal{N}_F$). $m_N, m_E$ and $m_F$ are the size of numerical data types in bytes used to represent nodes, edges, and node features in the implementation. Then memory usage when training (memory usage of Python clients) can be estimated by the estimation function $\mathcal{F}$. This function should be statistically determined by analyzing the memory usage of suitable number of graphs or graph partitions. $\mathcal{F}$ may depend on the implementation and libraries or frameworks used. In our implementation we statistically found out that $\mathcal{F}$ to be nearly perfect linear function of $inMemGraphSize$ ($\mathcal{F}(inMemGraphSize) = K_1 * inMemGraphSize + K_2$ with $K_1 = 3.53$ and $K_2 = 1.58$).

## V. Testing and Evaluation

In the following subsections we describe the process of evaluation and the results gathered during the execution of the process.

### A. Experiment Environment

The evaluations were conducted on a server computer having Intel(R) Xeon(R) CPU E7-4820 v3 @ 1.90GHz, 40 CPU cores (80 hardware threads via hyper threading), 64GB RAM, 32KB L1 (d/i) cache, 256K L2 cache, and 25600K L3 cache. It had 1.8TB hard disk drive. It was running on Ubuntu Linux version 16.04 with Linux kernel 4.4.0-148-generic.

**Algorithm 3** Partition Scheduler

**Input:** Map of partition to its memory estimations - $\mathcal{L}$, List of clients - $\mathcal{C}$, Total memory capacity of host $c$
**Output:** Map of partition to scheduled order per client in host $\mathcal{S}$

$sortedMemoryList \leftarrow$ (Sort $\mathcal{L}$ in descending order)
$order \leftarrow 1$
$availableMemory \leftarrow$ (Get available memory of the host)
$freeClients \leftarrow \mathcal{C}$
$\mathcal{S} \leftarrow \{\}$
**while** $n(\mathcal{S}) < n(sortedMemoryList)$ **do**
  $proceedToNextIter \leftarrow 0$
  **foreach** *partition, memory* $(p, m) \in sortedMemoryList$ **do**
    **if** $p \notin \mathcal{S}$ *AND* $m < availableMemory$ *AND* $freeClients \neq \emptyset$ **then**
      $w \leftarrow$ (Pop client from $freeClients$)
      $\mathcal{S}[w][p] \leftarrow order$
      $availableMemory \leftarrow availableMemory - m$
      $proceedToNextIter \leftarrow 1$
      $simulatedMem[w][p] \leftarrow m$ // Client, Partition and memory of partition yet to train
    **end**
  **end**
  $order \leftarrow order + proceedToNextIter$
  $minRemainingMem \leftarrow$ (Minimum value of $simulatedMem$)
  **foreach** *client, partition, memory yet to be trained* $(w, p, r) \in simulatedMem$ **do**
    $m \leftarrow \mathcal{L}[p]$
    **if** $r$ = $minRemainingMem$ **then**
      $availableMemory \leftarrow availableMemory + m$
      Remove $(p, r)$ from $simulatedMem$;
      Insert $w$ into $freeClients$;
    **else**
      $simulatedMem[w][p] \leftarrow r - minRemainingMem$
    **end**
  **end**
**end**
return $\mathcal{S}$

TABLE I: Memory usage of graph datasets

| Dataset | Size in storage (MB) | Size in memory (MB) | Memory takes to train (GB) |
|---|---|---|---|
| Twitter [23] | 157 | 640 | 3.84 |
| DBLP-V11 [30] | 9523 | 30266 | 107.48 |

### *B. Experiment Scenarios and Datasets*

We implement the scenario of link prediction in few popular graph datasets (their details are listed in Table II). The first dataset is the DBLP-V11 citation graph which consists of papers as vertices and citations between papers as edges (e.g., If paper X cites paper Y, then this adds an edge in the citation graph) [30]. The dataset is accessible from [2] and consists of 4,107,340 papers and 36,624,464 citation relationships. The total file size is 12GB. This dataset is originally available as JSON objects. Since we require a plain edge list file to upload to the JasmineGraph database server, format conversion was done using a JSON parser. After JSON parsing the edgelist file generated was 508MB and node features file was around 10GB. With link prediction, new possible citations among papers can be predicted.

**Algorithm 4** Memory Estimation

**Input:** List of graph partitions - $\mathcal{G}$, Size of in-memory data type used in bytes - $m_E, m_N, m_F$
**Output:** Map of estimated memory usage of Python clients in GB when training each graph partition - $\mathcal{L}_\mathcal{M}$

$\mathcal{L}_\mathcal{M} \leftarrow \{\}$;
**for** *each partition* $g \in \mathcal{G}$ **do**
  $\mathcal{N}_N \leftarrow$ (Number of nodes in $g$)
  $\mathcal{N}_E \leftarrow$ (Number of edges in $g$)
  $\mathcal{N}_F \leftarrow$ (Number of node features in $g$)
  $M_N \leftarrow \mathcal{N}_N * m_N$
  $M_E \leftarrow 2 * \mathcal{N}_E * m_E$
  $M_F \leftarrow \mathcal{N}_N * \mathcal{N}_F * m_F$
  $inMemGraphSize \leftarrow (M_N + M_E + M_F)/(1024 * 1024 * 1024)$;
  $\mathcal{L}_\mathcal{M}[g] \leftarrow \mathcal{F}(inMemGraphSize)$;
**end**

The second dataset is the Twitter dataset which is a social graph that contains the information about the ego networks of a subset of Twitter users [23]. Twitter is a large and popular online social network where users can follow other users and their activities as well as post online as "tweets" which can mention other users or trending topics. More specifically, the dataset is a directed graph where it contains the information of which user (vertex) follows whom. Furthermore, each user's features indicate whether a user has interacted with another Twitter user handle or a hashtag. These feature variables are therefore boolean variables. The source dataset has a feature list per ego net of Twitter handle and hashtags the users of that ego net have mentioned in their tweets. However, these feature lists are not the same for every ego net and combining all these feature lists directly will lead to a very large set of sparse attributes for a user node. Therefore, we decided to filter and keep only the most frequently appearing features based on a selected threshold. To be more specific, if a feature in the original source dataset has appeared more than 50 times across ego nets, we keep them as a user's feature variable. This resulted in 1007 features per vertex. Our experiment use case revolves around one of the basic use cases of social networks which is user recommendation. Since users are vertices and edges indicate which user follows whom, our task of link prediction is equivalent to suggesting new users to follow or interact with.

The third dataset is the Reddit dataset which is a large online discussion forum where discussions or posts related to certain topics are contained within "subreddits" [20]. Reddit users can follow subreddits to be able to see their posts and discussions

on their timeline. Our dataset is an undirected graph dataset of social interactions which contains information about how interacting users are shared across reddit posts in different subreddits. Each node corresponds to a reddit post where an edge between two nodes indicates that the two corresponding two posts have a shared user in its comments. Moreover, the nodes contain feature which describe the textual content in the corresponding post. Similar to the other datasets, the node features are boolean and each feature variable corresponds to a word which were filtered by us based on the frequency. With this dataset experiment, we aim to solve the use case of content/post recommendation. With link prediction, users from one post can be recommended another post to their timeline from an already following subreddit or a new subreddit to which might be interesting to the user.

TABLE II: Datasets.

| Dataset name | Vertices | Edges | No of features | Edgelist File Size (MB) | Feature Files Size (MB) |
|---|---|---|---|---|---|
| Twitter [23] | 81,306 | 1,768,149 | 1007 | 16 | 157 |
| Reddit [20] | 232,965 | 11,606,919 | 602 | 145 | 270 |
| DBLP-V11 [30] | 4,107,340 | 36,624,464 | 948 | 508 | 9523 |

### C. Training Process and the GCN Model Selection

Since we are focusing on Link prediction, we developed a link prediction model with GraphSAGE [15] which can be considered as an extension to GCN [19]. The model architecture is shown in Figure 3 and it consists of two GraphSAGE layers (shown as Node Embedding Generation) with one link classification layer. The GraphSAGE layers are supposed to generate node embeddings and feed to the link classification layer as pairs.

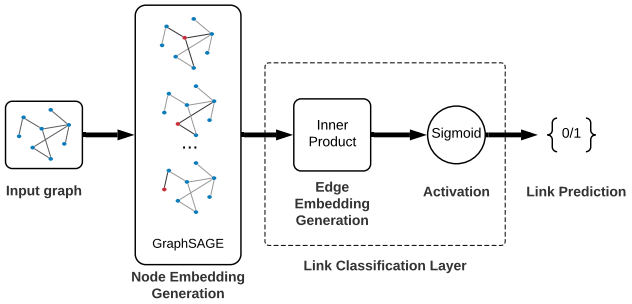


Fig. 3: ML model architecture.

Before training on a graph $G$, first we randomly sample fraction of all positive edges and same number of negative edges from $G$ to create a test set. Then we remove sample edges from $G$ and obtain reduced graph $G_{\text{test}}$. Then create evaluation and training sets in the same manner and obtain reduced graphs $G_{\text{eval}}$ and $G_{\text{train}}$. As we do the example labeling in this manner the training process should be considered as unsupervised.

Then in the training phase training examples with labels along with $G_{\text{train}}$ feed (batch size is 20) to the GraphSAGE layers and they are supposed to generate node embeddings and pass them to the link classification layer. Link classification layer is supposed to do edge embedding by getting inner product. The whole network is optimized end to end using Adam optimizer [18] with learning rate 0.01 and Binary cross entropy as the loss function.

Tensorflow 2 and Stellargraph [6] were used for the model development. Moreover, we used the same model in every experiment with every dataset because our main focus was to evaluate our graph learning system but not getting the highest accuracy possible.

### D. Experiments and Results

First, we trained and tested the link prediction model on the three datasets without our system (i.e., naive approach) to get understanding of general model performance on the datasets (See Table III). We could not train DBLP-V11 in our experiment environment because the available memory was not enough. We estimated the memory needed to train DBLP-V11 to be 107GB but the server we used had only 64GB of RAM.

TABLE III: General model performance

| Dataset | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|
| Twitter | 0.7887 | 0.9869 | 0.9576 | 0.8350 | 0.7233 |
| Reddit | 0.7174 | 0.9026 | 0.8037 | 0.7616 | 0.6587 |
| DBLP-V11 | ** | ** | ** | ** | ** |

Second, we trained and tested the link prediction model on the datasets on our system without using sheduling algorithm and assigning one partition per client parallelly to understand the effect of partitioning and federated training on evaluation metrices.

TABLE IV: Twitter - partitioned training without scheduling

| Partition count | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|
| 1 | 0.7887 | 0.9869 | 0.9576 | 0.8350 | 0.7233 |
| 2 | 0.7047 | 0.9831 | 0.9292 | 0.7700 | 0.6336 |
| 4 | 0.6395 | 0.9730 | 0.8672 | 0.7306 | 0.5861 |
| 8 | 0.6537 | 0.9844 | 0.8977 | 0.7412 | 0.5962 |
| 16 | 0.5936 | 0.9860 | 0.8441 | 0.7088 | 0.5538 |

TABLE V: Reddit - partitioned training without scheduling

| Partition count | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|
| 1 | 0.7174 | 0.9026 | 0.8037 | 0.7616 | 0.6587 |
| 2 | 0.7020 | 0.9559 | 0.8458 | 0.7625 | 0.6344 |
| 4 | 0.6836 | 0.9534 | 0.8201 | 0.7510 | 0.6202 |

We trained Twitter with partition count up to 16 and Reddit up to 4. The results are listed in Tables IV and V respectively. The Reddit dataset can not be trained parallely with more than

4 clients in 64GB of memory. As explained earlier DBLP-V11 dataset can not be trained as well.

We observed that when increasing the number of partitions the evaluation Accuracy, AUC, F1, and Precision dropped heavily in the Twitter dataset. This may be due to the information loss due to partitioning and federated training. But Reddit shows very slight amount of drop. Therefore, this may be highly dependent on the dataset.

Finally, we trained and tested the link prediction model on all three datasets using our system with the scheduling algorithm assigning multiple partitions to each client. As shown in the Table X the model was trained on Twitter dataset using clients count 2 and 4 using partition values respectively 4, 8, and 16 as well as 8 and 16. Twitter dataset was trained for 5 training rounds with each training round containing 3 epochs. Sampled train data percentage as well as the sampled test data percentages were set to 0.1 for each partition.

The model was trained on Reddit dataset (Table XI) using clients count 2 and 4 using partition values respectively 4, 8, and 16 as well as 8 and 16. Reddit dataset was trained for 5 training rounds with each training round containing 3 epochs. Sampled train data percentage as well as the sampled test data percentages were set to 0.1 for each partition. As explained above, unlike Twitter and Reddit datasets, we were unable to train DBLP-V11 without partitioning due to the limited computational resources. However, we were able to train on the DBLP-V11 graph dataset using the proposed approach.

Here we train DBLP-V11 using two clients and 16 partitions by assigning eight partitions per each client. We conduct experiments for five training rounds with each training round containing two epochs. We use the train data sampling rate as 0.01, and the test data sampling as 0.005 for each partition for the experiments on this dataset. After training with the described setup, we were able to train DBLP-V11 graphs within 20 hours and 34 minutes. By further analyzing the experiment environment, we observed that there is an unusual memory growth issue occurring during the experiments. Due to this reason, the DBLP-V11 dataset had to be trained on for two stages. During the first stage, we trained up to training round three and saved the model weights in the persistent storage. Based on the saved weights for the last training round (training round 3), we commenced the second stage and trained for another two rounds to complete the five training rounds. The results are shown in Table VI.

TABLE VI: DBLP-V11 performance

| Dataset | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|
| DBLP-V11 | 0.56529 | 0.99584 | 0.88943 | 0.69677 | 0.53630 |

As we can observe by increasing the number of partitions and clients we can greatly reduce the training time (Table VII and Table VIII). We have compared the best training times we got compared to conventional method in Table IX.

When we compare the accuracy values shown in Tables X and XI for Twitter and Reddit respectively we find that there is no significant accuracy drop when changing between partition counts.

TABLE VII: Partition count vs Execution time - Twitter.

| Client count | Partition Count | Elapsed Time (seconds) |
|---|---|---|
| 2 | 4 | 19186.66 |
| 2 | 8 | 16545.43 |
| 2 | 16 | 19575.20 |
| 4 | 8 | 11601.11 |
| 4 | 16 | 12922.13 |

TABLE VIII: Partition count vs Execution time - Reddit.

| Client count | Partition Count | Elapsed Time (seconds) |
|---|---|---|
| 2 | 4 | 23568.88 |
| 2 | 8 | 24268.64 |
| 2 | 16 | 22011.78 |
| 4 | 8 | 11505.92 |
| 4 | 16 | 15019.63 |

CPU and memory behavior of the entire system while running the experiments are highlighted in Figures 4, 5, 6, and 7. All these four charts indicate that after the use of our scheduling algorithm the memory usage efficiency increased (more free memory) while CPU usage also increased indicating the system's ability to handle large workloads.

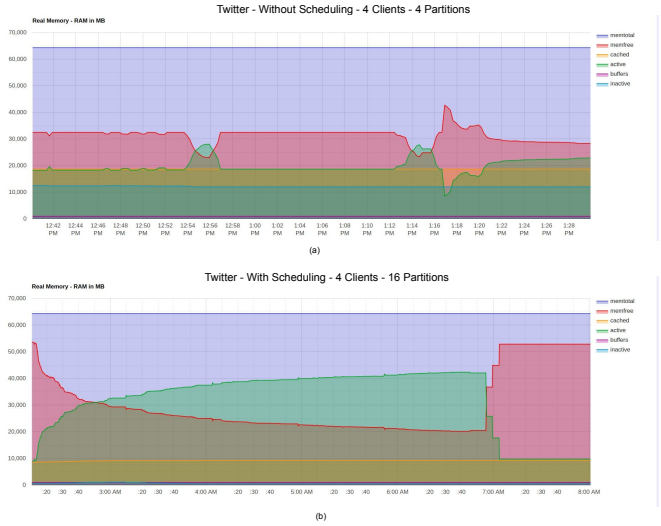


Fig. 4: Twitter - Naive vs scheduling memory consumption comparison.

## VI. Discussion

When comparing the evaluation metric values related to the test cases we do not find significant differences in terms of whether they are being partitioned or unpartitioned. But there was an unavoidable evaluation metric value loss due to the information loss that occurred during the partitioning process. However, this is an acceptable loss when compared to the considerable speedup that is achieved when increasing the parallelism of training is taken into account. Moreover, we have not spent a significant portion of time on fine tuning model or training parameters since our main focus is efficient use of system resources. Therefore, it is possible to increase

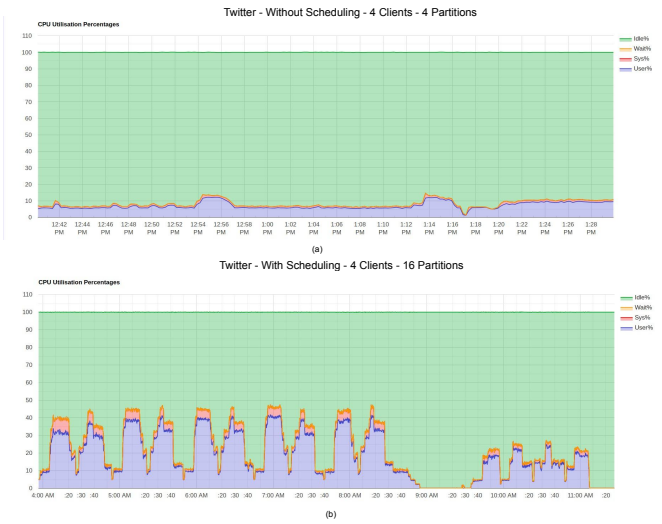


Fig. 5: Twitter - Naive vs scheduling CPU usage comparison.

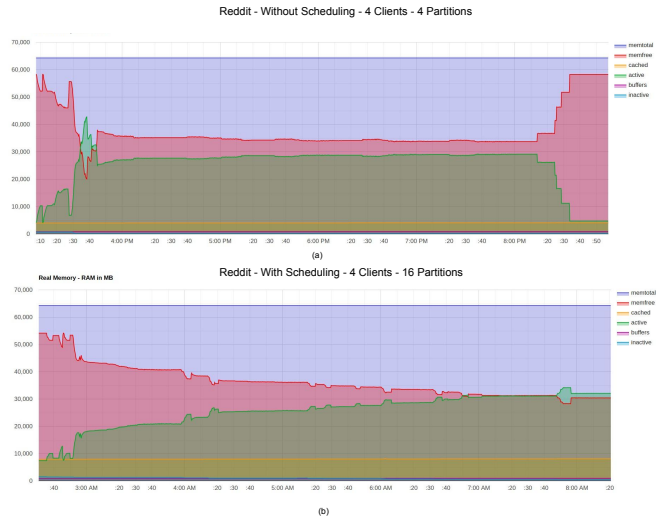


Fig. 6: Reddit - Naive vs scheduling memory consumption comparison.

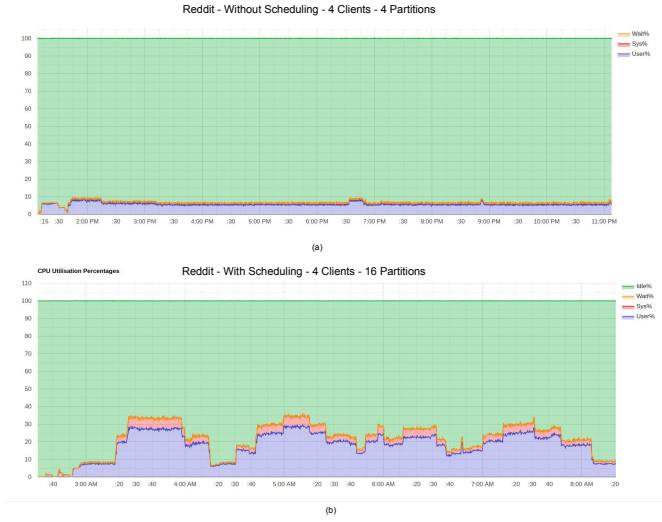


Fig. 7: Reddit - Naive vs scheduling CPU usage comparison.

TABLE IX: General model performance

| Dataset | Training time (Conventional) | Training time (With JasmineGraph) |
|---|---|---|
| Twitter | 10h 31min | 2h 6min (16 clients-16 partitions) |
| Reddit | 9h 8min | 3h 11min(4 clients-8 partitions) |

TABLE X: Twitter - partitioned training with scheduling

| Client count | Partition count | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|---|
| 2 | 4 | 0.6512 | 0.9659 | 0.8728 | 0.7352 | 0.5936 |
| 2 | 8 | 0.6042 | 0.9802 | 0.8539 | 0.7137 | 0.5620 |
| 2 | 16 | 0.6056 | 0.9957 | 0.9166 | 0.7166 | 0.5599 |
| 4 | 8 | 0.5900 | 0.9852 | 0.8656 | 0.7067 | 0.5514 |
| 4 | 16 | 0.6009 | 0.9941 | 0.8800 | 0.7142 | 0.5578 |

the performances of these models by hyperparameter tuning, increasing the number communication rounds and the number of epochs used in a communication round.

In the current setup the number of communication rounds and epochs were decided based on the experimental setup capacity. It is possible to increase those numbers by using higher computational resources. As seen in the experimental results with the DBLP-V11 dataset, we encounter a memory growth as training rounds proceed. We suspect that this growth is caused by certain Python library dependencies that exist during the training process. We believe that with the replacement of these dependencies with more primitive yet equally efficient libraries, we would be able to eliminate the memory growth. However, this does not mean that large graphs cannot be handled. Since our solution saves the model at the end of each training round, it is possible to restart the training process from where training stopped last.

Considering the training time of the graph datasets this can be reduced with the use of GPUs. In the current experiment setup models were trained using CPUs. However in the current setup there are multiple GraphSAGE models are training in parallel manner, due to this reason use of multiple GPUs is recommended. Furthermore, while capable of scaling horizontally to multiple hosts, our experiment results only speak of the performance of a vertically scaled system. This will be addressed as future work related to this research work and experiments.

## VII. Conclusion

Resource efficient machine learning based graph data processing has become a significant challenge in high performance graph data management and mining recently. In this paper we describe a resource efficient implementation of graph convolutional network (GCN) based link prediction conducted on top of an open source distributed graph database server. Our implementation is accessible from [2]. We implement our solution in such a way so that anyone can train a graph machine learning model for use cases such as node embedding,

[2]https://github.com/miyurud/jasminegraph

TABLE XI: Reddit - partitioned training with scheduling

| Client count | Partition count | Accuracy | Recall | AUC | F1 | Precision |
|---|---|---|---|---|---|---|
| 2 | 4 | 0.6927 | 0.9598 | 0.8331 | 0.7577 | 0.6262 |
| 2 | 8 | 0.6947 | 0.9545 | 0.8377 | 0.7576 | 0.6281 |
| 2 | 16 | 0.6830 | 0.9436 | 0.7753 | 0.7485 | 0.6209 |
| 4 | 8 | 0.6753 | 0.9630 | 0.7953 | 0.7479 | 0.6113 |
| 4 | 16 | 0.6271 | 0.9215 | 0.6631 | 0.7146 | 0.5878 |

link prediction, node classification, etc. written in Tensorflow in a distributed manner in a resource constrained environment. We use a graph partitioning scheme in order to divide the graph into subgraphs with minimal information loss that can be trained in parallel. Using our scheduling algorithm, we organize the partitions in such a way to ensure that the memory is not overloaded while improving efficiency of the training process. We conduct our experiments on link prediction use cases on the real world datasets from Twitter, Reddit, and DBLP-V11 utilizing the well known GraphSAGE mechanism. We show that our solution performs well while providing significant speedup. JasmineGraph was able to train a GCN on the largest dataset DBLP-V11(>10GB) in 20 hours and 24 minutes for 5 training rounds and 3 epochs by partitioning it into 16 partitions with 2 workers on a single server while the conventional training method could not process it at all due to lack of memory. The second largest dataset Reddit took 9 hours 8 minutes to train with conventional training while JasmineGraph took only 3 hours and 11 minutes in the same hardware giving 3 times improved performance. In case of Twitter dataset JasmineGraph was able to give 5 times improved performance. In future work, we plan to conduct more experiments on more real world datasets utilizing GPUs as well as horizontal scaling. Additionally, we plan to address the memory growth issue by re-evaluating the dependencies and replacing them as needed. We also hope to extend this work to the domain of privacy preserving federated learning between different organizations.

## REFERENCES


[1] Alibaba. Euler. URL: https://github.com/alibaba/euler, 2019.

[2] AMiner. Citation network dataset. URL: https://aminer.org/citation, 2019.

[3] Apache Software Foundation. Janusgraph. URL: https://janusgraph.org/, 2020.

[4] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[5] I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy. Machine learning on graphs: A model and comprehensive taxonomy, 2020.

[6] C. Data61. Stellargraph machine learning library. https://github.com/stellargraph/stellargraph, 2018.

[7] M. Dayarathna, S. Bandara, N. Jayamaha, M. Herath, A. Madhushan, S. Jayasena, and T. Suzumura. An x10-based distributed streaming graph database engine. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 243–252, Dec 2017.

[8] M. Dayarathna, I. Herath, Y. Dewmini, G. Mettananda, S. Nandasiri, S. Jayasena, and T. Suzumura. Acacia-rdf: An x10-based scalable distributed rdf graph database engine. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 521–528, 2016.

[9] M. Dayarathna, C. Houngkaew, H. Ogata, and T. Suzumura. Scalable performance of scalegraph for large scale graph analysis. In *2012 19th International Conference on High Performance Computing*, pages 1–9, Dec 2012.

[10] M. Dayarathna and T. Suzumura. Towards scalable distributed graph database engine for hybrid clouds. In *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 1–8, Nov 2014.

[11] M. Dayarathna and T. Suzumura. *High-Performance Graph Data Management and Mining in Cloud Environments with X10*, pages 173–210. Springer International Publishing, Cham, 2017.

[12] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2, July 2005.

[13] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, pages 1–8, 2005.

[14] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[15] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[16] A. Karunarathna, D. Senarath, A. Madhushanki, C. Weerakkody, M. Dayarathna, S. Jayasena, and T. Suzumura. Scalable graph convolutional network based link prediction on a distributed graph database server. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 107–115, 2020.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[18] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[19] T. N. Kipf and M. Welling. 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*, 2017.

[20] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 933–943, 2018.

[21] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019.

[22] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 556–559, New York, NY, USA, 2003. ACM.

[23] J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. NIPS'12, pages 539–547, USA, 2012. Curran Associates Inc.

[24] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[25] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM.

[26] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009.

[27] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, Sept. 2017.

[28] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[29] X. Shen, S. Pan, W. Liu, Y.-S. Ong, and Q.-S. Sun. Discrete network embedding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 3549–3555, 7 2018.

[30] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: Extraction and mining of academic social networks. In *KDD'08*, pages 990–998, 2008.