A STUDY OF LEARNING KNOWLEDGE GRAPH EMBEDDINGS AT LARGE SCALE

Harit Vishwakarma Suryadev Sahadevan Rajesh Vinith Venkatesan

1 INTRODUCTION

Knowledge graphs (KG) are used to represent entities and relationships among them. Apart from the explicit information present in a KG, it contains a wealth of information which can be inferred from it. It makes them of great interest and importance to various applications. In recent years, by using the deluge of data on web, industries try to capture this data in a knowledge graph to make their products intelligent. For example, Google's knowledge graph contains 70 billion assertions and nearly 1 billion entities, and they use the knowledge graph to interpret the semantics of the user's request and respond with the results that are related to the concept (Noy et al., 2019). Also, online retailers use knowledge graphs on their products to identify relationships between two products and recommend them to the users. Apart from these, there are several knowledge graphs publicly available for use such as FreeBase (Bollacker et al., 2008), DBpedia (Auer et al., 2007) for development and experiment purposes.

To work with these knowledge graphs, the application should be able to understand the relations between two entities in the graph. For this purpose, embeddings are used to embed the entities and relations in the graph into vector space so that the application can use them identify to relations between two entities such as link prediction, knowledge completion, entity classification, and further custom operations for knowledge inferences (Wang et al., 2017). Also, it is used by recommender systems and other machine learning applications to work with the categorical data efficiently. To compute the embeddings, several methods are available including TransE (Bordes et al., 2013), RESCAL (Nickel et al., 2011), and ComplEx (Trouillon et al., 2016). However, in most cases, knowledge graphs are too large to handle using the resources in a single machine. To process more than a billion nodes with a trillion edges in a shorter time-interval and with a single machine is a challenge.

There are several approaches to compute the graph embeddings in a distributed manner that have been proposed recently. For computing the embedding in parallel, one approach (Duong et al., 2019) is to split the graph into subgraphs and compute the embeddings for these subgraphs and reconcile these embeddings for the large graph. Also, Facebook has proposed a framework called PyTorch-BigGraph (Lerer et al., 2019) to compute the embeddings by distributed execution of these computations, and they partition the graph by nodes. Though there were several approaches proposed, they will differ in terms of resource usage, accuracy and time taken to execute the computations.

The methodology in which we partition the graph plays a major role in resource consumption and the accuracy of the results. Besides, there is no single way of partitioning that works well for all types of graphs. For example, a graph with a community structure can be partitioned between non-overlapping communities to minimize communication between workers to reduce network usage. Computing the knowledge graph embeddings is a resource-intensive task. Especially, if parameter servers are used, as in the case of BigGraph, then it requires lots of network bandwidth for parameter communication, and more memory usage for holding them. Also, in the knowledge graph new knowledge will be learned as time goes and these graphs will get updated frequently with new knowledge. This will force us to run these computations periodically to keep our products using the up-to-date graphs, and it becomes important for us to study these approaches from a systems perspective.

In this project, we will do an in-depth analysis of the workload properties and measure the performance metrics with different datasets. First, we will contribute by evaluating the accuracy and resource usage of different graph partitioning approaches for learning the knowledge graph embeddings. This will give us more insights into these approaches, and we can provide some improvements to the architecture or the methods followed in computing the graph embeddings in parallel. By using this data, we can suggest some modifications or improvements for these approaches leverage these hardware better. We feel this study could form as a basis for new improvements that can be made in computing the graph embedding efficiently in a distributed environment.

2 RELATED WORK

Pytorch-BigGraph (PBG) for learning knowledge graph embeddings at very large scale has been proposed recently (Lerer et al., 2019). It employs graph-partitioning and parameter server architecture for distributed training. Graph

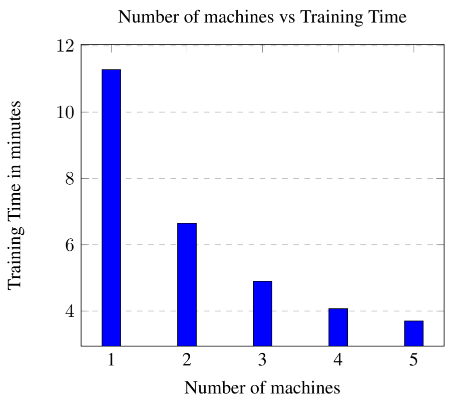

Figure 1. Speed up achieved by scaling up the training

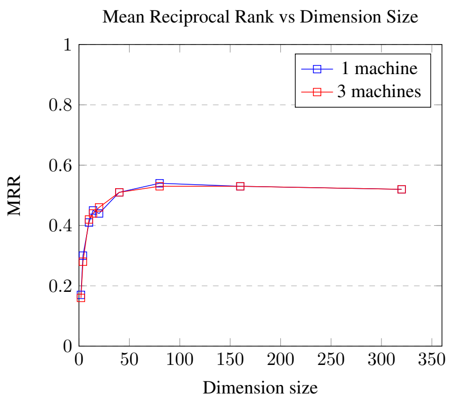

Figure 2. Variation in MRR (Mean Reciprocal Rank) when dimension size is increased

partitioning is done by randomly dividing the graph into partitions, and the edges are placed in the bucket based on its source and destination vertex's partition.

In another recent work (Duong et al., 2019) an algorithm for parallel computation of graph embeddings is proposed. In this paper the graph partitioning respects resource capacity constraints, (particularly memory constraints). Since each node learns embeddings for a subgraph, the embeddings learned at each node may be in different spaces. Which can compromise the quality of embeddings. So they select a few *anchor* nodes which are shared by all subgraphs and using these anchors the embeddings of subgraphs are reconciled. Moreover, they measure the quality of embeddings in terms of Pairwise Inner Product (PIP) and based on this give a bound on the difference in quality of embeddings learned in parallel and centralized systems.

In a similar approach to the above work, MILE (Liang et al., 2018) repeatedly coarsens the large graph and uses the existing embedding mechanism to compute the embedding for the coarsest graph. Further, it refines the embedding of the coarsest graph to the original graph using a method based on graph convolution neural network.

All of these work propose an approach to compute the graph embedding for large graph which cannot fit in a single machine, and they focus more on accuracy of the embedding values that are computed using these distributed execution. However, there was a less focus on system profiling and in-depth analysis of performance metrics.

3 Workload Characteristics

In this section, we discuss about how the quality of learning knowledge graph embeddings and the training time for it are affected by varying different parameters of the workload. For this we used the FB15k dataset (Bollacker et al., 2008) and Pytorch-BigGraph framework (Lerer et al., 2019) for our experimentation.

3.1 Quality of embedding increases when the dimension size increases up to a certain point.

The embedding dimension is a parameter that denotes the number of dimensions used to represent the entities as embeddings. The MRR (Mean Reciprocal Rank) is a measure of the relevance of the results produced during information retrieval.

From the figure 2, we could observe that the MRR (Mean Reciprocal Rank) increases as the size of the dimension increases, and the increase saturates after a certain dimension size. So it is important to find the right dimension size when training a specific knowledge-graph. If the higher dimension is chosen, more than what is optimal, then the system will be spending more time on compute to learn the embeddings, although it does not provide benefits in terms of quality.

3.2 Embedding quality is not affected greatly by increasing the number of machines

It is shown in the figure 2 that for the experiments with the 1 machine and 3 machines the trend and the values of MRR does not differ significantly with respect to increase in the dimension size. Similar to the single machine learning, for the 3 machines the MRR increases and saturates almost nearly at the same point and also overlaps with the plot of the single machine plot. We could infer that the embedding quality is not degraded or improved greatly by distributed training.

3.3 Increasing the number of machines reduces the training time.

For some of the workloads the single threaded application performs better (McSherry et al., 2015). However, for the

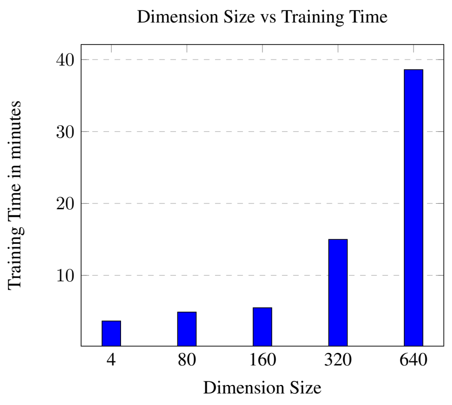

Figure 3. Dimension size's influence on Training Time

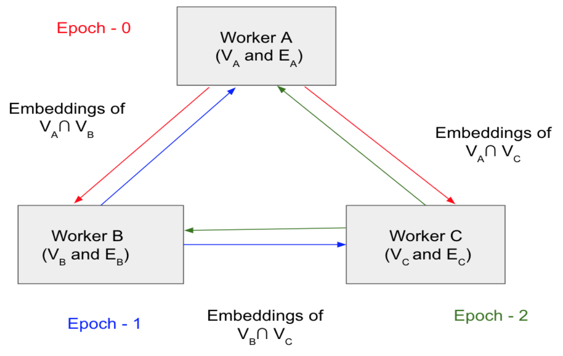

Figure 4. System design of the prototype for evaluation

knowledge graph embedding learning, we could observe (Figure 1) that the increase in the number of machines reduces the training time, as they are able to work on learning the embeddings of different partitions of the graph in parallel.

3.4 Increasing the dimension size increases the training time.

Previously, we discussed how the MRR increases by increasing the dimension size up to a certain point. In this section, we could infer from the figure 3 that the increase in dimension size increases the time it takes to learn the embeddings, as it requires additional system resources to compute the other values in the dimensions. Though the increase embedding provides a good MRR till a certain point, it comes with an additional cost. Hence, it will be good idea to learn the embeddings with large dimensions in distributed manner to train them faster.

Though some of the observation can be trivial, they emphasize that we could learn that the distributed learning of the embedding is helpful as they provide speed-up and more computational power to learn the higher dimensional embedding without degradation in the quality. However, for distributed learning, partitioning the graph plays an important role and with different methods of partitioning available it can influence the system resource usage, training time and time to converge. We will discuss about this in detail in the next section.

4 System Design

We implemented a simple prototype to study the performance of different partitions in PyTorch. The prototype is implemented in such a way that partitioning module is decoupled with the rest of the system so that it is easier to experiment with different partitioning algorithm. We also used the OpenKE (Han et al., 2018) module for leveraging different models present in it. For our experiments, we used the TransE (Bordes et al., 2013) model for evaluation.

4.1 Worker nodes

Each worker learns the embedding for one of the partition present in the shared file system. The worker uses the specified model to learn these embedding, and in our case, it is TransE. The worker has the knowledge of the vertices that are processed by other worker by reading the meta data from the shared file system. They communicate the embeddings to the other worker if they are learning embeddings for any common vertices (Refer Figure 4). The worker updates its embeddings by taking the average with the received embeddings. The communication between the workers happens at the epoch boundary.

4.2 Communication protocol

If all workers communicate with all the other workers at the epoch boundary, then there will be more network usage, and it can become a bottleneck. In addition, most of the learned embeddings in the previous epoch gets affected because of the new embeddings received from all the other workers. To overcome these problems, in our setup, only one of the worker communicates with all the other workers at the epoch boundary. As of now, this protocol works good when there are smaller number of workers.

However, in the future, we can extend this protocol by only subset of workers communicating with all the other workers at the epoch boundary instead of single worker, when the number of workers is larger than the number of epochs we have configured.

4.3 Partitioning Algorithms

In distributed training of knowledge graphs embeddings, the partitioning approach used will have a major impact on the performance. In our prototype, we implemented the following three partitioning approaches, similar to the ideas discussed in (Gonzalez et al., 2012) and (Devine et al., 2006), which were then used in our experiments.

4.3.1 Random Edge-Cut

In edge-cut partitioning, vertices are assigned to different partitions based on some heuristic. A common method to do this is the balanced p-way edge-cut where vertices are evenly assigned to partitions and the number of edges spanning partitions are minimized. In the random approach which we used, vertices are randomly assigned to partitions without considering edge distribution in any way. The algorithm for the random edge-cut partitioning is presented in Algorithm 1

Algorithm 1 Random Edge-Cut

```
1: for $edge\ in\ edges$ do
2:     $vertexMap[edge.src] \leftarrow edge$
3:     $vertexMap[edge.dst] \leftarrow edge$
4: for $vertex,\ edgeList\ in\ vertexMap$ do
5:     $randPart \leftarrow\ random(numParts)$
6:     for $edge\ in\ edgeList$ do
7:         $partitions[randPart] \leftarrow edge$
```

4.3.2 Random Vertex-Cut

In vertex-cut partitioning, edges are assigned to different partitions based on some heuristic. A common method to do this is the balance p-way vertex-cut where edges are evenly assigned to partitions and the number of replica vertices spanning partitions are minimized. In the random approach which we used, edges are randomly assigned to partitions with an equal distribution. The algorithm for the random vertex-cut partitioning is presented in Algorithm 2

Algorithm 2 Random Vertex-Cut

```
1: $edges \leftarrow permutation(edges)$
2: $eCount \leftarrow len(edges) \div numParts$
3: for $i = 0, 2, \ldots numParts - 1$ do
4:     for $j = i \times eCount, \ldots (i+1) \times eCount$ do
5:         $partitions[i] \leftarrow edges[j]$
```

4.3.3 Greedy load-aware Vertex-Cut

In contrast to random edge-cut and random vertex-cut partitioning algorithms which do not use any particular heuristic when partitioning, the greedy load-aware vertex-cut uses a good heuristic to distribute the load evenly across partitions. This approach tries to assign edges into the least-loaded partitions which already contains both the vertices of the edges. This in turn naturally tries to reduces the number of vertex replicas. When there are no common partitions between the vertices of an edge to which it is already assigned to, then we try to assign the edge to the least-loaded partition with at least one of the vertices already assigned to it. The algorithm for the greedy vertex-cut partitioning is presented in Algorithm 3

Algorithm 3 Greedy Load-Aware Vertex Cut

```
1: $edges \leftarrow permutation(edges)$
2: for $edge\ in\ edges$ do
3:     $eCount[edge.src] \mathrel{+}= 1$
4:     $eCount[edge.dst] \mathrel{+}= 1$
5: for $edge\ in\ edges$ do
6:     $src \leftarrow edge.src$
7:     $dst \leftarrow edge.dst$
8:     $srcParts \leftarrow vertexParts[src]$
9:     $dstParts \leftarrow vertexParts[dst]$
10:    $edgeParts \leftarrow srcParts \cap dstParts$
11:    if $edgeParts \neq \emptyset$ then
12:        $part \leftarrow edgeParts.pop()$
13:    else
14:        if $srcParts = \emptyset \wedge dstParts = \emptyset$ then
15:            $part \leftarrow min(partitions)$
16:        else if $srcParts \neq \emptyset \wedge dstParts \neq \emptyset$ then
17:            if $eCount[src]eCount[dst]$ then
18:                $part \leftarrow min(srcParts)$
19:            else
20:                $part \leftarrow min(dstParts)$
21:        else
22:            if $srcParts \neq \emptyset$ then
23:                $part \leftarrow min(srcParts)$
24:            else
25:                $part \leftarrow min(dstParts)$
26:    $partitions[part] \leftarrow edge$
27:    $eCount[src] \mathrel{-}= 1$
28:    $eCount[dst] \mathrel{-}= 1$
29:    $vertexParts[src] \leftarrow part$
30:    $vertexParts[dst] \leftarrow part$
```

5 EVALUATION

In this section, we will discuss about the results that we got from our experiments with the partition algorithms discussed in the previous section.

5.1 Experiment Setup

For the evaluation, we used cluster of 8 machines, and each machine with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz with 5 cores and memory of 32 GB. To do the experiments, we used both synthetic and real-world dataset. We used networkx package in python to generate graphs based on power-law model and stochastic block model for generating graphs with community structures. For real-world graphs, we used the Amazon product co-purchasing network from the Stanford Large Network Dataset Collection (sta).

The table 2 shows the dataset description that we used for our experiments. We can observe that the power-law based graph has more number of edges compared to the given number of vertices.

Dataset Description			
Name	Type	Vertices	Edges
Amazon-co-purchased	Real-world	26211	1234877
Power-law Based	Synthetic	20000	1986667
SBM Based	Synthetic	20000	1466482

Table 1. Description of Dataset used for the experiments

5.2 Discussion

For the below inferences, we conducted experiments with 8 partitions and embedding dimension as 10, and total number of epochs as 10.

5.2.1 *Distribution of edges across partitions varies based on the algorithm.*

In the table 2, you can observe that the distribution of edges across partitions varies for different algorithm. For the edge-cut, you can observe that there are more number of partitioned edges compared to the actual number. This is due to the duplicate vertices that are created in each partition because of the random placement. However, for the vertex-cut, you can find that the number of partitioned edges is nearly equal to the actual total. This helps in reduced computation in each machine as they have equal and lesser number of edges to compute the embeddings compared to the edge-cut.

5.2.2 *Training with Greedy load-aware Vertex-Cut executes slightly faster compared to the other partitioning algorithm in some cases.*

From the table 3, we can see that the greedy load-aware vertex-cut makes the training execute slightly faster compared to the random vertex-cut. Also, we could see that the random vertex-cut execute faster compared to random edge-cut partitioning algorithm.

The random vertex-cut is able to execute faster compared to the random edge-cut because it avoids the duplication of vertices, and it saves computational time. In the Greedy load-aware vertex cut, the savings come from the network communication because of the optimal placement of edges among the partitions. However, we found that for the other datasets the greedy vertex-cut executes nearly equal to the random vertex-cut, and both execute much better than the random edge-cut.

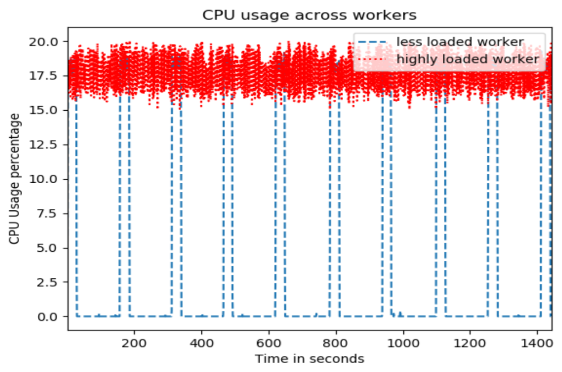

Figure 5. CPU usage difference because of skewed partition

5.2.3 *Skewed partition cause wastage of resources in the cluster.*

If the partition algorithm does not perform better distribution of edges, then it affects the training time and wastage of system resources. The figure 5 shows the CPU usage of two machines from the cluster which were learning embeddings for the entities in the power-law based graph. The partition algorithm that was used for this experiment was greedy algorithm without any load-awareness. We could notice from the figure that the less loaded machine was made to wait for the highly loaded machine to complete to begin the next epoch. However, if the partitioning was done evenly, then these idle CPU time could have been used for computation and made the training to complete faster.

5.2.4 *Greedy algorithm has lesser network communication compared to the other algorithms.*

Greedy algorithm has better placement compared to the other partition algorithm. A partition is preferred, if the edge's vertices are already present in the set of vertices the partition holds. As the intersection of vertices between the partitions are communicated, only less number of data are communicated between the machines for the data partitioned using the greedy algorithm. Refer figure 6.

5.2.5 *SBM based graphs, which has community structure, has lesser network communication compared to the other dataset*

In the figure 7, we compared our greedy load-aware implementation with amazon co-product dataset and SBM based dataset. We could observe that training the SBM based data set has lesser network communication compared to the other dataset. As the SBM based dataset has community structure,

Edges distribution across partition (Total Edges = 1234877)								
Partition Algorithm	Part - 0	Part - 1	Part - 2	Part - 3	Part - 4	Part - 5	Part - 6	Part - 7
Random Edge-Cut	225764	227463	227069	223854	223302	224495	221797	225840
Greedy load-aware Vertex-cut	158331	156199	154996	154691	153352	152828	152322	152158
Random Vertex-Cut	154359	154359	154359	154359	154359	154359	154359	154359

Table 2. Edges distribution across partitions based on different algorithm for Amazon co-product dataset

Training Time	
Partitioning Algorithm	Training Time (min)
Greedy Load-Aware Vertex-Cut	6.73
Random Vertex-Cut	8.78
Random Edge-Cut	11.07

Table 3. Preliminary results of training time on the partitioned data using different algorithms (Amazon co-product dataset)

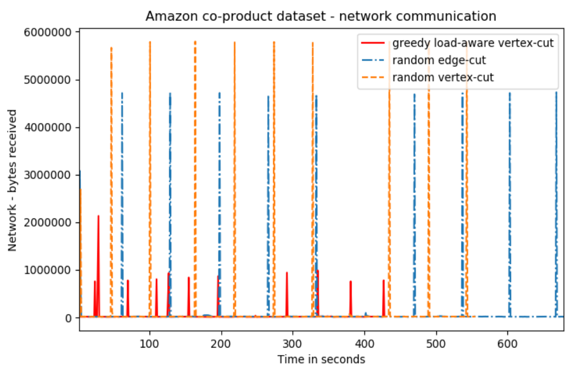

Figure 6. Comparison of network usage for different partition algorithm

greedy's partition algorithm does better placement of edges, by minimizing the number of common vertices across the partitions which reduces the network communication. We could infer from this experiment that graph structure plays an important role in the performance of the knowledge graph embedding learning.

6 Embedding Quality Evaluation

As discussed earlier, with distributed training using different graph partitioning algorithms we are able to uniformly distribute the workload and reduce the training time significantly. A natural question is, are we loosing out on embedding quality while doing distributed training? To understand the effect on quality we evaluate the embeddings on link prediction task for power-law and SBM graphs. Power-law graph has 1000 nodes and 4974 edges, SBM graph has 1000 nodes and 3689 edges. Please refer appendix for details of parameters used for generating these graphs. The edges are split into training (80%), validation (10%) and test (10%) sets. We partition the training set using Load aware Greedy Vertex Cut (LGVC), Random Vertex Cut (RVC) and Random Edge Cut (REC). For each algorithm we set number of partitions = 3.

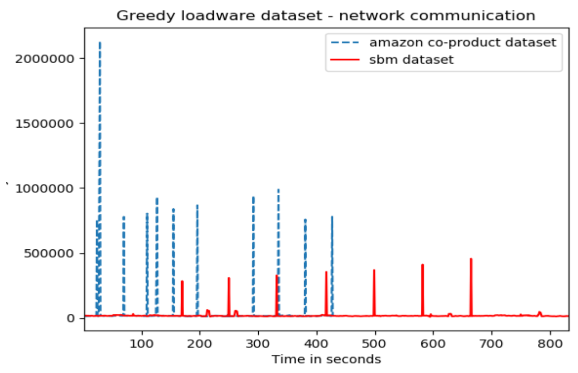

Figure 7. Comparison of network usage (bytes received) for different data-sets for the greedy load-aware algorithm

Method	Partitioning	MRR	Hits@10	Train Time
Dist. TransE	LGVC	0.0518	0.1338	47s
	RVC	0.0559	0.1257	47s
	REC	0.0857	0.2002	71s
Seq. TransE	–	0.0168	0.0291	65s
Random Emb.	–	0.0063	0.0100	0s

Table 4. Results on PowerLawGraph

Method	Partitioning	MRR	Hits@10	Train Time
Dist. TransE	LGVC	0.0228	0.0433	36s
	RVC	0.0312	0.0582	39s
	REC	0.0148	0.0216	56s
Seq. TransE	–	0.0184	0.0271	51s
Random Emb.	–	0.0067	0.0121	0s

Table 5. Results on SBM Graph

Next, we train distributed TransE method on the above synthetic training sets using the system and communication protocol mentioned in section 4. We observe the Mean Reciprocal Rank (MRR) and Hits@10 on the test data and total training time for the sequential training (centralized training) and distributed training with different partitioning methods. We run three worker processes (one for each partition), each worker uses the same hyper-parameters as used for sequential training. (Details of hyper-parameters are in Appendix). Results for power-law and SBM graphs are presented in Table 4 and Table 5 respectively.

For both graph types we observe, the quality of embeddings learned using distributed training (DT) is much better than the sequential (ST) method. We expected the quality of DT to be comparable to ST, but these results are surprising and we are interested in doing a thorough evaluation to establish this result. Given this, we can at least conclude that the embeddings of DT are giving reasonable performance on power-law graph.

At the same time we also observe that DT with LGVC and RVC partitioning takes significantly less training time than the sequential method and REC takes more training time than sequential method. Further, the time taken by LGVC is less than or equal to the time taken by RVC. This is due to uniform partitioning of edges and vertices in LGVC. Edge-Cuts take more time since they end up replicating lot of edges and almost equivalent to sequential training with added burden of synchronization.

Although DT does better than ST for SBM graph as well, the absolute numbers obtained by both of them are not good enough, indicating TransE itself could be limited in learning representations for SBM graphs. To the best of our knowledge, TransE has been evaluated on real-graphs only (mostly power-law graphs) and there are no studies or embedding methods for graphs following certain probabilistic models such as SBM. Doing such evaluation and developing appropriate methods adhering the underlying probabilistic model is another interesting problem which is out of scope for this study.

7 Conclusion and Future Work

Learning KG embeddings is an interesting problem because of its diverse applications. However, most real graphs are very large and absence of efficient algorithms and systems becomes a bottleneck in fully utilizing the potential of knowledge graph embeddings. In this work, we have evaluated the scalability aspects of this problem and proposed a new system with different graph partitioning methods and an efficient communication protocol for embedding reconciliation across workers. We evaluated partitioning methods on graphs with different structures and how does it affect the training time and quality of embeddings. The results are not thorough but indicate the use of load-aware graph partitioning algorithm is useful and also distributed training using our system is helpful. We also observed distributed training gets better embedding quality than sequential ones, and we need to establish this with thorough experiments. Aside from training, inference in large scale knowledge graphs is a challenging problem. For instance, amazon co-product graph with 1.2M edges took around 1s latency for one prediction when the embeddings of all nodes were in single machine. In addition, inference is even more challenging when the embeddings have to be sharded across machines which calls for novel serving systems and methods.

8 Appendix

Following are the parameters used for generating synthetic graphs, (please refer to networkx documentation for details about these parameters)

Parameters for SBM graph:
number of vertices = 1000
number of blocks = 3
intra-block edge probability = 0.01
inter-block edge probability = 0.001

Parameters for Power-law graph:
number of vertices = 1000
random edges per vertex = 5
triagle probability = 0.01

hyper-parameters used for Dist. and Seq. TransE :
number of epochs = 30
embedding size = 25
Margin = 5.0
Negative Samples size = 25
distance = ℓ_2

References

Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International The Semantic Web and 2Nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, pp. 722–735, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76297-3, 978-3-540-76297-3. URL http://dl.acm.org/citation.cfm?id=1785162.1785216.

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pp. 1247–1250, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376746. URL http://doi.acm.org/10.1145/1376616.1376746.

Bordes, A., Usunier, N., Garcia-Durán, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pp. 2787–2795, USA, 2013. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999792.2999923.

Devine, K. D., Boman, E. G., Heaphy, R. T., Bisseling, R. H., and Catalyurek, U. V. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pp. 124–124, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898953.1899056.

Duong, C. T., Yin, H., Hoang, T. D., Ba, T. G. L., Weidlich, M., Nguyen, Q. V. H., and Aberer, K. Parallel computation of graph embeddings, 2019.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 17–30, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387883.

Han, X., Cao, S., Xin, L., Lin, Y., Liu, Z., Sun, M., and Li, J. Openke: An open toolkit for knowledge embedding. In *Proceedings of EMNLP*, 2018.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., and Peysakhovich, A. Pytorch-biggraph: A large-scale graph embedding system. In *SysML*, 2019.

Liang, J., Gurukar, S., and Parthasarathy, S. MILE: A multi-level framework for scalable graph embedding. *CoRR*, abs/1802.09612, 2018. URL http://arxiv.org/abs/1802.09612.

McSherry, F., Isard, M., and Murray, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pp. 14–14, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831104.

Nickel, M., Tresp, V., and Kriegel, H.-P. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pp. 809–816, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5. URL http://dl.acm.org/citation.cfm?id=3104482.3104584.

Noy, N., Gao, Y., Jain, A., Narayanan, A., Patterson, A., and Taylor, J. Industry-scale knowledge graphs: Lessons and challenges. *Queue*, 17(2):20:48–20:75, April 2019. ISSN 1542-7730. doi: 10.1145/3329781.3332266. URL http://doi.acm.org/10.1145/3329781.3332266.

Trouillon, T., Welbl, J., Riedel, S., Gaussier, E., and Bouchard, G. Complex embeddings for simple link prediction. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 2071–2080. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045609.

Wang, Q., Mao, Z., Wang, B., and Guo, L. Knowledge graph embedding: A survey of approaches and applications. *IEEE Trans. Knowl. Data Eng.*, 29(12):2724–2743, 2017. URL http://dblp.uni-trier.de/db/journals/tkde/tkde29.html#WangMWG17.