Proceedings of the Fifteenth International AAAI Conference on Web and Social Media (ICWSM 2021)

MILE: A Multi-Level Framework for Scalable Graph Embedding

Jiongqian Liang [*†], **Saket Gurukar** [*], **Srinivasan Parthasarathy**
Department of Computer Science and Engineering, The Ohio State University
liang.420@osu.edu, gurukar.1@osu.edu, srini@cse.ohio-state.edu

Abstract

Recently there has been a surge of interest in designing graph embedding methods. Few, if any, can scale to a large-sized graph with millions of nodes due to both computational complexity and memory requirements. In this paper, we relax this limitation by introducing the MultI-Level Embedding (MILE) framework – a generic methodology allowing contemporary graph embedding methods to scale to large graphs. MILE repeatedly coarsens the graph into smaller ones using a hybrid matching technique to maintain the backbone structure of the graph. It then applies existing embedding methods on the coarsest graph and refines the embeddings to the original graph through a graph convolution neural network that it learns. The proposed MILE framework is agnostic to the underlying graph embedding techniques and can be applied to many existing graph embedding methods without modifying them. We employ our framework on several popular graph embedding techniques and conduct embedding for real-world graphs. Experimental results on five large-scale datasets demonstrate that MILE significantly boosts the speed (order of magnitude) of graph embedding while generating embeddings of better quality, for the task of node classification. MILE can comfortably scale to a graph with 9 million nodes and 40 million edges, on which existing methods run out of memory or take too long to compute on a modern workstation. Our code and data are publicly available with detailed instructions for adding new base embedding methods: https://github.com/jiongqian/MILE.

Introduction

In recent years, *network embedding* has attracted much interest due to its broad applicability for a range of tasks such as social influence prediction (Qiu et al. 2018b), network role discovery(Rossi and Ahmed 2014), and social recommender systems (Wu et al. 2018). While these new embedding methods often offer a competitive qualitative advantage over traditional approaches, many of them do not scale to large datasets (e.g., graphs with over 1 million nodes) since they are computationally expensive and often memory intensive. For example, random-walk-based embedding techniques such as DeepWalk (Perozzi, Al-Rfou, and Skiena 2014) and Node2Vec (Grover and Leskovec 2016), require a large amount of CPU time to generate a sufficient number of walks and train the embedding model. It takes over 10 hours for a single machine to achieve quality embeddings on a graph with a million nodes using such methods. To the best of our knowledge, none of the existing efforts examines how to scale up graph embedding in a **generic** way. We make the first attempt to close this gap. We are also interested in the related question of whether the quality of such embeddings can be improved along the way. Specifically, we ask:

1. Can we scale up the existing embedding techniques in an agnostic manner so that they can be directly applied to larger datasets?
2. Can the quality of such embedding methods be strengthened by incorporating the holistic view of the graph?

To tackle these problems, we propose a MultI-Level Embedding (MILE) framework for graph embedding. Our approach relies on a three-step process: **first**, we repeatedly coarsen the original graph into smaller ones by employing a hybrid matching strategy; **second**, we compute the embeddings on the coarsest graph using an existing embedding technique - note that graph embedding on the coarsest graph is inexpensive to compute and utilizes far less memory, and moreover intuitively can capture the global structure of the original graph (Karypis and Kumar 1998b; Satuluri and Parthasarathy 2009); and **third**, we propose a novel refinement model based on learning a graph convolution network to refine the embeddings from the coarsest graph to the original graph – learning a graph convolution network allows us to attain a refinement procedure that levers the dependencies inherent to the graph structure and the embedding method of choice. To summarize, we find:

- MILE is generalizable: Our MILE framework is agnostic to the underlying graph embedding techniques and treats them as black boxes.
- MILE is scalable: MILE can *significantly improve the scalability of the embedding methods* (**up to 30-fold**), by reducing the running time and memory consumption.
- MILE generates high-quality embeddings: In many cases, we find that the quality of embeddings improves by levering MILE (in some cases is in excess of 10%).

[*]Equal contribution
[†]Now at Google

Related Work

Network Embedding: Many techniques for graph or network embedding have been proposed in recent years. DeepWalk and Node2Vec generate truncated random walks on graphs and apply the Skip Gram by treating the walks as sentences (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016). LINE learns the node embeddings by preserving the first-order and second-order proximities (Tang et al. 2015). Following LINE, SDNE leverages deep neural networks to capture the highly non-linear structure (Wang, Cui, and Zhu 2016). Other methods construct a particular objective matrix and use matrix factorization techniques to generate embeddings, e.g., GraRep (Cao, Lu, and Xu 2015) and NetMF (Qiu et al. 2018a). This also led to the proliferation of network embedding methods for information-rich graphs, including heterogeneous information networks (Chang et al. 2015; Dong et al. 2017) and attributed graphs (Liang et al. 2018; Kipf and Welling 2017).

There are very few efforts, focusing on the scalability of network embedding (Yang et al. 2017; Huang, Li, and Hu 2017). First, such efforts are specific to a particular embedding strategy and do not generalize. Second, the scalability of such efforts is limited to moderately sized datasets. Finally, and notably, these efforts at scalability are actually orthogonal to our strategy and can potentially be employed along with our efforts to afford even greater speedup.

The closest work to this paper is HARP (Chen et al. 2018), which proposes a hierarchical paradigm for graph embedding based on iterative learning methods (e.g., DeepWalk and Node2Vec). However, HARP focuses on improving the quality of embeddings by using the learned embeddings from the previous level as the initialized embeddings for the next level, which introduces a huge computational overhead. Moreover, it is not immediately obvious how a HARP like a methodology would be extended to other graph embedding techniques (e.g., GraRep and NetMF) in an agnostic manner since such an approach would necessarily require one to modify the embedding methods to preset their initialized embeddings. In this paper, we focus on designing a general-purpose framework to scale up embedding methods treating them as black boxes.

Multi-level Community Detection: The multi-level approach has been widely studied for efficient community detection (Karypis and Kumar 1998b; Satuluri and Parthasarathy 2009; Dhillon, Guan, and Kulis 2007; Ruan et al. 2015). The key idea of these multi-level algorithms is to coarsen the original graph into a much smaller one, which is then partitioned into clusters. The partitions are then recovered from the coarse-grained graph to the original graph in a recursive manner. While our framework shares some ideas at a conceptual level with such efforts, the objectives are distinct in that we focus on graph embeddings while these methods work on graph partitioning and community discovery.

Problem Formulation

Let $\mathcal{G} = (V, E)$ be the input graph where V and E are respectively the node set and edge set. Let A be the adjacency matrix of the graph and we assume $\mathcal{G}$ is undirected, though our problem can be easily extended (Chung 2005; Gleich 2006; Satuluri and Parthasarathy 2009) to a directed graph. Table 1 shows the table of notations. We first define graph embedding:

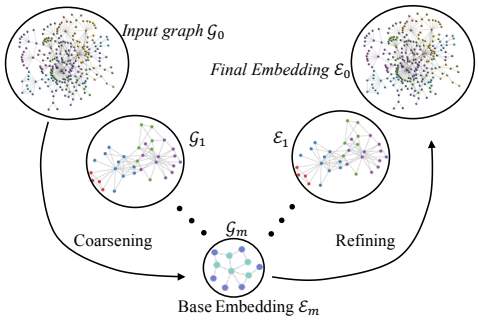


Figure 1: An overview of the multi-level embedding framework

Symbol	Definition
$\mathcal{G}_i$	the graph after i iterations of coarsening
V_i, E_i	vertex set, edge set of $\mathcal{G}_i$
A_i, D_i	the adjacency and degree matrix of $\mathcal{G}_i$
d	dimensionality of the embeddings
m	the total number of coarsening levels
$f(\cdot)$	the base embedding method applicable on $\mathcal{G}_i$
$\mathcal{E}_i$	the embeddings of nodes in $\mathcal{G}_i$
$M_{i,i+1}$	the matching matrix from $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$
$\mathcal{R}(\cdot)$	the embeddings refinement model
l	# layers in the graph convolution network

Table 1: The table of notations.

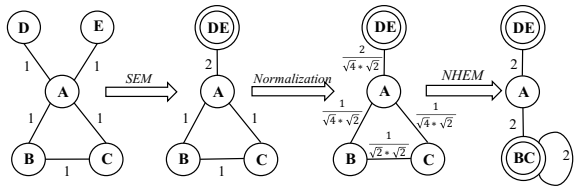
Definition 1 ***Graph Embedding*** *Given a graph $\mathcal{G} = (V, E)$ and a dimensionality d ($d \ll |V|$), the problem of graph embedding is to learn a d-dimension vector representation for each node in $\mathcal{G}$ so that graph properties are best preserved.*

Following this, a graph embedding method is essentially a mapping function $f : \mathbb{R}^{|V|\times|V|} \mapsto \mathbb{R}^{|V|\times d}$, whose input is the adjacency matrix A (or $\mathcal{G}$) and output is a lower dimension matrix. Motivated by the fact that the majority of graph embedding methods cannot scale to large datasets, we seek to speed up existing graph embedding methods without sacrificing quality. We formulate the problem as:

Given a graph $\mathcal{G} = (V, E)$ and a graph embedding method $f(\cdot)$, we aim to realize a strengthened graph embedding method $\hat{f}(\cdot)$ so that it is more scalable than $f(\cdot)$ while generating embeddings of comparable or even better quality.

Methodology

The MILE framework comprises three phases: graph coarsening, base embedding, and refinement (see Figure 1), described next.



(a) Using SEM and NHEM for graph coarsening

$$A_0 = \begin{array}{c} \begin{array}{ccccc} \text{A} & \text{B} & \text{C} & \text{D} & \text{E} \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad M_{0,1} = \begin{array}{c} \begin{array}{ccc} \text{A} & \text{BC} & \text{DE} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \\ \text{E} \end{array}$$

$$A_1 = M_{0,1}^T A_0 M_{0,1} = \begin{pmatrix} 0 & 2 & 2 \\ 2 & 2 & 0 \\ 2 & 0 & 0 \end{pmatrix}$$

(b) Adjacency matrix and matching matrix

Figure 2: Toy example for illustrating graph coarsening. (a) shows the process of applying Structural Equivalence Matching (SEM) and Normalized Heavy Edge Matching (NHEM) for graph coarsening. (b) presents the adjacency matrix A_0 of the input graph, the matching matrix $M_{0,1}$ corresponding to the SEM and NHEM matchings, and the derivation of the adjacency matrix A_1 of the coarsened graph using Eq. 2.

Graph Coarsening

In this phase, the input graph $\mathcal{G}$ (or $\mathcal{G}_0$) is repeatedly coarsened into a series of smaller graphs $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_m$ such that $|V_0| > |V_1| > ... > |V_m|$. In order to coarsen a graph from $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$, multiple nodes in $\mathcal{G}_i$ are collapsed to form super-nodes in $\mathcal{G}_{i+1}$, and the edges incident on a super-node are the union of the edges on the original nodes in $\mathcal{G}_i$. Here the set of nodes forming a super-node is called a *matching*. We propose a hybrid matching technique containing two matching strategies that can efficiently coarsen the graph while retaining the global structure. A toy example is shared in Figure 2.

Structural Equivalence Matching (SEM) : Given two vertices u and v in an unweighted graph $\mathcal{G}$, we call they are *structurally equivalent* if they are incident on the same set of neighborhoods. In figure 2a, node D and E are *structurally equivalent*. The intuition of *matching* structually equivalent nodes is that if two vertices are structurally equivalent, then their node embeddings will be similar. The novel SEM method – though dependent on graph structure – is highly effective on the real world datasets. Empirically, we found 5-20% of nodes to be structurally equivalent in many real world datasets. For example, during the first level of coarsening, YouTube has 172,906 nodes (or 86,453 pairs) out of 1,134,890 nodes that are found to be SEM (around 15%); Yelp has 875,236 nodes (or 437,618 pairs) out of 8,938,630 nodes are SEM (around 10%). Note that even more nodes are involved in SEM across iterations, as SEM is run iteratively at each coarsening level.

Normalized Heavy Edge Matching (NHEM) : Heavy edge matching is a popular matching method for graph coarsening (Karypis and Kumar 1998b). We select an unmatched node, say u, in the graph and find a large weighted edge (u, v) incident on node u such that node v is also unmatched. We then collapse nodes u and v into one super-node and mark them as matched. We repeat the matching process until there are no unmatched nodes or an unmatched node does not have an unmatched neighbor node. In this paper, we propose to normalize the edge weights when applying heavy edge matching using the formula as follows

$$W_i(u, v) = \frac{A_i(u, v)}{\sqrt{D_i(u, u) \cdot D_i(v, v)}}. \tag{1}$$

Here, the weight of an edge is normalized by the degree of the two vertices on which the edge is incident. Intuitively, it penalizes the weights of edges connected with high-degree nodes. We will show later that this normalization is tightly connected with the graph convolution kernel.

Hybrid Matching Method : We use a hybrid of two matching methods above for graph coarsening. To construct $\mathcal{G}_{i+1}$ from $\mathcal{G}_i$, we first find out all the structural equivalence matching (SEM) $\mathcal{M}_1$, where $\mathcal{G}_i$ is treated as an unweighted graph. This is followed by the searching of the normalized heavy edge matching (NHEM) $\mathcal{M}_2$ on $\mathcal{G}_i$. Nodes in each matching are then collapsed into a super-node in $\mathcal{G}_{i+1}$. Note that some nodes might not be matched at all and they will be directly copied to $\mathcal{G}_{i+1}$.

Formally, we build the adjacency matrix A_{i+1} of $\mathcal{G}_{i+1}$ through matrix operations. To this end, we define the *matching matrix* storing the matching information from graph $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$ as a binary matrix $M_{i,i+1} \in \{0, 1\}^{|V_i| \times |V_{i+1}|}$. The r-th row and c-th column of $M_{i,i+1}$ is set to 1 if node r in $\mathcal{G}_i$ will be collapsed to super-node c in $\mathcal{G}_{i+1}$, and is set to 0 if otherwise. Each column of $M_{i,i+1}$ represents a matching with the 1s representing the nodes in it. Each unmatched vertex appears as an individual column in $M_{i,i+1}$ with merely one entry set to 1. Following this formulation, we construct the adjacency matrix of $\mathcal{G}_{i+1}$ by using

$$A_{i+1} = M_{i,i+1}^T A_i M_{i,i+1}. \tag{2}$$

Algorithm 1 summarizes the steps of graph coarsening. For each iteration of coarsening, SEM is generated followed by NHEM (line 2-9). A key part of NHEM is to visit the vertices in **ascending** order of the number of neighbors (line 5). This is important to ensure that a large fraction of vertices can be matched in each step, and the graph can be coarsened significantly. Intuitively, vertices with a small number of neighbors have a limited choice of finding a match and should be given a higher priority for matching (otherwise, once their neighbors are matched by others, these vertices cannot be matched).

Key Intuitions: The graph coarsening phase significantly reduces the size of graph while maintaining the clustering structure and backbone of the original graph. The coarsening phase also potentially exposes the global structure of the graph to the base embedding method that it otherwise might not take into account thereby offering a potential efficacy gain (up to a point). This efficacy gain has also been observed elsewhere, in the context of stochastic flow algorithms (Satuluri and Parthasarathy 2009). Note that stochas-

Algorithm 1 Graph Coarsening Algorithm

Input: An input graph $\mathcal{G}_0 = (V_0, E_0)$, and coarsening levels m.
Output: Coarsened graphs $\mathcal{G}_{i+1}$ and $M_{i,i+1}$ for $0 \leq i \leq m-1$.

for $i = 1...m$ **do**
  $\mathcal{M}_1 \leftarrow$ all the structural equivalence matching in $\mathcal{G}_{i-1}$.
  Mark vertices in $\mathcal{M}_1$ as matched.
  $\mathcal{M}_2 = \emptyset$. ▷ storing normalized heavy edge matching
  Sort V_{i-1} by the number of neighbors in ascending order.
  for $v \in V_{i-1}$ **do**
    if v and u are not matched and $u \in$ Neighbors(v) **then**
      $(v, u) \leftarrow$ the normalized heavy edge matching for v.
      $\mathcal{M}_2 = \mathcal{M}_2 \cup (v, u)$, and mark both as matched.
    end if
  end for
  Compute matching matrix $M_{i-1,i}$ based on $\mathcal{M}_1$ and $\mathcal{M}_2$.
  Derive the adjacency matrix A_i for $\mathcal{G}_i$ using Eq. 2.
end for
Return $\mathcal{G}_{i+1}$ and $M_{i,i+1}$ for $0 \leq i \leq m-1$.

tic flow algorithms share some common mathematical abstraction to several graph embedding methods (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016; Qiu et al. 2018a). The embeddings learned by base embedding method on the coarsened graph can act as an effective initialization for the graph-topology aware refinement model.

Choice for Coarsening Level: Similar to other multi-level frameworks, such as graph partitioning (Karypis and Kumar 1998a,b; Satuluri and Parthasarathy 2009) and visualization (Harel and Koren 2000), the choice of coarsening level depends on the application domain and the graph properties. However, we found a small number of coarsening levels (from 2 to 4) usually yields the best-quality embeddings with decent speedup when the graph is of medium size ($\#nodes < 1,000,000$). For larger graph ($\#nodes > 1,000,000$), the embeddings tend to remain constantly high-quality even with coarsening levels from 6 to 8 while speedup increases even more. We defer our detailed discussion on this to the experiments section.

Base Embedding on Coarsened Graph

The size of the graph reduces drastically after each iteration of coarsening, halving the size of the graph in the best case. We coarsen the graph for m iterations and apply the graph embedding method $f(\cdot)$ on the coarsest graph $\mathcal{G}_m$. Denoting the embeddings on $\mathcal{G}_m$ as $\mathcal{E}_m$, we have $\mathcal{E}_m = f(\mathcal{G}_m)$. Since our framework is agnostic to the adopted graph embedding method, we can use any graph embedding algorithm for base embedding.

Refinement of Embeddings

The final phase of MILE is the embedding refinement. Given a series of coarsened graph $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_m$, their corresponding matching matrix $M_{0,1}, M_{1,2}, ..., M_{m-1,m}$, and the node embeddings $\mathcal{E}_m$ on $\mathcal{G}_m$, we seek to develop an approach to derive the node embeddings of $\mathcal{G}_0$ from $\mathcal{G}_m$. To this end, we first study an easier subtask: given a graph $\mathcal{G}_i$,

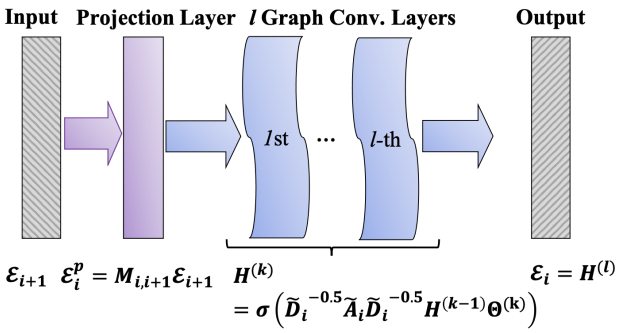


Figure 3: Architecture of the embedding refinement model

its coarsened graph $\mathcal{G}_{i+1}$, the matching matrix $M_{i,i+1}$ and the node embeddings $\mathcal{E}_{i+1}$ on $\mathcal{G}_{i+1}$, how to infer the embeddings $\mathcal{E}_i$ on graph $\mathcal{G}_i$. Once we solved this subtask, we can then iteratively apply the technique on each pair of consecutive graphs from $\mathcal{G}_m$ to $\mathcal{G}_0$ and eventually derive the node embeddings on $\mathcal{G}_0$. In this work, we propose to use a graph-based neural network model to perform embedding refinement.

Graph Convolution Network for Refinement Learning : Since we know the matching information between the two consecutive graphs $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$, we can easily project the node embeddings from the coarse-grained graph $\mathcal{G}_{i+1}$ to the fine-grained graph $\mathcal{G}_i$ using

$$\mathcal{E}_i^p = M_{i,i+1}\mathcal{E}_{i+1} \tag{3}$$

In this case, embedding of a super-node is directly copied to its original node(s). We call $\mathcal{E}_i^p$ the *projected embeddings* from $\mathcal{G}_{i+1}$ to $\mathcal{G}_i$, or simply *projected embeddings* without ambiguity. While this way of simple projection maintains some information of node embeddings, it has an obvious limitation that nodes will share the same embeddings if they are matched and collapsed into a super-node during the coarsening phase. This problem will be more serious when the embedding refinement is performed iteratively from $\mathcal{G}_m$, ..., $\mathcal{G}_0$. To address this issue, we propose to learn a *graph convolution network* (GCN) for embedding refinement (Kipf and Welling 2017). Specifically, we design a graph-based neural network model $\mathcal{E}_i = \mathcal{R}(\mathcal{E}_i^p, A_i)$, which derives the embeddings $\mathcal{E}_i$ on graph $\mathcal{G}_i$ based on the projected embeddings $\mathcal{E}_i^p$ (from the base method) and the graph adjacency matrix A_i (from the input graph).

Given graph G with adjacency matrix A, we consider the fast approximation of graph convolution from (Kipf and Welling 2017). The k-th layer of this neural network model is

$$H^{(k)}(X, A) = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k-1)}(X, A)\Theta^{(k)}\right) \tag{4}$$

where $\sigma(\cdot)$ is an activation function, $\Theta^{(k)}$ is a layer-specific trainable weight matrix, and $H^{(0)}(X, A) = X$. In this paper, we define our embedding refinement model as a l-layer graph convolution model

$$\mathcal{E}_i = \mathcal{R}\left(\mathcal{E}_i^p, A_i\right) \equiv H^{(l)}\left(\mathcal{E}_i^p, A_i\right). \tag{5}$$

The architecture of the refinement model is shown in Figure 3. The intuition behind this refinement model is to integrate the structural information of the current graph $\mathcal{G}_i$ into the projected embedding $\mathcal{E}_i^p$ by repeatedly performing the spectral graph convolution. Each layer of graph convolution network in Eq. 4 can be regarded as one iteration of embedding propagation in the graph following the re-normalized adjacency matrix $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. Note that this re-normalized matrix is well aligned with the way we conduct normalized heavy edge matching in Eq. 1.

Choice for Number of GCN Layers: The graph convolution model is often treated as a message passing operator (Hamilton, Ying, and Leskovec 2017) with the number of layers corresponding to the number of hops in the graph. In other words, l GCN layers correspond to aggregating structural information from all the l-hop neighbours for each node. On the one hand, we want l to be larger than 1 so that node embeddings can reflect connectivity structure beyond immediate neighbors. On the other hand, we also do not want too large an l as it will make the node embeddings homogeneous and less distinguishable across the graph due to the small-world property of real-world graphs. Similar to existing literature (Kipf and Welling 2017; Hamilton, Ying, and Leskovec 2017; Veličković, Cucurull et al. 2017) we find that setting l to 2 worked best in practice.

Intricacies of Refinement Learning : The learning of the refinement model is essentially learning $\Theta^{(k)}$ for each $k \in [1, l]$ according to Eq. 4. Here we study how to design the learning task and construct the loss function. Since the graph convolution model $H^{(l)}(\cdot)$ aims to predict the embeddings $\mathcal{E}_i$ on graph $\mathcal{G}_i$, we can directly run a base embedding on $\mathcal{G}_i$ to generate the "ground-truth" embeddings and use the difference between these embeddings and the predicted ones as the loss function for training. We propose to learn $\Theta^{(k)}$ on the coarsest graph and **reuse** them across all the levels for refinement. Specifically, we can define the loss function as the mean square error as follows

$$L = \frac{1}{|V_m|}\left\|\mathcal{E}_m - H^{(l)}(M_{m,m+1}\mathcal{E}_{m+1}, A_m)\right\|^2. \quad (6)$$

We refer to the learning task associated with the above loss function as *double-base* embedding learning. We point out, however, there are two key drawbacks to this method. First of all, the above loss function requires one more level of coarsening to construct $\mathcal{G}_{m+1}$ and an extra base embedding on $\mathcal{G}_{m+1}$. These two steps, especially the latter, introduce non-negligible overheads to the MILE framework. More importantly, $\mathcal{E}_m$ might not be a desirable "ground truth" for the refined embeddings. This is because most of the embedding methods are invariant to an orthogonal transformation of the embeddings, i.e., the embeddings can be rotated by an arbitrary orthogonal matrix (Hamilton, Ying, and Leskovec 2017). In other words, the embedding spaces of graph $\mathcal{G}_m$ and $\mathcal{G}_{m+1}$ can be totally different since the two base embeddings are learned independently. Even if we follow the paradigm in (Chen et al. 2018) and conduct base embedding on $\mathcal{G}_m$ using the simple projected embeddings from $\mathcal{G}_{m+1}$ ($\mathcal{E}_m^p$) as initialization, the embedding space does not naturally generalize and can drift during re-training. One possible solution is to use an alignment procedure to force the embeddings to be aligned between the two graphs (Hamilton, Leskovec, and Jurafsky 2016). But it could be very expensive.

Algorithm 2 Multi-Level Algorithm for Graph Embedding

Input: An input graph $\mathcal{G}_0 = (V_0, E_0)$, # coarsening levels m, and a base embedding method $f(\cdot)$.
Output: Graph embeddings $\mathcal{E}_0$ on $\mathcal{G}_0$.

1: Coarsen $\mathcal{G}_0$ into $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_m$ using proposed hybrid matching method.
2: Perform base embedding on the coarsest graph $\mathcal{G}_m$ (See Section.).
3: Learn the weights $\Theta^{(k)}$ using the loss function in Eq. 7.
4: **for** $i = (m-1)...0$ **do**
5: Compute the projected embeddings $\mathcal{E}_i^p$ on $\mathcal{G}_i$.
6: Use Eq. 4 and Eq. 5 to compute refined embeddings $\mathcal{E}_i$.
7: **end for**
8: Return graph embeddings $\mathcal{E}_0$ on $\mathcal{G}_0$.

In this paper, we propose a very simple method to address the above issues. Instead of conducting an additional level of coarsening, we construct a dummy coarsened graph by simply copying $\mathcal{G}_m$, i.e., $M_{m,m+1} = I$ and $\mathcal{G}_{m+1} = \mathcal{G}_m$. By doing this, we not only reduce one iteration of graph coarsening, but also avoid performing base embedding on $\mathcal{G}_{m+1}$ simply because $\mathcal{E}_{m+1} = \mathcal{E}_m$. Moreover, the embeddings of $\mathcal{G}_m$ and $\mathcal{G}_{m+1}$ are guaranteed to be in the same space in this case without any drift. With this strategy, we change the loss function for model learning as follows

$$L = \frac{1}{|V_m|}\left\|\mathcal{E}_m - H^{(l)}(\mathcal{E}_m, A_m)\right\|^2. \quad (7)$$

We minimize the difference between the generated embeddings and the embeddings generated from the refinement model (GCN based) so that the learnt refinement model could then be levered to generate embeddings in other coarsening levels. With the above loss function, we adopt gradient descent with back-propagation to learn the parameters $\Theta^{(k)}$, $k \in [1, l]$. In the subsequent refinement steps, we apply the same set of parameters $\Theta^{(k)}$ to infer the refined embeddings. We point out that the training of the refinement model is rather efficient as it is done on the coarsest graph. The embedding refinement process involves merely sparse matrix multiplications using Eq. 5 and is relatively affordable compared to conducting embedding on the original graph. With these different components, we summarize the whole algorithm of our MILE framework in Algorithm 2.

Discussion on Reusing $\Theta^{(k)}$ Across All Levels: Similar to GCN, $\Theta^{(k)}$ is a matrix of filter parameters and is of size $d * d$ (where d is the embedding dimensionality). Eq. 4 defines how the embeddings are propagated during embedding refinements, parameterized by $\Theta^{(k)}$. Intuitively, $\Theta^{(k)}$ defines how different embedding dimensions interact with each other during the embedding propagation. This interaction is dependent on graph structure and base embedding

Dataset	# Nodes	# Edges	# Classes
PPI	3,852	38,705	50
Blog	10,312	333,983	39
Flickr	80,513	5,899,882	195
YouTube	1,134,890	2,987,624	47
Yelp	8,938,630	39,821,123	22

Table 2: Dataset Information

method, which can be learned from the coarsest level. Ideally, we would like to learn this parameter $\Theta^{(k)}$ on consecutive levels. But this is not practical since this could be expensive as the graph gets more fine-grained (and defeat our purpose of scaling up graph embedding). This trick of "sharing" parameters across different levels is the trade-off between efficiency and effectiveness. To some extent, it is similar to GCN (Kipf and Welling 2017), where the authors share the same filter parameters $\Theta^{(k)}$ over the whole graph (as opposed to using different $\Theta^{(k)}$ for different nodes; see Eq (6) and (7) in(Kipf and Welling 2017)). Moreover, we empirically found this works well and is much more efficient. As we will reveal in the Experiments section, the shared $\Theta^{(k)}$ values do much better than alternative $\Theta^{(k)}$ choices during refinement (see Table 4).

Intuition and Rationale of Embedding Refinements: Refining the embeddings from a coarsened graph to its fine-grained graph boils down to two steps: a) projecting the node embeddings back to fine-grained graph based on correspondence matching; and b) propagating the projected embeddings locally, adjusting them based on the fine-grained graph structure. The first step is relatively straightforward through matrix multiplication (Eq. 3). For the second step, we repurpose the GCN model to propagate the node embeddings. There are mainly two reasons that GCN performs well in this part. First, the propagation rule in GCN is shown to be a first-order approximation of the **localized spectral filters** on graphs (Kipf and Welling 2017; Defferrard, Bresson, and Vandergheynst 2016) and as a result, the propagated embeddings can capture the local graph structure. Second, GCN contains learnable $\Theta^{(k)}$ and is capable of modeling the interaction between different embedding dimensions in the propagation process, as discussed above.

Experiments and Analysis

Datasets: The datasets used in our study (See Table 2) have seen prior use for evaluating representation learning methods (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016; Qiu et al. 2018a) and are primarily drawn from popular social media platforms while one is drawn from a well curated bioinformatics dataset. We preprocess the Yelp dataset following a procedure described in (Huang, Li, and Hu 2017)[1].

[1]Raw data: https://www.yelp.com/dataset_challenge/dataset

Base Embedding Methods: To demonstrate that MILE can work with different graph embedding methods, we explore several popular methods for graph embedding.

- **Random-walk based methods**: We select DeepWalk (Perozzi, Al-Rfou, and Skiena 2014) and Node2Vec (Grover and Leskovec 2016) as baseline methods for this group. We set the length of random walks as 80, number of walks per node as 10, and context windows size as 10. In Node2Vec, we set $p = 4.0$ and $q = 1.0$ which we found empirically to generate better results across all the datasets.
- **Edge reconstruction based method**: We select Line (Tang et al. 2015) as it is a representative work in this group. The number of edge samples is set to 100 million.
- **Matrix-factorization based methods** : GraRep (Cao, Lu, and Xu 2015) and NETMF (Qiu et al. 2018a) are two popular methods in this group. For GraRep, we set k=4. We varied this parameter but found this value to work well (also suggested by the original authors (Cao, Lu, and Xu 2015)). For NetMF, we set the window size to 10 and the rank h to 1024.
- **Deep neural network based methods:** We lever SDNE (Wang, Cui, and Zhu 2016) as an exemplar for this group. We set alpha to 0.2 and beta to 10.0. We varied these parameters and found these values to work well – these parameter values also work well in original paper.
- **Distributed embedding methods:** We compare MILE against Pytorch-biggraph (Lerer et al. 2019). We set the parameter negative batch sizes to 500 and set learning rate to 0.01. We varied these parameters and found these values to work well – these parameters fall in the suggested values by the authors (Lerer et al. 2019).

By showing the performance gain of using MILE on top of these methods, we want to ensure the contribution of this work is of broad interest to the community. We also want to reiterate that these methods are quite different in nature.

MILE-specific Settings: For all the above base embedding methods, we set the embedding dimensionality d as 128. When applying our MILE framework, we vary the coarsening levels m from 1 to 8 whenever possible. For the graph convolution network model, the self-loop weight λ is set to 0.05, the number of hidden layers l is 2, and $\tanh(\cdot)$ is used as the activation function, the learning rate is set to 0.001 and the number of training epochs is 200. The Adam Optimizer is used for model training.

Evaluation Metrics: We evaluate the quality of the embeddings through multi-label node classification (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016) and link prediction (Gurukar, Vijayan et al. 2019). Specifically, for node classification, we run a 10-fold cross validation using the embeddings as features and report the average Micro-F1 and average Macro-F1. For link prediction, we follow the setup described in (Gurukar, Vijayan et al. 2019). Specifically, we evaluate the link prediction performance in terms

of AUROC on 5-fold cross validation and report the average AUROC scores.

Running time: We present end-to-end wallclock time for scalability analysis. For MILE, the reported running time include the execution time of all the phases, including the training time of refinement model.

System Specifications: The experiments were conducted on a machine running Linux with an Intel Xeon E5-2680 CPU (28 cores, 2.40GHz) and 128 GB of RAM. We implement our MILE framework in `Python`. For all the five base embedding methods, we adapt the original code from the authors [2]. For SDNE, we lever the publicly available source code from here (Goyal and Ferrara 2018). We additionally use the TensorFlow package for the embedding refinement learning component. We lever the available parallelism (on 28 cores) for each method (e.g., the generation of random walks in DeepWalk and Node2Vec, the training of the refinement model in MILE, etc.).

MILE Framework Performance

We first evaluate the performance of our MILE framework when applied to different graph embedding methods. The performance of MILE with various base embedding methods – on different datasets and different coarsening levels – for node classification and link prediction is shown in Figure 4 and Figure 5, respectively.[3] We also investigate various design choices related to MILE in a subsequent section. The evaluation of various design choices are conducted through the lens of node classification, but similar results hold for link prediction. Note that a coarsening level of m=0, corresponds to the original embedding method. We make the following observations:

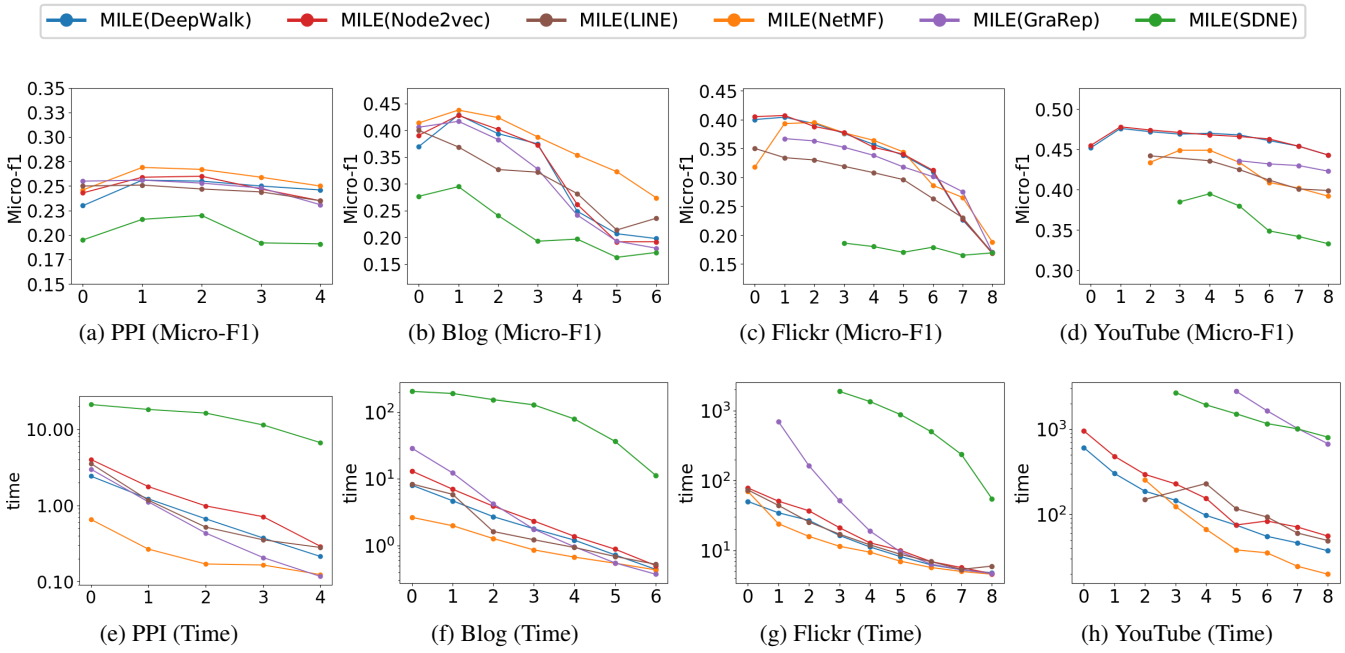
MILE is scalable. MILE greatly boosts the speed of the explored embedding methods. In the case of node classification, as shown in Figure 4, with a single level of coarsening (m=1), we are able to achieve speedup ranging from 1.5× to 3.4× (on PPI, Blog, and Flickr) while improving qualitative performance. Larger speedups are typically observed on GraRep, NetMF, and SDNE. Increasing the coarsening level m to 2, the speedup increases further (up to 14.4×), while the quality of the embeddings is comparable with the original methods reflected by Micro-F1. On YouTube, for the coarsening levels 6 and 8, we observe more than 10× speedup for DeepWalk, Node2Vec, Line, and SDNE. The execution of SDNE method on flickr dataset does not finish in 2 days, however, the execution of MILE(SDNE) with coarsen level 8 finishes in less than 1 hour. For NetMF on YouTube, the speedup is even larger – original NetMF runs out of memory within 9.5 hours while MILE (NetMF) only takes around 20 minutes ($m = 8$). In the case of link prediction, as shown in Figure 5, we observe that an increase in coarsening level results in a consistent decrease in the running time of embedding methods. The higher coarsening levels (m = 7,8) leads to 1.2× to 113× speedup for all the methods on BlogCatalog and PPI datasets. On Flickr and YouTube, the running time reduction for all the methods ranges from 10× to 29× while the quality of the embeddings is comparable (or even better) with respect to the original methods in terms of AUROC score.

Impact of MILE on embedding quality. In the case of node classification, as shown in Figure 4, for coarsening levels $m = 1$ or 2, we observe that MILE learnt embeddings are almost always better in quality across all the datasets and methods. Examples include MILE (DeepWalk, $m = 1$) on Blog/PPI, MILE (Line, $m = 1$) on PPI and MILE (NetMF, $m = 1$) on PPI/Blog/Flickr. Even with higher number of coarsening level ($m = 2$ for PPI/Blog/Flickr; $m = 6, 8$ for YouTube), MILE in addition to being much faster can still improve, qualitatively, over the original methods on most of the datasets, e.g., MILE (NetMF, $m = 2$) $\gg$ NetMF on PPI, Blog, and Flickr. In the case of link prediction, as shown in Figure 5, we observe that the increase in coarsening level results in a corresponding increase in AUROC scores in most of the cases. For Node2vec, LINE and NetMF, with a single level of coarsening (m=1), we see improvement in AUROC from 3% to 10% on PPI and BlogCatalog datasets. With higher coarsening levels (m=7, 8), we see a consistent improvement in AUROC from 1.4% to 10.7% for Node2vec, LINE, NetMF and GraRep methods on Flickr and YouTube datasets. For SDNE, on Blog and YouTube dataset, the link prediction performance across coarsening levels 2-5 remains competitive with original SDNE method. We observe a 2x-15x speed in this range, consistent with other methods that use MILE. As discussed when describing the key intuitions underpinning the coarsening-refinement strategy that MILE employs (in Section), the observed improvement on quality – for both node classification and link prediction – is likely due to the fact that the base embedding methods are exposed to a holistic view of the entire graph. This observation is consistent with those observed for stochastic flow clustering (Satuluri and Parthasarathy 2009).

MILE supports multiple embedding methods. We empirically show that MILE can work with different category of embedding methods on multiple real world datasets. We observe that MILE often improves both the quality and the efficiency of NetMF on all four datasets (for YouTube, NetMF runs out of memory). For the largest dataset, the speedups afforded exceed 30-fold. We observe that for GraRep, while speedups with MILE are consistently observed, the qualitative improvements if any, are smaller (for both YouTube and Flickr, the base method runs out of memory). For Line, even though its time complexity is linear to the number of edges (Tang et al. 2015), applying MILE framework on top of it still generates significant speed-up (likely due to the fact that the complexity of Line contains a larger constant factor k than MILE). On the other hand, MILE on top of Line generates better quality of embeddings on PPI and YouTube while falling a bit short on Blog and Flickr. For DeepWalk and Node2Vec, we again observe consistent improvements in scalability (up to 11-fold on the larger

[2] DeepWalk: https://github.com/phanein/deepwalk;
Node2Vec: http://snap.stanford.edu/node2vec/;
Line: https://github.com/thunlp/OpenNE
GraRep: https://github.com/thunlp/OpenNE;
NetMF: https://github.com/xptree/NetMF

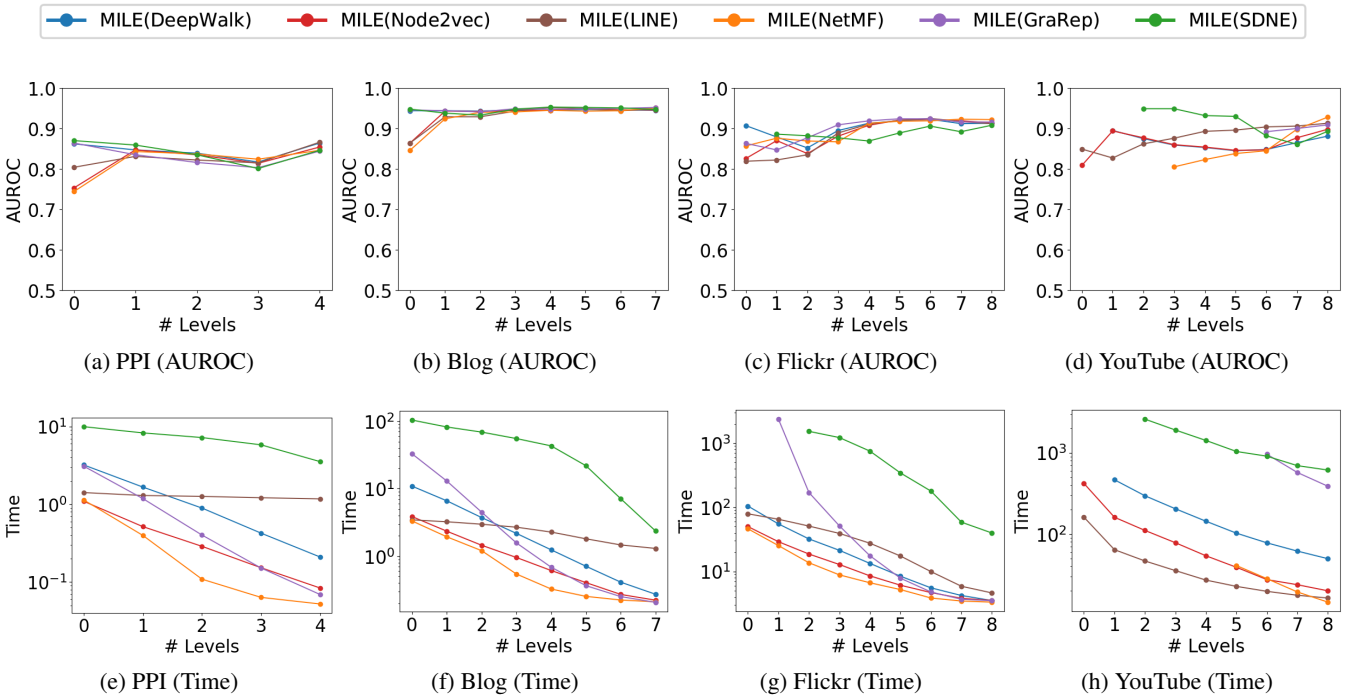
[3] We discuss the results of Yelp later.



(a) PPI (Micro-F1) (b) Blog (Micro-F1) (c) Flickr (Micro-F1) (d) YouTube (Micro-F1)

(e) PPI (Time) (f) Blog (Time) (g) Flickr (Time) (h) YouTube (Time)

Figure 4: Changes in the node classification performance as the number of coarsening levels in MILE increases (best viewed in color). Micro-F1 and running-time are reported in the first and second row respectively. Running time in minutes is shown in the logarithm scale. Note that # level $= 0$ represents the original embedding method without using MILE. Lines/points are missing for algorithms that use over 128 GB of RAM.



(a) PPI (AUROC) (b) Blog (AUROC) (c) Flickr (AUROC) (d) YouTube (AUROC)

(e) PPI (Time) (f) Blog (Time) (g) Flickr (Time) (h) YouTube (Time)

Figure 5: Changes in the link prediction performance as the number of coarsening levels in MILE increases. AUROC and running-time are reported in the first and second row respectively. Running time in minutes is shown in the logarithm scale. Note that # level $= 0$ represents the original embedding method without using MILE.

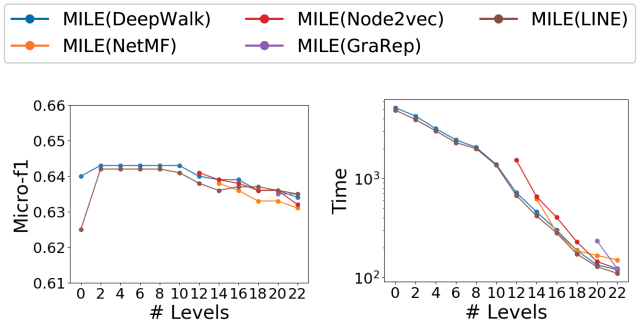
datasets) on the node classification task using MILE with a few levels of coarsening. However, when the coarsening level is increased, the additional speedup afforded (up to 17-fold) comes at a mixed cost to quality (micro-F1 drops slightly). We observe similar trends – improvements in embedding quality and reduction in the running time – in the case of link prediction experiments.

Impact of varying coarsening levels on MILE. In the case of node classification, as shown in Figure 4, when coarsening level m is small ($m = 1$ or 2), MILE tends to significantly improve the quality of embeddings while taking much less time. From $m = 0$ to $m = 1$, we see a clear jump of the Micro-F1 score on all the datasets across the base embedding methods. This observation is more evident on larger datasets (Flickr and YouTube). On YouTube, MILE (DeepWalk) with m=1 increases the Micro-F1 score by 5.3% while only consuming half of the time compared to the original DeepWalk. MILE (DeepWalk) continues to generate embeddings of better quality than DeepWalk until $m = 7$, where the speedup is $13\times$. As the coarsening level m in MILE increases, the running time drops dramatically while the quality of embeddings only decreases slightly. In the case of link prediction, as shown in Figure 5, the methods with higher coarsening levels (m=7,8) on all the evaluated datasets, except GraRep on PPI, show an improvement of AUROC with respect to the embedding performance on the original graph. For SDNE, we observe that as we vary the coarsening levels till coarsen level 5, the link prediction performance on Blog, Flickr, YouTube remains close to original method while we see improvement in the speedup. However on YouTube we see a drop in AUROC score with coarsening levels 6 and 8 for SDNE. For both node classification and link prediction, the execution time decreases at an almost exponential rate (logarithm scale on the y-axis in the second row of Figure 4 and Figure 5). On the other hand, the Micro-F1 score descends much more slowly (the first row of Figure 4), most of which are still better than the original methods.

Overall, the above experiments shows that MILE can not only accommodate existing embedding methods - treating them as a blackbox - but also provides nice trade-off between effectiveness and efficency, a useful lever for downstream tasks and use-cases.

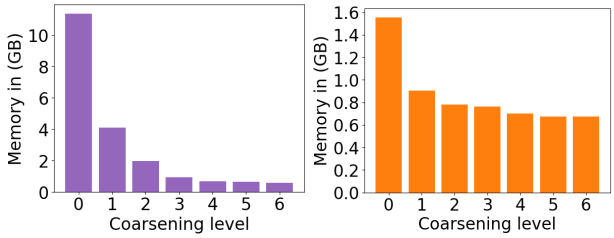
MILE: Large Graph Embedding

We now explore the scalability of MILE on the large Yelp dataset. None of the five graph embedding methods studied in this paper can successfully conduct graph embedding on Yelp within 60 hours on a modern machine with 28 cores and 128 GB RAM. Even extending the run-time deadline to 100 hours, we see DeepWalk and Line barely finish. Leveraging the proposed MILE framework now makes it much easier to perform graph embedding on this scale of datasets (see Figure 6 for the results). We observe that MILE significantly reduces the running time and improves the Micro-F1 score. For example, The Micro-f1 scores of original DeepWalk and Line are 0.640 and 0.625 respectively, which all take more than 80 hours. But using MILE with $m = 4$, the micro-F1 score improves to 0.643 (DeepWalk) and 0.642 (Line)



(a) Micro-F1 (b) Running Time

Figure 6: Running MILE on Yelp dataset. Lines/points are missing for algorithms that do not finish within 60 hours or use over 128 GB of RAM.



(a) MILE (GraRep) (b) MILE (NetMF)

Figure 7: Memory consumption of MILE (GraRep) and MILE (NetMF) on Blog with varied coarsening levels.

while achieving speedups of around $1.6\times$. Moreover, MILE reduces the running time of DeepWalk from 53 hours (coarsening level 4) to 2 hours (coarsening level 22) while reducing the Micro-F1 score just by 1% (from 0.643 to 0.634). Meanwhile, there is no change in the Micro-F1 score from the coarsening level 4 to 10, where the running time is improved by a factor of two. These results affirm the power of the proposed MILE framework on scaling up graph embedding algorithms while generating quality embeddings. We also observe similar performance gain and reduction in the running time for the link prediction task, however, due to paucity of space the link prediction results on Yelp dataset are not shared in this paper.

Memory Consumption

We also study the impact of MILE on reducing memory consumption. For this purpose, we focus on MILE (GraRep) and MILE (NetMF), with GraRep and NetMF as base embedding methods respectively. Both of these are embedding methods based on matrix factorization, which possibly involves a dense objective matrix and could be rather memory expensive. We do not explore DeepWalk and Node2Vec here since their embedding learning methods generate truncated random walks (training data) on the fly with almost negligible memory consumption (compared to the space storing the graph and the embeddings). Figure 7 shows the memory consumption of MILE (GraRep) and MILE(NetMF) as the coarsening level increases on Blog (results on other datasets are similar). We observe that MILE significantly reduces

	PPI		Blog	
	Mi-F1	Time	Mi-F1	Time
DeepWalk (DW)	23.0	2.4	37.0	8.0
MILE (DW)	25.6	1.2	42.9	4.6
HARP (DW)	24.1	3.0	41.3	9.8
Node2Vec (NV)	24.3	4.0	39.1	13.0
MILE (NV)	**25.9**	1.7	**42.8**	6.9
HARP (NV)	22.3	3.9	36.2	13.16
	Flickr		YouTube	
	Mi-F1	Time	Mi-F1	Time
DeepWalk	40.0	50.0	45.2	604.8
MILE (DW)	40.4	34.4	46.1	55.2
HARP (DW)	40.6	78.2	46.6	1727.7
Node2Vec	40.5	78.2	45.5	951.2
MILE (NV)	**40.7**	50.5	46.3	83.5
HARP (NV)	40.5	101.1	**47.2**	1981.3

Table 3: Comparisons of MILE with HARP.

the memory consumption as the coarsening level increases. Even with one level of coarsening, the memory consumption of GraRep and NetMF reduces by 64% and 42% respectively. The dramatic reduction continues as the coarsening level increases until it reaches 4, where the memory consumption is mainly contributed by the storage of the graph and the embeddings. This memory reduction is consistent with our intuition, since both # rows and # columns in the objective matrix reduce almost by half with one level of coarsening.

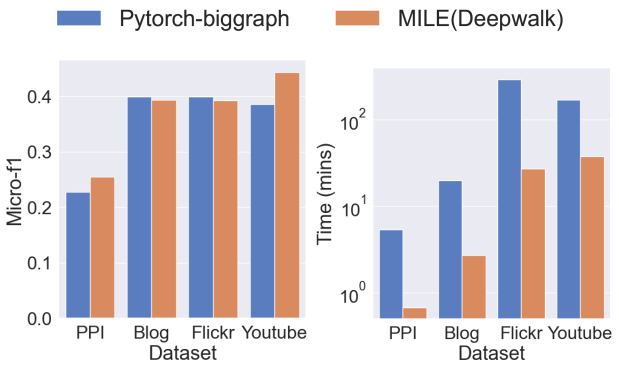
Comparing MILE with HARP

HARP is a multi-level method primarily for improving the quality of graph embeddings. We compare HARP with our MILE framework using DeepWalk and Node2vec as the base embedding methods[4]. Table 3 shows the performance of these two methods on the four datasets (coarsening level is 1 on PPI/Blog/Flickr and 6 on YouTube). We also observe similar node classification performance with Macro-f1 metric (not shown). From the table, we can observe that MILE generates embeddings of comparable quality with HARP. MILE performs much better than HARP on PPI and Blog, marginally better on Flickr and marginally worse on YouTube. However, MILE is significantly faster than HARP on all the four datasets (e.g. on YouTube, MILE affords a $31\times$ speedup). This is because HARP requires running the whole embedding algorithm on each coarsened graph, which introduces a **huge computational overhead**.

Comparing MILE with Pytorch-biggraph

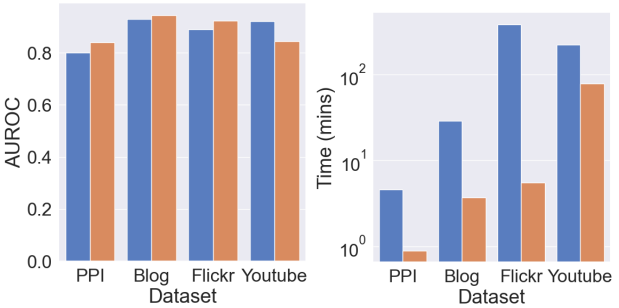
In this section, we compare the performance of MILE with Pytorch-biggraph. In Pytorch-biggraph, we set the number of workers to 28 (equal to the number of available cores). Figure 8 and Figure 9 respectively show the node classification and link prediction performance of both Pytorch-biggraph and MILE (Deepwalk) methods. Note that, in MILE, the speedup is driven by the coarsening level and higher speedup is achieved with higher coarsening level – as evident from the experiments shown in Figure 4 and Figure 5. For PPI and Blog dataset, we set coarsen level to 2 and for Flickr and YouTube dataset, we set coarsen level to 6. From Figure 8, we see that for the node classification task on PPI and YouTube dataset, MILE outperforms Pytorch-biggraph on classification quality metrics (higher-quality embeddings) while the running time of MILE is lower than Pytorch-biggraph. The lower running time of MILE is due to the proposed multi-level framework. From Figure 9, we observe that MILE outperforms Pytorch-biggraph for the link prediction task on three datasets namely PPI, Blog and Flickr with less running time compared to Pytorch-biggraph. This experiment shows that on a single system with 28-cores, MILE outperforms Pytorch-biggraph in terms of running time and also outperforms Pytorch-biggraph on downstream tasks such as node classification and link prediction.

[4]https://github.com/GTmac/HARP



(a) Node Classification (b) Running time

Figure 8: Node classification task: Comparisons of MILE with Pytorch-biggraph.



(a) Link Prediction (b) Running time

Figure 9: Link Prediction task: Comparisons of MILE with Pytorch-biggraph.

MILE Drilldown

Design Choices

We now study the role of the design choices we make within the MILE framework related to the coarsening and refinement procedures described. To this end, we examine alternative design choices and systematically examine their performance. The alternatives we consider are:

Random Matching (MILE-rm): For each iteration of coarsening, we repeatedly pick a random pair of connected nodes as a match and merge them into a super-node until no more matching can be found. The rest of the algorithm is the same as our MILE.

Simple Projection (MILE-proj): We replace our embedding refinement model with a simple projection method. In other words, we directly copy the embedding of a super-node to its original node(s) without any refinement ($\mathcal{E}_i^p = M_{i,i+1}\mathcal{E}_{i+1}$).

Averaging Neighborhoods (MILE-avg): For this baseline method, the refined embedding of each node is a weighted average node embedding of its neighborhoods (weighted by the edge weights). This can be regarded as an embeddings propagation method. We add self-loop to each node[5] and conduct the embeddings propagation for two rounds.

Untrained Refinement Model (MILE-untr): Instead of training the refinement model to minimize the loss defined as $L = \frac{1}{|V_m|}\left\|\mathcal{E}_m - H^{(l)}(\mathcal{E}_m, A_m)\right\|^2$, this baseline merely uses a fixed set of values for parameters $\Theta^{(k)}$ without training (values are randomly generated; other parts of the refinement model are the same, including $\tilde{A}$ and $\tilde{D}$).

Double-base Embedding for Refinement Training (MILE-2base): This method replaces the loss function as Eq. 7 with $L = \frac{1}{|V_m|}\left\|\mathcal{E}_m - H^{(l)}(M_{m,m+1}\mathcal{E}_{m+1}, A_m)\right\|^2$ for model training. It conducts one more layer of coarsening and base embedding (level $m+1$), from which the embeddings are projected to level m and used as the input for model training.

GraphSAGE as Refinement Model (MILE-gs): It replaces the graph convolution network in our refinement step with GraphSAGE (Hamilton, Ying, and Leskovec 2017)[6]. We choose max-pooling for aggregation and set the number of sampled neighbors as 100, as suggested by the authors. Also, concatenation is conducted instead of replacement during the process of propagation.

Results of MILE-variants

Table 4 shows the comparison of performance on these methods across the four datasets. Here, we focus on using DeepWalk for base embedding with coarsening level as $m = 1$ for PPI, Blog, and Flickr, while $m = 6$ for YouTube. Results are similar for the other embedding options we consider. We hereby summarize the key information derived from Table 4 as follows:

The matching methods used within MILE offer a qualitative benefit at a minimal cost to execution time. MILE generates better embeddings than MILE-rm using DeepWalk as the base embedding method. Though MILE-rm is slightly faster than MILE due to its random matching, its Micro-F1 score are consistently lower than of MILE.

The graph convolution based refinement learning methodology in MILE is particularly effective. Simple projection-based MILE-proj, performs significantly worse than MILE. The other two variants (MILE-avg and MILE-untr) which do not train the refinement model at all, also perform much worse than the proposed method. Note MILE-untr is the same as MILE except it uses a default set of parameters instead of learning those parameters. Clearly, the model learning part of our refinement method is a fundamental contributing factor to the effectiveness of MILE. Through training, the refinement model is tailored to the specific graph under the base embedding method in use. The overhead cost of this learning (comparing MILE with MILE-untr), can vary depending on the base embedding employed (for instance on the YouTube dataset, it is an insignificant 1.2% on DeepWalk but is still worth it due to qualitative benefits.

	PPI		Blog	
	Mi-F1	Time	Mi-F1	Time
DeepWalk	23	2.4	37	8.1
MILE (DW)	**25.6**	1.2	**42.9**	4.7
MILE-rm (DW)	25.3	1.01	40.4	3.6
MILE-proj (DW)	20.9	1.1	34.5	3.9
MILE-avg (DW)	23.5	1.1	37.7	3.8
MILE-untr (DW)	23.5	1.1	35.5	3.9
MILE-2base (DW)	25.4	2.2	35.6	6.7
MILE-gs (DW)	22.4	2.1	35.3	6.4
	Flickr		YouTube	
	Mi-F1	Time	Mi-F1	Time
DeepWalk	40	50.1	45.2	604.8
MILE (DW)	**40.4**	34.5	**46.1**	55.2
MILE-rm (DW)	38.9	26.6	44.9	55.1
MILE-proj (DW)	35.5	25.9	40.7	53.9
MILE-avg (DW)	37.2	25.9	41.4	55.2
MILE-untr (DW)	37.6	26.1	41.8	54.5
MILE-2base (DW)	37.7	53.3	41.6	94.7
MILE-gs (DW)	36.4	44.8	43.6	394.7

Table 4: MILE vs its variants. Except for the original methods, $m = 1$ for PPI/Blog/Flickr and $m = 6$ for YouTube. Time is in minutes.

Graph convolution refinement learning outperforms GraphSAGE. Replacing the graph convolution network with GraphSAGE for embedding refinement, MILE-gs does not perform as well as MILE. It is also computationally more expensive, partially due to its reliance on embeddings concatenation, instead of replacement, during the process the embeddings propagation (higher model complexity).

Double-base embedding learning is not effective. In the section – Intricacies of Refinement Learning – we discussed the issues with unaligned embeddings of the double-base embedding method for the refinement model learning. The performance gap between MILE and MILE-2base in Table 4 provides empirical evidence supporting our argument. This gap is likely caused by the fact that the base embeddings of level m and level $m+1$ might not lie in the same embedding space (rotated by some orthogonal matrix) (Hamilton, Ying, and Leskovec 2017). As a result, us-

[5]Self-loop weights are tuned to the best performance.

[6]Adapt code from https://github.com/williamleif/GraphSAGE

ing the projected embeddings $\mathcal{E}_m^p$ as input for model training (MILE-2base) is not as good as directly using $\mathcal{E}_m$ (MILE). Moreover, Table 4 shows that the additional round of base embedding in MILE-2base introduces a non-trivial overhead. On YouTube, the running time of MILE-2base is 1.6 times as much as MILE.

Conclusion

In this work, we propose a novel multi-level embedding (MILE) framework to scale up graph embedding techniques, without modifying them. Our framework incorporates existing embedding techniques as black boxes, and significantly improves the scalability of extant methods by reducing both the running time and memory consumption. Additionally, MILE also provides a lift in the quality of node embeddings in most of the cases. A fundamental contribution of MILE is its ability to learn a refinement strategy that depends on both the underlying graph properties and the embedding method in use.

Acknowledgments

This work is supported by the Air Force Research Laboratory under grant FA8650-19-2-2204 and by the National Institute of Health under grant NIH-1R01 HD088545-01A1. Ohio Supercomputer Center provided computational support under grant PAS0166.

References

Cao, S.; Lu, W.; and Xu, Q. 2015. Grarep: Learning graph representations with global structural information. In *CIKM*.

Chang, S.; Han, W.; Tang, J.; Qi, G.-J.; Aggarwal, C. C.; and Huang, T. S. 2015. Heterogeneous network embedding via deep architectures. In *KDD*, 119–128. ACM.

Chen, H.; et al. 2018. HARP: Hierarchical Representation Learning for Networks. In *AAAI*.

Chung, F. 2005. Laplacians and the Cheeger inequality for directed graphs. *Annals of Combinatorics* 9(1): 1–19.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*.

Dhillon, I. S.; Guan, Y.; and Kulis, B. 2007. Weighted graph cuts without eigenvectors a multilevel approach. In *PAMI*.

Dong, Y.; et al. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *KDD*.

Gleich, D. 2006. Hierarchical directed spectral graph partitioning. *Information Networks* .

Goyal, P.; and Ferrara, E. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151: 78–94.

Grover, A.; and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *KDD*.

Gurukar, S.; Vijayan, P.; et al. 2019. Network Representation Learning: Consolidation and Renewed Bearing. *arXiv preprint arXiv:1905.00987* .

Hamilton, W.; Ying, Z.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *NIPS*.

Hamilton, W. L.; Leskovec, J.; and Jurafsky, D. 2016. Diachronic word embeddings reveal statistical laws of semantic change. In *ACL*.

Harel, D.; and Koren, Y. 2000. A fast multi-scale method for drawing large graphs. In *GD*.

Huang, X.; Li, J.; and Hu, X. 2017. Accelerated attributed network embedding. In *SDM*.

Karypis, G.; and Kumar, V. 1998a. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20(1): 359–392.

Karypis, G.; and Kumar, V. 1998b. Multilevel k-way partitioning scheme for irregular graphs. In *JPDC*.

Kipf, T. N.; and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Lerer, A.; Wu, L.; Shen, J.; Lacroix, T.; Wehrstedt, L.; Bose, A.; and Peysakhovich, A. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *SysML*.

Liang, J.; Jacobs, P.; Sun, J.; and Parthasarathy, S. 2018. Semi-supervised Embedding in Attributed Networks with Outliers. In *SDM*.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *KDD*.

Qiu, J.; Dong, Y.; Ma, H.; Li, J.; Wang, K.; and Tang, J. 2018a. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *WSDM*.

Qiu, J.; Tang, J.; Ma, H.; Dong, Y.; Wang, K.; and Tang, J. 2018b. Deepinf: Social influence prediction with deep learning. In *KDD*.

Rossi, R. A.; and Ahmed, N. K. 2014. Role discovery in networks. *TKDE* .

Ruan, Y.; Fuhry, D.; Liang, J.; Wang, Y.; and Parthasarathy, S. 2015. Community discovery: Simple and scalable approaches. In *User Community Discovery*, 23–54. Springer.

Satuluri, V.; and Parthasarathy, S. 2009. Scalable graph clustering using stochastic flows: applications to community discovery. In *KDD*.

Tang, J.; Qu, M.; Wang, M.; Zhang, M.; Yan, J.; and Mei, Q. 2015. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, 1067–1077.

Veličković, P.; Cucurull, G.; et al. 2017. Graph attention networks. *ICLR* .

Wang, D.; Cui, P.; and Zhu, W. 2016. Structural deep network embedding. In *KDD*.

Wu, L.; et al. 2018. SocialGCN: An Efficient Graph Convolutional Network based Model for Social Recommendation. *ACL* .

Yang, C.; Sun, M.; Liu, Z.; and Tu, C. 2017. Fast network embedding enhancement via high order proximity approximation. In *IJCAI*.