LightNE: A Lightweight Graph Processing System for Network Embedding

Jiezhong Qiu*
Tsinghua University
qiujz16@mails.tsinghua.edu.cn

Laxman Dhulipala*
MIT CSAIL
laxman@mit.edu

Jie Tang*
Tsinghua University
jietang@tsinghua.edu.cn

Richard Peng*
Georgia Tech
rpeng@cc.gatech.edu

Chi Wang
Microsoft Research, Redmond
wang.chi@microsoft.com

ABSTRACT

We propose LightNE,[1] a cost-effective, scalable, and high quality network embedding system that scales to graphs with hundreds of billions of edges on a single machine. In contrast to the mainstream belief that distributed architecture and GPUs are needed for large-scale network embedding with good quality, we prove that we can achieve higher quality, better scalability, lower cost and faster runtime with shared-memory, CPU-only architecture. LightNE combines two theoretically grounded embedding methods NetSMF and ProNE. We introduce the following techniques to network embedding for the first time: (1) a newly proposed downsampling method to reduce the sample complexity of NetSMF while preserving its theoretical advantages; (2) a high-performance parallel graph processing stack GBBS to achieve high memory efficiency and scalability; (3) sparse parallel hash table to aggregate and maintain the matrix sparsifier in memory; and (4) Intel MKL for efficient randomized SVD and spectral propagation.

ACM Reference Format:
Jiezhong Qiu, Laxman Dhulipala, Jie Tang, Richard Peng, and Chi Wang. 2021. LightNE: A Lightweight Graph Processing System for Network Embedding. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3448016.3457329

1 INTRODUCTION

E-commerce and social networking companies today face the challenge of analyzing and mining graphs with billions of nodes, and tens of billions to trillions of edges. In recent years, a popular learning approach has been to apply network embedding techniques to obtain a vector representation of each node. These learned representations, or embeddings, can be easily consumed in downstream machine learning and recommendation algorithms. These representations are widely used in various online services and are updated frequently [25, 36, 39]. For example, one of the core item-recommendation systems at Alibaba with billions of items and users requires frequent re-embedding as both new users and items arrive online, and the underlying embedding must be quickly recomputed [36]. A similar system at LinkedIn computes embeddings of millions of individuals (nodes) offline and must periodically re-embed this graph to maintain high accuracy [25]. In both scenarios, computing embeddings must be done scalably and with low latency.

Despite a significant amount of research on developing sophisticated network embedding algorithms [7, 21, 32], using simple and scalable embedding solutions that potentially sacrifice a significant amount of accuracy remains the primary choice in the industry for dealing with large-scale graphs. For example, LinkedIn uses LINE [32] for embedding, which only captures local structural information within nodes' 1-hop neighborhoods. Alibaba embeds a 600-billion-node commodity graph by first partitioning it into 12,000 50-million-node subgraphs, and then embedding each subgraph separately with 100 GPUs running DeepWalk [21]. The reason is that in practice, graphs are updated frequently, and the embedding algorithms are often required to run every few hours [36].

While many new embedding systems have been proposed in the literature that demonstrate high accuracy for downstream applications, the high latency, limited scalability, and high computational cost prohibit these techniques from large scale deployment or commercial usage on massive datasets. For example, GraphVite [41] is a CPU-GPU hybrid system based on DeepWalk, which takes 20 hours to train on the Friendster graph (65M nodes and 1.8B edges) with 4 P100 GPUs. The cost of GraphVite for obtaining the embedding on this graph is 210 dollars measured by cloud virtual machine rent. One can estimate that embedding 10,000 such graphs (following the Alibaba approach) using GraphVite would amount to over 2 million dollars per run, which is prohibitively costly.

Motivated by the desire to obtain accurate, highly scalable, and cost-effective solutions that can embed networks with billions of nodes and hundreds of billions of edges, we design LightNE. Our design has the following objectives:

(1) **Scalable**: Embed graphs with 1B edges within 1.5 hours.
(2) **Lightweight**: Occupy hardware costs below 100 dollars measured by cloud rent to process 1B to 100B edges.
(3) **Accurate**: Achieve the highest accuracy in downstream tasks under the same time budget and similar resources.

Our Techniques. To reduce both cost and latency, we use a single-machine shared-memory environment equipped with multi-core

*Part of the work was done while the authors visited Microsoft Research.
[1]Our code is available at https://github.com/xptree/LightNE.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457329

CPUs, which are ubiquitous from cloud-providers today. Furthermore, to optimize our processing times and fully utilize the system, we avoid using SSDs or other external storage and instead utilize enough RAM so that both the input graph and all the intermediate steps can fit into memory, e.g., 1.5TB of RAM. Purchasing or renting a system with sufficient RAM and multi-core CPU(s) is the dominant cost of our system. Even with such a simple architecture, we successfully meet all our design goals by leveraging the following techniques and building an integrated system:

Firstly, we combine two lines of advances on efficient and effective network embedding techniques, **sample-based approximation** and **spectral approximation** of random walks stemming from the original DeepWalk: NetSMF [22] and ProNE [40]. Instead of using the general-purpose and computationally inefficient stochastic gradient descent method from most other solutions, both NetSMF and ProNE perform principled, cheap matrix operations on graphs to leverage the unique characteristics of real-world graphs such as sparsity, power-law degree distribution, and spectral properties. We combine both sample-based approximation and spectral approximation of random walks to achieve high accuracy levels while maintaining both of their advantages of low resource consumption and efficiency on real-world graphs.

Secondly, we propose **a new sampling algorithm** that reduces the number of required random-walk samples of original NetSMF by a factor of #edges/#vertices. On real-world graphs, it achieves a 10-100× reduction in the number of samples. Our algorithm is grounded in **spectral graph sparsification theory**.

Thirdly, we optimize our system for commodity shared-memory architectures and performing sparse matrix operations by (1) utilizing state-of-the-art **shared-memory graph processing techniques**, including parallel graph compression and efficient bulk-parallel operations, (2) integrating efficient parallel data structures, and techniques such as **sparse hash tables** for random walk samplers, and (3) a new **randomized SVD** subroutine based on Intel Math Kernel Library (MKL). These techniques enable us to achieve between 4–32× speedup over state-of-the-art network embedding systems, while also experiencing a similar order of magnitude cost improvement, all while maintaining or improving accuracy. Our memory efficiency enables us to scale to graphs significantly larger than those processed by single-machine embedding systems today. In particular, we show that using LightNE we can embed one of the largest publicly available graphs, the WebDataCommons hyperlink 2014 graph, with over 100 billion edges in under 2 hours.

Compared to three large-scale systems: GraphVite, PyTorch-BigGraph, and NetSMF, and using the tasks and the largest datasets evaluated by each system LightNE takes an order of magnitude lower latency and cost, while achieving the state-of-the-art accuracy. Compared to ProNE, our accuracy is significantly higher while the latency is comparable. In addition, we show that our system can scale to networks with billions of nodes, and hundreds of billions of edges on a single machine, which has never been demonstrated by any existing network embedding systems, including ProNE.

2 RELATED WORK

We review related work of network embedding algorithms/systems.

Network Embedding Algorithms. Over the last decade, network embedding algorithms have been extensively studied. A survey can be found in [10]. From an optimization aspect, recent network embedding algorithms fall into three main categories. The first category uses general-purpose stochastic gradient descent to optimize a logistic loss and follows the skip-gram model framework [18]. Methods belonging to this category include DeepWalk [21], LINE [32], and node2vec [7]. To date, the only bounds on sample efficiency and convergence rate for these methods require additional assumptions [4]. The second category uses singular value decomposition (SVD) to obtain the best low-rank approximations [6]. Examples of methods in this category include GraRep [2], HOPE [20], NetMF [23], NetSMF [22], and ProNE [40]. LightNE also belongs to this category. Graph Neural Networks represent the third line of network embedding algorithms [1]. Such methods include GCN [12], GAT [35], GIN [37], GraphSAGE [10] and PinSAGE [39]. These algorithms usually rely on vertex attributes, as well as supervised information. They are beyond the scope of this paper because our focus is on graphs with no additional information.

Network Embedding Systems. Due to the efficiency challenges posed by large graphs, several systems for embedding large graphs have been developed. We give a brief overview of the most related and comparable ones. *GraphVite* [41] is a CPU-GPU hybrid network embedding system based on DeepWalk [21] and LINE [32]. It uses CPU to conduct graph operations and GPU to perform linear algebra operations. The system is bounded by GPU memory, which in most cases is at most 32GB per GPU: embedding graphs with billions of vertices often require hundreds of Gigabytes of parameter memory. This limit constraints GraphVite to repeatedly updating only a small part of the embedding matrix. *PyTorch-BigGraph* [15] is a distributed memory system based on DeepWalk [21] and LINE [32]. It uses graph partition for load balancing and a shared parameter server for synchronization. LightNE is designed for shared memory machines, where communication is much cheaper than distributed memory systems. *NetSMF* [22] is a network embedding system based on sparse matrix factorization. The system is built on OpenMP and Eigen3 (a C++ template library for linear algebra). On large graphs, NetSMF is still time-consuming due to its poor implementation of the graph processing system and the shortcoming of Eigen3 in supporting sparse matrix operations. Our proposal contains a redesign of NetSMF that focuses on the performances on graphs and sparse matrices. A detailed experimental comparison of LightNE and NetSMF is in Section 5. *NPR* [38] is a recently proposed network embedding system built upon Matlab. It derives embeddings from the pairwise personalized PageRank (PPR) matrix. Although it is also based on random walks, it omits a step of taking the entry-wise logarithm of the random walk matrix before factorization, which is a required step by NetMF and NetSMF for establishing the equivalence to DeepWalk. Due to that omission, NPR is able to operate on the original graph efficiently while the others must construct the random walk matrix exactly or approximately.

3 BACKGROUND AND ALGORITHM

3.1 Background

We provide a self-contained background of the fundamental embedding techniques of our system. The list of notations used in this paper can be found in Table 1. We take a matrix-oriented view of graph embedding: the resulting embedding vectors are simply

Table 1: Notation used throughout this paper.

Notation	Description	Notation	Description
G	input network	b	#negative samples
V	vertex set, $\|V\| = n$	T	context window size
E	edge set, $\|E\| = m$	$\boldsymbol{X}$	$n \times d$ embedding matrix
$\boldsymbol{A}$	adjacency matrix	k	spectral propagation steps
$\boldsymbol{D}$	degree matrix	$\boldsymbol{D} - \boldsymbol{A}$	graph Laplaician $\boldsymbol{L}$
vol (G)	volume of G	$\boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}$	normalized Laplacian $\mathcal{L}$
M	# edge samples	d	embedding dimension

Algorithm 1: PathSampling.

```
Procedure PathSample(G, u, v, r)
    Let a random edge (u, v) be given.
    Sample a random number s uniformly in [0, r − 1].
    u' ← random walk u for s steps on graph G
    v' ← random walk v for r − 1 − s steps on graph G.
    return edge (u', v')
```

the rows of a $n \times d$ matrix $\boldsymbol{X}$ that we compute from the original adjacency matrix $\boldsymbol{A}$ of $n \times n$, where n and d denote the number of nodes and the embedding dimension, respectively. This interpretation of embedding gives us the flexibility of utilizing multiple matrix processing tools and synthesizing them.

NetMF. We begin with the matrix factorization approach introduced in [23], which showed that most network embedding methods up to that point in time, including DeepWalk [21] LINE [32], and node2vec [7], can be described as factorizing a matrix polynomial of the adjacency matrix $\boldsymbol{A}$ and degree matrix $\boldsymbol{D}$ of the graph. Formally, for an unweighted, undirected graph, $\boldsymbol{A}$ is the matrix with 1 in every entry with an edge, and 0 everywhere else; and the matrix $\boldsymbol{D}$ is the diagonal matrix where the i-th diagonal entry contains the degree of the i-th vertex. The core result by [23] is that DeepWalk can be viewed as approximately factorizing the following matrix

$$\boldsymbol{M} \triangleq \text{trunc\_log}^{\circ}\left(\frac{\text{vol}(G)}{b}\frac{1}{T}\sum_{r=1}^{T}(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}\boldsymbol{D}^{-1}\right) \tag{1}$$

in which T represents the length of random walks (by default $T = 10$), trunc_log$^{\circ}$ is the truncated logarithm applied entry-wise to a matrix (trunc_log$(x) = \max\{0, \log x\}$), and vol$(G) = 2m$ is the total number of edges in G. Moreover, LINE approximately factorizes a matrix in the same form but for $T = 1$. The bottleneck of factorizing the matrix in Equation (1) is that $(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}$ tends to be a dense matrix as the increase of r, and thus constructing the matrix is cost-prohibitive even before the factorization can be performed, due to the sheer amount of memory required. Note that the truncated logarithm is critical for embedding quality and cannot be omitted, otherwise there exists a shortcut to the factorization without constructing the dense matrix, similar to NPR [38].

NetSMF. One approach to mitigate the increased construction cost for constructing the matrix in Equation (1) is through the sampling of random walks. Qiu et al. [22] showed that an r-step random walk matrix $(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}$ could be approximated by repeating the PathSampling algorithm (Algo. 1). The result of applying Algo. 1 is a r-step walk in $\boldsymbol{A}$ and contributes to a non-zero entry to a sparsified version of $(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}$. Building upon an analysis of sparsification of random walk matrix polynomials [3], Qiu et al. [22] showed that a nearly-linear number of samples w.r.t. the number of edges in G (i.e., m) is sufficient to make a spectral approximation of $(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}$. They then demonstrated experimentally that this matrix could be used in place of the dense random walk matrix used by NetMF. Improving the scalability of NetSMF, especially the efficient sampling of random walks, is the starting point of our system.

ProNE. ProNE [40] proposed to firstly conduct Singular Value Decomposition (SVD) on a matrix $\boldsymbol{M}$ with each entry defined to be $\boldsymbol{M}_{uv} \triangleq \log\left(\frac{\boldsymbol{A}_{uv}}{\boldsymbol{D}_u}\frac{\sum_j(\sum_i \boldsymbol{A}_{ij}/\boldsymbol{D}_i)^{\alpha}}{b(\sum_i \boldsymbol{A}_{iv}/\boldsymbol{D}_i)^{\alpha}}\right)$, which is a modulated normalized graph Laplacian, with $b = 1$ and $\alpha = 0.75$ by default. Given the factorized embedding matrix $\boldsymbol{X}$, ProNE applies a filter to each column of the matrix using a low degree polynomial in the normalized graph Laplacian matrix $\mathcal{L} \triangleq \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}$, i.e., $\sum_{r=0}^{k} c_r \mathcal{L}^{r}\boldsymbol{X}$, where c_r's are chosen to be coefficients of Chebyshev polynomials and k is set to around 10. We will utilize the same choice of parameters for this spectral propagation step, but instead apply it to the factorization of the sparsified NetMF matrix in Equation (1).

3.2 LightNE: Algorithm Design

From an algorithm design perspective, our design of LightNE combines NetSMF and ProNE. In particular, LightNE consists of two steps. The first step is NetSMF with a novel edge downsampling algorithm, which significantly improves sample complexity. The second step is to enhance NetSMF embedding using ProNE's spectral propagation. We then introduce the two steps in detail.

Step 1: NetSMF with Edge Downsampling. NetSMF [22] has proposed an efficient PathSampling algorithm to approximate the r-step random walk matrix, $(\boldsymbol{D}^{-1}\boldsymbol{A})^{r}$, with roughly $O(m)$ samples. However, for graphs with billions of edges, there is still an urgent need to further reduce its sample complexity while preserving its theoretical advantages. Our approach is to downsample edges that will be added to the sparsifier. We do this by adding a further layer of sampling to Algo. 1 — for each sampled edge $e = (u, v)$, we flip a coin that comes up heads with some probability p_e, then only apply Algo. 1 and add the sampled edge to the sparsifier with adjusted weight $A_{u,v}/p_e$ if the coin comes up heads. Such a sampling method is a special case of importance sampling — adjusting edge weights ensures the downsampled graph is an unbiased estimation to the original graph in terms of the graph Laplacian:

Theorem 3.1 (Unbiasness of Edge Downsampling, Sec 6.5 in [34]). *Let the graph Laplacian of the original graph be $\boldsymbol{L}_G \triangleq \boldsymbol{D} - \boldsymbol{A}$. Note that $\boldsymbol{L}_G = \sum_{(u,v)\in E} A_{u,v}\boldsymbol{L}_{u,v}$ where $\boldsymbol{L}_{u,v}$ is the Laplacian matrix of the graph with just one unweighted edge between u and v. Also denote the downsampled graph to be H, then we have $\mathbb{E}[\boldsymbol{L}_H] = \sum_{e=(u,v)\in E} p_e \frac{A_{u,v}}{p_e}\boldsymbol{L}_{u,v} = \boldsymbol{L}_G$.*

In theory, setting sampling probability p_e as an upper bound of the effective resistance [31] guarantees an accurate approximation of input graph G with high probability, i.e., $p_e \leftarrow \min(1, CA_{u,v}R_{u,v})$ where $R_{u,v}$ is the effective resistance between u and v, and C is some constant. However, how to quickly approximate the effective resistances remains an open problem [14, 31]. Our choice is to adopt degree sampling. For an edge $e = (u, v)$, we set the sampling probability $p_e \leftarrow \min(1, CA_{u,v}(d_u^{-1} + d_v^{-1}))$. Here $d_u = \sum_v A_{u,v}$ is the degree of u. The following theorem from Lovász et al. [16]

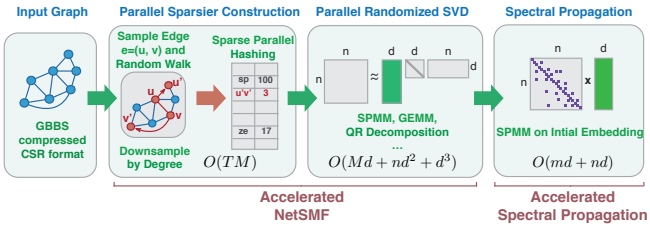


Figure 1: System overview of LightNE.

showed that the quantity $d_u^{-1} + d_v^{-1}$ is a simple but good upper bound to the effective resistance, especially for expander graphs:

Theorem 3.2 (Corollary 3.3 in [16]). *For* $\forall u, v \in V$, $\frac{1}{2}\left(\frac{1}{d_u} + \frac{1}{d_v}\right) \leq R_{u,v} \leq \frac{1}{1-\lambda_2}\left(\frac{1}{d_u} + \frac{1}{d_v}\right)$, *where* $1 - \lambda_2$ *is the spectral gap of the normalized graph Laplacian.*

Such a scheme ensures that the total number of edges kept in expectation is $O(nC)$: we can show that for any vertex u, we have $\sum_v A_{uv} d_u^{-1} = 1$ by the definition of d_u. Furthermore, we can increase the constant C in order to increase the concentration of the samples we pick. In this work, we set $C = \log(n)$.

Experimentally, this downsampling has negligible effects on the qualities of the embedding we produce, but significantly reduces the edge count: most of our graphs initially have at least 10 times as many edges as vertices, and the random walk graphs have even more edges. We believe this can be justified using the guarantees of this degree-sampling scheme in well-connected cases [30]. For example, the spectral gap (i.e., $1 - \lambda_2$ in Theorem 3.2) of the Blogcatalog graph [33] is about 0.43 [24], and it is widely believed that most web graphs are well connected, too [19].

Overall, the first step of LightNE provides an $O(n \log n)$-sparse but accurate estimator to the NetMF matrix in Equation (1). By conducting randomized SVD on the sparsifer such that $\boldsymbol{M} \approx \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, the NetSMF embedding matrix is defined as $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$.

Step 2: Embedding Enhancement with Spectral Propagation. Once the embedding $\boldsymbol{X}$ is obtained, we apply spectral propagation to further improve its quality. Following ProNE, the final embedding is enhanced by applying a polynomial $\boldsymbol{X} \leftarrow \sum_{r=0}^{k} c_r \mathcal{L}^r \boldsymbol{X}$, where k is set to be 10, $\mathcal{L} \triangleq \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}$ is the normalized graph Laplacian matrix and c_r's are the coefficients of Chebyshev polynomials.

4 LightNE: System Design

Overview. We make a series of system optimizations to enable LightNE in a CPU-only shared-memory machine. As introduced in Section 3.2, LightNE consists of two steps — NetSMF and spectral propagation. From a system perspective, the NetSMF step can be further decomposed into two sub-steps — parallel sparsifier construction and parallel randomized SVD. In this section, we present our acceleration techniques for these components. In Section 4.1, we introduce a new graph processing system, GBBS, which we leverage throughout LightNE. In Section 4.2, we discuss how we optimize sparsifer construction with GBBS and sparse parallel hashing, which enable us to aggregate and construct the sparsifier in memory efficiently. Lastly, in Section 4.3, we describe our randomized SVD and spectral propagation implementation using Intel MKL. An overview of our design can be found in Figure 1.

4.1 Sparse Parallel Graph Processing

LightNE involves intensive graph operations, such as performing random walks, querying vertex degrees, random accessing a neighbor of a vertex, etc. In this work, we build on the Graph Based Benchmark Suite (GBBS) [5], which extends the Ligra [26] interface with additional purely-functional primitives such as maps, reduces, filters over both vertices and graphs. We chose GBBS because it is performant, relatively simple to use, and has already been shown to scale to real-world networks with billions–hundreds of billions of edges on a single machine, achieving state-of-the-art running times for many fundamental graph problems.

Compression. An important design consideration for LightNE is to embed very large graphs on a single machine. Although the CSR format is normally regarded as a good compressed graph representation [11], we need to further compress this data structure and reduce memory usage. Our approach builds on state-of-the-art parallel graph compression techniques, which enable both fast parallel graph encoding and decoding. In particular, we adopt the *parallel-byte* format from Ligra+ [28]. In *sequential byte coding*, we store a vertex's neighbor list by difference encoding consecutive vertices, with the first vertex difference encoded with respect to the source. A decoder processes each difference one at a time, and sums the differences into a running sum which gives the ID of the next neighbor. Unfortunately, this process is entirely sequential, which could be costly for high-degree vertices and thus inhibit parallelism. The *parallel-byte* format from Ligra+ breaks the neighbors of a high-degree vertex into blocks, where each block contains a configurable number of neighbors. Each block is internally difference-encoded with respect to the source. As each block can have a different compressed size, the format also stores offsets from the start of the vertex to the start of each block. In what follows, when we refer to *compressed* graphs, we mean graphs in the CSR format where neighbor lists are compressed using the parallel-byte format.

To the best of our knowledge, we are the first to introduce GBBS and Ligra+ to the network embedding problem.

4.2 Parallel Sparsifier Construction

As described in Section 3.1, building the sparsifier requires: (1) generating a large number of edge samples using the PathSampling in Algo. 1 and (2) aggregating the sampled edges to count the frequency each distinct edge appears. After ensuring that the input graph is compressed, the main challenge in our design is to efficiently construct and store the sparsifier in memory. In this section, we use both the purely-functional primitives and the parallel compression techniques in GBBS to scalably and memory-efficiently conduct the PathSampling. We further employ sparse parallel hashing to aggregate the sampled edges and construct the sparsifer.

Parallel Per Edge PathSampling by GBBS. A natural idea is to repetitively call PathSampling(G, u, v, r) (Algo. 1) with a uniformly sampled edge (u, v) and a uniformly sampled path length $r \in [T]$. Unfortunately, this approach is challenging to implement on compressed graphs — it requires an efficient way to sample and access a random edge. Straightforward methods store all edges in an array which enables $O(1)$ random access, or perform binary search on the prefix sums of vertex degrees and select the chosen edge incident to a particular vertex. The former would require a prohibitive amount

Algorithm 2: Downsampled Per-Edge PathSampling.

```
1 Procedure DownSampledPerEdgePathSampling(G, T)
2     G.MapEdges(function (e = (u, v)) →
3         n_e ← ⌊M/m⌋+random variable from Bernoulli ({M/m})
4         for i ← 1 to n_e do
5             p ← Uniform[0, 1]; r ← Uniform[1, T]
6             if p < p_e then
7                 (u', v') ← PathSampling(G, u, v, r)
8                 Add (u', v') with weight 1/p_e to the sparsifier )
```

of memory for our largest networks, and the latter would require extra $O(\log n)$ time for binary searching each sample.

Instead, we propose Algo. 2, which describes an equivalent process that has the benefit of being more cache and memory-friendly, and works seamlessly alongside compression. The idea is to map over the edges in parallel, and for each edge $e = (u, v)$ we run PathSampling (Algo. 1) n_e times where n_e is $\lfloor M/m \rfloor$ plus a Bernoulli random variable with mean $\{M/m\}$.[2] Since each edge e is sampled independently, the expected number of samples is exactly M.[3] And it is easy to see that $\Theta(M)$ samples are drawn with high probability by standard concentration bounds. After sampling a value n_e from this random variable for a given edge (u, v), we perform n_e many random walks from (u, v) treating it as though it had been selected uniformly at random in the original process. We further incorporate the edge downsampling (as introduced in Section 3.2) into Alg. 2 to reduce the sample complexity. After drawing random variable n_e (Algo. 2, Line 3), for each of the n_e times this edge e is sampled, we flip a coin that comes up heads with probability p_e (Algo. 2, Line 5). We then apply Algo. 1 and add the sampled edge pair to the sparsifier with adjusted weight $1/p_e$ only if the coin comes up heads (Algo. 2, Line 7-8). Our implementation of the above idea uses the MapEdges primitive (Algo. 2, Line 2) in GBBS, which applies a user-defined function over every edge in parallel. The user-defined function (Algo. 2, Line 3-8) conducts the downsampled per-edge sampling we introduced above.

Lastly, we observe that implementing random walks in the shared-memory setting requires efficiently fetching the i-th edge incident to a vertex during the walk. This is because we simulate the random walk one step at a time by first sampling a uniformly random 32-bit value, and computing this value modulo the vertex degree. Fetching an arbitrary incident edge is trivial to implement for networks stored in CSR (without extra compression) by simply fetching the offset for a vertex and accessing its i-th edge. However, for graphs in CSR where adjacency information is additionally compressed in the parallel-byte format, we may need to decode an *entire block* in order to fetch the i-th edge. To help mitigate this cost, we chose a block size of 64 after experimentally evaluating the trade-off between the compressed size of the graph in memory, and the latency of fetching arbitrary edges incident to vertices. We note that further optimizations of this approach, such as batching multiple random walks accessing the same (or nearby vertices) together, to mitigate the cost of accessing these vertices' edges would require a careful analysis of the overhead for shuffling the data via a semisort [8], or a partial radix-sort [13] vs. the overhead for performing random reads. Optimizations of this flavor to further improve locality may be an interesting direction for future work.

Sparse Parallel Hashing. Next, we turn to how the sparsifier is constructed and represented in memory. After running Algo. 2 which generates many weighted edges, we need to count the frequency each distinct edge is sampled. We considered several different techniques for this aggregation problem in the shared-memory setting, including (1) generating per-processor lists of the edges and then merging the lists using the efficient sparse-histogram introduced in GBBS [5] and (2) storing the edges and partial-counts in per-processor hash tables that are periodically merged.

Ultimately, we found that the fastest and most memory-efficient method across all of our inputs was to use sparse parallel hashing. The construction used in this paper is folklore in the parallel algorithms literature, and we refer to Maier et al. [17] for a detailed explanation of the folklore algorithm. In a nutshell, our parallel hash table stores a distinct entry for each edge that is ever sampled, along with a count. Threads can access the table in parallel, and collisions are resolved using linear probing. Note that we do not require deletions in this setting. When multiple samples are drawn for a single edge, the counts are atomically incremented using the atomic xadd instruction. We note that the xadd instruction is significantly faster than a more naive implementation of a fetch-and-add instruction using compare-and-swap in a while loop when there is contention on a single memory location, and xadd is only negligibly slower in the light-load case [27]. Our implementation is lock-free and ensures that the exact count of each edge is computed, since our implementation uses atomic instructions to ensure that each sample is accounted for.

4.3 Randomized SVD and Spectral Propagation

Randomized SVD. After constructing the sparsifier, the next step is to efficiently perform randomized SVD and obtain the initial embedding. The randomized SVD [9] involves excessive linear algebra operations, which are well-supported and highly-optimized by the Intel MKL library. For example, its random projection is, in essence, a product of an $n \times n$ sparse matrix and a dense $n \times d$ Gaussian random matrix, which are implemented in MKL's Sparse BLAS Routines. Other examples include the Gram-Schmidt process and SVD on the projected matrices, which are all supported by Intel MKL LAPACK routines. We list the pseudo-code of randomized SVD [9] and the corresponding Intel MKL routines in Algo. 3.

Spectral Propagation. Besides randomized SVD, the spectral propagation step also involves linear algebra operations. Note that the spectral propagation step is highly efficient. It does not need to evaluate the higher powers of $\mathcal{L}$, but rather only applies repeated Sparse Matrix-Matrix multiplication (SPMM) between a sparse $n \times n$ Laplacian matrix $\mathcal{L}$ and a dense $n \times d$ embedding matrix, which can also be handled by MKL Sparse BLAS routines.

5 END-TO-END EVALUATIONS

In this section, we evaluate LightNE on nine graph datasets, summarized in Table 3. These datasets fall into three natural groups by scale: (1) large graphs in previous works, such as Friendster studied by GraphVite and OAG studied by NetSMF; (2) some of the largest publicly-available graphs where no previous results on network

[2] $\lfloor \cdot \rfloor$ is the floor function, and $\{\cdot\}$ is the fractional part of a number.
[3] $\mathbb{E}[n_e] = M/m$, and $\mathbb{E}[\sum_{e \in E} n_e] = \sum_{e \in E} \mathbb{E}[n_e] = M$.

Algorithm 3: Randomized SVD.

Procedure RandomizedSVD($\boldsymbol{A}$, d)
  Sample Gaussian random matrix $\boldsymbol{O}$ and $\boldsymbol{P}$ // vsRngGaussian
  Gaussian random projection $\boldsymbol{Y} = \boldsymbol{A}^\top \boldsymbol{O}$ // mkl_sparse_s_mm
  Orthonormalize $\boldsymbol{Y}$ // LAPACKE_sgeqrf, LAPACKE_sorgqr
  Compute $\boldsymbol{B} = \boldsymbol{AY}$ // mkl_sparse_s_mm
  Gaussian random projection $\boldsymbol{Z} = \boldsymbol{BP}$ // cblas_sgemm
  Orthonormalize $\boldsymbol{Z}$ // LAPACKE_sgeqrf, LAPACKE_sorgqr
  Compute $\boldsymbol{C} = \boldsymbol{Z}^\top \boldsymbol{B}$ // cblas_sgemm
  Run SVD on $\boldsymbol{C} = \boldsymbol{U}\Sigma\boldsymbol{V}^\top$ // LAPACKE_sgesvd
  return $\boldsymbol{ZU}$, Σ, $\boldsymbol{YV}$ // cblas_sgemm

Table 2: Hardware configurations and their most similar counterparts in Azure. N/A indicates it is not reported in the original paper.

		vCores	RAM	GPU	Price ($/h)
System	GraphVite	N/A	256 GB	4X P100	N/A
	PBG	48	256 GB	0	N/A
	NetSMF	64	1.7 TB	0	N/A
	LightNE	88	1.5 TB	0	N/A
Azure	NC24s v2	24	448 GiB	4X P100	8.28
	E48 v3	48	384 GiB	0	3.024
	M64	64	1024 GiB	0	6.669
	M128s	128	2,048 GiB	0	13.338

embeddings exist; (3) small-sized benchmarks standard to the network embedding literature, such as Blogcatalog and Youtube. We use the large graphs from (1) to demonstrate the effectiveness and efficiency of our method. The experiments on the very large graphs from (2) further demonstrate that our system can scale beyond previous work. The small graphs from (3) are used to verify the effectiveness of LightNE, although they are not our main target scenarios. We set up our evaluation in Section 5.1 and then report experimental results in the three groups, respectively.

5.1 Experimental Setup

Hardware Configuration. For LightNE, all experiments are conducted on a server with two Intel® Xeon® E5-2699 v4 CPUs (88 virtual cores in total) and 1.5 TB memory.

Accuracy Metrics. We follow the tasks and evaluation metrics in the original proposals. When comparing to PBG on LiveJournal, we evaluate the link prediction task with metrics to be mean rank (MR), mean reciprocal rank (MRR), and HITS@10. When comparing to GraphVite on Hyperlink-PLD , we evaluate the link prediction task with metric to be AUC. For the rest of the datasets, the task is node classification, and the metric is Mico/Macro F1.

Efficiency Metrics. We compare both the time and cost efficiency of different systems. Time efficiency is measured by running time, while cost efficiency is measured by estimated cost. Our cost estimation is based on the pricing ($) on Azure Cloud (The AWS price is very similar). We search the most suitable Azure instance for each system and then use its price per hour multiplied by the running time to estimate the cost. As shown in Table 2, we assume GraphVite uses NC24s v2, PBG uses E48 v3, while NetSMF and LightNE use M128s. The reason why we present cost efficiency is that different systems have different hardware requirements — GraphVite is a CPU-GPU hybrid system, while LightNE and NetSMF are CPU applications. Thus we use cloud rent price as a measure for the value of different hardware configurations.

5.2 Large Graphs

We compare to three systems that are designed for large graphs: PyTorch-BigGraph, GraphVite, and NetSMF. We use the tasks, datasets, hyper-parameters, and evaluation scripts provided by the corresponding papers' GitHub repos in making these comparisons.

5.2.1 Comparison with PyTorch-BigGraph (PBG). We compare with PBG on the LiveJournal dataset [4]. For LightNE, we set $T = 5$ by cross-validation. The results are reported in the following table:

	Time	Cost	MR	MRR	Hits@10
PBG	7.25 h	$21.95	4.25	0.87	0.93
LightNE	**16 min**	**$2.76**	**2.13**	**0.91**	**0.98**

Not only does LightNE achieve better performance regarding all metrics, but it also reduces time and cost by one order of magnitude. Specifically, LightNE is 27× faster and 8× cheaper than PBG.

5.2.2 Comparison with GraphVite. GraphVite offers the evaluation of link prediction task on Hyperlink-PLD and node classification task on Friendster-small and Friendster. The performance of LightNE is obtained by setting $T = 5$ via cross-validation. LightNE achieves AUC score 96.7 and outperforms GraphVite's 94.3. Limited by space, we report Micro-F1 for the two node classification tasks (Friendster-small and Friendster), and the conclusion for Macro-F1 is the same. For LightNE, cross-validation shows that the best performance is obtained by setting $T = 1$. The following is the performance when the label ratio is 1%, 5%, and 10%:

Metric	Dataset	Label Ratio (%)	1	5	10
Micro-F1	Friendster-small	GraphVite	76.93	87.94	89.18
		LightNE	**84.53**	**93.20**	**94.04**
	Friendster	GraphVite	72.47	86.30	88.37
		LightNE	**80.72**	**91.11**	**92.34**
AUC	Hyperlink-PLD	GraphVite	94.3	LightNE	**96.7**

As we can see, LightNE is significantly better than GraphVite. This again shows that our method is competitive.

As for efficiency, LightNE can embed Hyperlink-PLD in 30 min, 11× faster than GraphVite. Moreover, LightNE achieves 29× and 32× speedup respectively in Friendster-small and Friendster, and saves the cost by orders of magnitude (22× cheaper on Friendster-small and 25× cheaper on Friendster). The detailed efficiency comparison between LightNE and GraphVite is summarized as follows:

		Friendster-small	Hyperlink-PLD	Friendster
Time	GraphVite	2.79 h	5.36 h	20.3 h
	LightNE	**5.83 min**	**29.77 min**	**37.6 min**
Cost	GraphVite	$28.84	$44.38	$209.84
	LightNE	**$1.30**	**$6.62**	**$8.36**

5.2.3 Comparing with NetSMF and ProNE+. NetSMF and ProNE are redesigned and used as building blocks of LightNE. In this section,

[4] PBG reports results on LiveJournal, YouTube and Twitter, but only releases configuration for LiveJournal in the official github repository.

Table 3: Datasets statistics.

	Small Graphs ($\|E\| \leq 10M$)		Large Graphs ($10M < \|E\| \leq 10B$)					Very Large Graphs ($\|E\| > 10B$)	
	BlogCatalog	YouTube	LiveJournal	Friendster-small	Hyperlink-PLD	Friendster	OAG	ClueWeb-Sym	Hyperlink2014-Sym
$\|V\|$	10,312	1,138,499	4,847,571	7,944,949	39,497,204	65,608,376	67,768,244	978,408,098	1,724,573,718
$\|E\|$	333,983	2,990,443	68,993,773	447,219,610	623,056,313	1,806,067,142	895,368,962	74,744,358,622	124,141,874,032

Table 4: Comparison on OAG with label ratio 0.001%, 0.01%, 0.1% and 1%.

Metric	Method	Time	0.001%	0.01%	0.1%	1%
Micro	NetSMF (M=8Tm)	22.4 h	30.43	31.66	35.77	38.88
	ProNE+	21 min	23.56	29.32	31.17	31.46
	LightNE-Small	20.9 min	23.89	30.23	32.16	32.35
	LightNE-Large	1.53 h	**44.50**	**52.89**	**54.98**	**55.23**
Macro	NetSMF (M=8Tm)	22.4 h	7.84	9.34	13.72	17.82
	ProNE+	21 min	10.47	10.30	9.83	9.79
	LightNE-Small	20.9 min	10.90	11.92	11.59	11.57
	LightNE-Large	1.53 h	**25.85**	**35.72**	**38.18**	**38.53**

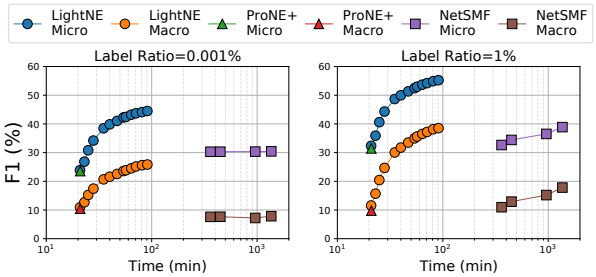


Figure 2: Efficiency-effectiveness trade-off curve of LightNE.

we focus on investigating how LightNE combines and strengthens the advantages of both. The dataset we adopt is OAG [29], the largest graph from NetSMF. We vary the number of edge samples M conducted by LightNE from $0.1Tm$ to $20Tm$. We denote the configuration with the fewest edge samples ($M = 0.1Tm$) as LightNE-Small, and the one with the most edge samples ($M = 20Tm$) as LightNE-Large. As for NetSMF, we enumerate its edge samples M from $\{1Tm, 2Tm, 4Tm, 8Tm\}$[5]. Both LightNE and NetSMF set $T = 10$. The Python implementation released by ProNE paper [40] is inefficient, and it "requires 29 hours to embed a network of hundreds of millions of nodes" (quote from the abstract of ProNE paper [40]). For fair comparison, we re-implement ProNE to benefit from our system optimizations (highly optimized GBBS for graph processing and MKL for linear algebra operations). Our re-implementation has comparable accuracy to the original one on datasets used in ProNE [40], but much faster. We refer to this new implementation as ProNE+. The predictive performance of LightNE, NetSMF and ProNE+ is shown in Table 4 and Figure 2.

Comparing LightNE-Large with NetSMF. As shown in Table 4, LightNE-Large achieves 14.9× speedup (1.53h v.s. 22.4h) and significant performance gain (on average 52.3% and 201.7% relatively better regarding Micro and Macro F1, respectively).

Comparing LightNE-Small with ProNE+. As shown in Table 4, not only does LightNE-Small run faster than ProNE+ (20.9 min v.s. 21 min), but also outperforms ProNE+ significantly (averagely +0.78 Micro F1 and +1.4 Macro F1).

[5]We can not run the experiment with $M = 10Tm$ in NetSMF paper because it needs 1.7TB memory but our machine has only 1.5TB memory.

Overall, LightNE is a Pareto-optimal solution that strictly dominates ProNE+ or NetSMF — for either ProNE+ or NetSMF, one can find a configuration of LightNE in Figure 2 that is faster and more accurate. Moreover, there is a clear trade-off between efficiency and effectiveness in LightNE. That means a user can configure and adjust LightNE flexibly according to his/her time/cost budgets and performance requirements. Comparing LightNE to NetSMF and ProNE+ also suggests that spectral propagation plays the role of "standing on the shoulder of giants" — the quality of the enhanced embedding heavily relies on that of the initial one.

5.2.4 Ablation Study on the OAG Dataset. Next, we conduct ablation studies on the OAG dataset.

Ablation Study on Running Time. We break down the running time of LightNE, NetSMF, and ProNE+, as shown in Table 5. LightNE consists of three stages — parallel sparsifier construction, randomized SVD, and spectral propagation. In contrast, (1) NetSMF does not have the third stage; (2) ProNE+ directly factorizes a simple graph Laplacian matrix, so it doesn't have the first stage;

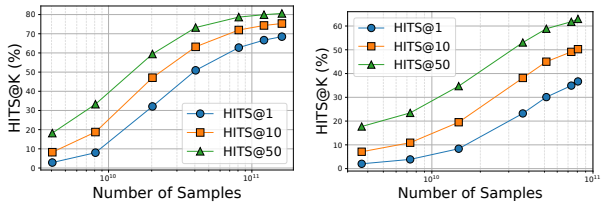
Comparing LightNE-Large with NetSMF, LightNE achieves 33x speedup when constructing the sparsifier, showing the advantages of the sparse parallel graph processing, the downsampling algorithm, and the sparse parallel hashing. It also achieves 4.8x speedup when factorizing the sparse matrix, showing the advantage of Intel MKL over Eigen3 in the implementation of randomized SVD.

Comparing LightNE-Small with ProNE+, both methods take 8.2 min in spectral propagation, while the factorization step in LightNE is slightly faster than that of ProNE+. The main reason is that the matrix factorized by LightNE-Small is sparser than ProNE+. Note that ProNE+ has exactly m non-zeros in its matrix to be factorized. However, for LightNE-Small with $M = 0.1Tm = m$ samples, the actual number of non-zeros in the sparsifier could be fewer than m, due to the downsampled PathSampling algorithm (Algo. 2).

Ablation Study on Sample Size. Comparing to NetSMF with $8Tm$ samples, LightNE-Large is able to draw up to $20Tm$ (2.5x) samples and achieves significantly better performance. The large sample size can be attributed to (1) compressed GBBS, (2) downsampling technique, and (3) sparse parallel hashing. However, the uncompressed OAG graph (in CSR format) occupies only 16GB; thus, the effect to compressed GBBS is negligible given our machine has 1.5TB memory (though compressed GBBS plays a big role in very large graphs, ref. Section 5.3). We turn off the downsampling in LightNE and gradually increase its number of samples until out-of-memory. We observe that we can have at most $12.5Tm$ samples without downsampling, which is 56.3% greater than NetSMF's $8Tm$. The above analysis suggests that the sparse parallel hashing is another contributor to the larger sample size — it increases affordable sample size by 56.3%, and downsampling further increases by 60%. This is because the shared-memory hash table in LightNE can significantly save memory. However, NetSMF maintains a thread-local sparsifer in each thread and merges them at the end of sampling.

Table 5: The running time distribution of LightNE, NetSMF and ProNE+. NA means the algorithm does not have the corresponding stage.

Time	Parallel Sparsifier Construction	Randomized SVD	Spectral Propagation
LightNE-Large	32.8 min	49.9 min	8.1 min
NetSMF (M=8Tm)	18 h	4 h	NA
LightNE-Small	1.4 min	10.5 min	8.2 min
ProNE+	NA	12.0 min	8.2 min



(a) ClueWeb-Sym **(b)** Hyperlink2014-Sym

Figure 3: HITS@K ($K = 1, 10, 50$) of LightNE w.r.t. the number of samples.

5.3 Very Large Graphs

To further illustrate the lightweightness and scalability of LightNE, we test LightNE on two very large 100-billion scale graphs, ClueWeb-Sym, and Hyperlink2014-Sym, as shown in Table 3. It is worth noting that none of the existing network embedding systems can handle such large graphs in a single machine. For example, it takes 564GB memory to store ClueWeb-Sym's 74 billion unweighted edges. Furthermore, the Hyperlink2014 graph is one of the largest publicly available graph today, and very few graph processing systems or graph algorithms have been applied to a graph of this magnitude, in any setting [5]. However, by adopting the graph compression from GBBS [5], we are able to reduce the size of ClueWeb-Sym to 107GB. Moreover, by leveraging the downsampling technique in Section 3.2, we are able to maintain a $O(n \log n)$ sparsifier, which only requires a modest amount of memory and enables us to apply randomized SVD without an excessive memory footprint.

To evaluate the performance of LightNE on very large graphs, we adopt link prediction to be the evaluation task, as vertex labels are not available for these graphs. We follow PBG to set up link prediction evaluation — we randomly exclude 0.00001% edges from the training graph for evaluation. When training LightNE on the two very large graphs, we skip the spectral propagation step (due to memory issue) and set $T = 2$ as well as $d = 32$. After training, the ranking metrics on the test set are obtained by ranking positive edges among randomly sampled corrupted edges. We gradually increase the number of edge samples until it reaches the 1.5TB memory bottleneck. For each experiment, LightNE needs fewer than two hours for training. Figure 3 presents the HITS@K with $K = 1, 10, 50$ of LightNE with different numbers of edge samples M. As we can see, the more samples we draw by edge sampling, the higher accuracy LightNE can achieve. Moreover, the trend of growth shown in Figure 3 suggests that the performance in these datasets can be further improved if we can overcome the memory bottleneck by, for example, using a machine with larger memory, or designing compressed hash tables and linear algebra tools.

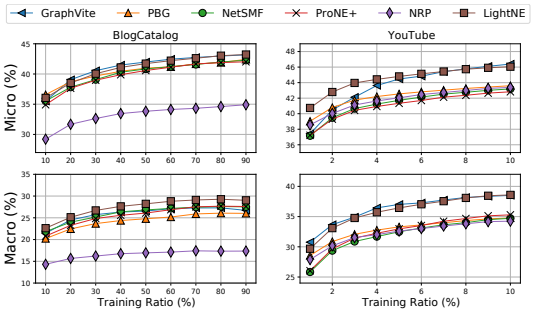


Figure 4: Predictive performance on small graphs.

5.4 Small Graphs

As shown in Figure 4, we compare the prediction performance of LightNE against all the baselines[6] in BlogCatalog and YouTube. In BlogCatalog, LightNE outperforms all the baselines consistently in terms of Macro-F1, and achieves comparable results to GraphVite regarding Micro-F1. In YouTube, the right panel of Figure 4 suggests that LightNE, together with GraphVite, consistently yields the best results among all the methods. In particular, LightNE shows better Micro-F1 than GraphVite when the training ratio is small (1-6%). We also highlight that ProNE+ performs consistently worse than LightNE, again showing that enhancing a simple embedding via spectral propagation may yield sub-optimal performance. The experiments on small graphs demonstrate the effectiveness of our system, though the system is mainly designed for larger graphs.

6 CONCLUSION

In this work, we present LightNE, a single-machine shared-memory system that significantly improves the efficiency, scalability, and accuracy of state-of-the-art network embedding techniques. LightNE combines two advanced network embedding algorithms, NetSMF and ProNE, to achieve state-of-the-art performance on nine benchmarking graph datasets, compared to three recent network embedding systems—GraphVite, PyTorch-BigGraph, and NetSMF. By incorporating sparse parallel graph processing techniques, and other parallel algorithmic techniques like sparse parallel hashing and high-performance parallel linear algebra, LightNE is able to learn high-quality embeddings for graphs with hundreds of billions of edges in a few hours, all at a modest cost.

In the future, as we discussed in Section 5.2, we plan to study techniques for reducing the memory bottleneck on our hash table and SVD implementation. Designing efficient compression techniques for these data structures is a promising avenue to achieve even better performance for massive real-world networks. We also would like to study large-scale network embedding in a streaming or dynamic setting.

ACKNOWLEDGMENTS

Jiezhong Qiu and Jie Tang were supported by the National Key R&D Program of China (2018YFB1402600), NSFC for Distinguished Young Scholar (61825602), and NSFC (61836013). Richard Peng was partially supported by the US National Science Foundation under grant number 18462.

[6] NRP is from github.com/AnryYang/nrp. PBG's hyper-parameters for YouTube haven't been officially released, so we set them by cross-validation.

REFERENCES

[1] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).

[2] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. Grarep: Learning graph representations with global structural information. In *CIKM '15*. 891–900.

[3] Dehua Cheng, Yu Cheng, Yan Liu, Richard Peng, and Shang-Hua Teng. 2015. Efficient Sampling for Gaussian Graphical Models via Spectral Sparsification. *Proceedings of The 28th Conference on Learning Theory* (2015), 364–390.

[4] Christopher De Sa, Kunle Olukotun, and Christopher Ré. 2014. Global convergence of stochastic gradient descent for some non-convex matrix problems. *arXiv preprint arXiv:1411.1134* (2014).

[5] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 393–404.

[6] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.

[7] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD '16*. 855–864.

[8] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *SPAA '15*. 24–34.

[9] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. 2011. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review* 53, 2 (2011), 217–288.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data(base) Engineering Bulletin* 40 (2017), 52–74.

[11] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[12] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. (2017).

[13] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing indirect memory references with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 299–312.

[14] Ioannis Koutis, Alex Levin, and Richard Peng. 2012. Improved Spectral Sparsification and Numerical Algorithms for SDD Matrices. In *29th International Symposium on Theoretical Aspects of Computer Science*. Citeseer, 266.

[15] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287* (2019).

[16] László Lovász et al. 1993. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty* 2, 1 (1993), 1–46.

[17] Tobias Maier, Peter Sanders, and Roman Dementiev. 2016. Concurrent hash tables: Fast and general?(!). *ACM SIGPLAN Notices* 51, 8 (2016), 1–2.

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[19] Abedelaziz Mohaisen, Aaram Yun, and Yongdae Kim. 2010. Measuring the mixing time of social graphs. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 383–389.

[20] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric transitivity preserving graph embedding. In *KDD '16*. 1105–1114.

[21] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD '14*. ACM, 701–710.

[22] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. Netsmf: Large-scale network embedding as sparse matrix factorization. In *The World Wide Web Conference*. ACM, 1509–1520.

[23] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *WSDM '18*. ACM, 459–467.

[24] Jiezhong Qiu, Chi Wang, Ben Liao, Richard Peng, and Jie Tang. 2020. A Matrix Chernoff Bound for Markov Chains and Its Application to Co-occurrence Matrices. *NeurIPS '20* (2020).

[25] Rohan Ramanath, Hakan Inan, Gungor Polatkan, Bo Hu, Qi Guo, Cagri Ozcaglar, Xianren Wu, Krishnaram Kenthapadi, and Sahin Cem Geyik. 2018. Towards Deep and Representation Learning for Talent Search at LinkedIn. In *CIKM '18*. 2253–2261.

[26] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[27] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *SPAA*.

[28] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *DCC*.

[29] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Hsu, and Kuansan Wang. 2015. An overview of microsoft academic service (mas) and applications. In *WWW '15*. 243–246.

[30] D. Spielman and S. Teng. 2011. Spectral Sparsification of Graphs. *SIAM J. Comput.* 40, 4 (2011), 981–1025.

[31] Daniel A Spielman and Nikhil Srivastava. 2011. Graph sparsification by effective resistances. *SIAM J. Comput.* 40, 6 (2011), 1913–1926.

[32] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW '15*. 1067–1077.

[33] Lei Tang and Huan Liu. 2009. Relational learning via latent social dimensions. In *KDD '09*.

[34] Shang-Hua Teng. 2016. Scalable algorithms for data and network analysis. *Foundations and Trends® in Theoretical Computer Science* 12, 1–2 (2016), 1–274.

[35] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[36] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *KDD '18*. 839–848.

[37] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[38] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S Bhowmick. 2020. Homogeneous network embedding for massive graphs via reweighted personalized PageRank. *Proceedings of the VLDB Endowment* 13, 5 (2020), 670–683.

[39] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *KDD '18*. 974–983.

[40] Jie Zhang, Yuxiao Dong, Yan Wang, Jie Tang, and Ming Ding. 2019. ProNE: fast and scalable network representation learning. In *IJCAI '19*. 4278–4284.

[41] Zhaocheng Zhu, Shizhen Xu, Jian Tang, and Meng Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *The World Wide Web Conference*. ACM, 2494–2504.