BOOSTING LARGE-SCALE GRAPH EMBEDDING WITH MULTI-LEVEL GRAPH COARSENING

by
TAHA ATAHAN AKYILDIZ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
May 2020

BOOSTING LARGE-SCALE GRAPH EMBEDDING WITH MULTI-LEVEL GRAPH COARSENING

Approved by:

Asst. Prof. Kamer Kaya .....
(Thesis Supervisor)

Assoc. Prof. Hüsnü Yenigün .....

Assoc. Prof. Hasan Sözer .....

Date of Approval: September 4, 2020

ABSTRACT

BOOSTING LARGE-SCALE GRAPH EMBEDDING WITH MULTI-LEVEL GRAPH COARSENING

TAHA ATAHAN AKYILDIZ

Computer Science and Engineering, Master's Thesis, 2020

Thesis Supervisor: Asst. Prof. Kamer Kaya

Keywords: Graph coarsening, graph embedding, GPU, parallel graph algorithms, link prediction

Graphs can be found anywhere from protein interaction networks to social networks. However, the irregular structure of graph data constitutes an obstacle for running machine learning tasks such as link prediction, node classification, and anomaly detection. Graph embedding is the process of representing graphs in a multi-dimensional space, which enables machine learning tasks to be run on graphs. Although, embedding is proven to be advantageous by a series of works, it is compute-intensive. Current embedding approaches either can not scale to large graphs or they require expensive hardware for such purposes. In this work we propose a novel, parallel multi-level coarsening method to boost the performance of graph embedding both in terms of speed, and accuracy. We integrate the proposed coarsening approach into a GPU-based graph embedding tool called *Gosh*, which is able to embed large-scale graphs with a single GPU at a fraction of the time compared to the state-of-the-art. When coarsening is introduced, the run-time of *Gosh* improves by 14× while scoring greater *AUCROC* for the majority of medium-scale graphs. For the largest graph in our data-set with 66 million vertices, and 1.8 billion edges, embedding takes under an hour, and 93.4% AUCROC is achieved. Moreover, we investigate the relationship between quality of the coarsening on the quality of the embeddings. Our preliminary experiments show that the coarsening decisions must be balanced and the proposed coarsening strategy *novel* performs well for graph embedding.

ÖZET

BÜYÜK ÖLÇEKLİ ÇİZGE GÖMME İŞLEMLERİNİ İYİLEŞTİRMEK İÇİN ÇOK KATMANLI ÇİZGE İNDİRGEME

TAHA ATAHAN AKYILDIZ

Bilgisayar Bilimi, Yüksek Lisans Tezi, 2020

Tez Danışmanı: Asst. Prof. Kamer Kaya

Anahtar Kelimeler: çizge irileştirme, çizge katıştırma, ekran kartı, bağlantı tahmini, paralel algoritmalar

Çizgeler protein etkileşim ağlarından sosyal ağlara hemen her yerde bulunmaktadır. Fakat çizgelerin düzensiz veri yapısı, çizgelerin üzerinde bağlantı tahmini, düğüm sınıflama ve aykırılık belirleme gibi makine öğrenmesi görevleri çalıştırmak adına bir engel teşkil etmektedir. Çizge gömme, çizgeleri çok boyutlu bir uzayda tanımlayarak, çizgeler üzerinde makine öğrenmesi görevlerinin kolayca çalıştırılabilmesini sağlamaktadır. Literatürde bir dizi çalışma bu metodun faydalarını göstermiş olsa da, çizge gömme yoğun işlem teşkil eden bir metottur. Güncel gömme uygulamaları, ya büyük ölçekli çizgeleri işleyememekte ya da işlemek için pahalı bir donanım gerektirmektedir. Bu çalışmada orijinal, paralel ve çok katmanlı bir çizge indirgeme metodu ileri sürmekteyiz. Bu metot çizge gömmenin performansını hem zaman hem de doğruluk açısından geliştirmektedir. Bu çalışmada, bahsedilen metot, *Gosh* adlı büyük ölçekli çizgeleri tek bir ekran kartı ile işleyebilen bir çizge gömme uygulamasına entegre edilmiştir. Çizge indirgeme metodu entegre edildiğinde, *Gosh* ortalama olarak 14 kat daha hızlı çalışmakta ve orta büyüklükteki çizgelerin çoğunda daha başarılı AUCROC değerleri elde etmektedir. Veri setindeki, 66 milyon düğüm ve 1.8 milyar bağlantı bulunan en büyük çizgeyi, *Gosh*, 93.4% AUCROC elde ederken işlemi bir saatin altında tamamlamıştır. Bunlara ek olarak bu çalışmada, çizge indirgeme kalitesinin çizge gömme kalitesine etkisini incelemekteyiz. Deneylerimiz indirgeme sürecinin dengeli olması gerektiğini ve ileri sürülen indirgeme metodunun çizge gömme açısından üstün bir performans sergilediğini göstermektedir.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Kamer Kaya for his endless support, and my family and friends for supporting me throughout my work in Sabancı.

To my family & friends

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Graphs are ubiquitous. They can be found anywhere from social and communication networks to co-occurrence and protein interaction networks, and many more. Judicious analysis of graphs yields far-reaching insights to many areas of research and industry. Recently, there has been a growing interest in the literature in representing graph vertices in vector space, where a vertex is represented by a relatively small number of dimensions. This type of low dimensional representation of graphs, namely *graph embedding*, paves the way to running machine learning tasks such as link prediction, node classification, and anomaly detection on graphs. However, graph embedding is a computation-intensive process, where naive implementations do not scale to real-world sized graphs. One way to tackle this issue is to reduce the size of the graphs without disturbing the structural properties of the original graph. A popular method in the literature, *graph coarsening*, is an efficient, and effective way for approximating large graphs with smaller ones. Our preliminary experiments show that leveraging *graph coarsening* not only improves the run-time, but also the quality of the embedding.

In the literature, there have been a series of works which proposed powerful graph embedding methods. However, these approaches, even with parallel implementations, can not scale to real-world sized graphs which are relatively larger. Other works that utilized coarsening (Chen, Perozzi, Hu & Skiena, 2017; Liang, Gurukar & Parthasarathy, 2018) are also unable to scale to larger graphs, and they do not have parallel implementations for coarsening. The only GPU implementation, *GraphVite* (Zhu et al., 2019), is able to relax this limitation but requires multiple GPUs, and does not leverage coarsening.

In this thesis, we propose a novel, parallel, and multi-level coarsening algorithm for graph embedding. The algorithm shrinks graphs efficiently and prevents giant vertex sets from forming. This is achieved by introducing a new coarsening method called MultiEdgeCollapse, where the vertices are sorted in terms of their degree and processed in descending order. Moreover, vertices that have a relatively higher degree are not permitted to merge. We also present *Gosh*, a CPU-GPU hybrid,

multi-level graph embedding tool that leverages coarsening for boosting both the speed of the tool and the quality of the embeddings generated. With a single GPU, *Gosh* can handle any graph that fits on the host memory. First, a coarsened set of graphs is obtained by iteratively shrinking the graph. Then starting from the coarsest graph, GPU embedding is executed, and the initial embedding is obtained. Then using the coarsening information, the embedding is expanded for the training on the next level. This process is repeated until embedding is executed on the original graph, and the final embedding is obtained.

The contributions of the thesis can be summarized as follows:

- A novel multi-level coarsening algorithm is proposed, which is able to shrink graphs efficiently and boost the performance of graph embedding in terms of both speed and accuracy. When coarsening is introduced, the run-time of *Gosh* improves by 14× on medium-scale graphs. Moreover the version with coarsening scores greater *AUCROC* for 5 out of 8 graphs.

- We further introduce a parallel version of the novel coarsening algorithm, which generates similar coarsening sets compared to the sequential implementation while being up to 7.5× faster. On the largest graph in our data-set, parallel coarsening constitutes an 80% improvement on the run-time of *Gosh*.

- To the best of our knowledge, we are the first ones to analyze the quality of the coarsening on the quality of the embeddings. An extensive set of experiments using four different coarsening strategies demonstrates that *smart* coarsening has a positive impact on the quality of the embeddings.

- Multilevel coarsening and smart work distribution across levels enable *Gosh* to generate accurate embeddings at a fraction of the time compared to the state-of-the-art. For instance, on the graph `com-lj`, *GraphVite*, a state-of-the-art GPU-based embedding tool, spends around 11 minutes to reach 98.33% AUCROC score on the task of link prediction, while *Gosh* is able to score 98.33% in a single minute. Furthermore, according to Zhu et al. (2019), *GraphVite* takes 20 hours with 4 Tesla P100 GPUs on the graph `com-friendster` which has 60 million vertices and 1.8 billion edges. On a single Titan X GPU, *Gosh* reaches 93.4% link prediction AUCROC score within 45 minutes.

The rest of the thesis is organized as follows: in Chapter 2, the notation used in the thesis is given. In Chapter 3, the high-level description of *Gosh* is outlined. Following that, in Chapter 4, graph coarsening, and in Chapter 5, graph embedding, are described in detail by summarizing related work from the literature. The proposed algorithms are also introduced in Chapters 4 and 5. In Chapter 6, these algorithms

are judiciously evaluated, and comparisons with state-of-the-art tools are provided. Chapter 7 concludes the thesis.

2. BACKGROUND AND NOTATION

A graph $G = (V, E)$ is a collection of nodes represented by V, and E represents the connection information between V, where $E \subseteq (V \times V)$. $\forall (u, v) \in E$, if G is undirected $(u, v) \in E \implies (v, u) \in E$. On the other hand, if G is directed (u, v) does not imply (v, u). The set of outgoing neighbors of a vertex $u \in V$ is denoted as $\Gamma^+(u) = \{v \in V : (u, v) \in E\}$. Similarly, the incoming neighbors of a vertex $v \in V$ is denoted as $\Gamma^-(v) = \{u \in V : (u, v) \in E\}$. The set of all neighbors of a vertex $u \in V$ is denoted as $\Gamma(v) = \Gamma^-(v) \bigcup \Gamma^+(v)$. For undirected graphs, $\Gamma(v) = \Gamma^+(v) = \Gamma^-(v)$.

An embedding of G is a matrix $\mathbf{M}$ of size $|V| \times d$, where $|V|$, and d is the amount of rows, and columns respectively. A row $\mathbf{M}[v]$ is a vector of features representing a vertex $v \in V$. Various random-walk based embedding methods are proposed in the literature (Grover & Leskovec, 2016; Perozzi, Al-Rfou & Skiena, 2014; Tang, Qu, Wang, Zhang, Yan & Mei, 2015; Tsitsulin, Mottin, Karras & Müller, 2018; Zhu et al., 2019). For all of the mentioned works, updating the embedding vector is similar where stochastic gradient decent algorithm is utilized for optimization. *Gosh* employs the embedding method of *VERSE* (Tsitsulin et al., 2018), which is empirically shown to be faster and has a smaller memory footprint compared to the state-of-the-art. Furthermore, *VERSE* can be operated with various similarity measures Q, which is especially effective for different machine learning tasks. *VERSE* defines two distribution on a vertex, where the first one sim^v_Q is computed using the similarity measure Q, and the other one sim^v_E is computed by calculating the cosine similarities of the embedding vector $\mathbf{M}[v]$ to every other vertex $u \in V$. As a post-processing step, a soft-max layer is applied to normalize the obtained distributions. *VERSE* aims to minimize the difference between sim^v_E, and sim^v_Q, which is also discussed as minimizing the Kullback-Leibler divergence by Tsitsulin et al. (2018).

During *VERSE* embedding, a logistic regression classifier is trained in order to distinguish the positive samples selected from sim^v_Q, and the negative samples selected from a noise distribution N. To be more precise, $\forall v \in V$, e number of positive updates, and $e \times n_s$ number of negative updates are executed. In Algorithm 1, the

Algorithm 1: UPDATEEMBEDDING

Data: $\mathbf{M}[v]$, $\mathbf{M}[sample]$, b, lr
Result: $\mathbf{M}[v]$, $\mathbf{M}[sample]$
1 $score \leftarrow b - \sigma(\mathbf{M}[v] \odot \mathbf{M}[sample]) \times lr$;
2 $\mathbf{M}[v] \leftarrow \mathbf{M}[v] + \mathbf{M}[sample] \cdot score$;
3 $\mathbf{M}[sample] \leftarrow \mathbf{M}[sample] + \mathbf{M}[v] \cdot score$;

details of a single update on the embedding is depicted. Given the embedding vector $\mathbf{M}[v]$ of the source $v \in V$, the embedding vector $\mathbf{M}[sample]$ of the sample, b for indicating the sign of the update, and lr as the learning rate; first a score is calculated using the learning rate, sign indicator, and the sigmoid σ of the dot product of the vectors. Then using the calculated score both embeddings are updated accordingly.

Given $G = (V, E)$, graph coarsening is the process of structurally approximating G with a new graph $G' = (V', E')$ such that G' has fewer vertices and edges. This is done through means of collapsing (disjoint) sets of vertices in G into super-vertices which will form the vertex set of G'.

In a multi-level setting, the initial graph $G_0 = G$ is coarsened in multiple levels and a set $\mathcal{G} = \{G_0, G_1, \cdots, G_{D-1}\}$ of graphs is generated where G_{D-1} is the coarsest, i.e., the smallest graph. In this work, we evaluate the *efficiency* of a coarsening level based on the rate of shrinking defined as

$$(|V_{i-1}| - |V_i|)/|V_{i-1}|.$$

We followed a vertex-centric measurement since the size of the embedding matrix and the number of samples required for an iteration change with respect to the number of vertices. We also consider the *effectiveness* of the overall coarsening strategy which compares the embedding quality of a strategy to that of another for the same graph embedded with the same parameters.

The notation used in the thesis is given in Table 2.1.

Symbol	Definition
$G_0 = (V_0, E_0)$	The original graph to be embedded.
$G_i = (V_i, E_i)$	Represents a graph, which is coarsened i times.
$\Gamma^+(u)$	The set of outgoing neighbors of vertex u.
$\Gamma^-(u)$	The set of incoming neighbors of vertex u.
$\Gamma(u)$	Neighborhood of u, i.e., $\Gamma^+_{G_i}(u) \bigcup \Gamma^-_{G_i}(u)$.
d	# features per vertex, i.e., dimension of the embedding.
n_s	# negative samples per vertex.
σ	Sigmoid function.
sim_m	Similarity metric used in training.
e	Total number of epochs that will be performed
lr	Learning rate.
D	Total amount of coarsening levels.
$\mathcal{G}$	The set of coarsened graphs created from a graph $G = G_0$.
p	Smoothing ratio for epoch distribution.
e_i	# epochs for coarsening level i.
$\mathbf{M}_i$	Embedding matrix obtained for G_i.
$\mathcal{M}$	The set of mappings used in coarsening.
map_i	Mapping information from G_{i-1} to G_i.
$\mathcal{V}_i$	The partitioning of vertex set V_i.
$\mathcal{P}_i$	The partitioning of embedding matrix $\mathbf{M}_i$.
K_i	# parts in $\mathcal{V}_i$.
P_{GPU}	# embedding parts to be placed on the GPU.
S_{GPU}	# sample pools to be placed on the GPU.
B	# positive samples per vertex in a single sample pool.

Table 2.1 Notation used in the thesis.

3. GOSH IN A NUTSHELL

Given a graph G_0, *Gosh* generates the embedding matrix $\mathbf{M}_0$ (see Algorithm 2). Mainly, two stages are required for this process namely, coarsening and training:

1.1 A set, $\mathcal{G} = \{G_0, G_1, \ldots, G_{D-1}\}$, of coarsened graphs is created iteratively (see left of Figure 3.1), where a *super* vertex/node $v \in V_i$ represents one or more vertices $u, .. \in V_{i-1}$ (Line 1). The mapping information is also stored for the graph for the correct projection of the embedding vectors, which is performed in the next stage.

1.2 The training process starts from the coarsest graph G_{D-1} in order to generate the first embedding matrix $\mathbf{M}_{D-1}$. Then the embedding vectors are projected to the respective locations in $\mathbf{M}_{D-2}$, and the training continues using G_{D-2}. To generalize the embedding, matrix $\mathbf{M}_i$ is trained with the graph $\mathbf{G}_i$ and projected to embedding matrix $\mathbf{M}_{i-1}$ (Lines 3- 10).

The training process is repeated until $\mathbf{M}_0$ is obtained (see right of Figure 3.1). To obtain $\mathbf{M}_{i-1}$ from $\mathbf{M}_i$ the mapping information of $\mathbf{G}_{i-1}$ is used, where $M_i[u] = M_{i-1}[v]$ iff $u \in V_i$ is a super node of $v \in V_{i-1}$.

Gosh is implemented in such a way that it provides support for large-scale graphs,

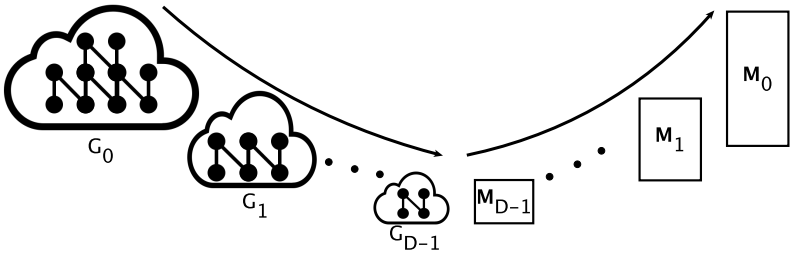


Figure 3.1 Multilevel embedding performed by *Gosh*: first, the coarsened set of graphs are generated. Then, the embedding matrices are trained until $\mathbf{M}_0$ is obtained. The original graph is coarsened down into a smaller graph, and then that graph is coarsened, and so on. The smallest graph is then embedded, and its embeddings are projected onto the higher graph, and then that graph is embedded and that continues upwards until the original graph is embedded.

which the memory footprint of the embedding matrix, and the graph itself exceeds the memory of the device. The size of the matrix is approximately 16G for practical sizes, e.g., $|V| = 128$M and $d = 128$. To add, with double precision, one needs to have 128GB memory on the device to store the entire matrix. This is not possible for contemporary devices. *Gosh* can handle graphs of all sizes with a single GPU.

Algorithm 2: *Gosh*

Data: G_0, n_s, lr, lr_d, p, e, $threshold$, P_{GPU}, S_{GPU}, B
Result: $\mathbf{M}$

```
1  G ← MultiEdgeCollapse (G_0, threshold);
2  Randomly initialize M_{D-1};
3  for i from D-1 to 1 do
4      e_i ← calculateEpochs(e, p, i);
5      if G_i and M_i fits into GPU then
6          CopyToDevice(G_i, M_i);
7          M_i ← TrainInGPU (G_i, M_i, n_s, lr, lr_d, e_i);
8      else
9          M_i ← LargeGraphGPU(G_i, M_i, n_s, lr, lr_d, e_i, P_GPU, S_GPU, B);
10     M_{i-1} ← ExpandEmbedding(M_i, map_{i-1});
11 return M_0;
```

For all the graphs in G, if both G_i and $\mathbf{M}_i$ can fit in the GPU memory (Line 5), G_i and the projection of $\mathbf{M}_i$ is directly copied to the device. *Thanks to coarsening, this is the case for many of the levels during the embedding process even when the original graph is huge.* For this case, the embedding process is completed in a single step (Lines 6-7), where the samples are generated on the GPU. Otherwise, the samples are generated on the CPU and the embedding is carried out by copying respective portions of the samples, $\mathbf{M}_i$ and $\mathbf{G}_i$ in batches (Line 9) (see Figure 3.2). This thesis focuses on the coarsening algorithm and multi-level embedding which reduces the cost significantly. The details of the case where the graph and the embedding vector do not fit in the GPU memory can be found in (Akyildiz, Aljundi & Kaya, 2020).

The multi-level nature of *Gosh* brings out an interesting problem: *how one will distribute the epoch budget e to the levels?* A naive approach would be to distribute the epochs evenly through out the levels. If fewer epochs are reserved for the higher levels the embedding process will be faster. To add, the corresponding embedding matrices will have a significant impact on the overall process as they are projected to further levels. On the contrary, if more epochs are reserved for the higher levels the embedding will be more fine tuned. Based on our preliminary experiments, where we tried a combination of uniform, and geometric distribution of the epochs, *Gosh* performs best with a mixed strategy. As a default, a portion p of the epochs are

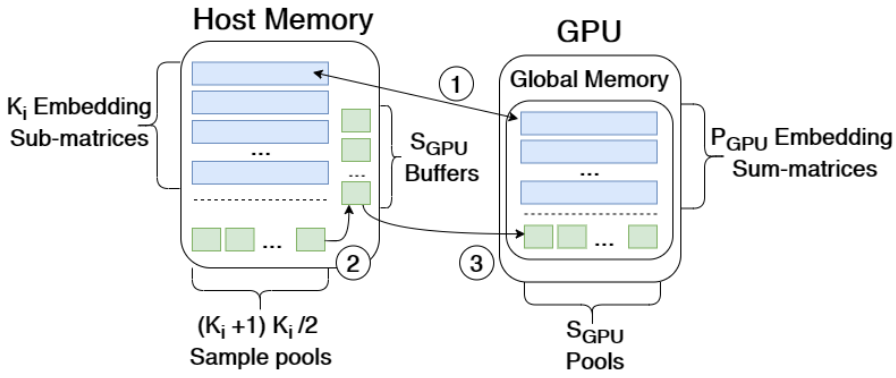


Figure 3.2 Memory model of large graphs algorithm embedding graph G_i. 1) Embedding sub-matrices are copied between the host and GPU as needed, 2) When sample pool $S_i^{j,k}$ is ready, it is copied to an empty buffer. 3) When a sample pool on the GPU is used up, it is replaced by the next sample pool from the buffer.

distributed uniformly. The remaining $p-1$ epochs are distributed geometrically. Training at level i uses $e_i = e/D + e'_i$ epochs where e'_i is half of e'_{i+1}. The value p is called the *smoothing ratio* and is left as a configurable parameter for the user to establish an interplay between the performance and accuracy. The epoch distribution strategy is also left as a configurable parameter, where the user can choose the default, or a completely uniform, or geometric distribution.

Learning rate is another configurable parameter of *Gosh* that significantly affects the quality of multi-level embedding. Once again, because of the multi-level nature of the algorithm, another question arises: *how to set the learning strategy for each level?* In short, *Gosh* uses the same initial learning rate for all the levels. In other words, the algorithm resets the learning rate as the initial input for the training of each $\mathbf{M}_i$ and decrease it after each epoch. The learning rate for epoch j at the ith level is equal to $lr \times \max\left(1 - \frac{j}{e_i}, 10^{-4}\right)$.

4. GRAPH COARSENING

As data becomes an integral part of our daily lives in terms of both personal and commercial use, the amount of data generated, and stored is increasing rapidly. Similarly, real world graph data is also getting larger and larger each and every day. Although large amounts of data is desirable for applications, processing such data on a modern processor is becoming impractical due to overlong run-times. This is especially troublesome for graph algorithms, where the time complexity scales with the amount of vertices and edges in the graph. To tackle this problem, researchers focused on finding generic ways to simplify graphs, where the main goal is to decrease the size of the graph while preserving its structural properties. There exist two approaches in the literature; namely, *graph sparsification* and *graph coarsening*:

Graph sparsification aims to decrease the amount of edges that are present in the graph while preserving the amount of vertices in the graph. In other words, the sparsified graph is an approximation of the original graph. In recent years, it is shown that any arbitrary graph can be represented by a sparser version in terms of pairwise distances (Peleg & Schäffer, 1989), eigenvalues (Spielman & Teng, 2008), and cuts (Tsay, Lovejoy & Karger, 1999). Moreover, these techniques are also utilized in applications where the amount of edges constitutes a bottleneck (Batson, Spielman, Srivastava & Teng, 2013; Calandriello, Lazaric, Koutis & Valko, 2018).

Graph coarsening, similar to graph sparsification, reduces the number of edges, and additionally reduces the number of vertices by grouping vertices under super vertices. This process results in a coarser version of the input graph, where each vertex in the coarse graph represents/contains one or more vertices in the original graph. Due to its ability to shrink input data, it is mainly, but not exclusively, used to accelerate algorithms that have a high time complexity. In the literature, coarsening is widely adopted in algorithms that apply a multi-level setting. Graph partitioning (Hendrickson & Leland, 1995; Karypis & Kumar, 1998a) and visualization (Harel & Koren, 2000; Hu, 2005) research pioneered the adoption of graph coarsening algorithms in computer science. Recent studies show that graph coarsening is further utilized in machine learning research that operates on graph-structured data (Gavish, Nadler

& Coifman, 2010; Lafon & Lee, 2006). Furthermore a series of works shows how graph coarsening is utilized in Convolutional Nueral Networks (CNNs) (Bronstein, Bruna, LeCun, Szlam & Vandergheynst, 2017; Bruna, Zaremba, Szlam & LeCun, 2013). Although coarsening is widely adopted, and is utilized in various applications, unlike graph sparsification, it does not have an established theory in the literature. Graph coarsening approaches, and implementations vary greatly especially in different branches of graph research. Although there have been recent attempts to demystify graph coarsening (Loukas, 2018), its success still remains a mystery.

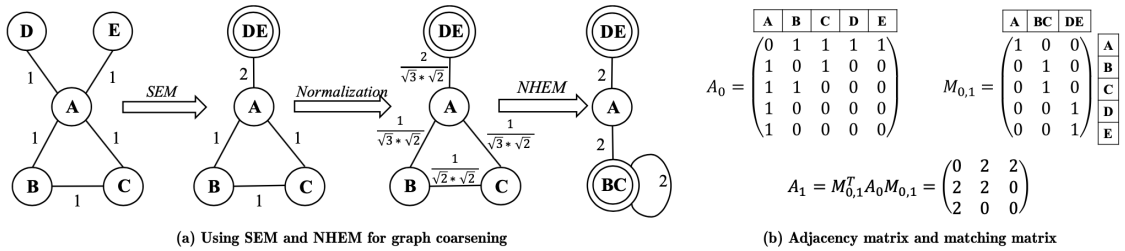
4.1 Graph Embedding Frameworks that Utilize Coarsening

To the best of our knowledge, there are two studies in the literature, which apply multi-level graph embedding by the means of coarsening (Chen et al., 2017; Liang et al., 2018). Intriguingly, coarsening is applied in order to improve different paradigms of the respective algorithms. While MILE (Liang et al., 2018) aims to accelerate the embedding process, HARP (Chen et al., 2017) proposes that the multi-level setting can be utilized to boost the accuracy of preexisting embedding algorithms.

4.1.1 MILE: A Multi-Level Framework for Scalable Graph Embedding

MILE (Multi-Level Embedding Framework) proposes a novel algorithm to relax computational complexity and memory requirement limitations of preexisting embedding methods. MILE shows that contemporary embedding algorithms cannot scale to millions of vertices, and edges on a modern processor. MILE tackles this problem by repeatedly shrinking the graph into smaller ones by utilizing a hybrid matching algorithm. Then it only runs embedding on the smallest graph, where the embedding method can be selected from the existing methods in the literature. Finally, it refines the embedding for the coarsest graph through a Graph Convolutional Neural Network (GCNN) up to the original graph, resulting with the final embedding.

MILE applies a hybrid matching scheme, which includes SEM (Structural Equivalence Matching) and NHEM (Normalized Heavy Edge Matching), as demonstrated



(a) Using SEM and NHEM for graph coarsening

(b) Adjacency matrix and matching matrix

Figure 4.1 MILE Coarsening Strategy

in Figure 4.1 (Karypis & Kumar, 1998b). With SEM, two vertices are matched if and only if they are incident on the same set of neighbourhoods, where the vertices are interchangeable and structurally equivalent. With NHEM, for an unmatched vertex u, a neighbour of u, v, is matched where the weight of the edge (u, v) is the largest. For NHEM, edges are normalized in a fashion where the ones connecting to high-degree vertices are penalized, which in turn prevents forming huge super vertices. To coarsen a graph, first SEM is applied and all the structurally equivalent vertices are matched. Then the edges are normalized. Normalization is followed by NHEM and all the matched vertices are collapsed under respective super vertices. This processes is applied iteratively until the coarsest graph is obtained.

After the coarsest graph is obtained, a baseline method for embedding is run, and an embedding for the coarsest graph is generated. For the embedding algorithm, MILE provides the following approaches by Perozzi et al. (2014), Cao, Lu & Xu (2015), Grover & Leskovec (2016) and Qiu, Dong, Ma, Li, Wang & Tang (2018), but it can be extended to any baseline method.

For the final step of the MILE framework, first, the embedding vectors of the coarser graphs are directly projected to the embedding of the larger graphs. All the vertices that are collapsed under the same super vertex are assigned the same embedding vector. This constitutes a big problem, where the problem only gets more serious as more levels of coarsening is introduced. Hence, during projection, embedding vectors are refined with the help of a GCN. GCN takes the projections, and graph adjacency matrix as input. Embeddings go trough l convolution layers and the refined versions are provided as an output. We refer the reader to (Kipf & Welling, 2016a) for more on GCNs.

MILE is evaluated for node classification (Perozzi et al., 2014). The data-sets used in the experiments range from 4 thousand vertices and 37 thousand edges to 9 million vertices and 40 million edges. According to the results, MILE is able to speed up the embedding process up to an order of magnitude (depending on the level of coarsening), without degrading the quality of the embeddings. For some graphs,

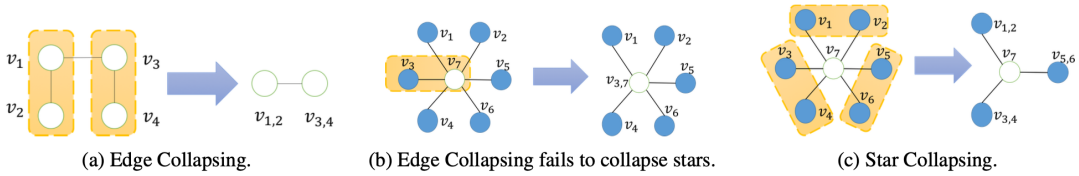


Figure 4.2 HARP Coarsening Strategy

MILE performs better than the baseline algorithm and for some graphs it performed worse without a substantial difference. However, since embedding is only applied to the last level, as the coarsening levels increase the quality of the final embedding decreases. Although MILE improves the run time of embedding algorithms without degrading the quality of the embeddings, it still comes short for graphs which have tens of millions of vertices and billions of edges. The main bottleneck for MILE is the coarsening process which is analyzed in Chapter 6.

4.1.2 HARP: Hierarchical Representation Learning for Networks

HARP (Hierarchical Representation Learning for Networks) by Chen et al. (2017) is a general meta-strategy to improve state-of-the-art algorithms for graph embedding. It recursively coarsens the input graph to get a set of smaller graphs with the same structure of the original graph. Unlike MILE, after coarsening, HARP runs embedding on all the levels. First, embedding is run on the coarsest level, then the vectors of the generated embedding are projected to the embedding of the larger graph. This process is repeated until the learning phase is completed on the original graph. The authors claim that this schema addresses several shortcomings of state-of-the-art (Grover & Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015) embedding algorithms. First, all of the sample-based, state-of-the-art embedding algorithms focus on extracting information from the imminent neighbourhood of vertices. According to the authors, this completely ignores long-distance global patterns. Second, since all the algorithms utilize stochastic gradient decent, without a multi-level setting, the learning process can get stuck on a local minima.

HARP also applies a hybrid coarsening scheme. The scheme has two key parts; edge collapse, and star-collapse for preserving first-order, and second-order proximity respectively. With edge collapse (Hu, 2005), the vertices that have an edge in between are collapsed such that no vertex can be collapsed more than once. On the other hand, *star collapse* is an efficient coarsening method for graphs with large

degree (hub) vertices, where edge collapse needs a lot of iterations to coarsen. With star collapse, low degree peripheral vertices are mapped to the same super vertex as shown in Figure 4.2. The hybrid scheme first applies star collapse, and than adopts edge collapse for each coarsening step. Coarsening is applied recursively until a small enough graph, which has less than 100 vertices, is obtained.

The performance of HARP is evaluated on Node Classification with three graphs ranging from (3 thousand vertices, 5 thousand edges and 6 classes) to (10 thousand vertices, 333 thousand edges, and 39 classes). For the experiments, DeepWalk (Perozzi et al., 2014), Node2Vec (Grover & Leskovec, 2016), and Line (Tang et al., 2015) are used as baseline methods. HARP performed better than the respective baselines. It improved Line, DeepWalk, and Node2Vec 7, 5, and 2 percent on average respectively. Although HARP conceptually proves that coarsening boosts the quality of the embeddings, it can not scale to millions of vertices, and edges.

4.2 GOSH Coarsening

Gosh employs a fast algorithm to keep the structural information within the coarsed graphs while maximizing the coarsening *efficiency* and *effectiveness*. Coarsening efficiency at the ith level is measured by the rate of shrinking defined as

$$(|V_{i-1}| - |V_i|)/|V_{i-1}|.$$

On the other hand, the effectiveness is measured in terms of its embedding quality compared to other possible coarsenings of the same graph embedded with the same parameters. An agglomerative coarsening approach MultiEdgeCollapse, which generates vertex clusters in a way similar to the one used in (Chen et al., 2017) is adapted. In the ith level, given $G_i = (V_i, E_i)$, the vertices in V_i are processed one by one. If v is not marked, it is marked, and mapped to a cluster, i.e., a new vertex in V_{i+1} and its edges are processed. If an edge $(v, u) \in E_i$, where u is not marked, u is added to v's cluster. Then, all of the vertices in v's cluster are shrunk into a *super* vertex $v_{sup} \in G_{i+1}$.

MultiEdgeCollapse preserves both the first- and second-order proximites (Tang et al., 2015) in a graph. The former measures the pairwise connection between vertices, and the latter represents the similarity between vertices' neighborhoods. It achieves that by collapsing the vertices that belong to the same neighborhood

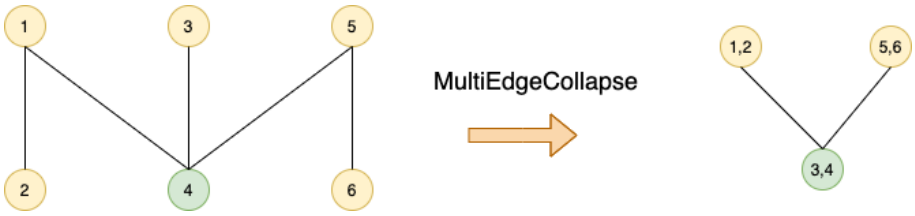


Figure 4.3 MULTIEDGECOLLAPSE: Since green vertex (4) has the highest degree it is processed first. Unlike vertex 3, vertices 4 and 5 can not be mapped to the same super vertex since their degrees are bigger than the density of the graph. Hence they are mapped to the same super with vertices 2, and 6 respectively.

around a local, hub vertex. However, if this process is handled carelessly, two, giant hub vertices can be merged. It is observed that this degrades the effectiveness and efficiency of the coarsening. The effectiveness degrades since the structural equivalence is not preserved in the lower levels of the coarsening, where most of the vertices are represented by a small set of super vertices. Furthermore having a small set of giant supers inhibits the graph from being coarsened further, resulting in an insufficient efficiency. To mitigate this, a new condition for matching is introduced to the algorithm, where $u \in V_i$ can not be put into the cluster of $v \in V_i$ if $|\Gamma_{G_i}(u)|$ and $|\Gamma_{G_i}(v)|$ are both larger than $\frac{|E_i|}{|V_i|}$. Consequently, assuming that the hub vertices will have a higher degree than the density of G_i, two of them can no longer be in the same cluster. Preliminary experiments show that this simple rule has a significant effect on both the efficiency and the effectiveness of the coarsening. The rule is further improved by changing the threshold from $\frac{|E_i|}{|V_i|}$ to $\frac{|E_i|}{|V_i|} + 1$ in order to be able to coarsen cliques.

As mentioned above, when a vertex is marked and added to a cluster, its edges are not processed further and it does not contribute to the coarsening. Performing the coarsening with an arbitrary ordering may degrade the efficiency since large vertices can be locked by the vertices with small neighborhoods. Hence, when an edge $(u, v) \in E_i$ is used for coarsening for a hub-vertex $v \in V_i$, to maximize efficiency, we prefer $u \in V_i$ to be inserted in to the cluster of origin v. To provide this, an ordering is procured by sorting the vertices with respect to their respective degrees and this ordering is used during coarsening. This ensures processing vertices with a higher degree before the vertices with smaller neighborhoods and this results in a substantial increase in the coarsening efficiency.

The details of the coarsening phase are given in Algorithm 3. The algorithm takes an uncoarsened graph $G = G_0$ and returns the set of coarsened graphs $\mathcal{G}$ along with the mapping information to be used to project the embedding matrices $\mathcal{M}$. $\mathcal{G}$ and $\mathcal{M}$ are initialized as $\{G_0\}$ and empty set, respectively. Starting from $i = 0$, the coarsening continues until a graph G_{i+1} with less than *threshold* vertices is

generated. As mentioned above, first the vertices in G_i are sorted with respect to their neighborhood sizes. Then the coarsening is performed and a smaller G_{i+1} is generated. We also store the mapping information map_i used to shrink G_i to G_{i+1}. This will be used later to project the embedding matrix M_{i+1} obtained for G_{i+1} to initialize the matrix M_i for G_i. To add, $threshold = 100$ is used for all the experiments in the paper which is the default value for *Gosh*.

Algorithm 3: MultiEdgeCollapse

```
   Data: G_0 = (V_0, E_0), threshold
   Result: 𝒢, ℳ
 1 𝒢 ← {G_0}, ℳ ← ∅, i ← 0;
 2 while |V_i| > threshold do
 3     order ← Sort(G_i);
 4     for v ∈ V_i do map_i[v] ← −1;
 5     δ ← |E_i|/|V_i|;
 6     cluster ← 0;
 7     for v in order do
 8         if map_i[v] = −1 then
 9             map_i[v] ← cluster;
10             cluster ← cluster + 1;
11             foreach (v,u) ∈ E_i do
12                 if |Γ_{G_i}(v)| ≤ δ or |Γ_{G_i}(u)| ≤ δ then
13                     if map_i[u] = −1 then
14                         map_i[u] ← map_i[v];
15     G_{i+1} ← Coarsen(G_i, map_i);
16     𝒢 ← 𝒢 ∪ {G_{i+1}}, ℳ ← ℳ ∪ {map_i}, i ← i + 1;
```

4.2.1 Complexity analysis:

All the algorithms, coarsening and embedding, use the Compressed Sparse Row (CSR) graph data structure. In CSR, an array, *adj* holds the neighbors of every vertex in the graph consecutively. This array is a list of all the neighbors of vertex 0, followed by all the neighbors of vertex 1, and so on. Another array, *xadj*, holds the starting indices of each vertex's neighbors in *adj*, with the last index being the number of edges in the graph. In other words, the neighbors of vertex i are stored in the array *adj* from $adj[xadj[i]]$ until $adj[xadj[i+1]]$.

MultiEdgeCollapse has three stages; sorting (line 3), mapping (lines 7–14) and coarsening (line 15). A version of *counting sort* is implemented for the first

stage with a time complexity of $\mathcal{O}(|V|+|E|)$. For mapping, the algorithm traverses all the edges in the graph. This has a time complexity of $\mathcal{O}(|V|+|E|)$. Finally, coarsening the graph requires sorting the vertices with respect to their mappings and going through all the vertices and their edges within the CSR, which also has a time complexity of $\mathcal{O}(|V|+|E|)$.

4.3 Parallel GOSH Coarsening

As the literature suggests, when the embedding is performed on the CPU, embedding dominates the total execution time. Nonetheless, with fast embedding as in *Gosh*, this is not the case. Thus, coarsening on the CPU is parallelized for *Gosh*.

For parallelization, we employ locks, which are needed mainly for two reasons: First, two threads can attempt to map the same vertex to two different mapped vertices at the same time. Second, a thread might attempt to map a vertex v (line 14 of Algorithm 3) while another is currently on the process of mapping (line 9) other vertices to v; this makes v both a mapped and a mapping vertex. Both of these occurrences lead to inconsistent coarsenings due to race conditions. To avoid race conditions, we use a lock per each entry of map_i. To update $map_i[v]$ and $map_i[u]$ as in lines 9 and 14, the thread first tries to lock $map_i[v]$ and $map_i[u]$, respectively. If the lock is obtained, the process continues. Otherwise, the thread skips the current candidate and continues with the next vertex. One caveat is the update on the counter *cluster*. Hence, instead of using a separate variable for super vertex ids, the parallel version uses the hub-vertex id for mapping. That is $map_i[v]$ is set to v unlike line 9 of the sequential algorithm. With this implementation, map_i does not provide a mapping to actual vertex IDs in G_{i+1}. This can be fixed in $\mathcal{O}(|V|)$ time via sequential traversals of the map_i array, which first detect/count the vertices that has $map_i[v] = v$ and reset the map_i values for all.

The parallel coarsened graph construction is not straightforward. After the mapping, the degrees of the (super) vertices in G_{i+1} are not yet known. To alleviate that, we allocate a private E_{i+1}^j region in the memory to each thread t_j, $1 \leq j \leq \tau$. These threads create the edge lists of the new vertices on these private regions which are then merged on a different location of size $|E_{i+1}|$. To do that, first a sequential scan operation is performed to find the region in E_{i+1} for each thread. Then, the private information is copied to E_{i+1}.

An important problem that needs to be addressed for all the steps above is load imbalance. Since the degree distribution on the original graph can be skewed and becomes more skewed for the coarsened graphs, a static vertex-to-thread assignment can reduce the performance. Hence, *Gosh* uses a dynamic scheduling strategy, which uses small batch sizes for all the steps above.

4.4 Grappolo

Thus far, this chapter elaborated on different coarsening strategies that are used for graph embedding. However, these strategies lack formal justification. The performance of the aforementioned coarsening algorithms can only be evaluated with the quality of the embeddings that are generated. Although it has not been directly used for graph embedding, high-quality, state-of-the-art clustering and community detection algorithms have been proposed in the literature. A well known community detection tool *Grappolo* (Halappanavar, Lu, Kalyanaraman & Tumeo, 2017; Lu, Halappanavar & Kalyanaraman, 2014) is selected for further investigating the effect of coarsening on the embeddings.

Grappolo is a CPU-parallel clustering tool. The tool is built upon the Louvain method (Blondel, Guillaume, Lambiotte & Lefebvre, 2008) which is an efficient, greedy, and iterative solution for generating hierarchy of communities (i.e., clusters). The main idea of Louvain is to maximize the modularity of the clusters. A cluster has a high modularity if it has dense connections in between the cluster and sparse connections with vertices belonging to different clusters. In a multilevel setting, at ith level, for $G_i = (V_i, E_i)$, the structure $P_i = \left(C_1^{(i)}, C_2^{(i)}, \ldots, C_k^{(i)}\right)$ is the communities where $1 \leq k \leq |V_i|$, the modularity of the graph is calculated with the following expression:

$$Q_i = \frac{1}{2m} \sum_{j \in V_i} e_{j \to C^{(i)}(j)} - \sum_{C \in P_i} \left(\frac{a_c}{2m} \times \frac{a_c}{2m} \right) \tag{4.1}$$

where $e_{j \to C^{(i)}(j)}$ denotes the sum of the edge weights in $E_{j \to C^{(i)}(j)}$ which is the set of edges that connects vertex $j \in V_i$ to the vertices in its community $C^{(i)}(j)$. The value a_C denotes the sum of the edge weights of all the vertices in community C (Lu et al., 2014; Newman & Girvan, 2004).

To maximize the modularity, the Louvain method applies the following steps iteratively: first, each vertex $v \in G$ is assigned to their own cluster C_i, where $|V_i| = |C_i|$ holds true after the initialization. Then for every vertex, the modularity gain for moving that vertex to its neighbouring communities are calculated. If all the move gains are negative it means that the current vertex already belongs to the correct community. Hence, no move operations are carried out. Alternatively, if there exists a move with a positive gain, the vertex is assigned to a neigbouring community with the maximum modularity gain. After each vertex is processed in this fashion, all the communities are collapsed under respective super vertices, where a new coarse version of the graph, $G_i = (V_i, E_i)$, is created. The output of each iteration, G_i, is the input for the next iteration. The algorithm is terminated once the modularity score converges.

Experimental results display the success of *Grappolo*. According to the results, its parallel implementation is able to produce communities with a better modularity output compared to the sequential implementation of the Louvain method. Moreover, *Grappolo* is up to 16× faster than the vanilla algorithm using 32 threads. Its success comes from the heuristics constructed for extracting parallelism out of the algorithm.

Although it is excelled for modularity maximization, as mentioned above, *Grappolo* is not proposed for graph embedding in a multilevel setting. In order to utilize *Grappolo*, *Gosh* is adapted to work with *Grappolo* clusters. First the code is patched to print out the communities for intermediate iterations. Then a graph re-constructor is written in order to generate a graph from *Grappolo* cluster information. The rest of the algorithm is the same with the original one, where the set of coarsened graphs is processed to extract an embedding as mentioned in the previous chapter.

5. GRAPH EMBEDDING

The first works on *graph embedding* are introduced in the early 2000s (Belkin & Niyogi, 2001; Roweis & Saul, 2000). These algorithms are developed utilizing dimensionality reduction techniques. For this approach, the connection information between the vertices is represented as a matrix. This matrix is then factorized to obtain an embedding. The main goal is to preserve the structural properties of the graph. However, there are two major problems with factorization based approaches. Firstly, the approach to factorize the matrix varies in terms of the properties of the graph, which inhibits such approaches to be generalized. Secondly, these approaches have a time complexity of $O(V^2)$, thus can not scale to real-world size graphs.

In recent years, similar to many areas of research, deep neural network based methods became popular for *graph embedding*. Wang, Cui & Zhu (2016) explored the possibility of preserving both the first, and second-order proximities by the means of deep auto encoders. Cao, Lu & Xu (2016) combined random surfing with deep auto encoders. However, both approaches are computationally expensive. Moreover, for each vertex, the global neighbourhood is required as input. Kipf & Welling (2016b) relaxed this limitation by defining a convolution operation on the graph, which iteratively aggregates the local neighborhood.

We encourage the reader to read (Cai, Zheng & Chang, 2017; Goyal & Ferrara, 2018) for more information on matrix factorization, and deep neural-network-based *graph embedding* approaches. In the rest of this chapter, first, random-walk-based graph embedding approaches will be presented. Then, *Gosh* embedding will be described in detail.

5.1 Random-Walk-based Graph Embedding

Random-walk-based methods are used in the literature for approximating the centrality and analyzing the similarity between vertices. Recently, a series of works (Grover & Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015; Tsitsulin et al., 2018) demonstrated the effectiveness of random walks in graph embedding. Furthermore, Zhu et al. (2019) showed that with a GPU implementation the run-time can be improved immensely without loosing any accuracy.

5.1.1 DeepWalk: Online Learning of Social Representations

DeepWalk (Perozzi et al., 2014) proposes a novel approach to graph embedding. The authors generalized the recent advancements in language modelling, and supervised feature learning (Bengio, Courville & Vincent, 2012) for this task. Especially, the advancements in language modelling, as in representing words as vectors (Mikolov, Chen, Corrado & Dean, 2013), paved the way for *DeepWalk*. The tool learns social representations of a graph's vertices $V_i \in G_i$ as $\mathbf{M}_i$ by utilizing a series of short random walks. Through these random walks, *DeepWalk* captures neighborhood similarity, and community membership. The authors state that their algorithm provides adaptable, community aware, low dimensional and continuous embeddings.

DeepWalk consists of two parts; a random walk generator, and following that, an update procedure. The random walk generator first samples a vertex. Then one of its neighbours is selected uniformly randomly. This constitutes a step of the walk. For *DeepWalk*, exactly t steps are taken to generate a complete walk, where t is a configurable parameter for the algorithm. Then all the collocations of the vertices visited in the walk are generated, and the embedding vectors are updated. Updates are carried out using *SkipGram* (Mikolov et al., 2013) algorithm.

DeepWalk is evaluated on the machine learning task of node classification. The authors distance themselves from previous work (Neville & Jensen, 2002), where, unlike *DeepWalk*, the label information is used in training. Three graphs are used for the evaluation. The largest one has one million vertices, and three million edges. The experiments are carried out in an iterative manner where the percentage of the number of label vertices is increased in each iteration. The experiments revealed that *DeepWalk* performs better than the competition. Only in a few experiments, *SpectralClustering* (Tang & Liu, 2011) performs better than *DeepWalk*. *DeepWalk*'s representations provide up to 10% higher F1-scores while in various experiments *DeepWalk* provides better scores with less labeled nodes.

5.1.2 LINE: Large-scale Information Network Embedding

LINE (Tang et al., 2015) proposes a graph embedding approach with a novel objective function which preserves the local, and global graph structures for various types of graphs. Moreover, *LINE* introduces a novel edge-sampling algorithm that tackles the shortcomings of the classical SGD (Stochastic Gradient Decent) algorithm. The efficiency and the effectiveness of the algorithm are demonstrated through empirical experiments.

LINE presents two new concepts for measuring the similarity between the vertices; first-order proximity and second-order proximity. The first-order proximity is defined as the local pairwise proximity between two vertices $u, v \in V_i$, i.e., the vertex set of the coarsened graph in the ith level, where the weight on the edge (u, v) indicates the magnitude. If no edge is incident between (u, v) then the first-order proximity is 0. The second order proximity of $u, v \in V_i$, is determined by the similarity between the neighborhoods of the respective vertices. Let S_v as the set containing the first-order proximities of $v \in V_i$ to $\forall u \in V_i$. The second-order proximity between u, and v is determined by the similarity between S_v, and S_u. If there is no common element in S_u, and S_v then the second-order proximity is zero.

To take the second-order proximity into account, first a random *source* vertex is selected. Then, a neighbour of the *source*, the *context* vertex, is chosen randomly proportional to the edge weights, where the vertices which have a stronger connection have a higher probability to be selected. The updates on the *source* vertex are performed on the original embedding, and the updates on the *context* vertex are performed on the *context embedding* which is a separate data structure used to carry the second-order information to its neighbor vertices. For this approach, the first-order proximity, and the second-order proximity is captured by the original embedding, and the context embedding respectively. Finally, the two embeddings are concatenated, and provided as the output.

The algorithm is evaluated on the machine learning task of node classification. The data-set which is used to evaluate *LINE* is comparatively larger than that of the data-set used for *DeepWalk*. The data-set consists of five graphs, where the smallest one contains 1 million vertices, and 3 million edges, and the largest one contains 2 million vertices, and 1 billion edges. As reported in the work, *LINE* not only decreases the run-time compared to the previous approaches but also improves the quality of the embeddings.

5.1.3 VERSE: Versatile Graph Embeddings from Similarity Measures

To the best of our knowledge, VERSE (Tsitsulin et al., 2018) is the latest of a series of works (Grover & Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015) that presents new similarity measures for graph embedding. VERSE also introduces a versatile framework that explicitly learns *any* similarity measure for graph vertices. The authors argue that real-world tasks rely on a mix of three different kinds of properties namely; community structure, roles, and structural equivalence. They state that a feature-learning algorithm should be able to capture all three properties. VERSE is able to achieve such a task by its inherent design. As stated, *any* similarity measure can be incorporated into VERSE without having the need to change its core.

By default VERSE has three instantiations that utilize different similarity measures sim_m; (i) Personalized Page Rank (PPR), (ii) Adjacency Similarity, and (iii) SimRank. Personalized Page-Rank similarity (Page, Brin, Motwani & Winograd, 1999) is a well-known similarity measure. Given an initial distribution s, the similarity measure can be defined as

$$\pi_s = (\alpha \times s) + (1 - \alpha) \times \pi_s \times A \tag{5.1}$$

where π_s is the current similarity vector and A is the normalized adjacency matrix. The average explored size of the neighbourhood is determined by the damping factor α. As shown by Page et al. (1999), a random walk with stopping probability of (1-α) converges to *PPR*. Thus, a sample in the *PPR* algorithm constitutes the starting vertex and the last visited vertex of the respective random walk. The adjacency similarity measure is actually a sub variant of *PPR*, where the damping factor is zero. For this measure, only the imminent neighbors of the starting vertex can be sampled. It is a powerful measure for tasks that require extracting first-order proximity like link prediction. The last measure, SimRank (Jeh & Widom, 2002), measures the structural equivalence (similarity) of vertices. In a nutshell, for SimRank, if two vertices' neighbourhoods are similar then the vertices are similar too. Although, VERSE performs best with SimRank, it is an exhaustive measure with a time complexity of $O(n^4)$. Hence, it is infeasible to use SimRank as a similarity measure when working with large-scale graphs.

VERSE is evaluated on various different machine learning tasks such as link prediction, node classification, node clustering, and graph reconstruction. The size of the graphs used for the experiments range from 10 thousand vertices, and 178 thousand edges to 3 million vertices, and 234 million edges. For the experiments VERSE uses three different variants; original VERSE with *PPR* similarity measure, HSVERSE

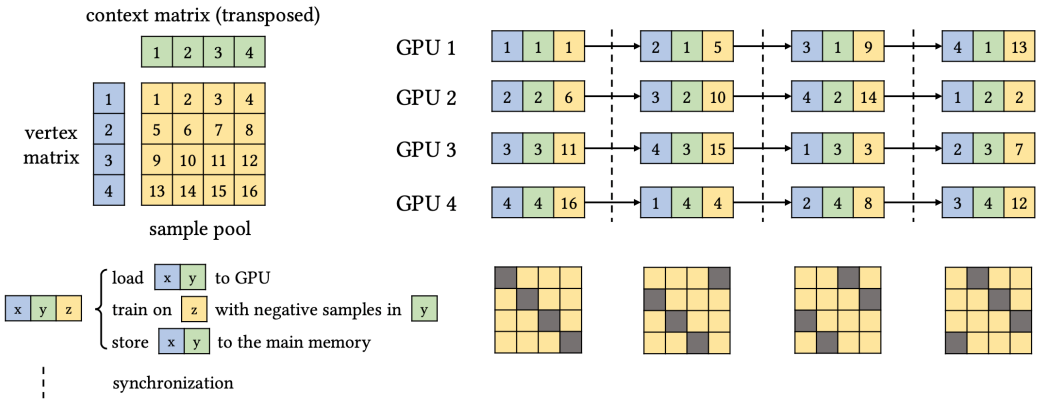


Figure 5.1 *GraphVite* Embedding with Multiple GPUs (Zhu et al., 2019)

which selects the best similarity measure out of the aforementioned three, and an exhaustive version FVERSE which is not scalable. Although the VERSE variants outperform the competition in general, for some graphs and tasks, DeepWalk and LINE surpasses the three, and for others, they produce comparable results.

5.1.4 GraphVite

To the best of our knowledge, *GraphVite* (Zhu et al., 2019) is the first, and besides *Gosh*, the only GPU-based graph embedding algorithm. As stressed in previous sections, CPU-based embedding algorithms are unable to scale to graphs with tens of millions of vertices. *GraphVite* successfully relaxes this limitation by introducing a CPU-GPU hybrid system, where augmented edge samples are generated in the CPU, and embedding is performed utilizing multiple GPUs. Moreover, an efficient synchronization algorithm is proposed to reduce the communication cost between CPUs and GPUs.

The first stage of *GraphVite* is to augment the original network with random walks. For this, *GraphVite* follows an online strategy similar to (Tang et al., 2015), where the samples are generated on the fly. To generate a batch of positive samples, a departure vertex is selected uniformly randomly. Then a random walk is performed starting from the selected vertex, where vertex pairs within a predefined distance are picked as positive samples. Finally, samples from multiple walks are gathered in a single sample pool and shuffled to increase the performance of optimization (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra & Riedmiller, 2013). In the embedding stage, the training process is divided into small fragments and

distributed among multiple GPUs (see Figure 5.1). For n GPUs, the *vertex* and *context* embedding are divided into n parts. This results in an $n \times n$ partition of the sampling pool, where a pair of blocks that does not share a row or column can be used for training concurrently without the need for synchronization. One down-side for this approach is that negative samples can only be generated from the blocks that are in the GPU. However, selecting negative samples from the entire graph would require extensive CPU-GPU communication which in turn would decrease the speed immensely.

For the evaluation of *GraphVite*, four graphs ranging from 1 million vertices and 5 million edges to 65 million vertices and 1.8 billion edges are used. Node classification and link prediction are used as machine learning tasks. *GraphVite* is able to obtain speedups up to 19× with 6 CPU cores and a single GPU. The speedup increases to 51× with 24 CPU cores and 4 GPUs while preserving the quality of the embeddings. Although size limitations are relaxed to a certain degree with *GraphVite*'s state-of-the-art CPU-GPU hybrid implementation, to embed graphs which have more than 12 million vertices, *GraphVite* requires multiple GPUs. Furthermore, a graph with 65 million vertices and 1.8 billion edges takes 20 hours to train using 4 Tesla P100 GPUs. Thus, as shown in Section 6.3, there is still a substantial amount of room for improvement.

5.2 GOSH Embedding

The learning step for *Gosh* is GPU parallel, lock-free, and configurable. The SGD-based updates, which are based on the similarity measures described in 5.1.3, are utilized for training. Unlike CPUs, working on a GPU makes an efficient, lock-free implementation harder to achieve. Niu et al. Niu, Recht, Re & Wright (2011) suggest that a lock-free SGD implementation, that have no mechanics to prevent race conditions, performs similar to a race-condition-free implementation in terms of the learning quality on multi-core processors. However, unlike multi-core processors, GPU's can run millions of threads in parallel. Our preliminary experiments show that with GPU's, race conditions significantly deteriorate the quality of the learning, hence the quality of the embeddings. Thus, *Gosh* follows a moderately more restricted implementation which is still not race free.

To reduce the number of race conditions one needs to carefully utilize the architecture of the device. First the epochs are synchronized using CUDA. This ensures

that no two epochs are processed concurrently. Furthermore, given an epoch, *Gosh* traverses V_i in parallel, and assigns *source* vertices to a single GPU-warp, where multiple positive/negative samples are used to update the embedding (see Algorithm 1) one after another (see Algorithm 4). The aforementioned features of *Gosh* ensures that a vertex $v \in V_i$ cannot be a source vertex for two concurrent updates. However, v can be selected as a positive, or a negative sample by another warp while v is assigned to a warp as a *source* vertex. Similarly, v can be sampled by two different source vertices concurrently. Although the updates on $\mathbf{M}_i[v]$ can be disturbed by race conditions, the synchronization implemented in *Gosh* are sufficient enough to robustly perform the embedding process (see Section 6.3).

For graphs that can fit in the device memory, as shown in Algorithm 4, both positive and negative samples are generated during training. Given a *source* vertex $v \in V_i$, the positive sample is selected uniformly random from $\Gamma_{G_i}(v)$. Moreover, negative samples are selected from a uniformly-random noise distribution, which is modelled over V_i. Depending on the similarity measure, considering each graph is G_i is highly sparse, negative sample selection has a very little probability of error. Algorithm 1 shows the updates performed after each sampling in Algorithm 4 in lines 4, 7.

Gosh utilizes shared memory for the updates on *source* vertices. During the updates, given a *source* vertex $v \in V_i$, a positive and n_s negative updates are performed. This constitutes $(1 + n_s) \times d$ accesses to $\mathbf{M}_i[v]$. Even for practical sizes like $n_s = 3, d = 128$; global memory access hinders the performance of *Gosh*. To improve the performance, first, $\mathbf{M}_i[v]$ is copied to shared memory. Then all updates for positive, and negative samples are performed on the shared memory. Finally, $\mathbf{M}_i[v]$ is copied back to global memory. Unlike the embedding vectors of *source* vertices, the embedding vectors of the sampled vertices $\mathbf{M}_i[u]$, where $u \in V_i$, are only updated once. Hence, these vectors are kept in global memory, where the reads and writes are performed in a round-robin fashion. This way of access on $\mathbf{M}_i[u]$ is coalesced. $\mathbf{M}_i[u][j + (32 \times k)]$ is accessed by thread j at the kth access where 32 is the number of threads within a warp.

5.2.1 Small Dimensions

Originally, in *Gosh*, an update on a sample $(u, v) \in V_i$ is carried out by a warp. To be more specific, we choose a source vertex $v \in V_i$, and carry out s negative and a positive update. These updates on the *source* vertex are all handled by a single warp. This is important for coalesced access (see 5.2). However, assuming that

Algorithm 4: TRAININGPU

Data: G_i, $\mathbf{M}_i$, n_s, lr, e_i
Result: $\mathbf{M}_i$

```
1 for j = 0 to e_i − 1 do
2     lr' ← lr × max(1 − j/e_i, 10^−4);
      /* Each src below is assigned to a GPU warp                    */
3     for ∀src ∈ V_i in parallel do
4         u ← GETPOSITIVESAMPLE(G_i);
5         UPDATEEMBEDDING(M_i[src], M_i[u], 1, lr' );
6         for k = 1 to n_s do
7             u ← GETNEGATIVESAMPLE(G_i);
8             UPDATEEMBEDDING(M_i[src], M_i[u], 0, lr');
```

a warp contains 32 threads, if one wants to train an embedding which has $d < 32$ then $32 - d$ number of threads will remain idle which yields to the under utilization of the device. Hence, the original implementation performs poorly on embeddings with smaller dimensions. To mitigate this problem a specialized implementation for dimensions smaller than 32 is integrated to *Gosh*. The number of threads responsible for a source vertex is set as the smallest multiple of 8 larger than or equal to d, i.e., 8 or 16. Hence, depending on d, we can assign 2 or 4 samples to a single warp.

6. EXPERIMENTS

In this chapter, first the system configurations, state-of-the-art tools, and datasets used in the experiments are described. Then, the evaluation pipeline will be described in detail. Following that, experiments on coarsening performance and quality will be displayed. Lastly, results on embedding quality will be given, and the speed-up breakdown of *Gosh* is presented.

System configuration: For the experiments, a single machine with 2 sockets, each with 8 Intel E5-2620 v4 CPU cores running at 2.10GHz with two hyper-threads per core (32 logical cores in total), and 198GB RAM is used. To avoid the effects of hyper-threading, only 16 threads are used for parallel executions. The GPU experiments use a single Titan X Pascal GPU with 12GB of memory.

The server has `Ubuntu 4.4.0-159` as the operating system. CPU codes are compiled with `gcc 7.3.0` with `-O3`. For CPU parallelization, `OpenMP` multi-threading is used. For GPU implementations and compilation, `nvcc` with `CUDA 10.1` and optimization flag `-O3` are used. The GPUs are connected to the server via `PCIe 3.0 x16`. For GPU implementations, all the relevant information for the calculations are stored on the device, unified memory is not used.

Tools used for evaluation: The following state-of-the-art tools are selected to compare and evaluate the performance of *Gosh*:

VERSE: PPR similarity measure and $\alpha = 0.85$ are used as recommended by the authors. For *VERSE*, the epoch number and the learning rate are set to $e \in \{600, 1000, 1400\}$ and $lr = 0.0025$, respectively. Out of the three runs, the best AUCROC score is reported.

GraphVite: The default values are used for the hyper-parameters as recommended by the authors and LINE is chosen as the base embedding method. Two settings are created for *GraphVite*; a fast setting with $e = 600$ epochs, and a slow setting with $e = 1000$ epochs.

MILE: The following parameters are used for the model: DeepWalk as a base

embedding method, MD-GCN as a refinement method, 8 levels of coarsening, and a learning rate of $lr = 0.001$. *MILE* does not allow for the number of epochs to be configured.

For *Gosh*, four different versions, *Gosh*-ultra-fast, *Gosh*-fast, *Gosh*-normal, and *Gosh*-slow is used with the parameters given in Table 6.2. The configurations differ in terms of the number of epochs, smoothing ratio, and learning rate. As the number of epochs e and the smoothing ratio p decrease, lr increases to compensate for the reduced amount of work on the original graph with faster learning. The only difference between *Gosh*-ultra-fast, and *Gosh*-fast is the amount of epochs designated for training. Furthermore, we include a version of *Gosh* which does not perform coarsening. This configuration spends all of the epochs on the original graph. In addition to these differences, for medium-scale graphs, a larger number of epochs is used for each configuration compared to large-scale graphs.

For *Gosh* and *VERSE*, a single epoch is defined as executing $|E|$ number of updates on the embedding M_i in order to match the definition of an epoch given by *GraphVite* Zhu et al. (2019) for the fairness of the comparisons.

Datasets: We use various graphs in the evaluation process to cover many structural variations and to evaluate the tools in terms of performance and quality as fairly and thoroughly as possible. The graphs differ in terms of their origin, the number of vertices, and density. The properties of these graphs are given in Table 6.1. The medium-scale graphs, with less than 10M vertices, and large-scale ones are separated in the table.

6.1 Evaluation Pipeline

The embedding quality of *Gosh*, *VERSE*, *MILE*, and *GraphVite* are evaluated with link prediction, which is one of the most common machine learning tasks used in the literature to evaluate graph embedding tools (Grover & Leskovec, 2016; Lerer, Wu, Shen, Lacroix, Wehrstedt, Bose & Peysakhovich, 2019; Tsitsulin et al., 2018; Zhu et al., 2019).

For evaluation, the input graph G is split into train and test sub-graphs as $G_{train} = (V_{train}, E_{train})$ and $G_{test} = (V_{test}, E_{test})$. G_{train} contains 80% of the edges of G, where G_{test} contains the remaining 20%. To make sure $V_{test} \subseteq V_{train}$, all the isolated vertices in G_{train} are removed. In addition to this, all $(u, v) \in G_{test}$ edges, where u or

Table 6.1 Medium- and large-scale graphs used in the experiments. Thanks to Leskovec & Krevl (2014) for `com-dblp`, `com-amazon`, `soc-pokec`, `wiki-topcats`, `com-orkut`, `com-lj`, `soc-LiveJournal`, and `com-friendster`; to Rossi & Ahmed (2015) for `soc-sinaweibo`, and `twitter_rv`; to Meusel (2015) for `hyperlink2012`; to Mislove et al. (2007) for `youtube`.

Graph	**|V|**	**|E|**	**Density**
`com-dblp`	317,080	1,049,866	3.31
`com-amazon`	334,863	925,872	2.76
`youtube`	1,138,499	4,945,382	4.34
`soc-pokec`	1,632,803	30,622,564	18.75
`wiki-topcats`	1,791,489	28,511,807	15.92
`com-orkut`	3,072,441	117,185,083	38.14
`com-lj`	3,997,962	34,681,189	8.67
`soc-LiveJournal`	4,847,571	68,993,773	14.23
`hyperlink2012`	39,497,204	623,056,313	15.77
`soc-sinaweibo`	58,655,849	261,321,071	4.46
`twitter_rv`	41,652,230	1,468,365,182	35.25
`com-friendster`	65,608,366	1,806,067,135	27.53

v are $\notin G_{train}$ are also removed. Then, the target algorithm is executed with a newly generated G_{train} to get an embedding. By using the generated embedding, a Logistic Regression model is trained using `LogisticRegression` module from `scikit-learn` for medium scale graphs. For the relatively larger ones, $|V_{train}| > 10M$, classical logistic regression exceeds the memory limitations of our system. Thus, for such graphs, the `SGDClassifier` module from `scikit-learn` with a Logistic Regression solver is used. Finally, the existence of the edges in G_{test} is predicted by the model, and the $AUCROC$ score is reported (Fawcett, 2006).

For the prediction pipeline, two matrices $\mathbf{R}_{train}$ and $\mathbf{R}_{test}$ are created. Each (row) vector $\mathbf{R}_{train}$ is generated by the element-wise multiplication of two vectors from $\mathbf{M}_0[v]$ and $\mathbf{M}_0[u] \in \mathbf{M}_0$, where $u, v \in V_0$. This represents either a positive or a negative sample. To be more precise, $\mathbf{R}_{train}$ contains $|E_{train}|$ amount of positive samples, where a positive sample corresponds to an edge $(u, v) \in E_{train}$. Moreover, the same number of negative samples from $(V_{train} \times V_{train}) \setminus E_{train}$ is generated and added as vectors to $\mathbf{R}_{train}$ to make a balanced training set. $\mathbf{R}_{test}$ is created in a similar fashion by using G_{test} instead of G_{train} as the source of the samples. Unlike $\mathbf{R}_{test}$, for the last column of $\mathbf{R}_{train}$, a label representing a positive or negative sample is concatenated to the end of the vector.

Table 6.2 *Gosh* configurations, fast, normal and small for medium-scale and large-scale graphs. A version with no coarsening is also used in the experiments.

Configuration	p	lr	e_{normal}	e_{large}
Ultra-fast	0.1	0.050	400	-
Fast	0.1	0.050	600	100
Normal	0.3	0.035	1000	200
Slow	0.5	0.025	1400	300
No coarsening	-	0.045	1000	200

6.2 Coarsening Experiments

6.2.1 Experiments on Coarsening Performance

The impact of the features described in Section 4.2 on coarsening efficiency, and quality is demonstrated in Table 6.3. To clearly display the effects of the heuristical optimization, e.g., sorting and preventing two large vertices to merge, three different versions of *Gosh* coarsening are prepared.

- The *naive* version does not include processing the vertices in terms of their degree and allows two large vertices to merge if they are mapped. For both relatively smaller and larger graphs, this version performs poorly regarding coarsening efficiency.

- An improved version of naive is *ordered*, where the vertices are sorted, and processed in descending order. With this version, a slight improvement in coarsening efficiency is noted. However, the coarsening quality degrades substantially; it regresses from 0.964 to 0.921 for *com-friendster* (Table 6.3). This is caused by an underlying problem which is amplified by sorting. When two *hub* vertices $u, v \in V_i \wedge (u, v) \in E_i \wedge \Gamma(v) \geq \Gamma(u)$, are merged, the coarsening efficiency decreases immensely. A substantial amount of vertices might have been mapped by u. This potential reduction will not happen, since u is locked and can not be treated as a mapping vertex.

- With the *optimized* version, which applies both of the heuristical optimizations, this problem is mitigated, where the largest graph in the data-set *com-friendster* is reduced from 63 million vertices to only 823 vertices in ten iterations. Furthermore a substantial improvement is also demonstrated regarding

Table 6.3 Performance of *Gosh* coarsening with naive, ordered, and optimized version, and $\tau = 16$ threads.

Graph	V	AUCROC	T_t (s)	Level	T_c (s)	$\lvert V_i \rvert$	$\frac{\lvert E_i \rvert}{\lvert V_i \rvert}$
youtube	naive	0.959	106.50	1	0.13	1021590	7.75
				2	0.06	636896	4.84
	ordered	0.956	79.57	1	0.17	1021590	7.75
				2	0.04	480714	4.18
				3	0.03	378836	2.84
	optimized	0.98	9.28	1	0.17	1021590	7.75
				2	0.04	302683	12.27
				3	0.02	87718	32.78
				4	0.01	21356	94.29
				5	0.01	4436	216.26
				6	-	895	324.06
				7	-	380	298.84
				8	-	195	182.82
com-friendster	naive	0.964	43260.00	1	80.6	62603304	46.16
				2	15.99	28689906	64.23
	ordered	0.921	44912.40	1	61.99	62603304	46.16
				2	9.5	26666122	44.72
	optimized	0.972	2316.51	1	133.09	62603304	46.16
				2	98.9	20410136	131.66
				3	36.69	6390812	370.22
				4	7.36	1298192	842.45
				5	2.02	194157	1685.39
				6	0.5	29415	3424.18
				7	0.18	8370	4094.04
				8	0.06	3421	3019.52
				9	0.01	1814	1795.11
				10	0.01	1100	1098.59
				11	-	823	822

coarsening quality; 1%, and 2% increase is observed in Table 6.3 for graphs *com-friendster* and *youtube*, respectively. This further highlights the importance of an effective and efficient coarsening, which is judiciously analyzed in Section 6.2.2.

In Table 6.4, a brief comparison of *MILE* and *Gosh* with 16 threads on the graph `com-orkut` is shown. Since *MILE* does not have a stopping criterion for coarsening, the same amount of coarsening levels is used for both algorithms. While coarsening a graph of 3 million vertices and 100 million edges, *Gosh* is 264 times faster than *MILE*. Moreover, *Gosh* is significantly more efficient regarding the number of vertices obtained at each level. For instance, in 8 levels, *Gosh* shrinks `com-orkut` to only 230 vertices, while *MILE* shrinks it to 12062 vertices. Although this does not relate

Table 6.4 *MILE* vs *Gosh* coarsening on `com-orkut`. A parallel coarsening with $\tau = 16$ threads is used for *Gosh*.

	i	**Time (s)**	$\lvert V_i \rvert$		i	**Time (s)**	$\lvert V_i \rvert$
MILE	0	-	3056838	*Gosh*	0	-	3056838
	1	249.77	1535168		1	4.44	975132
	2	237.39	768804		2	1.23	213707
	3	184.72	384752		3	0.62	46667
	4	151.24	192507		4	0.16	8084
	5	139.23	96308		5	0.03	2000
	6	128.47	48183		6	0.01	701
	7	117.75	24107		7	< 0.01	375
	8	99.73	12062		8	< 0.01	275
	Total	1308.31	-		Total	6.60	-

the quality of the coarsening to the quality of embedding, it is important in terms of performance, since the training time is affected by the number of vertices at each level.

Table 6.5 provides the details of the coarsenings obtained from the sequential and parallel coarsening algorithms with $\tau = 2, 4, 8, 16$ threads. As the results show, parallel coarsening reaches a similar number of levels and the graphs at the last-level are of similar sizes, except *hyperlink2012*, where the runs with $\tau = 4, 8, 16$ perform relatively poorly compared to the sequential run, and the run with $\tau = 2$. Although a performance decrease is observed, we found that the issue can be solved by tuning *Gosh* accordingly. By using *coarsening stopping threshold* of 83% instead of 80% *Gosh* is able to score similar results on `hyperlink2012` for all the runs. With a similar coarsening quality, the parallel algorithm is 5–8× faster compared to the sequential counterpart. As described in Section 4.2.1, the time complexity is $\mathcal{O}(|V| + |E|)$ and in practice, $|E|$ dominates the workload. Although there are other parameters, the variation in the speedups is in concordance with the variation in the number of edges. For instance, `soc-sinaweibo` only has 200M edges and yields the smallest speedup value of 5.41×. On the other hand, the largest speedup 7.57× is obtained for `com-friendster`, which is the largest in our data-set with 1.8B edges.

6.2.2 Experiments on Coarsening Quality

To demystify the impact of coarsening on the quality of the embeddings, four coarsening strategies are integrated into *Gosh*. All of them merge the vertices in the current level with a super-vertex which can be considered as the center of the corre-

Table 6.5 Execution times, the number of levels and the size of the last-level graphs for sequential and parallel coarsening with $\tau = 2, 4, 8, 16$ threads for the large-scale graphs. The training graph with a split ratio of 0.8 is used for all the graphs. For `hyperlink2012` Coarsening Stopping Threshold of 0.83 is used for $\tau = 4, 8$, and 16.

Graph	τ	**Time (s)**	Speedup	D	$\lvert V_{D-1}\rvert$	$\lvert E_{D-1}\rvert$
	1	287.824	-	13	234	53792
	2	224.93	1.28×	13	125	15442
`hyperlink2012`	4	128.26	2.24×	13	100	9884
	8	77.73	3.7×	13	117	13560
	16	46.88	6.14×	13	117	13572
	1	117.606	-	10	230	52632
	2	80.35	1.28×	10	187	34778
`soc-sinaweibo`	4	46.05	2.55×	10	198	39000
	8	28.1	4.19×	10	209	43472
	16	21.76	5.41×	9	358	127716
	1	534.92	-	15	132	17000
	2	400.24	1.34×	13	159	24580
`twitter_rv`	4	226.05	2.37×	13	117	13454
	8	131.18	4.1×	13	113	12630
	16	77.07	6.94×	13	127	16002
	1	2154.48	-	11	901	810874
	2	1549.43	1.39×	11	768	589056
`com-friendster`	4	782.53	2.75×	11	765	584460
	8	424.93	5.07×	11	788	620156
	16	284.465	7.57×	11	795	631230

sponding cluster (which will be a single vertex in the next level). That is in all, the vertices are gathered around the super-vertices to form a cluster. Similarly, all the coarsening approaches have the potential to significantly reduce the training time since the graphs obtained in the multi-level setting are smaller than the original one. However, how they approach to the problem, and more formally, how much they have respect for the topology, i.e., the proximity information, during the coarsening are completely different. In fact, they are devised to create a coarsening spectrum and better understand the impact on the AUCROC scores. At one hand of the spectrum, there is *anti* which always bets against the proximity and coarsens only independent sets. The next one, *random*, form random clusters. The proposed strategy used in *Gosh*, *novel*, merges only the neighbor vertices and favors both first and second-order proximities. However, it also has heuristic optimizations that prevent merging hub vertices, and favors the gathering of low-degree vertices around the hub vertices. Hence, it can be placed as next to *random* in the spectrum. The last strategy, *grappolo*, totally respects to the graph structure and proximities. It forms the clusters and refine them to obtain the *best* community structure and maximize the modularity to the most. Hence, by nature, compared to the other three, it hides more edges during the coarsening levels and obtains sparser graphs. It can be placed to the other end of the spectrum.

- *anti*: this version is similar to the random version. The only difference is that another condition is introduced for mapping. If $(u, v) \in E_i$, u, and v can not be mapped under the same super vertex.

- *random*: with random coarsening, for each vertex $v \in V_i$, if the vertex is not mapped, a vertex $u \in V_i$ is selected uniformly randomly. If u is not mapped, it is mapped to the same *super* vertex $k \in V_{i+1}$ as v. $|\Gamma(v)|$ number of selections are executed for each vertex.

- *novel*: this version refers to the original coarsening strategies that is used in *Gosh*. For more information turn to Sections 4.2, and 4.3.

- *grappolo*: this version uses the code provided by (Lu et al., 2014). For more information turn to Section 4.4.

As the previous experiments, e.g, Table 6.3, show, coarsening is an indispensible tool to cope with big graphs during graph embedding. It boosts the training performance by several orders of magnitude and enables better utilization of the memory-restricted accelerators such as GPUs. Hence, we can assume that it will always be *on* for *Gosh*, as well as the future and better embedding tools. However, it is hard to demystify which coarsening strategy is the best and why. We devise this experiment

to delve into this and grasp the nature of the optimization led by the decisions taken during the coarsening in a more detailed way. It will not be fair if the strategies are considered as the competitors. For instance, *grappolo* is a modularity maximization tool and it is not proposed for graph embedding (as *novel* is not proposed to maximize modularity). In short, the three other strategies are judiciously chosen to form the spectrum with *novel* as described above.

Clustering similar vertices, which share an edge or similar to each other, is useful since these vertices will have the same embedding vector when the multilevel embedding moves up, i.e., uncoarsens, throughout the process. The strategies *anti* and *random* located at one hand of the spectrum cannot leverage this behaviour. However, from a different angle, if graph topology and proximities are the ultimate, sole information to be exploited, most of the edges will be hidden in the lower levels. That is, there will be less (positive) information to learn in these levels. In fact, considering the runtime performance, this is a desired thing to have. Furthermore, one important benefit of a multilevel setting is the ability to take big leaps to a good solution which also makes the optimization process effectively jump over local minimas. If there exist less information to process one cannot exploit this property well. We conjecture that there must be some form of balance in between these two behaviours.

Table 6.6 The performance of *Gosh* integrated with different types of coarsening. The training graph with a split ratio of 0.8 is used for all the graphs. *Gosh*-normal is used for the experiments.

Graph	Algorithm	T_t **(s)**	D	$\|V_{D-1}\|$	$\|E_{D-1}\|$
`youtube`	anti	10.33	7	172	29260
	random	8.26	8	127	15998
	novel	8.10	8	202	37848
	grappolo	287.824	4	10997	20039
`twitter_rv`	anti	1400.69	10	279	77562
	random	1195.78	11	195	37584
	novel	1031.19	12	158	24780
	grappolo	4586.12	4	11720	40782

For a glimpse of how these coarsening strategies behave, one can look at Table 6.6. The first three strategies perform more levels compared to *grappolo*. For instance, on `youtube` and `twitter_rv`, *grappolo* takes 4 levels to coarsen the graph having 20 and 40 thousand edges, respectively. On the other hand, the other three strategies can reach the same number of edges in at least 8 and 10 levels, respectively. As expected, the behaviour of *anti* and *random* are almost similar in terms of coarsening. As Table 6.6 shows, for both `youtube` and `twitter_rv`, *random* goes one level deeper and reaches a smaller amount of vertices in the coarsest level. In addition, the

time-to-train values for *grappolo* are higher, since the epochs are distributed to less number of levels containing relatively larger graphs.

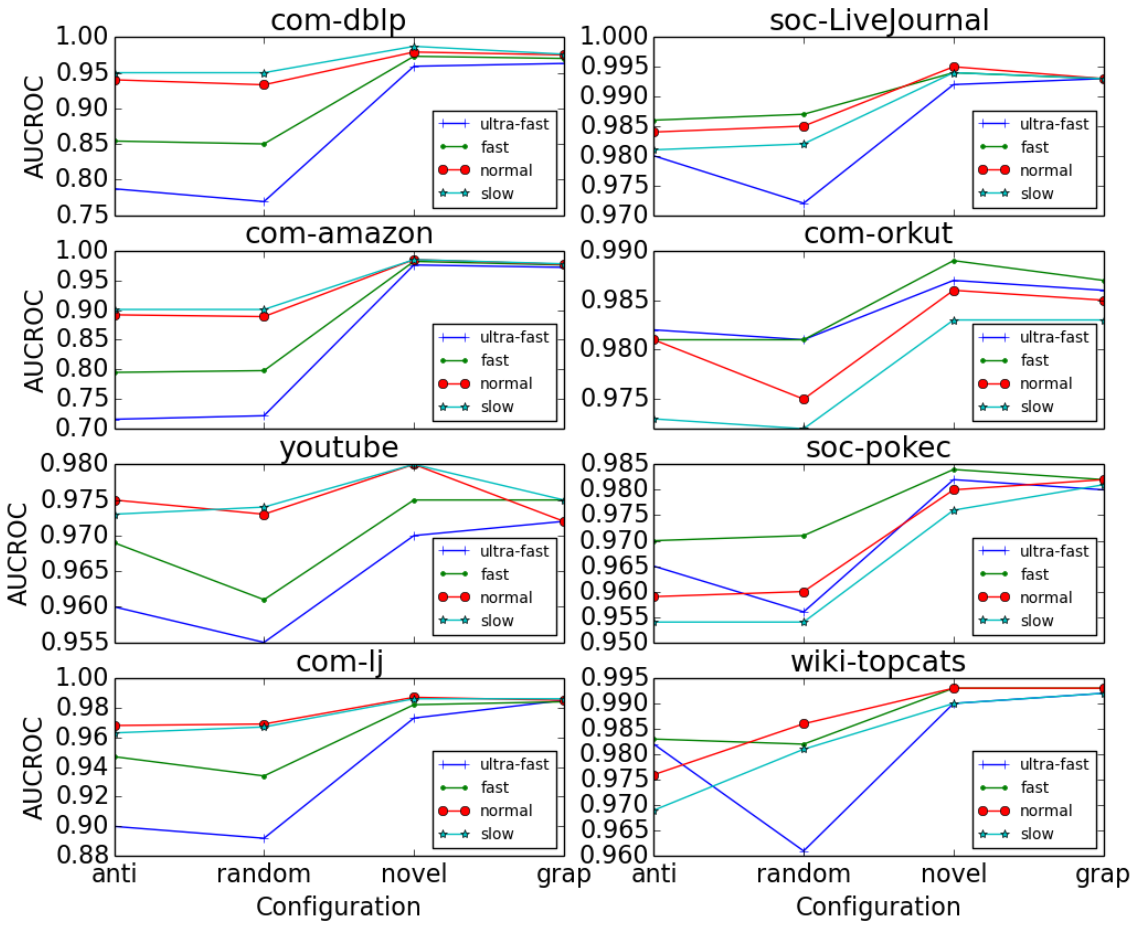


Figure 6.1 Medium-scale graph results for different coarsening strategies and configurations.

The embedding performance of the aforementioned strategies on different *Gosh* configurations are given in Figures 6.1 and 6.2 for medium- and large-scale graphs respectively. In terms of embedding performance, for the configurations ultra-fast and fast, *anti* has a slight edge over *random*, and for the rest, the performance of both strategies are not distinguishable. For both strategies, the AUCROC scores significantly improve when the number of epochs increases, i.e., when the configurations are changed from ultra-fast to slow. We believe that this happens due to bad decisions which ignore vertex proximities. Note that with *anti*, and also for most of the cases in *random*, vertices in independent sets will have the same embedding vector at the beginning of each level. When the number of epochs increases, *Gosh* has more fuel to fix the negative impact of the coarsening decisions which do not take the proximities into account. Although in a smaller scale, such an improvement is also observed for *novel* in medium-scale graphs. On these graphs, *novel* almost always perform better than *anti* and *random*. This emphasizes the importance of

taking the proximity information into account during coarsening, on the quality of the embeddings.

Especially for large-scale graphs in Figure 6.2, *novel* suffers to generate a quality embedding with a low epoch budget, where *grappolo* scores the best with the exception of `twitter_rv`. Note that as explained above, the approaches do not use the same number of levels since *grappolo* produces significantly less coarsening levels. That is the amount of work done by *grappolo* are substantially larger than the rest for fast configurations. In fact, compared to *novel*, *grappolo* does 28×, and 4.5× more work on graphs `youtube` and `twitter_rv` with normal configuration. Still, it seems to have less AUCROC variation when the configuration and the number of epochs change. When the epoch budget is increased, *novel* becomes superior to *grappolo* for most of medium- and large-scale graphs. Since *novel* tends to generate smaller graphs with less number of vertices compared to *grappolo*, it can perform larger optimization steps, i.e., updates that change the embedding of more vertices at once. For instance, on `youtube` with slow configuration, for the original graph, 70, and 220 epochs are reserved for *novel*, and *grappolo* respectively. Hence, for *novel*, the embedding on the original graph in the final level can be considered as a *fine tuning* whereas it is one of the main steps in *grappolo*. Furthermore, for *novel*, using more levels also make the embedding process much faster since most of the epochs are spent on smaller graphs. That being said, a good coarsening strategy should not create thousands of levels since in this case, most of the epoch budget will be used for coarsened and similar graphs. Although we do not know the optimal number for each graph, based on our experience and the experiments we performed (presented in the next subsection), using around 10 levels creates a good balance in terms of runtime and embedding performance.

To better profile the relative performance of the coarsening strategies, Figures 6.3, 6.4, 6.5, 6.6, and 6.7 are provided. For these figures the respective algorithms, and configurations are compared against the best performing result for each graph. T (x-axis) represents the distance of a result relative to the best performing result in terms of $AUCROC$. Furthermore, a point on the chart, $T = y$, and $\#ofgraphs = x$, indicates the amount of graphs x, where the respective version scores an $AUCROC$ at most $y\%$ worse than the best. The experiments until this point show that *grappolo* is the most stable coarsening strategy. That is the variation in the AUCROC scores is less when *Gosh* is configured differently. Indeed, due to the epoch distribution strategy employed by *Gosh*, using less number of levels increase the amount of the work performed. However, using less, as *novel* does, has the advantage of performing more impactful moves. As mentioned above, *novel* uses less epochs in the last level for fine tuning compared to *grappolo*. As Figure 6.3 shows, *grappolo* is the best

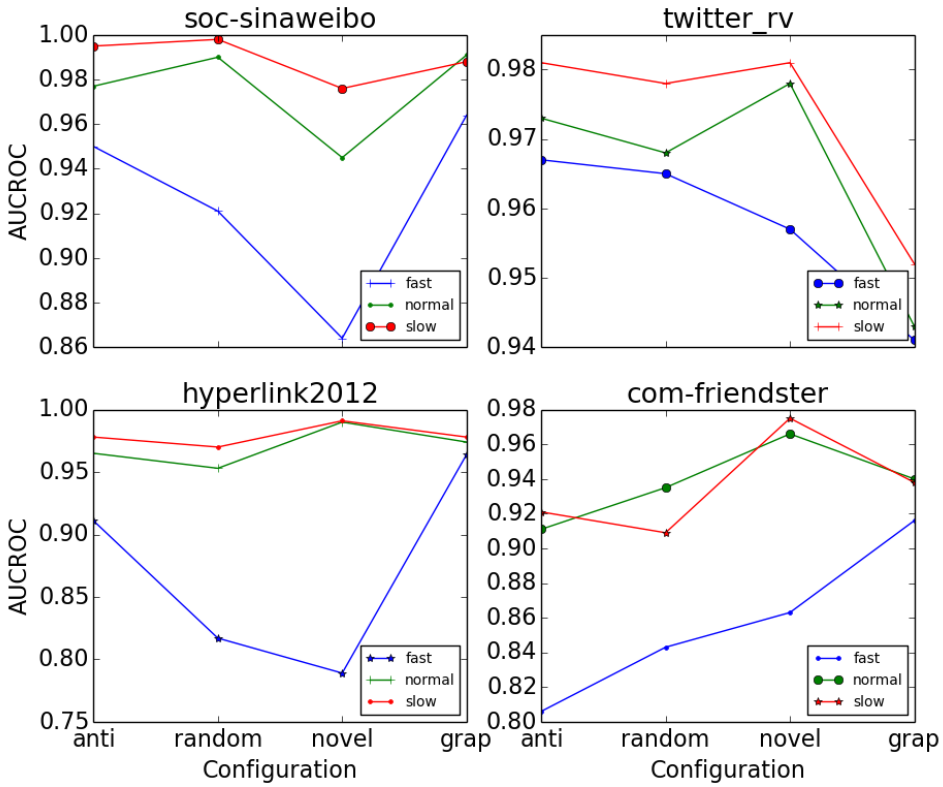


Figure 6.2 Large-scale graph results for different coarsening strategies and configurations.

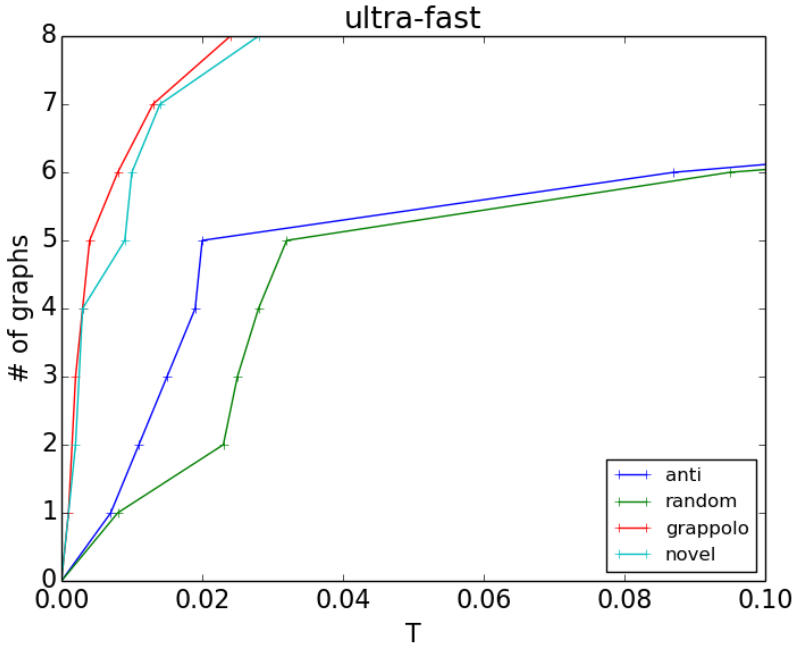


Figure 6.3 Performance profile of *Gosh* using ulta-fast configuration with different coarsening strategies for the entire data-set.

coarsening strategy with the ultra-fast configuration in terms of AUCROC scores (also having relatively more expensive execution times). We believe that for cheap configurations, the proposed strategy *novel* does not have enough number of epochs to fine tune the embeddings in the last level. However, when *Gosh* configurations move from ultra-fast to slow (Figures 6.3–6.6), the proposed coarsening strategy becomes better, and stays faster than *grappolo* in terms of embedding time. Figure 6.7 shows the performance profile of all coarsening-configuration pairs. Similar conclusions follow: *anti* and *random* need more epochs to obtain a decent performance. Although, *grappolo* is more robust and successful with less epochs, the *novel*, which uses a more balanced strategy, becomes better in terms of AUCROC when the number of epochs is increased.

6.2.2.1 Experiments on Coarsening Depth

Through Table 6.7, the effects of running *Gosh* with different coarsening levels D on the amount of work, and the quality of the embedding are presented. As described in Section 5.2, due to the epoch distribution strategy of *Gosh*, when the coarsening depth increases the total amount of work that is reserved for the finer levels decreases. For separate runs of *Gosh* with different coarsening levels, one may expect a decrease both on the training time, and the quality of the output, since the amount of epochs transferred from the finer levels to coarser levels will take less amount of time, and the updates on the coarser levels will be less instrumental. While the former is true, according to our preliminary experiments, and the analysis by Chen et al. (2017) the latter is the other way around. *Gosh* coarsening groups similar vertices under the same *super* vertex, which makes a single update on the coarser levels more powerful. This not only compensates for the reduced amount of work but also boosts the performance of *Gosh* for larger D.

Table 6.8 Link prediction results on medium-scale graphs. Every data-point is the average of 15 results. *VERSE* and *Gosh* uses $\tau = 16$ threads. *MILE* is a sequential tool. Both *GraphVite* and *Gosh* uses the same GPU. The speedup values are computed based on the execution time of *VERSE*.

Graph	**Algorithm**	T_{total} **(s)**	**Speedup**	**AUCROC**(%)
com-dblp	*VERSE*	247.99	1.00×	**97.82**
	MILE	136.65	1.81×	97.65
	GraphVite-fast	13.97	17.70×	97.80
	GraphVite-slow	19.93	12.40×	**98.08**
	Gosh-fast	0.72	344.43×	96.45
	Gosh-normal	2.08	119.23×	97.38

	Gosh-slow	3.84	64.58×	97.63
	Gosh-NoCoarse	29.97	8.27×	93.31
com-amazon	*VERSE*	216.18	1.00×	97.71
	MILE	146.29	1.48×	**98.14**
	GraphVite-fast	12.45	17.36×	97.40
	GraphVite-slow	16.84	12.83×	97.82
	Gosh-fast	0.69	313.30×	97.20
	Gosh-normal	1.88	114.99×	98.29
	Gosh-slow	3.59	60.22×	**98.43**
	Gosh-NoCoarse	24.60	8.79×	90.13
com-lj	*VERSE*	12502.72	1.00×	**98.86**
	MILE	3948.62	3.17×	80.19
	GraphVite-fast	373.58	33.47×	98.04
	GraphVite-slow	644.43	19.40×	98.33
	Gosh-fast	16.27	768.45×	96.82
	Gosh-normal	55.01	227.28×	98.33
	Gosh-slow	153.72	81.33×	**98.46**
	Gosh-NoCoarse	675.25	18.52×	98.32
com-orkut	*VERSE*	45994.93	1.00×	**98.65**
	MILE	11904.31	3.86×	90.38
	GraphVite-fast	1246.38	36.90×	98.02
	GraphVite-slow	2199.25	20.91×	**98.05**
	Gosh-fast	43.30	1062.24×	97.35
	Gosh-normal	185.12	248.46×	97.63
	Gosh-slow	487.33	94.38×	97.69
	Gosh-NoCoarse	2301.89	19.98×	97.64
wiki-topcats	*VERSE*	8709.48	1.00×	**99.31**
	MILE	4953.68	1.76×	86.04
	GraphVite-fast	310.47	28.05×	96.42
	GraphVite-slow	544.06	16.01×	96.28
	Gosh-fast	11.34	768.03×	98.13
	Gosh-normal	40.76	213.68×	98.33
	Gosh-slow	93.86	92.79×	**98.50**
	Gosh-NoCoarse	549.65	15.85×	98.51
youtube	*VERSE*	1365.36	1.00×	**98.04**
	MILE	1328.62	1.03×	94.17
	GraphVite-fast	63.90	21.37×	97.07
	GraphVite-slow	104.76	13.03×	97.45
	Gosh-fast	2.76	494.70×	96.16
	Gosh-normal	7.15	190.96×	97.78
	Gosh-slow	15.32	89.12×	**97.93**
	Gosh-NoCoarse	158.60	8.61×	97.16
soc-pokec	*VERSE*	9182.53	1.00×	**98.32**
	MILE	2848.78	3.22×	85.75
	GraphVite-fast	370.73	24.77×	**97.42**
	GraphVite-slow	607.07	15.13×	97.37
	Gosh-fast	16.34	561.97×	96.34
	Gosh-normal	54.66	167.99×	96.49
	Gosh-slow	131.06	70.06×	96.67
	Gosh-NoCoarse	598.95	15.33×	97.28
soc-LiveJournal	*VERSE*	14965.76	1.00×	**97.61**
	MILE	6210.58	2.41×	80.84
	GraphVite-fast	745.33	20.08×	99.23
	GraphVite-slow	1209.95	12.37×	**99.31**
	Gosh-fast	29.74	503.22×	98.58
	Gosh-normal	112.72	132.77×	98.87
	Gosh-slow	183.64	81.50×	98.76
	Gosh-NoCoarse	1348.74	11.10×	98.88

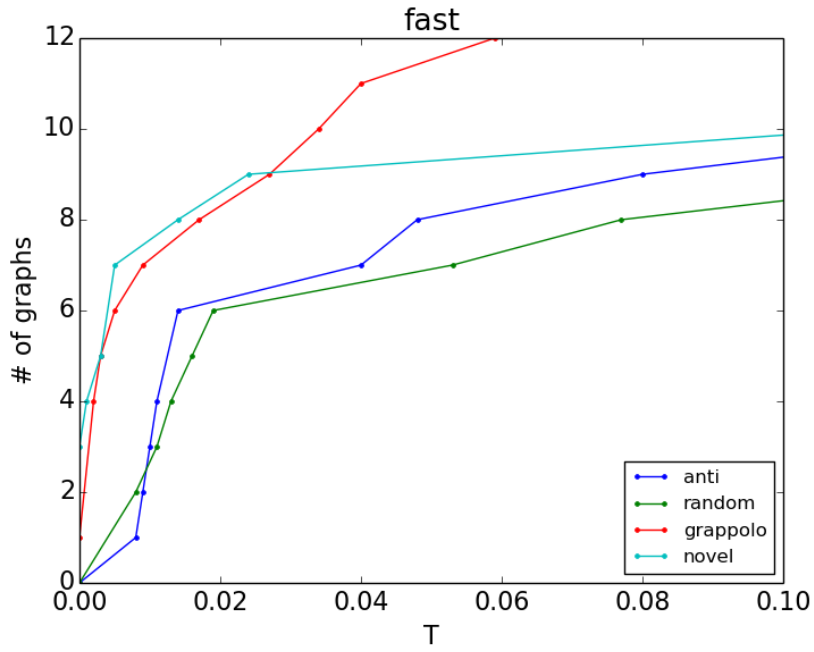


Figure 6.4 Performance profile of *Gosh* using fast configuration with different coarsening strategies for the entire data-set.

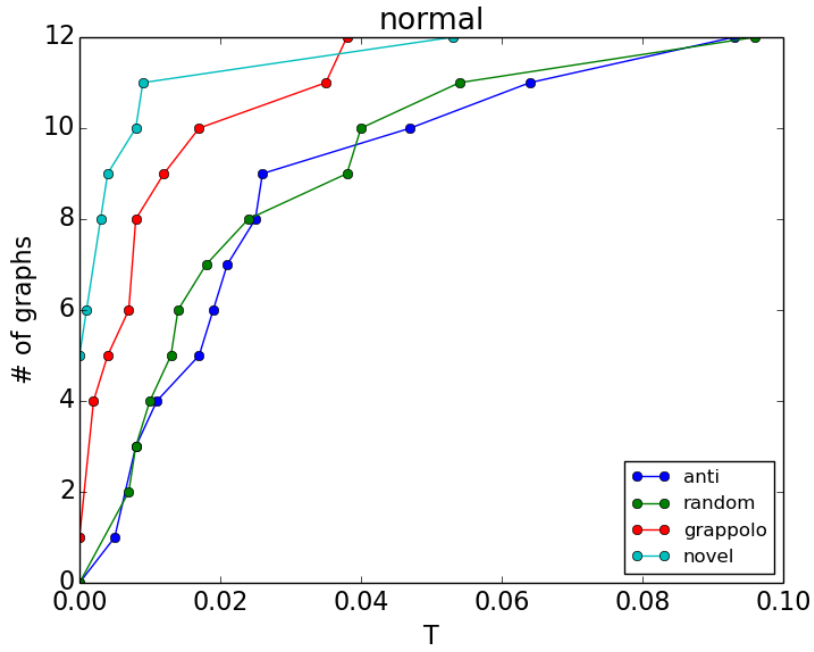


Figure 6.5 Performance profile of *Gosh* using normal configuration with different coarsening strategies for the entire data-set.

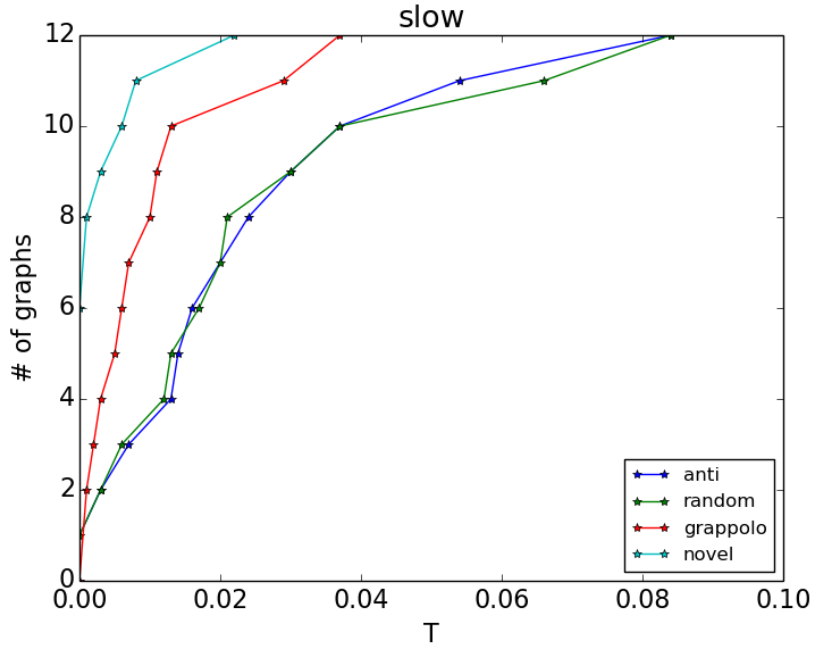


Figure 6.6 Performance profile of *Gosh* using slow configuration with different coarsening strategies for the entire data-set.

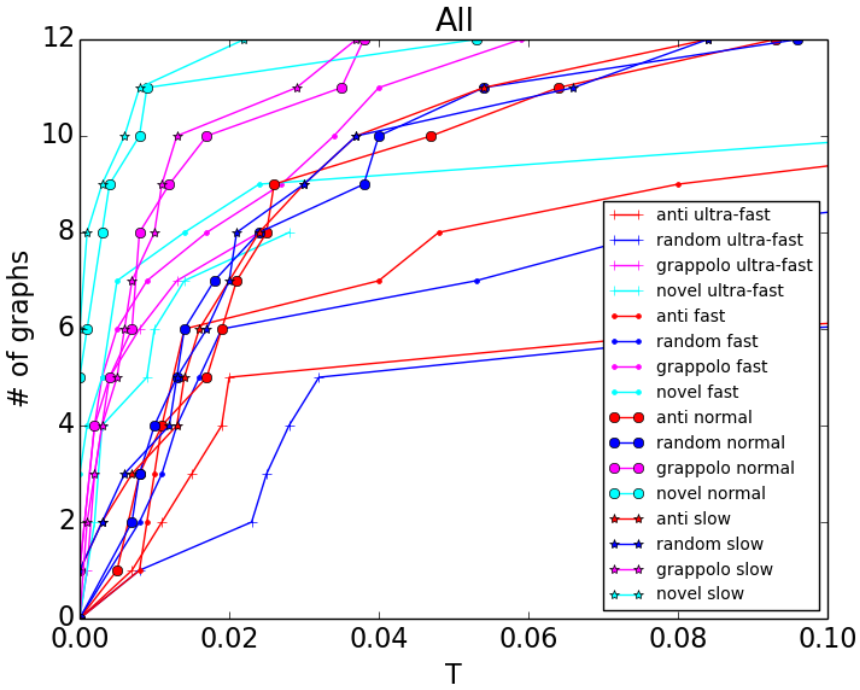


Figure 6.7 Performance profile of *Gosh* with different coarsening strategies, and embedding configurations for the entire data-set. Colors, and markers represent the configuration, and the coarsening strategy respectively.

Table 6.7 The performance of *Gosh* is displayed for coarsening levels 2,3,5, and 7. The training graph, with a split ratio of 0.8, is used for all the graphs.

Graph	# of Levels	**Time (s)**	AUCROC
com-dblp	2	15.47	0.976
	3	11.12	0.977
	5	5.32	0.977
	7	3.10	0.979
com-amazon	2	14.29	0.969
	3	10.45	0.98
	5	5.10	0.984
	7	2.95	0.985
youtube	2	69.15	0.974
	3	45.07	0.972
	5	19.98	0.975
	7	11.25	0.980
soc-pokec	2	482.84	0.965
	3	326.63	0.963
	5	136.50	0.969
	7	75.07	0.975
com-lj	2	539.76	0.972
	3	365.27	0.973
	5	157.48	0.978
	7	86.62	0.985
com-orkut	2	1756.51	0.974
	3	1117.92	0.975
	5	474.36	0.979
	7	263.20	0.983
wiki-topcats	2	370.3	0.977
	3	226.45	0.980
	5	95.71	0.985
	7	54.85	0.992
soc-LiveJournal	2	1056.7	0.989
	3	713.24	0.989
	5	307.47	0.991
	7	166.80	0.993

6.3 Embedding Experiments

Tables 6.8, and 6.9 provide the execution times and AUCROC scores of the tools evaluated on medium-scale and large-scale graphs, respectively. For evaluation, *VERSE* results are determined as a baseline, and the speedups are provided relative to *VERSE*. The following observations can be made from the experiments:

- *Gosh*-fast is an accelerated solution that produces accurate embeddings, which is faster compared to all the systems under evaluation. It can achieve a speedup over *VERSE* of up to three orders of magnitude and an average of 600× with a maximum loss in AUCROC of 2% and an average loss of 1.16%. When compared to *MILE*, it is superior in terms of AUCROC in three-quarters of the graphs while being at least two orders of magnitude faster.
- At an average loss in AUCROC of 0.54%, *GraphVite* can achieve an average speedup of 23.44×.
- *Gosh*-normal not only demonstrates the speed/quality trade-off of *Gosh* but also its flexibility. Switching from *Gosh*-fast to *Gosh*-normal, although the speed reduces on average by a factor of 3, AUCROC scores increase by 0.76% on average.
- *Gosh*-slow demonstrates the flexibility of *Gosh*, where its accuracy comes close to the best tool for every graph. Compared to *VERSE*, *Gosh*-slow has an average loss of 0.24% in AUCROC, however it still has an average speedup of 79.24×.
- To compare *Gosh* with the state-of-the-art GPU implementation *GraphVite*, the best AUCROCs scores are used from several runs with different configurations. For half of the graphs, *Gosh* produce better AUCROC scores compared to *GraphVite*. The values are similar; on average, *Gosh* achieves 0.16% higher AUCROCs than *GraphVite*, while being 5.2× faster than *GraphVite* on average.

6.3.1 Large-scale graphs

Since the number of edges is significantly larger than medium-scale graphs, and the amount of work in an epoch scales with $|E_i|$, for larger graphs, embedding with a

smaller number of epochs is sufficient. Consequently, compared to medium-scale graphs, lower number of epochs is used for the relatively larger ones.

GraphVite results are not reported since, for all the large-scale graphs, the executable runs out of CPU memory on our machine. We find that *GraphVite* on `hyperlink2012` is reported to achieve 94.3% link-prediction AUCROC after an embedding for 5.36 hours using four Tesla P100s GPUs Zhu et al. (2019). *Gosh*-normal achieves an AUCROC of 97.20% after an embedding taking only 0.2 hours using a single Titan X GPU (26.8× speedup). It is also reported that *GraphVite* takes 20.3 hours on `com-friendster` Zhu et al. (2019) where *Gosh*-normal requires only 0.76 hours (26.7× speedup).

MILE ran out of memmory for `twitter_rv`, and `com-friendster`, and cannot embed `hyperlink2012` and `soc-sinaweibo` before the 12 hour timeout.

As shown in Table 6.9, *VERSE* times out on 3 out of the 4 graphs, where `soc-sinaweibo` is the only graph that an embedding is generated. Compared to *Gosh*-slow, it scores a 0.52% higher AUCROC while being 26× slower.

Table 6.9 Link prediction results on large graphs. Every data-point is the average of 6 results. *GraphVite* and *MILE* fail to embed any of the graphs due to excessive memory usage or an execution time larger than 12 hours. $\tau = 16$ threads used for both *VERSE* and *Gosh*.

Graph	**Algorithm**	**Time (s)**	**Speedup**	**AUCROC** (%)
`hyperlink2012`	*VERSE*	Timeout	-	-
	Gosh-fast	201.02	-	87.60
	Gosh-normal	724.09	-	97.20
	Gosh-slow	1676.93	-	98.00
`soc-sinaweibo`	*VERSE*	20397.79	1.00×	99.89
	Gosh-fast	48.88	417.30×	70.27
	Gosh-normal	352.86	57.81×	97.00
	Gosh-slow	759.85	26.84×	99.37
`twitter_rv`	*VERSE*	Timeout	-	-
	Gosh-fast	261.08	-	91.78
	Gosh-normal	994.46	-	97.36
	Gosh-slow	2128.70	-	98.50
`com-friendster`	*VERSE*	Timeout	-	-
	Gosh-fast	680.33	-	85.17
	Gosh-normal	2720.82	-	93.40
	Gosh-slow	5000.96	-	94.98

6.3.2 Experiments on Small Dimensions

The performance of *Gosh* is analyzed with running different number of vertices in a single warp for small d values. The results on `com-orkut` and `soc-LiveJournal` are given in Table 6.10. With SM, we observe 2.63× and 1.84× speedups for $d = 8$ and 16, respectively. Moreover, for `soc-LiveJournal`, we obtain 2.70× and 1.85× speedups for $d = 8$ and $d = 16$. As expected, with or without SM, $d = 32$ timings are almost the same. Without small-dimension technique (SM), *Gosh* takes approximately the same time for $d = 8, 16$ and 32 although 4× and 2× less work is performed for $d = 8$ and 16.

Table 6.10 Performance of *Gosh* with (SM = Yes) & without (SM = No) small-dimension embedding and $\tau = 16$ threads.

Graph	SM	d	Time (s)	Graph	SM	d	Time (s)
com-orkut	No	8	63.72	soc-LiveJournal	No	8	40.13
		16	64.20			16	40.46
		32	64.95			32	41.22
	Yes	8	24.27		Yes	8	14.86
		16	34.98			16	21.82
		32	64.54			32	40.93

6.4 Speed Up Break-Down

As shown in Figure 6.8, a comparison of intermediate versions of *Gosh* is reported over the 16-thread CPU implementation. The experiments are conducted with six graphs; two large-scale graphs (`com-friendster`, and `hyperlink2012`), and four medium-scale graphs. The results for the versions that do not include coarsening on large-scale graphs are not reported due to time constraints.

There is a significant difference between the two GPU versions that do not use coarsening. *Naive GPU* implementation results in an average slowdown of 3.3×, where the *Optimized GPU* version scores 5.4× faster. The results emphasize the hardware oriented programming nature of GPUs. For the *Optimized GPU* version global memory is organized to have coalesced accesses, and shared memory is utilized to reduce the number of global memory accesses.

The biggest jump is introduced by the version *Sequential Coarsening*, which scores an average speedup of 45× over the CPU version while maintaining the embedding

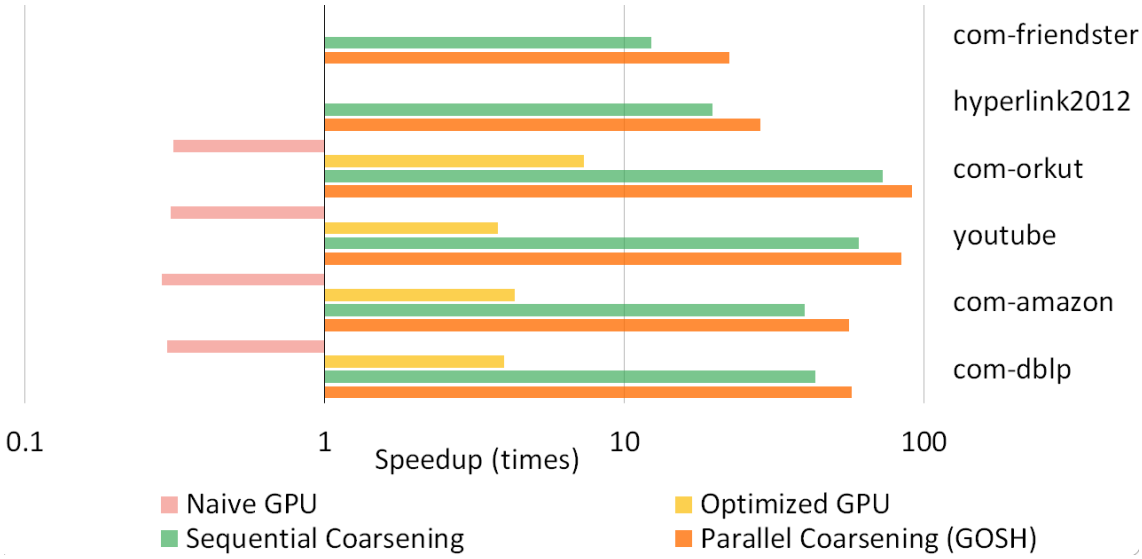


Figure 6.8 The speedups obtained from running intermediate versions of *Gosh* compared to our multi-core CPU implementation with 16 threads.

quality as shown in Table 6.8. This is due to the cumulative nature of the updates on the coarsened graphs. A single update on a *super* vertex is propagated to all the vertices it contains.

The performance of *Gosh* is further improved with the addition of parallel coarsening. The performance difference between *Parallel Coarsening* and *Sequential Coarsening* is greater for larger graphs. For instance, on `com-friendster`, sequential, parallel coarsening, and training with normal configurations take 2468.52, 235.38, and 2720.82 seconds respectively (Table 6.5). In other words, parallel coarsening results in an 80% improvement on performance.

7. CONCLUSION

In this thesis, a novel, parallel, and multi-level coarsening algorithm is proposed for boosting the performance of large-scale graph embedding. The algorithm leverages a newly designed, agglomerative coarsening approach called MultiEdgeCollapse. By comparing intermediate versions of the approach, the effectiveness of newly introduced features are presented. Furthermore a comparison between sequential, and parallel coarsening is provided. Compared to the sequential one, the parallel algorithm is able to generate coarsenings in similar quality, while being 6.5× faster on average. The parallel algorithm outperforms state-of-the-art coarsening techniques both in terms of efficiency, and speed. Compared to state-of-the-art, the algorithm is able to obtain a graph which is 44× smaller, while being 264× faster. Furthermore the performance in terms of embedding quality is evaluated by comparing the algorithm to a high quality, and to two low quality coarsening algorithms. The experiments shows that the quality of the coarsening positively effects the quality of the embedding, where the proposed algorithm outperforms the rest in three-quarters of the data-set.

A CPU-GPU hybrid, high quality and multi-level graph embedding tool is also presented. The tool is able to embed any directed, or undirected graph on a *single* GPU, which applies a partitioning schema that is able to generate samples during embedding. The high-level architecture of the tool which includes work distribution, partitioning and coarsening, and the techniques which are leveraged to minimize GPU idling, and maximize GPU utilization are described in detail. Our preliminary experiments shows that the proposed tool out performs the-sate-of-art by 27×, while generating an embedding which is similar or better in quality. To add, the quality of the embeddings are evaluated with the machine learning task of link prediction. Lastly, a new, and flexible performance evaluation pipeline for graph embedding, which can evaluate various tools on different machine learning tasks is also outlined.

As future work, we would like to extend our work to various different machine learning task, i.e, node classification, and anomaly detection. We believe that it is important to investigate the performance of the coarsening on different machine

learning tasks, and explore the possibility of integrating additional metrics to the coarsening.

BIBLIOGRAPHY

Akyildiz, T. A., Aljundi, A. A., & Kaya, K. (2020). Gosh: Embedding big graphs on small hardware. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA. Association for Computing Machinery.

Batson, J., Spielman, D., Srivastava, N., & Teng, S.-H. (2013). Spectral sparsification of graphs: Theory and algorithms. *Communications of the ACM*, *56*, 87–94.

Belkin, M. & Niyogi, P. (2001). Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, (pp. 585–591)., Cambridge, MA, USA. MIT Press.

Bengio, Y., Courville, A., & Vincent, P. (2012). Representation learning: A review and new perspectives.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, *2008*(10), P10008.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, *34*(4), 18–42.

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2013). Spectral networks and locally connected networks on graphs.

Cai, H., Zheng, V. W., & Chang, K. C.-C. (2017). A comprehensive survey of graph embedding: Problems, techniques and applications.

Calandriello, D., Lazaric, A., Koutis, I., & Valko, M. (2018). Improved large-scale graph learning through ridge spectral sparsification. In Dy, J. & Krause, A. (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, (pp. 688–697)., Stockholmsmässan, Stockholm Sweden. PMLR.

Cao, S., Lu, W., & Xu, Q. (2015). Grarep: Learning graph representations with global structural information. In *Proc. 24th ACM Int. Conf. on Info. and Knowledge Management*, CIKM '15, (pp. 891–900)., NY, USA. ACM.

Cao, S., Lu, W., & Xu, Q. (2016). Deep neural networks for learning graph representations. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, (pp. 1145–1152). AAAI Press.

Chen, H., Perozzi, B., Hu, Y., & Skiena, S. (2017). Harp: Hierarchical representation learning for networks.

Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recogn. Lett.*, *27*(8), 861–874.

Gavish, M., Nadler, B., & Coifman, R. R. (2010). Multiscale wavelets on trees, graphs and high dimensional data: Theory and applications to semi supervised learning. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, (pp. 367–374)., Madison, WI, USA. Omnipress.

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

Grover, A. & Leskovec, J. (2016). node2vec: Scalable feature learning for networks.

Halappanavar, M., Lu, H., Kalyanaraman, A., & Tumeo, A. (2017). Scalable static and dynamic community detection using grappolo. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, (pp. 1–6).

Harel, D. & Koren, Y. (2000). A fast multi-scale method for drawing large graphs. volume 6, (pp. 183–196).

Hendrickson, B. & Leland, R. (1995). A multi-level algorithm for partitioning graphs. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, (pp. 28–28).

Hu, Y. (2005). Efficient and high quality force-directed graph drawing. *Mathematica Journal*, *10*, 37–71.

Jeh, G. & Widom, J. (2002). Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, (pp. 538–543)., New York, NY, USA. Association for Computing Machinery.

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

Karypis, G. & Kumar, V. (1998b). Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, *48*(1), 96 – 129.

Kipf, T. N. & Welling, M. (2016a). Semi-supervised classification with graph convolutional networks.

Kipf, T. N. & Welling, M. (2016b). Semi-supervised classification with graph convolutional networks.

Lafon, S. & Lee, A. B. (2006). Diffusion maps and coarse-graining: a unified framework for dimensionality reduction, graph partitioning, and data set parameterization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *28*(9), 1393–1403.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., & Peysakhovich, A. (2019). Pytorch-biggraph: A large-scale graph embedding system.

Leskovec, J. & Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Liang, J., Gurukar, S., & Parthasarathy, S. (2018). Mile: A multi-level framework for scalable graph embedding.

Loukas, A. (2018). Graph reduction with spectral and cut guarantees.

Lu, H., Halappanavar, M., & Kalyanaraman, A. (2014). Parallel heuristics for scalable community detection.

Meusel, R. (2015). The graph structure in the web – analyzed on different aggregation levels. *Journal of Web Science*, *1*, 33–47.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space.

Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., & Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, (pp. 29–42)., New York, NY, USA. ACM.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Neville, J. & Jensen, D. (2002). J. neville and d. jensen (2000). iterative classification in relational data. proceedings of the aaai 2000 workshop learning statistical.

Newman, M. E. J. & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, *69*(2).

Niu, F., Recht, B., Re, C., & Wright, S. J. (2011). Hogwild! a lock-free approach to parallelizing stochastic gradient descent. In *Proc. 24th Int. Conf. on Neural Information Processing Systems*, NIPS'11, (pp. 693–701)., NY, USA. Curran Associates Inc.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. In *WWW 1999*.

Peleg, D. & Schäffer, A. A. (1989). Graph spanners. *Journal of Graph Theory*, *13*(1), 99–116.

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proc. 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, KDD '14, (pp. 701–710)., NY, USA. ACM.

Qiu, J., Dong, Y., Ma, H., Li, J., Wang, K., & Tang, J. (2018). Network embedding as matrix factorization. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining - WSDM '18*.

Rossi, R. A. & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Roweis, S. T. & Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, *290*(5500), 2323–2326.

Spielman, D. A. & Teng, S.-H. (2008). Spectral sparsification of graphs.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). Line: Large-scale information network embedding. In *Proc. 24th Int. Conf. on World Wide Web*, (pp. 1067–1077). IW3C2.

Tang, L. & Liu, H. (2011). Leveraging social media networks for classification. *Data Min. Knowl. Discov.*, *23*, 447–478.

Tsay, A., Lovejoy, W., & Karger, D. (1999). Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, *24*, 383–413.

Tsitsulin, A., Mottin, D., Karras, P., & Müller, E. (2018). Verse: Versatile graph embeddings from similarity measures. In *Proc. World Wide Web Conference*, WWW '18, (pp. 539–548)., Republic and Canton of Geneva, CHE. IW3C2.

Wang, D., Cui, P., & Zhu, W. (2016). Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (pp. 1225–1234)., New York, NY, USA. Association for Computing Machinery.

Zhu, Z., Xu, S., Tang, J., & Qu, M. (2019). Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*, WWW '19, (pp. 2494–2504)., NY, USA. ACM.