Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank

Renchi Yang∗, Jieming Shi†, Xiaokui Xiao†, Yin Yang§, Sourav S. Bhowmick∗
∗School of Computer Science and Engineering, Nanyang Technological University, Singapore
†School of Computing, National University of Singapore, Singapore
§College of Science and Engineering, Hamad Bin Khalifa University, Qatar

∗{yang0461,assourav}@ntu.edu.sg,†{shijm,xkxiao}@nus.edu.sg,§yyang@hbku.edu.qa

ABSTRACT

Given an input graph G and a node $v \in G$, homogeneous network embedding (HNE) maps the graph structure in the vicinity of v to a compact, fixed-dimensional feature vector. This paper focuses on HNE for massive graphs, *e.g.*, with billions of edges. On this scale, most existing approaches fail, as they incur either prohibitively high costs, or severely compromised result utility.

Our proposed solution, called Node-Reweighted PageRank (NRP), is based on a classic idea of deriving embedding vectors from pairwise personalized PageRank (PPR) values. Our contributions are twofold: first, we design a simple and efficient baseline HNE method based on PPR that is capable of handling billion-edge graphs on commodity hardware; second and more importantly, we identify an inherent drawback of vanilla PPR, and address it in our main proposal NRP. Specifically, PPR was designed for a very different purpose, *i.e.*, ranking nodes in G based on their *relative* importance from a source node's perspective. In contrast, HNE aims to build node embeddings considering the *whole* graph. Consequently, node embeddings derived directly from PPR are of suboptimal utility.

The proposed NRP approach overcomes the above deficiency through an effective and efficient node reweighting algorithm, which augments PPR values with node degree information, and iteratively adjusts embedding vectors accordingly. Overall, NRP takes $O(m \log n)$ time and $O(m)$ space to compute all node embeddings for a graph with m edges and n nodes. Our extensive experiments that compare NRP against 18 existing solutions over 7 real graphs demonstrate that NRP achieves higher result utility than all the solutions for link prediction, graph reconstruction and node classification, while being up to orders of magnitude faster. In particular, on a billion-edge Twitter graph, NRP terminates within 4 hours, using a single CPU core.

1. INTRODUCTION

Given a graph $G = (V, E)$ with n nodes, a *network embedding* maps each node $v \in G$ to a compact feature vector in $\mathbb{R}^k$ ($k \ll n$), such that the embedding vector captures the graph features surrounding v. These embedding vectors are then used as inputs in downstream machine learning operations [41, 46, 55]. A *homogeneous network embedding (HNE)* is a type of network embedding that reflects the topology of G rather than labels associated with nodes or edges. HNE methods have been commonly applied to various graph mining tasks based on neighboring associated similarities, including node classification [35], link prediction [3], and graph reconstruction [30]. This paper focuses on HNE computation on massive graphs, *e.g.*, social networks involving billions of connections. Clearly, an effective solution for such a setting must be highly scalable and efficient, while obtaining high result utility.

HNE is a well studied problem in the data mining literature, and there are a plethora of solutions. However, most existing solutions fail to compute effective embeddings for large-scale graphs. For example, as we review in Section 2, a common approach is to learn node embeddings from random walk simulations, *e.g.*, in [16, 31]. However, the number of possible random walks grows exponentially with the length of the walk; thus, for longer walks on a large graph, it is infeasible for the training process to cover even a considerable portion of the random walk space. Another popular methodology is to construct node embeddings by factorizing a proximity matrix, *e.g.*, in [64]. The effectiveness of such methods depends on the proximity measure between node pairs. As explained in Section 2, capturing multi-hop topological information generally requires a sophisticated proximity measure; on the other hand, the computation, storage and factorization of such a proximity matrix often incur prohibitively high costs on large graphs.

This paper revisits an attractive idea: constructing HNEs by taking advantage of *personalized PageRank (PPR)* [20]. Specifically, given a pair of nodes $u, v \in G$, the PPR value $\pi(u, v)$ of v with respect to u is the probability that a random walk from u terminates at v. PPR values can be viewed as a concise summary of an infinite number of random walk simulations, which, intuitively, should be helpful in building node embeddings. Realizing the full potential of PPR for scalable HNE computation, however, is challenging. One
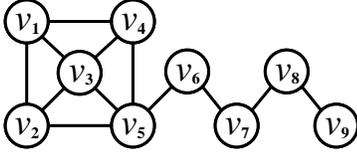
Figure 1: An example graph G.

Table 1: PPR for v_2 and v_9 in Fig. 1 ($\alpha = 0.15$).

v_i	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\pi(v_2, v_i)$	0.15	0.269	0.188	**0.118**	0.17	0.048	0.029	0.019	0.008
$\pi(v_4, v_i)$	0.15	**0.118**	0.188	0.269	0.17	0.048	0.029	0.019	0.008
$\pi(v_7, v_i)$	0.036	0.043	0.056	0.043	0.093	0.137	0.29	0.187	**0.12**
$\pi(v_9, v_i)$	0.02	0.024	0.031	0.024	0.056	0.083	**0.168**	0.311	0.282

major hurdle is cost: materializing the PPR between each pair of nodes clearly takes $O(n^2)$ space for n nodes (*e.g.*, in [59]), and evaluating even a single PPR value can involve numerous random walk simulations (*e.g.*, in [42, 65]).

Further, we point out that even without considering computational costs, it is still tricky to properly derive HNEs from PPR values. The main issue is that PPR was designed to serve a very different purpose, *i.e.*, ranking nodes in G based on their relative importance from a source node's perspective. In other words, PPR is essentially a *local* measure. On the other hand, HNE aims to summarize nodes from the view of the whole graph. To illustrate this critical difference, consider the example in Fig. 1 with nodes v_1-v_9. Observe that between the node pair v_2 and v_4, there are three different nodes connecting them, *i.e.*, v_1, v_3 and v_5. In contrast, there is only one common neighbor between v_9 and v_7. Intuitively, if we were to predict a new edge in the graph, it is more likely to be (v_2, v_4) than (v_9, v_7). For instance, in a social network, the more mutual friends two users have, the more likely they know each other [4]. However, as shown in Table 1, in terms of PPR values, we have $\pi(v_9, v_7) = 0.168 > \pi(v_2, v_4) = 0.118$, which tends to predict (v_9, v_7) over (v_2, v_4) and contradicts with the above intuition. This shows that PPR by itself is *not* an ideal proximity measure, at least for the task of link prediction. This problem is evident in PPR-based HNE methods, *e.g.*, in [42, 65], and a similar issue limits the effectiveness of a recent proposal [59], as explained in Section 2.

This paper addresses both the scalability and result utility issues of applying PPR to HNE computation with a novel solution called *Node-Reweighted PageRank* (NRP). Specifically, we first present a simple and effective baseline approach that overcomes the efficiency issue of computing node embeddings using PPR values. The main proposal NRP then extends this baseline by addressing the above-mentioned deficiency of conventional PPR. Specifically, NRP augments PPR values with additional reweighting steps, which calibrate the embedding of each node to its in- and out- degrees. Intuitively, when a node has a large number of neighbors (*e.g.*, v_2 in Fig. 1), its embedding vector should be weighted up in accordance with its degree, so as to reflect the importance of the node from the perspective of the whole graph. In NRP, node reweighting is performed using an effective and scalable algorithm that iteratively adjusts node embeddings, whose cost is small compared to PPR computations. Overall, NRP takes $O\left(k(m + kn)\log n\right)$ time and $O(m + nk)$ space to construct length-k embeddings of all nodes in a graph with n nodes and m edges. In the common case that k is small and the graph is sparse, the above complexities reduce to $O(m\log n)$ time and $O(m)$ space.

We have conducted extensive experiments on 7 popular real datasets, and compared NRP against 18 existing HNE solutions on three tasks: link prediction, graph reconstruction and node classification. In all settings, NRP achieves

the best result utility. Meanwhile, with a few exceptions, NRP is often orders of magnitude faster than its competitors. In particular, on a Twitter graph with 1.2 billion edges, NRP terminates within 4 hours on a single CPU core.

2. RELATED WORK

Network embedding is a hot topic in graph mining, for which there exists a large body of literature as surveyed in [5, 11, 61]. Here we review the HNE methods that are most relevant to this work.

Learning HNEs from random walks. A classic methodology for HNE computation is to learn embeddings from random walk simulations. Earlier methods in this category include DeepWalk [31], LINE [39], node2vec [16] and Walklets [32]. The basic idea is to learn the embedding of a node v by iteratively "pulling" the embeddings of positive context nodes (*i.e.*, those that are on the random walks originating from v) towards that of v, and "pushing" the embeddings of negative context nodes (*i.e.*, the nodes that are not connected to v) away from v. Subsequent proposals [8, 36] construct a multilayer graph over the original graph G, and then perform random walks on different layers to derive more effective embeddings. Instead of using a predefined sampling distribution, SeedNE [15] adaptively sample negative context nodes in terms of their informativeness. GraphCSC-M [9] learns the embeddings of different centrality-based random walks, and combines these embeddings into one by weighted aggregation. Recent techniques APP [65] and VERSE [42] improve the quality of embeddings by refining the procedures for learning from PPR-based random walk samples. However, neither of them addresses the deficiency of conventional PPR as described in Section 1.

The main problem of random-walk-based HNE learning in general is their immense computational costs (proportional to the number of random walks), which can be prohibitive for large graphs. The high running time could be reduced with massively-parallel hardware, *e.g.*, in PBG [26], and/or with GPU systems, *e.g.*, in Graphy [67]. Nevertheless, they still incur a high financial cost for consuming large amounts of computational resources.

Learning HNEs without random walks. HNEs can also be learned directly from the graph structure using a deep neural network, without performing random walks. Training such a deep neural network, however, also incurs very high computational overhead, especially for large graphs [42]. Notably, SDNE [44] and DNGR [7] employ multi-layer auto-encoders with a target proximity matrix to learn embeddings. GAE [22] combines the graph convolutional network [23] and auto-encoder models to learn embeddings. PRUNE [25] utilizes a Siamese neural network to preserve both pointwise mutual information and global PageRank of nodes. NetRA [60] and DRNE [43] learn embeddings by feeding node sequences to a long short-term memory model

(LSTM). DVNE [66] learns a Gaussian distribution in the Wasserstein space with the deep variational model as the latent representation of each node. GA [1] applies graph attention mechanism to a closed-form expectation of the limited random-walk co-occurrence matrices [31] to learn the embeddings. GraphGAN [45], ANE [12] and DWNS [13] adopt the popular generative adversarial networks (GAN) to accurately model the node connectivity probability. As demonstrated in our experiments, none of these methods scale to large graphs.

Constructing HNEs through matrix factorization. Another popular methodology for HNE is through factorizing a *proximity matrix* $\mathbf{M} \in \mathbb{R}^{n \times n}$, where n is the number of nodes in the input graph G, and each entry $\mathbf{M}[i,j]$ signifies the *proximity* between a pair of nodes $v_i, v_j \in G$. The main research question here is how to choose an appropriate $\mathbf{M}$ that (i) captures the graph topology well and (ii) is efficient to compute and factorize on large graphs. Specifically, to satisfy (i), each entry $\mathbf{M}[i,j] \in \mathbf{M}$ should accurately reflect the proximity between nodes v_i, v_j via indirect connections, which can be long and complex paths. Meanwhile, to satisfy (ii) above, the computation / factorization of $\mathbf{M}$ should be done in memory. This means that $\mathbf{M}$ should either be sparse, or be efficiently factorized without materialization. In addition, note that for a directed graph G, the proximity is also directed, meaning that it is possible that $\mathbf{M}[i,j] \neq \mathbf{M}[j,i]$. Thus, methods that require $\mathbf{M}$ to be symmetric are limited to undirected graphs and cannot handle directed graphs.

Earlier factorization-based work, including [2, 6, 40, 52], directly computes $\mathbf{M}$ before factorizing it to obtain node embeddings. For instance, spectral embedding [40] simply outputs the top k eigenvectors of the Laplacian matrix of an undirected graph G as node embeddings. This method has limited effectiveness [16, 31], as the Laplacian matrix only captures one-hop connectivity information for each node. To remedy this problem, one idea is to construct a higher-order proximity matrix $\mathbf{M}$ to capture multi-hop connectivity for each node [6, 52, 56]. However, such a higher-order proximity matrix $\mathbf{M}$ is usually no longer sparse; consequently, materializing $\mathbf{M}$ becomes prohibitively expensive for large graphs due to the $O(n^2)$ space complexity for n nodes.

Recent work [30, 63, 64] constructs network embeddings without materializing $\mathbf{M}$, to avoid extreme space overhead. Many of these methods, however, rely on the assumption that $\mathbf{M}$ is symmetric; consequently, they are limited to undirected graphs as discussed above. For instance, AROPE [64] first applies an eigen-decomposition on the adjacency matrix $\mathbf{A}$ of an undirected graph G, and then utilizes the decomposition results to derive each node's embedding to preserve proximity information, without explicitly constructing $\mathbf{M}$. Similar approaches have been adopted in [30, 63]. In particular, RandNE [63] uses a Gaussian random projection of $\mathbf{M}$ directly as node embeddings without factorization, in order to achieve high efficiency, at the cost of lower result utility.

The authors of [34] prove that random-walk-based methods such as as DeepWalk, LINE and node2vec essentially perform matrix factorizations. Thus, they propose NetMF, which factorizes a proximity matrix $\mathbf{M}$ that approximates the closed form representation of DeepWalk's implicit proximity matrix. However, NetMF requires materializing a dense $\mathbf{M}$, which is infeasible for large graphs. NetSMF [33] improves the efficiency of NetMF by sparsifying $\mathbf{M}$ using the

theory of spectral sparsification. However, NetSMF is still rather costly as it requires simulating a large number of random walks to construct $\mathbf{M}$. ProNE [62] learns embeddings via matrix factorization with the enhancement of spectral propagation. However, ProNE is mainly designed for node classification, and its accuracy is less competitive for other tasks such as link prediction and graph reconstruction. GRA [27] iteratively fine-tunes the proximity matrix to obtain enhanced result effectiveness, at the expense of high computuational costs.

HNE via Approximate Personalized PageRank. Although the idea of using PPR as the proximity measure to be preserved in the embeddings is often mentioned in random-walk-based solutions [42, 65], these methods largely fail to scale due to numerous random walk simulations for the costly training process. A recent work STRAP [59] obtains better scalability by computing and factorizing a PPR-based proximity matrix instead. Specifically, STRAP builds node embeddings by factorizing the *transpose proximity matrix*, defined as $\mathbf{M} = \mathbf{\Pi} + \mathbf{\Pi}^{\top}$, where $\mathbf{\Pi}$ and $\mathbf{\Pi}^{\top}$ represent the approximate PPR matrices of the original graph G and its transpose graph (*i.e.*, obtained by reversing the direction of each edge in G), respectively.

The space and time complexities of STRAP are $O(\frac{n}{\delta})$ and $O(\frac{m}{\delta} + nk^2)$, respectively, where δ is the error threshold for PPR values. In the literature of approximate PPR processing (*e.g.*, [38, 48, 50, 51, 53]), δ is commonly set to $\frac{1}{n}$, which would lead to prohibitively high space (*i.e.*, $O(n^2)$) and time (*i.e.*, $O(mn + nk^2)$) costs in STRAP. Instead, in [59], the authors fix δ to a constant 10^{-5} and only retain PPR values greater than $\frac{\delta}{2}$, which compromises result utility. Even so, STRAP is still far more expensive than the proposed solution NRP, as shown in our experiments.

Further, as explained in Section 1, conventional PPR is not an ideal proximity measure for the purpose of HNE due to the former's relative nature; this problem propagates to STRAP which uses the PPR-based transpose proximity measure, *i.e.*, $\pi(u,v) + \pi(v,u)$ for each node pair $u, v \in G$. For instance, in the example of Table 1, we have $\pi(v_7, v_9) + \pi(v_9, v_7) = 0.288 > \pi(v_2, v_4) + \pi(v_4, v_2) = 0.236$, indicating that STRAP also tends to predict (v_9, v_7) over (v_2, v_4) in link prediction, which is counter-intuitive as we have explained in Section 1.

Other HNE methods. There also exist several techniques that generate embeddings without random walks, neural networks or matrix factorization. In particular, NetHiex [28] applies expectation maximization to learn embeddings that capture the neighborhood structure of each node, as well as the latent hierarchical taxonomy in the graph. RaRE [17] considers both the proximity and popularity of nodes, and derive embeddings by maximizing a posteriori estimation using stochastic gradient descent. GraphWave [14] represents each node's neighborhood via a low-dimensional embedding by leveraging heat wavelet diffusion patterns, so as to capture structural roles of nodes in networks. node2hash [47] transplants the feature hashing technique for word embeddings to embed nodes in networks. A common problem with the above methods is that they do not aim to preserve proximity information between nodes; consequently, they are generally less effective for tasks such as link prediction and graph reconstruction, as demonstrated in our experiments in Section 5.

Table 2: Frequently used notations.

Notation	Description
$G=(V,E)$	A graph G with node set V and edge set E
n,m	The numbers of nodes and edges in G, respectively
$d_{in}(v_i)$	The in-degree of node v_i
$d_{out}(v_i)$	The out-degree of node v_i
$\mathbf{A},\mathbf{D},\mathbf{P}$	The adjacency, out-degree and transition matrices of G
α	The random walk decay factor
k	The dimensionality of the embedding vectors
$\mathbf{X},\mathbf{Y}$	The forward and backward embeddings, respectively
$\overrightarrow{w}_v, \overleftarrow{w}_v$	The forward and backward weights for v's forward and backward embeddings, respectively

3. SCALABLE PPR COMPUTATION AND FACTORIZATION

This section presents ApproxPPR, a simple and effective baseline approach to HNE that obtains node embeddings through factorizing a conceptual approximate PPR proximity matrix. Unlike previous methods, ApproxPPR scales to billion-edge graphs without seriously compromising result quality. ApproxPPR forms the foundation of the our main proposal NRP, presented in Section 4. In what follows, Section 3.1 overviews ApproxPPR and formally defines the main concepts. Section 3.2 presents the main contribution in ApproxPPR: a scalable approximate PPR factorization algorithm. Table 2 summarizes frequent notations used throughout the paper.

3.1 Overview

As mentioned in Section 1, given an input graph $G = (V, E)$, the goal of HNE is to construct a size-k embedding for each node $v \in G$, where k is a user-specified per-node space budget. The input graph G can be either directed or undirected. For simplicity, in the following we assume that G is directed; for an undirected graph, we simply replace each undirected edge (u, v) with two directed ones with opposing direction, i.e., (u, v) and (v, u). Note that the capability to handle directed graphs is an advantage of our methods, compared to existing solutions that are limited to undirected graphs, e.g., [28, 33, 62–64].

In a directed graph, each node plays two roles: as the incoming end and outgoing end of edges, respectively. These two roles can have very different semantics. For instance, in a social graph, a user can deliberately choose to *follow* users who he/she is interested in, and is *followed* by users who are interested in him/her. The follower relationships and followee relationships of the same user should have different representations. This motivates building two separate embedding vectors $\mathbf{X}_v$ and $\mathbf{Y}_v$ for each node v, referred to as the *forward* and *backward* embeddings of v, respectively. In our solutions, we assign equal space budget (i.e., $\frac{k}{2}$) to $\mathbf{X}_v$ and $\mathbf{Y}_v$.

One advantage of ApproxPPR is that, it uses the PPR proximity matrix to do the factorization, without actually matrializing the matrix. Specifically, the definition of PPR is based on random walks, as follows. Suppose that we start a random walk from a source node u. At each step, we terminate the walk with probability α, and continue the walk (i.e., moving on to a random out-neighbor of u) with probability $1 - \alpha$. Then, for each node $v \in G$, we define its

PPR $\pi(u, v)$ with respect to source node u as the probability that the walk originating from u terminates at v.

Formally, let $\mathbf{\Pi}$ be an $n \times n$ matrix where $\mathbf{\Pi}[i, j] = \pi(v_i, v_j)$ for the i-th node v_i and j-th node v_j in G, and $\mathbf{P}$ be the probability transition matrix of G, i.e., $\mathbf{P}[i, j] = \frac{1}{d_{out}(v_i)}$, where v_j is an out-neighbor of v_i and $d_{out}(v_i)$ denotes the out-degree of v_i. Then,

$$\mathbf{\Pi} = \sum_{i=0}^{\infty} \alpha(1-\alpha)^i \cdot \mathbf{P}^i. \qquad (1)$$

ApproxPPR directly uses $\mathbf{\Pi}$ as the proximity matrix, i.e., $\mathbf{M} = \mathbf{\Pi}$. The goal is then to factorize $\mathbf{\Pi}$ into the forward and backward embeddings of nodes of the input graph G, such that for each pair of nodes u and v, i.e.:

$$\mathbf{X}_u \mathbf{Y}_v^\top \approx \pi(u, v) \qquad (2)$$

Remark. Note that directly computing $\mathbf{\Pi}$ (and subsequently factorizing it into the node embeddings $\mathbf{X}$ and $\mathbf{Y}$) is infeasible for a large graph. In particular, $\mathbf{\Pi}$ is a dense matrix that requires $O(n^2)$ space for n nodes, and Eq. (1) involves summing up an infinite series. To alleviate this problem, we could apply an approximate PPR algorithm [49, 51, 53] to compute the top-L largest PPR values for each node in G, which reduces the space overhead to $O(nL)$. Unfortunately, even the state-of-the-art approximate top-L PPR algorithm, i.e., TopPPR, is insufficient for our purpose. Specifically, TopPPR takes $O\left(\frac{L^{\frac{1}{4}}n^{\frac{3}{4}}\log n}{\sqrt{gap_\rho}}\right)$ time to compute the approximate top-L PPR values for each node, where $gap_\rho \leq 1$ is a parameter that quantifies the difference between the top-L and non-top-L PPR values [53]. To approximate the entire $\mathbf{\Pi}$, we would need to invoke TopPPR for every node, which incurs time super-quadratic to n. Empirically, Ref. [53] reports that running TopPPR on a billion-edge Twitter graph (used in our experiments as well) takes about 15 seconds CPU time, for $L = 500$. The same graph contains over 41 million nodes, meaning that running TopPPR on each of them would cost over 19 years of CPU time, which is infeasible even for a powerful computing cluster. While it is theoretically possible to reduce computational costs by choosing a small L and/or a large error threshold in TopPPR, doing so would lead to numerous zeros in $\mathbf{\Pi}$, which seriously degrades the result quality. We address this challenge in the next subsection with a simple and effective solution.

3.2 PPR Approximation

Observe that our goal is to obtain the node embeddings $\mathbf{X}$ and $\mathbf{Y}$, rather than the PPR matrix $\mathbf{\Pi}$ itself. Thus, the main idea of ApproxPPR is to integrate the computation and factorization of $\mathbf{\Pi}$ in the same iterative algorithm. Specifically, according to Eq. (1), $\mathbf{\Pi}$ can be viewed as the weighted sum of proximity values of different orders, i.e., one-hop proxmity, two-hop proximity, etc. Therefore, instead of first computing $\mathbf{\Pi}$ and then factorizing this *dense* matrix into node embeddings, we can instead start by factorizing the *sparse* first-order proximity matrix (i.e., $\mathbf{P}$) into the initial embeddings $\mathbf{X}$ and $\mathbf{Y}$, and then iteratively refine $\mathbf{X}$ and $\mathbf{Y}$, thereby incorporating higher-order information into them. This allows us to avoid the substantial space and computational overheads incurred for the construction and factorization of the $n \times n$ dense matrix $\mathbf{\Pi}$.

First, we consider a *truncated* version of $\mathbf{\Pi}$ as follows:

$$\mathbf{\Pi}' = \sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \cdot \mathbf{P}^i, \qquad (3)$$

where ℓ_1 is a relative large constant (*e.g.*, $\ell_1 = 20$). In other words, we set

$$\mathbf{\Pi}' = \mathbf{\Pi} - \alpha\mathbf{I} - \left(\sum_{i=\ell_1+1}^{+\infty} \alpha(1-\alpha)^i \cdot \mathbf{P}^i\right),$$

where $\mathbf{I}$ denotes an $n \times n$ identity matrix. The rationale is that when i is sufficiently large, $\alpha(1-\alpha)^i$ is small, in which case $\sum_{i=\ell_1+1}^{+\infty} \alpha(1-\alpha)^i \cdot \mathbf{P}^i$ becomes negligible. In addition, $\alpha\mathbf{I}$ only affects the PPR $\pi(u,u)$ from each node u to itself, which has no impact on our objective in Eq. (2) since we only concern the PPR values between different nodes.

To decompose $\mathbf{\Pi}'$, observe that

$$\mathbf{\Pi}' = \left(\sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \cdot \mathbf{P}^{i-1}\right) \mathbf{D}^{-1}\mathbf{A},$$

where $\mathbf{A}$ is the adjacency matrix of G, and $\mathbf{D}$ is an $n \times n$ diagonal matrix where the i-th diagonal element is $d_{out}(v_i)$. Instead of applying exact singular value decomposition (SVD) that is very time consuming, we factorize $\mathbf{A}$ using the BKSVD algorithm in [29] for randomized SVD, obtaining two $n \times k'$ matrices $\mathbf{U}, \mathbf{V}$ and a $k' \times k'$ diagonal matrix $\mathbf{\Sigma}$ given inputs $\mathbf{A}$ and k', such that $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \approx \mathbf{A}$. In short, BKSVD reduces $\mathbf{A}$ to a low-dimensional space by the Gaussian random projection and then performs SVD on the low-dimensional matrix. Given a relative error threshold ϵ, BKSVD guarantees a $(1+\epsilon)$ error bound for spectral norm low-rank approximation, which is much tighter than the theoretical accuracy bounds provided by previous truncated SVD algorithms [10, 19, 37].

Given the output $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$ from BKSVD, we set

$$\mathbf{X}_1 = \mathbf{D}^{-1}\mathbf{U}\sqrt{\mathbf{\Sigma}} \text{ and } \mathbf{Y} = \mathbf{V}\sqrt{\mathbf{\Sigma}}.$$

After that, we compute

$$\mathbf{X}_i = (1-\alpha)\mathbf{P}\mathbf{X}_{i-1} + \mathbf{X}_1 \text{ for } i = 2, \ldots, \ell_1,$$

and set $\mathbf{X} = \alpha(1-\alpha)\mathbf{X}_{\ell_1}$. This results in

$$\mathbf{X} = \sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \mathbf{P}^{i-1}\mathbf{X}_1 \text{ and}$$
$$\mathbf{X}\mathbf{Y}^\top = \sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \mathbf{P}^{i-1} \cdot \mathbf{X}_1\mathbf{Y}^\top$$

Note that $\mathbf{X}_1\mathbf{Y}^\top \approx \mathbf{D}^{-1}\mathbf{A} = \mathbf{P}$. It can be verified that $\mathbf{X}\mathbf{Y}^\top \approx \mathbf{\Pi}'$. Particularly, the following theorem establishes the accuracy guarantees of ApproxPPR.

THEOREM 1. *Given $\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}$, the dimensionality k', the random walk decay factor α, the number of iterations ℓ_1 and error threshold ϵ for BKSVD as inputs to Algorithm 1, it returns embedding matrices $\mathbf{X}$ and $\mathbf{Y}$ ($\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n \times k'}$) that satisfy, for every pair of nodes $(u,v) \in V \times V$ with $u \neq v$,*

$$\left|\mathbf{\Pi}[u,v] - (\mathbf{X}\mathbf{Y}^\top)[u,v]\right|$$
$$\leq (1+\epsilon)\sigma_{k'+1}(1-\alpha)(1-(1-\alpha)^{\ell_1}) + (1-\alpha)^{\ell_1+1},$$

and for every node $u \in V$,

$$\sum_{v \in V} \left|\mathbf{\Pi}[u,v] - (\mathbf{X}\mathbf{Y}^\top)[u,v]\right|$$
$$\leq \sqrt{n}(1+\epsilon)\sigma_{k'+1}(1-\alpha)(1-(1-\alpha)^{\ell_1}) + (1-\alpha)^{\ell_1+1},$$

where $\sigma_{k'+1}$ is the $(k'+1)$-th largest singular value of $\mathbf{A}$.

Algorithm 1: ApproxPPR

Input: $\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}, \alpha, k', \ell_1, \epsilon$.
Output: $\mathbf{X}, \mathbf{Y}$.
1   $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}] \leftarrow \mathsf{BKSVD}(\mathbf{A}, k', \epsilon)$;
2   $\mathbf{X}_1 \leftarrow \mathbf{D}^{-1}\mathbf{U}\sqrt{\mathbf{\Sigma}}, \quad \mathbf{Y} \leftarrow \mathbf{V}\sqrt{\mathbf{\Sigma}}$;
3   **for** $i \leftarrow 2$ *to* ℓ_1 **do**
4     $\lfloor \; \mathbf{X}_i \leftarrow (1-\alpha)\mathbf{P}\mathbf{X}_{i-1} + \mathbf{X}_1$;
5   $\mathbf{X} \leftarrow \alpha(1-\alpha)\mathbf{X}_{\ell_1}$;
6   **return** $\mathbf{X}, \mathbf{Y}$;

PROOF. *See our technical report [58] for the proof.* $\square$

Theorem 1 indicates that the PPR value between any pair of nodes preserved in the embedding vectors $\mathbf{X}$ and $\mathbf{Y}$ has absolute error at most $(1+\epsilon)\sigma_{k'+1}(1-\alpha)(1-(1-\alpha)^{\ell_1}) + (1-\alpha)^{\ell_1+1}$ and average absolute error of $\frac{1}{\sqrt{n}}(1+\epsilon)\sigma_{k'+1}(1-\alpha)(1-(1-\alpha)^{\ell_1}) + \frac{1}{n}(1-\alpha)^{\ell_1+1}$. Observe that the accuracy of the preserved PPR is restricted by ϵ and $\sigma_{k'+1}$, namely the accuracy of the low-rank approximation, *i.e.*, BKSVD.

Finally, we use $\mathbf{X}_v$ and $\mathbf{Y}_v$ as the initial forward and backward embeddings, respectively, for each node v. Algorithm 1 summarizes the pseudo-code for this construction of $\mathbf{X}$ and $\mathbf{Y}$. Next, we present a concrete example.

EXAMPLE 1. *Given input graph G in Fig. 1 and input parameters $k' = 2$, $\alpha = 0.15, \ell_1 = 20$, we run Algorithm 1 on G. It first applies BKSVD on the adjacency matrix $\mathbf{A} \in \mathbb{R}^{9 \times 9}$, which produces $\mathbf{X}_1 \in \mathbb{R}^{9 \times 2}$ and $\mathbf{Y} \in \mathbb{R}^{9 \times 2}$ as shown in Fig. 2.*

ApproxPPR *first sets $\mathbf{X} = \mathbf{X}_1$. Then, in each of the following iterations, the algorithm updates $\mathbf{X}$ to $0.85 \cdot \mathbf{P}\mathbf{X} + \mathbf{X}_1$. After repeating this process for $\ell_1 - 1 = 19$ iterations,* ApproxPPR *scales $\mathbf{X}$ by the weight $\alpha(1-\alpha) = 0.1275$ and returns us $\mathbf{X}$ and $\mathbf{Y}$ as in Fig. 2. The inner product between $\mathbf{X}_u$ and $\mathbf{Y}_v$ approximates $\pi(u,v)$. For example, consider node pairs $\langle v_2, v_4 \rangle$ and $\langle v_9, v_7 \rangle$:*

$$\mathbf{X}_{v_2}\mathbf{Y}_{v_4}^\top = [-0.18, 0.004] \cdot [-0.668, -0.359]^\top = 0.119,$$
$$\mathbf{X}_{v_9}\mathbf{Y}_{v_7}^\top = [-0.157, 0.236] \cdot [-0.105, 0.633]^\top = 0.166,$$

which are close to $\pi(v_2, v_4)$ and $\pi(v_9, v_7)$ in Table 1 respectively. $\square$

Time Complexity. By the analysis in Ref. [29], applying BKSVD on $\mathbf{A}$ requires $O\left((mk' + nk'^2)\frac{\log n}{\epsilon}\right)$ time, where ϵ is a constant that controls the tradeoff between the efficiency and accuracy of SVD. In addition, Lines 2, 4, and 5 in Algorithm 1 respectively run in $O(mk')$ time. Therefore, the overall time complexity of Algorithm 1 is

$$O\left(\left(\frac{\log n}{\epsilon} + \ell_1\right)mk' + \frac{\log n}{\epsilon}nk'^2\right),$$

which equals $O\left(k(m + kn)\log n\right)$ when ϵ and ℓ_1 are regarded as constants.

4. PROPOSED NRP ALGORITHM

The ApproxPPR algorithm presented in the previous section directly uses PPR as the proximity measure. However, as explained in Section 1, PPR by itself is not suitable for our purpose since it is a local measure, in the sense that PPR

$$\mathbf{Y} = \begin{bmatrix} \mathbf{Y}_{v_1} \\ \mathbf{Y}_{v_2} \\ \mathbf{Y}_{v_3} \\ \mathbf{Y}_{v_4} \\ \mathbf{Y}_{v_5} \\ \mathbf{Y}_{v_6} \\ \mathbf{Y}_{v_7} \\ \mathbf{Y}_{v_8} \\ \mathbf{Y}_{v_9} \end{bmatrix} = \begin{bmatrix} -0.652, & 0.243 \\ -0.668, & -0.359 \\ -0.823, & -0.142 \\ -0.668, & -0.359 \\ -0.737, & 0.547 \\ -0.314, & -0.42 \\ -0.105, & 0.633 \\ -0.094, & -0.225 \\ -0.071, & 0.818 \end{bmatrix}, \; \mathbf{X}_1 = \begin{bmatrix} -0.217, & -0.121 \\ -0.223, & 0.091 \\ -0.206, & 0.008 \\ -0.223, & 0.091 \\ -0.184, & -0.13 \\ -0.157, & 0.4 \\ -0.083, & -0.16 \\ -0.047, & 0.481 \\ -0.032, & -0.034 \end{bmatrix}, \cdots, \mathbf{X} = \begin{bmatrix} \mathbf{X}_{v_1} \\ \mathbf{X}_{v_2} \\ \mathbf{X}_{v_3} \\ \mathbf{X}_{v_4} \\ \mathbf{X}_{v_5} \\ \mathbf{X}_{v_6} \\ \mathbf{X}_{v_7} \\ \mathbf{X}_{v_8} \\ \mathbf{X}_{v_9} \end{bmatrix} = \begin{bmatrix} -0.182, & -0.014 \\ -0.18, & 0.004 \\ -0.14, & -0.002 \\ -0.18, & 0.004 \\ -0.13, & -0.008 \\ -0.182, & 0.075 \\ -0.126, & 0.072 \\ -0.092, & 0.141 \\ -0.157, & 0.236 \end{bmatrix}$$

Figure 2: Illustration of Example 1 for the ApproxPPR algorithm.

values are *relative* with respect to the source node. Consequently, PPR values for different source nodes are essentially incomparable, which is the root cause of the counterintuitive observation in the example of Fig. 1 and Table 1.

In the proposed algorithm NRP, we address the deficiency of PPR through a technique that we call *node reweighting*. Specifically, for any two nodes u and v, we aim to find forward and backward embeddings such that:

$$\mathbf{X}_u \mathbf{Y}_v^\top \approx \overrightarrow{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v \qquad (4)$$

where $\pi(u, v)$ is the PPR value of v with respect to node u as source, $\overrightarrow{w}_u$ and $\overleftarrow{w}_v$ are weights assigned to u and v, respectively. In other words, we let $\mathbf{X}_u^\top \mathbf{Y}_v$ preserve a scaled version of $\pi(u, v)$. The goal of NRP is then to find approximate node weights so that Eq. (4) properly expresses the proximity between nodes u and v. In NRP, the node weights are learned through an efficient optimization algorithm, described later in this section. The proposed node reweighting overcomes the deficiency of PPR, which is confirmed in our experiments.

In the following, Section 4.1 explains the choice of node weights in NRP. Sections 4.2 and 4.3 elaborate on the computation of node weights. Section 4.4 summarizes the complete NRP algorithm.

4.1 Choice of Node Weights

As discussed before, the problem of PPR as a proximity measure is that it is a relative measure with respect to the source node. In particular, the PPR value does not take into account the number of out-going and in-coming edges that each node has. To address this issue, NRP assigns to each node u a *forward weight* $\overrightarrow{w}_u$ and a *backward weight* $\overleftarrow{w}_u$, and uses $\overrightarrow{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v$ instead of $\pi(u, v)$ to gauge the strength of connection from u to v, as in Eq. (4). To compensate for the lack of node degree data in PPR values, we choose the forward and backward weights such that

$$\forall u \in V, \sum_{\forall v \in V \setminus u} (\overrightarrow{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v) \approx d_{out}(u), \text{ and}$$
$$\forall v \in V, \sum_{\forall u \in V \setminus v} (\overrightarrow{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v) \approx d_{in}(v). \qquad (5)$$

In other words, for any nodes $u, v \in G$, we aim to ensure that (i) the "total strength" of connections from u to other nodes is roughly equal to the out-degree $d_{out}(u)$ of u, and (ii) the total strength of connection from other nodes to v is roughly to equal the in-degree $d_{in}(v)$ of v. The rationale is that if u has a large out-degree, then it is more likely to be connected to other nodes, and hence, the proximity from u to other nodes should be scaled up accordingly. The case for a node v with a large in-degree is similar. In Section 5, we empirically show that this scaling approach significantly

improves the effectiveness of our embeddings for not just link prediction but also other important graph analysis task such as graph reconstruction.

4.2 Learning Node Weights

Given the output $\mathbf{X}$ and $\mathbf{Y}$ of ApproxPPR (Algorithm 1), we use $\mathbf{X}_v \mathbf{Y}_v^\top$ as an approximation of $\pi(u, v)$ for any two different nodes u and v. Then we formulate an objective function O for tuning node weights according to Eq. (5):

$$O = \min_{\overrightarrow{w}, \overleftarrow{w}} \sum_v \left\| \sum_{u \neq v} \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) - d_{in}(v) \right\|_2$$
$$+ \sum_u \left\| \sum_{v \neq u} \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) - d_{out}(u) \right\|_2 \qquad (6)$$
$$+ \lambda \sum_u \left(\|\overrightarrow{w}_u\|_2 + \|\overleftarrow{w}_u\|_2 \right),$$

subject to $\forall u \in V, \overrightarrow{w}_u, \overleftarrow{w}_u \geq \frac{1}{n}$.

To explain, recall that we use $\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v$ to quantify the strength of connection from u to v, and hence, for any fixed u (resp. v), $\sum_{u \neq v} \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right)$ measures the total strength of connections from u to other nodes (resp. from other nodes to v). Therefore, by minimizing O, we aim to ensure that the total strength of connections starting from (resp. ending at) each node u is close to u's out-degree (resp. in-degree), subject to a regularization term $\lambda \sum_u \left(\|\overrightarrow{w}_u\|_2 + \|\overleftarrow{w}_u\|_2 \right)$. In addition, we require that $\overrightarrow{w}_u, \overleftarrow{w}_u \geq \frac{1}{n}$ for all nodes u to avoid negative node weights.

We derive an approximate solution for Eq. (6) using coordinate descent [54]: We start with an initial solution $\overrightarrow{w}_v = d_{out}(v)$ and $\overleftarrow{w}_v = 1$ for each node v, and then iteratively update each weight based on the other $2n - 1$ weights. In particular, for any node v^*, the formula for updating $\overleftarrow{w}_{v^*}$ is derived by taking the partial derivative of the objective function in Eq. (6) with respect to $\overleftarrow{w}_{v^*}$:

$$\frac{\partial O}{\partial \overleftarrow{w}_{v^*}} = 2 \Big[\left(\left(\sum_{u \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top \right)^2 \overleftarrow{w}_{v^*}$$
$$- d_{in}(v^*) \left(\sum_{u \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top$$
$$+ \sum_u \left(\sum_{v \neq u, v \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top$$
$$+ \sum_{u \neq v^*} \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \right)^2 \overleftarrow{w}_{v^*}$$
$$- \left(\sum_u d_{out}(u) \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top + \lambda \overleftarrow{w}_{v^*} \Big]$$
$$= 2(a_3 - a_2 - a_1) + 2(b_1 + b_2 + \lambda) \overleftarrow{w}_{v^*},$$

where

$$a_1 = \left(\sum_u d_{out}(u) \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top,$$

$$a_2 = d_{in}(v^*) \left(\sum_{u \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top,$$

$$a_3 = \sum_u \left(\sum_{v \neq u, v \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top, \quad (7)$$

$$b_1 = \sum_{u \neq v^*} \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \right)^2,$$

$$b_2 = \left(\left(\sum_{u \neq v^*} \overrightarrow{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top \right)^2.$$

We identify the value of $\overleftarrow{w}_{v^*}$ that renders the above partial derivative zero, *i.e.*, $\frac{\partial O}{\partial \overleftarrow{w}_{v^*}} = 0$. If the identified $\overleftarrow{w}_{v^*}$ is smaller than $\frac{1}{n}$, then we set it to $\frac{1}{n}$ instead to avoid negativity. This leads to the following formula for updating backward weight $\overleftarrow{w}_{v^*}$:

$$\overleftarrow{w}_{v^*} = \max \left\{ \frac{1}{n}, \frac{a_1 + a_2 - a_3}{b_1 + b_2 + \lambda} \right\} \quad (8)$$

The formula for updating $\overrightarrow{w}_{u^*}$ is similar and included in our technical report [58] for brevity.

By Eq. (8), each update of $\overleftarrow{w}_{v^*}$ requires computing a_1, a_2, a_3, b_1 and b_2. Towards this end, a straightforward approach is to compute these variables directly based on their definitions in Eq. (7). This, however, leads to tremendous overheads. In particular, computing a_1, a_2, and b_2 requires a linear scan of $\mathbf{X}_u$ for each node u, which requires $O(nk')$ time. Deriving b_1 requires computing $\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top$ for each node u, which incurs $O(nk'^2)$ overhead. Furthermore, computing b_3 requires calculating $\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v$ for all $u \neq v \neq v^*$, which takes $O(n^2 k'^2)$ time. Therefore, each update of $\overleftarrow{w}_{v^*}$ takes $O(n^2 k'^2)$, which leads to a total overhead of $O(n^3 k'^2)$ for updating all $\overleftarrow{w}_{v^*}$ once. Apparently, this overhead is prohibitive for large graphs. To address this deficiency, in Section 4.3, we present a solution that reduces the overhead to $O(nk'^2)$ instead of $O(n^3 k'^2)$.

4.3 Accelerating Weight Updates

We observe that the updates of different node weights share a large amount of common computation. For example, for any node v^*, deriving a_1 always requires computing $\sum_u d_{out}(u) \overrightarrow{w}_u \mathbf{X}_u$. Intuitively, if we are able to reuse the result of such common computation for different nodes, then the overheads of our coordinate descent algorithm could be significantly reduced. In what follows, we elaborate how we exploit this idea to accelerate the derivation of $a_1, a_2, a_3, b_1,$ and b_2.

Computation of a_1, a_2, b_2. By the definitions of a_1, a_2, b_2 in Eq. (7),

$$a_1 = \boldsymbol{\xi} \mathbf{Y}_{v^*}^\top, \; a_2 = d_{in}(v^*)(\boldsymbol{\chi} - \overrightarrow{w}_{v^*} \mathbf{X}_{v^*}) \mathbf{Y}_{v^*}^\top,$$

$$\text{and } b_2 = \left((\boldsymbol{\chi} - \overrightarrow{w}_{v^*} \mathbf{X}_{v^*}) \mathbf{Y}_{v^*}^\top \right)^2, \quad (9)$$

$$\text{where } \boldsymbol{\xi} = \sum_u d_{out}(u) \overrightarrow{w}_u \mathbf{X}_u, \text{ and } \boldsymbol{\chi} = \sum_u \overrightarrow{w}_u \mathbf{X}_u.$$

Eq. (9) indicates that the a_1 values of all nodes $v^* \in V$ share a common $\boldsymbol{\xi}$, while a_2 and b_2 of each node v^* have $\boldsymbol{\chi}$ in common. Observe that both $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$ are independent of any backward weight. Motivated by this, we propose to first compute $\boldsymbol{\xi} \in \mathbb{R}^{1 \times k'}$ and $\boldsymbol{\chi} \in \mathbb{R}^{1 \times k'}$, which takes $O(nk')$ time. After that, we can easily derive $a_1, a_2,$ and b_2 for any node with precomputed $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$. In that case, each update

of a_1, a_2, and b_2 takes only $O(k')$ time, due to Eq. (9). This leads to $O(nk')$ (instead of $O(n^2 k')$) total computation time of a_1, a_2, and b_2 for all nodes.

Computation of a_3. Note that

$$a_3 = \sum_u \left(\sum_v \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top$$

$$- \sum_u \left(\overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \overleftarrow{w}_{v^*} \right) \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top$$

$$- \sum_v \left(\overrightarrow{w}_v \mathbf{X}_v \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \overrightarrow{w}_v \mathbf{X}_v \mathbf{Y}_{v^*}^\top$$

$$+ \left(\overrightarrow{w}_{v^*} \mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top \overleftarrow{w}_{v^*} \right) \overrightarrow{w}_{v^*} \mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top,$$

which can be rewritten as:

$$a_3 = \boldsymbol{\rho}_1 \boldsymbol{\Lambda} \mathbf{Y}_{v^*}^\top - \overleftarrow{w}_{v^*} \mathbf{Y}_{v^*} \boldsymbol{\Lambda} \mathbf{Y}_{v^*}^\top - \boldsymbol{\rho}_2 \mathbf{Y}_{v^*}^\top$$

$$+ \overleftarrow{w}_{v^*} \left(\mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top \right)^2 \overrightarrow{w}_{v^*}^2,$$

$$\text{where } \boldsymbol{\Lambda} = \sum_u \overrightarrow{w}_u^2 (\mathbf{X}_u^\top \mathbf{X}_u), \; \boldsymbol{\rho}_1 = \sum_v \overleftarrow{w}_v \mathbf{Y}_v, \quad (10)$$

$$\text{and } \boldsymbol{\rho}_2 = \sum_v \left(\overrightarrow{w}_v^2 \cdot \overleftarrow{w}_v \left(\mathbf{X}_v \mathbf{Y}_v^\top \right) \mathbf{X}_v \right).$$

Observe that $\boldsymbol{\Lambda}$ is independent of any backward weight. Thus, it can be computed once and reused in the computation of a_3 for all nodes. Meanwhile, both $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ are dependent on all of the backward weights, and hence, cannot be directly reused if we are to update each backward weight in turn. However, we note that $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be *incrementally* updated after the change of any single backward weight. Specifically, suppose that we have computed $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ based on Eq. (10), and then we change the backward weight of v^* from $\overleftarrow{w}'_{v^*}$ as $\overleftarrow{w}_{v^*}$. In that case, we can update $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ as:

$$\boldsymbol{\rho}_1 = \boldsymbol{\rho}_1 + \left(\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*} \right) \mathbf{Y}_{v^*},$$

$$\boldsymbol{\rho}_2 = \boldsymbol{\rho}_2 + \left(\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*} \right) \overrightarrow{w}_{v^*}^2 \left(\mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top \right) \mathbf{X}_{v^*}. \quad (11)$$

Since $\overleftarrow{w}_{v^*}, \overleftarrow{w}'_{v^*} \in \mathbb{R}$ and $\mathbf{X}_{v^*}, \mathbf{Y}_{v^*} \in \mathbb{R}^{1 \times k'}$, each of such updates takes only $O(k')$ time.

The initial values of $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be computed in $O(nk')$ time based on Eq. (10), while $\boldsymbol{\Lambda}$ can be calculated in $O(nk'^2)$ time. Given $\boldsymbol{\Lambda}$, $\boldsymbol{\rho}_1$, and $\boldsymbol{\rho}_2$, we can compute a_3 for any node v^* in $O(k'^2)$ time based on Eq. (10). Therefore, the total time required for computing a_3 for all nodes is $O(nk'^2)$, which is an significant reduction from the $O(n^3 k'^2)$ time required by the naive solution described in Section 4.2.

Approximation of b_1. We observe that the value of b_1 is insignificant compared to b_2. Thus, we propose to approximate its value instead of deriving it exactly, so as to reduce computation cost. By the inequality of arithmetic and geometric means, we have:

$$\tfrac{1}{k'} b_1 \leq \sum_{u \neq v^*} \overrightarrow{w}_u^2 (\sum_{r=1}^{k'} \mathbf{X}_u[r]^2 \mathbf{Y}_{v^*}[r]^2) \leq b_1. \quad (12)$$

Let $\boldsymbol{\phi}$ be a length-k' vector where the r-th ($r \in [1, k']$) element is

$$\boldsymbol{\phi}[r] = \sum_u \overrightarrow{w}_u^2 \mathbf{X}_u[r]^2. \quad (13)$$

We compute $\boldsymbol{\phi}$ in $O(nk')$ time, and then, based on Eq. (12), we approximate b_1 for each node in $O(k')$ time with

$$b_1 \approx \tfrac{k'}{2} \sum_{r=1}^{k'} \mathbf{Y}_{v^*}[r]^2 (\boldsymbol{\phi}[r] - \overrightarrow{w}_{v^*}^2 \mathbf{X}_{v^*}[r]^2). \quad (14)$$

Algorithm 2: updateBwdWeights

Input: $G, k', \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}, \mathbf{X}, \mathbf{Y}$.
Output: $\overleftarrow{\mathbf{w}}$
1   Compute $\boldsymbol{\xi}, \boldsymbol{\chi}, \boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \boldsymbol{\Lambda},$ and $\boldsymbol{\Phi}$ based on Eq. (9), (10), and (13);
2   **for** $r \leftarrow 1$ *to* k' **do**
3     $\lfloor \quad \boldsymbol{\phi}[r] = \sum_u \overrightarrow{w}_u^2 \mathbf{X}_u[r]^2;$
4   **for** $v^* \in V$ *in random order* **do**
5   Compute a_1, a_2, a_3, b_1, b_2 by Eq. (9), (10), and (14);
6     $\overleftarrow{w}'_{v^*} = \overleftarrow{w}_{v^*};$
7     $\overleftarrow{w}_{v^*} = \max\left\{\frac{1}{n}, \frac{a_1 + a_2 - a_3}{b_1 + b_2 + \lambda}\right\};$
8     $\boldsymbol{\rho_1} = \boldsymbol{\rho_1} + \left(\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}\right)\mathbf{Y}_{v^*};$
9     $\boldsymbol{\rho_2} = \boldsymbol{\rho_2} + \left(\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}\right)\overrightarrow{w}_{v^*}^2\left(\mathbf{X}_{v^*}\mathbf{Y}_{v^*}^\top\right)\mathbf{X}_{v^*}$
10 **return** $\overleftarrow{\mathbf{w}}$;

Algorithm 3: NRP

Input: Graph G, embedding dimensionality k, thresholds ℓ_1, ℓ_2, random walk decay factor α and error threshold ϵ
Output: Embedding matrices $\mathbf{X}$ and $\mathbf{Y}$.
1   $k' \leftarrow k/2;$
2   $[\mathbf{X}, \mathbf{Y}] \leftarrow \mathsf{ApproxPPR}(\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}, \alpha, k', \ell_1, \epsilon);$
3   **for** $v \in V$ **do**
4     $\lfloor \quad \overrightarrow{w}_v = d_{out}(v), \ \overleftarrow{w}_v = 1;$
5   **for** $l \leftarrow 1$ *to* ℓ_2 **do**
6     $\overleftarrow{\mathbf{w}} = \mathsf{updateBwdWeights}(G, k', \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}, \mathbf{X}, \mathbf{Y});$
7     $\overrightarrow{\mathbf{w}} = \mathsf{updateFwdWeights}(G, k', \overrightarrow{\mathbf{w}}, \overleftarrow{\mathbf{w}}, \mathbf{X}, \mathbf{Y});$
8   **for** $v \in V$ **do**
9     $\lfloor \quad \mathbf{X}_v = \overrightarrow{w}_v \cdot \mathbf{X}_v, \ \mathbf{Y}_v = \overleftarrow{w}_v \cdot \mathbf{Y}_v;$
10 **return** $\mathbf{X}, \mathbf{Y}$;

$$\boldsymbol{\xi} = [-8.1453, \ -7.6509], \ \boldsymbol{\chi} = [-3.5227, \ -3.2933],$$
$$\boldsymbol{\rho_1} = [-4.2126, \ -3.7234], \ \boldsymbol{\rho_2} = [-1.2659, \ -1.1678],$$
$$\boldsymbol{\Lambda} = \begin{bmatrix} 1.4478 & 1.3308 \\ 1.3308 & 1.2575 \end{bmatrix}, \ \boldsymbol{\phi} = [1.4478, \ 1.2575].$$

Figure 3: Illustration for Example 2

Therefore, the total cost for approximating b_1 for all nodes is $O(nk')$.

Summary. As a summary, Algorithm 2 presents the pseudocode of our method for updating the backward weight of each node. The algorithm first computes $\boldsymbol{\xi}, \boldsymbol{\chi}, \boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \boldsymbol{\Lambda}, \boldsymbol{\phi}$ in $O(nk'^2)$ time (Lines 1-3). After that, it examines each node's backward weight in random order, and computes a_1, a_2, a_3, b_1, b_2 by Eq. (9), (10), and (14), which takes $O\left(k'^2\right)$ time per node (Line 5). Given a_1, a_2, a_3, b_1, b_2, the algorithm updates the backward weight examined, and then updates $\boldsymbol{\rho_1}$ and $\boldsymbol{\rho_2}$ in $O(k')$ time (Lines 7-9). The total time complexity of Algorithm 2 is $O(nk'^2)$, which is significantly better than the $O(n^3k'^2)$-time method in Section 4.2. We illustrate Algorithm 2 with an example.

EXAMPLE 2. *Suppose that we invoke Algorithm 2 given graph G in Fig. 1, $k' = 2$, $\mathbf{X}$ and $\mathbf{Y}$ from Example 1, and the following $\overleftarrow{w}$ and $\overrightarrow{w}$:*

$$\overleftarrow{w} = [1,1,1,1,1,1,1,1,1], \quad \overrightarrow{w} = [3,3,4,3,4,2,2,2,1].$$

The algorithm first computes $\boldsymbol{\xi}, \boldsymbol{\chi}, \boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \boldsymbol{\Lambda}$ and $\boldsymbol{\phi}$ according to Eq. (9), (10), and (13). Fig. 3 shows the results.

Then, we update each backward weight in a random order with the above precomputed values. Let's pick $\overleftarrow{w}_{v_1}$ for the first update. According to Eq. (9), (10) and (14), we do not need to perform summations over all 9 nodes as in Eq. (7) but some multiplications between a 2×2 matrix and a length-2 vector, as well as inner products between length-2 vectors, yielding the following results fast:

$$a_1 = \boldsymbol{\xi}\mathbf{Y}_{v_1}^\top = 7.7968,$$
$$a_2 = 2(\boldsymbol{\chi} - 2\mathbf{X}_{v_1})\mathbf{Y}_{v_1}^\top = 5.903,$$
$$a_3 = \boldsymbol{\rho_1}\boldsymbol{\Lambda}\mathbf{Y}_{v_1}^\top - \mathbf{Y}_{v_1}\boldsymbol{\Lambda}\mathbf{Y}_{v_1}^\top - \boldsymbol{\rho_2}\mathbf{Y}_{v_1}^\top = 8.1324,$$
$$b_1 = \sum_{r=1}^{2}\mathbf{Y}_{v_1}[r]^2(\boldsymbol{\phi}[r] - \overrightarrow{w}_{v_1}^2\mathbf{X}_{v_1}[r]^2) = 0.9683,$$
$$b_2 = \left((\boldsymbol{\chi} - 2\mathbf{X}_{v_1})\mathbf{Y}_{v_1}^\top\right)^2 = 8.7113.$$

Let $\lambda = 0$. *The backward weight for v_1 is updated as*

$$\overleftarrow{w}_{v_1} = \max\left\{\frac{1}{9}, \frac{a_1 + a_2 - a_3}{b_1 + b_2}\right\} = 0.5752,$$

and then $\boldsymbol{\rho_1}$ and $\boldsymbol{\rho_2}$ are updated accordingly with the updated $\overleftarrow{w}_{v_1}$ based on Eq. (11) before proceeding to the next backward weight. $\qquad\square$

Remark. The forward weights $\overrightarrow{w}_{v^*}$ can be learned using an algorithm very similar to Algorithm 2, with the same space and time complexities. For brevity, we include the details in our technical report [58].

4.4   Complete NRP Algorithm and Analysis

Algorithm 3 presents the pseudo-code for constructing embeddings with NRP. Given a graph G, embedding dimensionality k, random walk decay factor α, thresholds ℓ_1, ℓ_2 and relative error threshold ϵ, it first generates the initial embedding matrices $\mathbf{X}$ and $\mathbf{Y}$ using Algorithm 1 (Lines 1-2, see Section 3.2 for details). After that, it initializes the forward and backward weights for each node (Lines 3-4) and then applies coordinate descent to refine the weights (Lines 5-7). In particular, in each epoch of the coordinate descent, it first invokes Algorithm 2 to update each backward weight once (Line 6), and then applies a similar algorithm to update each forward weight once (Line 7, see Algorithm updateFwdWeights in our technical report [58]). The total number of epochs is controlled by ℓ_2, which we set to $O(\log n)$ for efficiency. After the coordinate descent terminates, NRP multiplies the forward (resp. backward) embedding of each node by its forward (resp. backward) weight to obtain the final embeddings (Lines 8-9).

Complexity Analysis. NRP has three main steps: Algorithm 1, Algorithm 2, and Algorithm updateFwdWeights. By the analysis of time complexity in Section 3.2, Algorithm 1 runs in $O\left(k(m + kn)\log n\right)$ time, and its space overhead is determined by the number of non-zero entries in the matrices, which is $O(m + nk)$. For Algorithm 2 and Algorithm updateFwdWeights, each epoch takes $O(nk'^2)$ time, as analysed in Section 4.3. Hence, the time complexities of Algorithm 2 and Algorithm updateFwdWeights are both $O(nk'^2)$ when the number of epochs ℓ_2 is a constant. In addition, the space costs of Algorithm 2 and Algorithm updateFwdWeights depend on the size of $\boldsymbol{\xi}, \boldsymbol{\chi}, \boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \boldsymbol{\Lambda}, \boldsymbol{\phi}$ and the number of weights, which is bounded by $O(nk')$. As a result, the time

Table 3: Dataset statistics ($K = 10^3$, $M = 10^6$, $B = 10^9$).

Name	$|V|$	$|E|$	Type	#labels
Wiki	4.78K	184.81K	directed	40
BlogCatalog	10.31K	333.98K	undirected	39
Youtube	1.13M	2.99M	undirected	47
TWeibo	2.32M	50.65M	directed	100
Orkut	3.1M	234M	undirected	100
Twitter	41.6M	1.2B	directed	-
Friendster	65.6M	1.8B	undirected	-

complexity of Algorithm 3 is $O\left(k(m + kn)\log n\right)$ and its space complexity is $O(m + nk)$.

5. EXPERIMENTS

We experimentally evaluate our proposed method, *i.e.*, NRP, against 18 existing methods, including 4 classic ones and 14 recent ones, on three graph analysis tasks: link prediction, graph reconstruction, and node classification. We also study the efficiency of all methods and analyze the parameter choices of NRP. All experiments are conducted using a single thread on a Linux machine powered by an Intel Xeon(R) E5-2650 v2@2.60GHz CPU and 96GB RAM.

5.1 Experimental Settings

Datasets. We experiment with seven real networks that are used in previous work [16, 30, 42], including two billion-edge networks: directed network *Twitter* [24] and undirected network *Friendster* [57]. Table 3 shows the dataset statistics. For *Wiki, BlogCatalog, Youtube*, and *Orkut*, we use the node labels suggested in previous work [16, 31, 42]. For *TWeibo*, we collect its node tags from [21] and only keep the top 100 tags in the network, following the practice in [42].

Competitors. We evaluate NRP against eighteen existing methods, including four classic methods (*i.e.*, DeepWalk, node2vec, LINE and DNGR) and fourteen recent methods, many of which have not been compared against each other in previous work. To our knowledge, we are the first to systematically evaluate such a large number of existing network embedding techniques. We categorize the eighteen existing methods into four groups as follows:

1. factorization-based methods: AROPE [64], RandNE [63], NetSMF [33], ProNE [62], and STRAP [59];

2. random-walk-based methods: DeepWalk [31], LINE [39], node2vec [16], PBG [26], APP [65], and VERSE [42];

3. neural-network-based methods: DNGR [7], DRNE [43], GraphGAN [45], and GA [1];

4. other methods: RaRE [18], NetHiex [28] and GraphWave [14].

Parameter Settings. For NRP, we set $\ell_1 = 20, \ell_2 = 10, \alpha = 0.15, \epsilon = 0.2$, and $\lambda = 10$. Note that $\ell_1 = 20$ means up to 20-order proximities can be preserved in the embeddings, and most forward and backward weights converge with $\ell_2 = 10$ epochs. For fair comparison, the random walk decay factor, α, is set to 0.15, in all PPR-based methods, including VERSE, APP, STRAP, and NRP. We use the default parameter settings of all competitors as suggested in
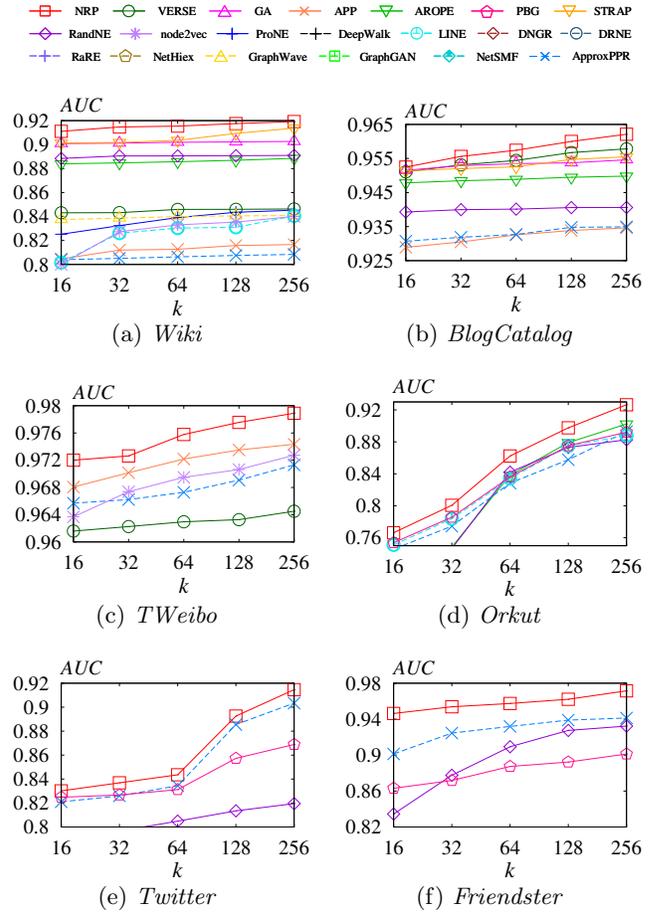

Figure 4: Link prediction results vs. embedding dimensionality k (best viewed in color).

their papers. For instance, the error threshold δ in STRAP is set to 10^{-5} as suggested in [59]. We obtain the source codes of all competitors from their respective authors. Unless otherwise specified, we set the embedding dimensionality k of each method to 128.

Note that AROPE, RandNE, NetHiex, GraphWave, NetSMF and ProNE are designed for undirected graphs only. For a thorough evaluation, we still report their performance on the directed graphs, *i.e.*, *Wiki*, *TWeibo*, and *Twitter*, by omitting the direction of each edge when feeding the graphs as input to these methods.

In addition, the following methods are designed for some specific tasks: *e.g.*, GraphWave and DRNE for structural role discovery; node2vec, DeepWalk, LINE, DNGR, NetSMF and ProNE for node classification or network visualization. For completeness, we evaluate all methods over three commonly used tasks, namely, link prediction, node classification, and graph reconstruction. We exclude a method if it cannot report results within 7 days.

5.2 Link Prediction

Link prediction aims to predict which node pairs are likely to form edges. Following previous work [64], we first remove 30% randomly selected edges from the input graph G, and then construct embeddings on the modified graph G'. After that, we form a testing set E_{test} consisting of (i) the node

pairs corresponding to the 30% removed edges, and (ii) an equal number of node pairs that are not connected by any edge in G. Note that on directed graphs, each node pair (u, v) is ordered, *i.e.*, we aim to predict whether there is a directed edge from u to v.

Given a method's embeddings, we compute a score for each node pair (u, v) in the testing set based on embedding vectors of u and v, and then evaluate the method's performance by the *Area Under Curve (AUC)* of the computed scores. Following their own settings, for AROPE, RandNE, NetHiex, NetSMF and ProNE, the score for (u, v) is computed as the inner product u and v's embedding vectors; for NRP, ApproxPPR, APP, GA, and STRAP, the score equals the inner product of u's forward vector and v's backward vector. For RaRE, we apply the probability function described in [17] for computing the score for (u, v). For DeepWalk, LINE, node2vec, DNGR, DRNE, GraphGAN, and GraphWave, we use the *edge features* approach [28]: (i) for each node pair (u, v) in G, concatenate u's and v's embeddings into a length-$2k$ vector; (ii) sample a training set of node pairs E'_{train} (with same size as E_{test}), such that half of the node pairs are from G' and the other half are node pairs not connected in G; (iii) feed the length-$2k$ vectors of node pairs in E'_{train} into a logistic regression classifier; (iv) then use the classifier to obtain the scores of node pairs in E_{test} for link prediction. For VERSE and PBG, the inner product approach only works for undirected graphs, since VERSE and PBG generate only one embedding vector per node, due to which the inner product approach cannot differentiate (u, v) from (v, u). Therefore, on directed graphs, we also employ the aforementioned edge features approach for VERSE and PBG.

Fig. 4 shows AUC of each method when k varies from 16 to 256. NRP consistently outperforms all competitors, by a significant margin of up to 3% on *Orkut* and *Friendster*, and a large margin of 0.5% to 2% on other graphs. Compared with the best competitor, *i.e.*, AROPE, NRP achieves a considerable gain of 1.9% on *Orkut* when $k = 128$. Note that NRP outperforms all the PPR-based competitors, including ApproxPPR, APP, VERSE and STRAP, over all datasets, which confirms the efficacy of our reweighting scheme in NRP, and validates our analysis of the conventional PPR deficiency in Section 1. Moreover, we observe that VERSE is worse on directed graphs, *i.e., Wiki* and *TWeibo*, although it is the best competitor on undirected graph *Blog-Catalog*. This is because that VERSE generates only one embedding vector per node, making it fail to capture the asymmetric transitivity (*i.e.,* direction of edges) in directed graphs [30, 65], which is critical for link prediction. Our method, NRP, instead generates two embedding vectors per node and successfully distinguishes the edge directions and thus is more promising. STRAP and GA cannot efficiently handle large graphs (*i.e., Youtube, TWeibo, Orkut, Twitter* and *Friendster*), since they require the materialization of a large $n \times n$ matrix, which is extremely costly in terms of both space and time; in contrast, NRP does not require to do so. NRP also consistently outperforms AROPE by about 2% absolute improvement on all graphs. For the other competitors, their performance is also less than satisfactory, as shown in the figures. In summary, for link prediction, NRP yields considerable performance improvements compared with the state-of-the-art methods, over graphs with various sizes.

5.3 Graph Reconstruction

Following previous work, for this task, we (i) take a set S of node pairs from the input graph G, (ii) compute the score of each pair using the same approach as in link prediction, and then (iii) examine the top-K node pairs to identify the fraction of them that correspond to the edges in G. This fraction is referred to as the *precision@K* of the method considered. On *Wiki* and *BlogCatalog*, we let S be the set of all possible node pairs. Meanwhile, on *Youtube* and *TWeibo*, following previous work [63, 64], we construct S by taking a 1% sample of the $\binom{n}{2}$ possible pairs of nodes. We exclude the results on *Orkut* and *Twitter* since 1% of all the possible node pairs from these two graphs are excessively large.

Fig. 5 shows the performance of all methods for graph reconstruction, varying K from 10 to 10^6. For readability, we split the results of each dataset into two sub-figures in vertical, and each sub-figure compares NRP against a subset of the competitors. NRP outperforms all competitors consistently on all datasets. NRP remains highly accurate when K increases to 10^4 or even 10^5, while the precisions of other methods, especially GA, AROPE, RandNE, APP, VERSE and STRAP, drop significantly. Specifically, NRP achieves at least 90% precision when K reaches 10^4 on *Wiki*, *Blogcatalog* and *TWeibo*, which means at least 10% absolute improvement over state-of-the-art methods. In addition, on *Youtube*, NRP achieves 2-8% absolute improvement over the best competitors, including VERSE. The superiority of NRP over the other PPR-based methods, *i.e.*, ApproxPPR, APP, VERSE and STRAP, in graph reconstruction demonstrates the power of our reweighting scheme. Meanwhile, the improvements over all other methods like GA, AROPE and RandNE implies that NRP accurately captures the structural information of the input graph via PPR.

5.4 Node Classification

Node classification aims to predict each node's label(s) based on its embeddings. Following previous work [42], we first construct network embeddings from the input graph G, and use the embeddings and labels of a random subset of the nodes to train a one-vs-all logistic regression classifier, after which we test the classifier with the embeddings and labels of the remaining nodes. In particular, for NRP, ApproxPPR, APP, GA, and STRAP, we first normalize the forward and backward vectors, respectively, of each node v, and then concatenate them as the feature representation of v before feeding it to the classifier. Note that the embeddings produced by NRP are weighted versions of that produced by ApproxPPR, and thus, they have the same feature representation for each node v after the normalization, for the task of node classification.

Fig. 6 shows the Micro-F1 score achieved by each method when the percentage of nodes used for training varies from 10% to 90% (*i.e.*, 0.1 to 0.9 in the figures). The Macro-F1 results are qualitatively similar and thus omitted for the interest of space. NRP consistently outperforms all competitors on *Wiki* and *TWeibo*, and has comparable performance to ProNE on *BlogCatalog* and *Youtube*. Specifically, on *Wiki*, NRP achieves an impressive improvement of at least 3% in Micro-F1 over existing methods and about 1% lead on *TWeibo*, which is considerable in contrast to that of our competitors. This demonstrates that NRP can accurately capture the graph structure via PPR. On *BlogCatalog* and *Youtube*, NRP, NetHiex, VERSE and ProNE all achieve com-
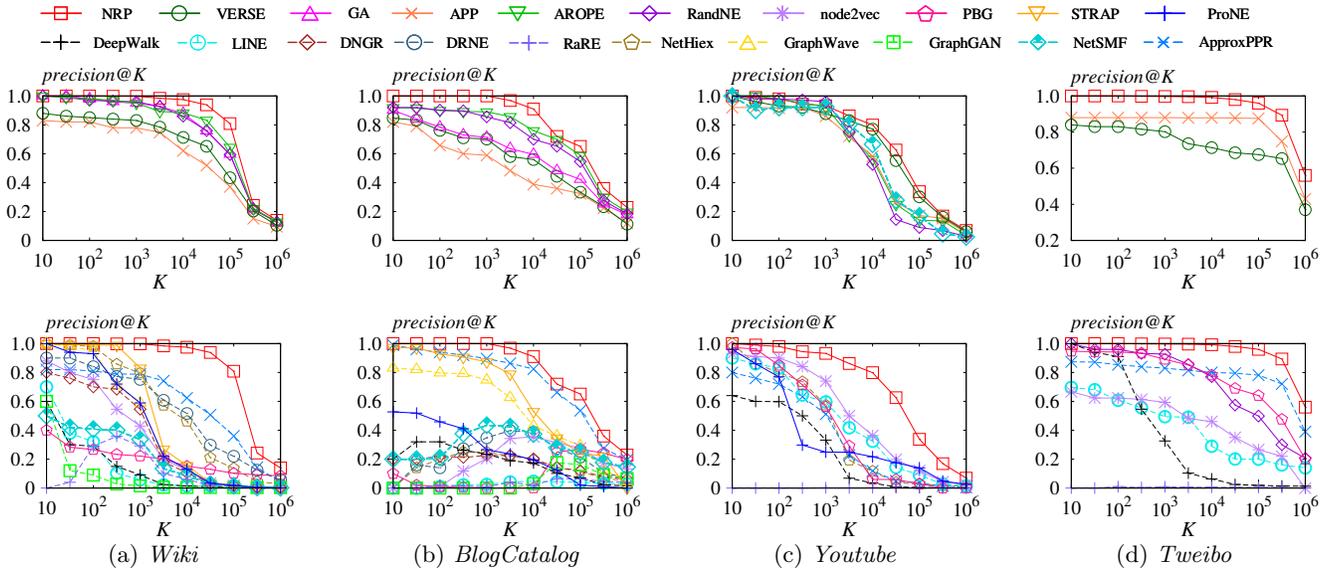
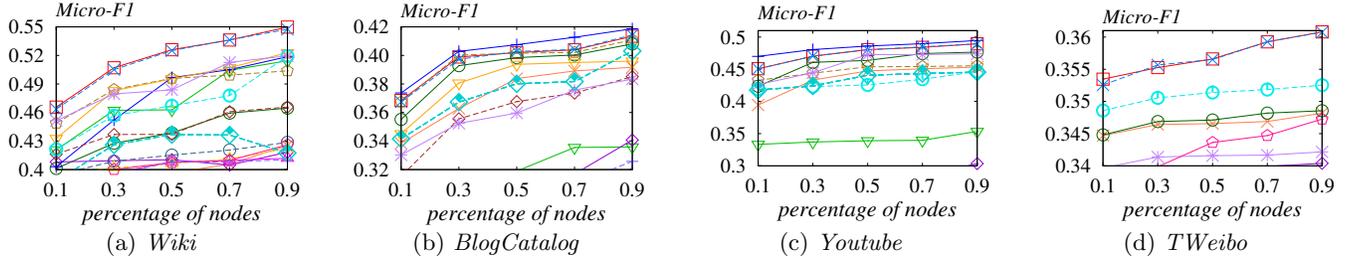Figure 5: Graph reconstruction results vs. K (best viewed in color).

Figure 6: Node classification results (best viewed in color).

parable performance. ProNE is slightly better than NRP, but note that ProNE can only handle undirected graphs and is specifically designed for node classification task by employing graph spectrum and graph partition techniques. NetHiex also requires the input graphs to be undirected. VERSE cannot achieve the same high-quality performance on directed graphs (Fig. 6a and 6d) as it does on undirected graphs (Fig. 6b and 6c). The reason is that VERSE only generates one embedding vector per node, and neglects the directions of edges in the directed graphs, while our method NRP can preserve the directions. Typically, NRP achieves consistent and outstanding performance for node classification task over all the real-world graphs.

5.5 Efficiency

Fig. 7 plots the time required by each method to construct embeddings, when k is varied from 16 to 256. Note that the Y-axis is in log-scale, and that the reported time excludes the overheads for loading datasets and outputting embeddings. We also omit any method with processing time exceeding 7 days. For a fair comparison, all methods are ran with a single thread.

Among all methods tested, NRP strikes the best balance between effectiveness and efficiency, and is up to 2 orders of magnitude faster than all methods except ApproxPPR, ProNE, RandNE and AROPE. In addition, as illustrated in Fig. 4, 5, and 6, RandNE and AROPE are both less effec-

tive compared to NRP for the three tasks. Furthermore, the results of RandNE, AROPE, and ProNE on directed graphs (*i.e.*, *Wiki*, *TWeibo* and *Twitter*) are all inferior to those of NRP, as shown in Fig. 4, 5, and 6. This is because RandNE, AROPE, and ProNE are specifically designed for undirected graphs instead of directed graphs. ProNE is also inferior to NRP in terms of link prediction and graph reconstruction on undirected graphs. Although ApproxPPR runs faster than NRP, it is less effective than the latter in terms of link prediction and graph reconstruction, due to the deficiency of conventional PPR discussed in 1. Neither GA nor STRAP is able to scale to large graphs, which again manifests the power of our scalable PPR computation. The remaining methods either rely on expensive training phases (*e.g.,* DeepWalk and VERSE), or require constructing a huge matrix (*e.g.,* NetSMF), thereby failing to handle large graphs efficiently as well.

5.6 Parameter Analysis

We study the effect of varying the parameters in NRP, including α, ϵ, ℓ_1 and ℓ_2, for link prediction on *Wiki*, *Blogcatalog* and *Youtube* datasets. Note that α is the decay factor in PPR (Eq. (1) in Section 3.1); ϵ is the error threshold of BKSVD used in our PPR approximation (Algorithm 1); ℓ_1 is the number of iterations for computing PPR (Algorithm 1); ℓ_2 is the number of epochs for reweighting node embeddings (Algorithm 3). The AUC results are shown in Fig. 8,
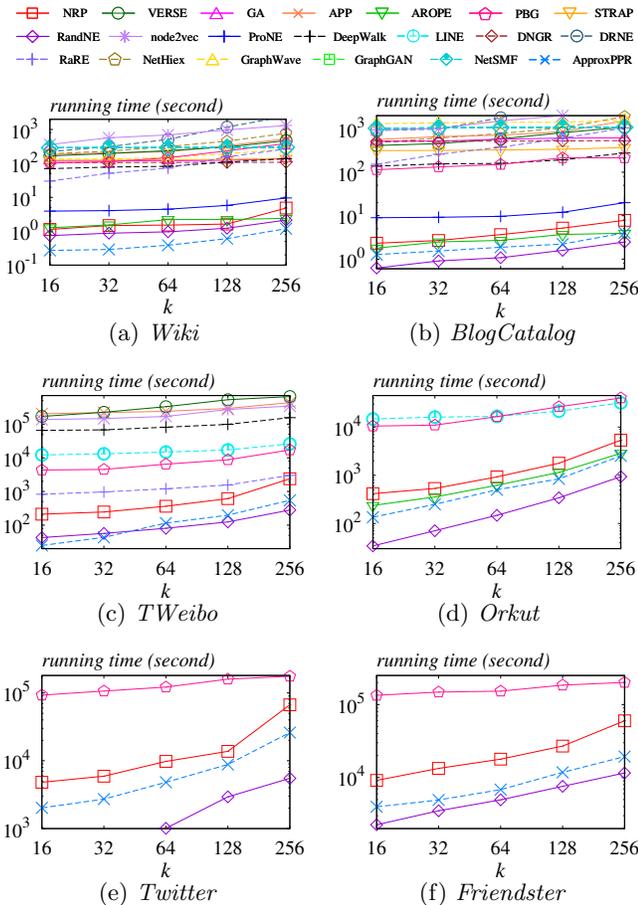
Figure 7: Running time vs. embedding dimensionality k (best viewed in color).

Figure 8: Link prediction results with varying parameters (best viewed in color).

when one of the parameters is varied, the others are kept as default values in Section 5.1.

Fig. 8a displays the AUC of NRP when we vary the decay factor α from 0.1 to 0.9. As α increases, the performance downgrades since only limited local neighborhoods of nodes are preserved and high-order proximities are failed to be captured in the embeddings, which is consistent with the observation in [42,59]. When $\alpha = 0.1$ or 0.2, the AUC score is the highest, which holds on all the three datasets. And thus our choice of $\alpha = 0.15$ makes sure that the best efficacy is achieved.

The AUC result of NRP when varying ϵ from 0.1 to 0.9 is depicted in Fig. 8b. According to Theorem 1, ϵ influences the accuracy of our PPR approximation. As shown in Fig. 8b, when ϵ is increased (*i.e.*, the error caused by BKSVD is larger), the AUC performance of the embedding decreases, especially on *Youtube* dataset. Therefore, we set ϵ to 0.2, which has the same excellent performance as 0.1 but is computationally more efficient.

In Fig. 8c, observe that the AUC of NRP grows significantly when we vary ℓ_1 from 1 to 15, and keeps stable and excellent for larger ℓ_1 from 15 to 40, which holds for all the three datasets. Recall that the accuracy of our PPR approximation is affected by ℓ_1 as well and when ℓ_1 increases, the approximate PPR scores are more accurate. According to Fig. 8c, our choice of $\ell_1 = 20$ is proper and robust.
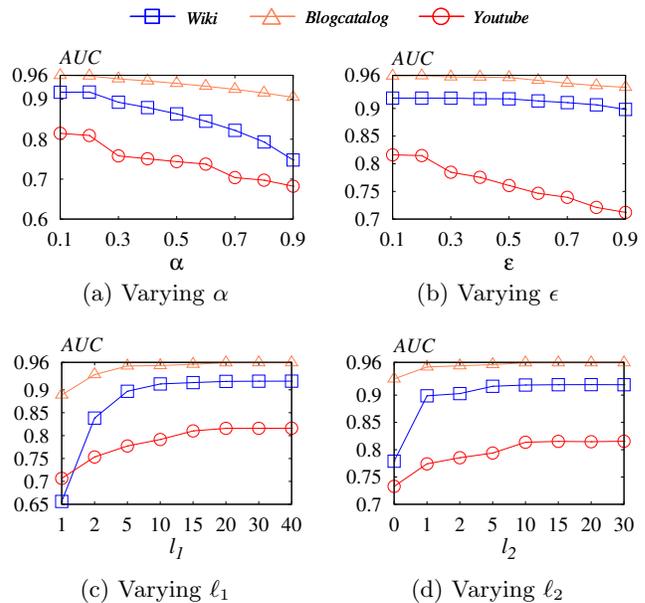
Fig. 8d shows the AUC of NRP when we vary ℓ_2 from 0 to 30. The AUC increases significantly when ℓ_2 is increased from 0 to 10, and then keeps stable for larger ℓ_2 values, which is consistent across the three datasets. When $\ell_2 = 0$, it is equivalent to disable our reweighting scheme and only use the conventional PPR for embedding, which is significantly inferior to the case that our NRP reweighting scheme is enabled, *e.g.*, when $\ell_2 = 10$. Specifically, on *Wiki*, the AUC is increased from 0.78 to 0.91 when ℓ_2 is varied from 0 to 10. This validates our insight about the drawback of vanilla PPR for embeddings and demonstrates the power of our proposed reweighting scheme. Further, Fig. 8d also shows that our reweighting scheme converges quickly when the epochs are increased.

6. CONCLUSION

This paper presents NRP, a novel, efficient and effective approach for homogeneous network embedding. NRP constructs embedding vectors based on personalized PageRank values and reweights each node's embedding vectors based on an objective function concerning the in-/out- degree of each node. We show that NRP runs in time almost linear to the size of the input graph, and that it requires less than four hours to process a graph with 1.2 billion edges. Extensive experiments on real data also demonstrate that NRP considerably outperforms the state of the arts in terms of the accuracy link prediction, graph reconstruction and on node classification tasks. As for future work, we plan to study how to extend NRP to handle attributed graphs.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] S. Abu-El-Haija, B. Perozzi, R. Al-Rfou, and A. A. Alemi. Watch your step: Learning node embeddings via graph attention. In *NeurIPS*, pages 9180–9190, 2018.

[2] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *WWW*, pages 37–48, 2013.

[3] L. Backstrom and J. Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.

[4] M. J. Brzozowski and D. M. Romero. Who should i follow? recommending people in directed social networks. In *Fifth International AAAI Conference on Weblogs and Social Media*, 2011.

[5] H. Cai, V. W. Zheng, and K. C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *TKDE*, 30(9):1616–1637, 2018.

[6] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *CIKM*, pages 891–900, 2015.

[7] S. Cao, W. Lu, and Q. Xu. Deep neural networks for learning graph representations. In *AAAI*, 2016.

[8] H. Chen, B. Perozzi, Y. Hu, and S. Skiena. HARP: hierarchical representation learning for networks. In *AAAI*, 2018.

[9] H. Chen, H. Yin, T. Chen, Q. V. H. Nguyen, W.-C. Peng, and X. Li. Exploiting centrality information with graph convolutions for network representation learning. In *ICDE*, pages 590–601, 2019.

[10] K. L. Clarkson and D. P. Woodruff. Low-rank approximation and regression in input sparsity time. *STOC*, pages 81–90, 2013.

[11] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *TKDE*, 31(5):833–852, 2018.

[12] Q. Dai, Q. Li, J. Tang, and D. Wang. Adversarial network embedding. In *AAAI*, 2018.

[13] Q. Dai, X. Shen, L. Zhang, Q. Li, and D. Wang. Adversarial training methods for network embedding. In *WWW*, pages 329–339, 2019.

[14] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec. Learning structural node embeddings via diffusion wavelets. In *KDD*, pages 1320–1329, 2018.

[15] H. Gao and H. Huang. Self-paced network embedding. In *KDD*, pages 1406–1415, 2018.

[16] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, pages 855–864, 2016.

[17] Y. Gu, Y. Sun, Y. Li, and Y. Yang. Rare: Social rank regulated large-scale network embedding. In *WWW*, pages 359–368, 2018.

[18] Y. Gu, Y. Sun, Y. Li, and Y. Yang. Rare: Social rank regulated large-scale network embedding. In *WWW*, pages 359–368, 2018.

[19] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[20] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[21] Kaggle, 2012. https://www.kaggle.com/c/kddcup2012-track1.

[22] T. N. Kipf and M. Welling. Variational graph auto-encoders. *NeurIPS Workshop*, 2016.

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

[24] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[25] Y.-A. Lai, C.-C. Hsu, W. Chen, M.-Y. Yeh, and S.-D. Lin. Prune: Preserving proximity and global ranking for network embedding. In *NeurIPS*, pages 5257–5266, 2017.

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

[27] X. Liu, T. Murata, K.-S. Kim, C. Kotarasu, and C. Zhuang. A general view for network embedding as matrix factorization. In *WSDM*, pages 375–383, 2019.

[28] J. Ma, P. Cui, X. Wang, and W. Zhu. Hierarchical taxonomy aware network embedding. In *KDD*, pages 1920–1929, 2018.

[29] C. Musco and C. Musco. Randomized block krylov methods for stronger and faster approximate singular value decomposition. In *NeurIPS*, pages 1396–1404, 2015.

[30] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *KDD*, pages 1105–1114, 2016.

[31] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *KDD*, pages 701–710, 2014.

[32] B. Perozzi, V. Kulkarni, H. Chen, and S. Skiena. Don't walk, skip!: Online learning of multi-scale network embeddings. In *ASONAM*, pages 258–265, 2017.

[33] J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J. Tang. Netsmf: Large-scale network embedding as sparse matrix factorization. In *WWW*, pages 1509–1520, 2019.

[34] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In *WSDM*, pages 459–467, 2018.

[35] P. Radivojac, W. T. Cark, T. R. Oron, A. M. Schnoes, T. Wittkop, A. Sokolov, K. Graim, C. Funk, K. Verspoor, and et. al. A large-scale evaluation of computational protein function prediction. *Nature methods*, 10(3):221, 2013.

[36] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo. struc2vec: Learning node representations from structural identity. In *KDD*, pages 385–394, 2017.

[37] T. Sarlos. Improved approximation algorithms for large matrices via random projections. In *FOCS*, pages 143–152, 2006.

[38] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang. Realtime top-k personalized pagerank over large graphs on gpus. *PVLDB*, 13(1):15–28, 2019.

[39] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.

[40] L. Tang and H. Liu. Leveraging social media networks for classification. *DMKD*, 23(3):447–478, 2011.

[41] R. Trivedi, B. Sisman, X. L. Dong, C. Faloutsos, J. Ma, and H. Zha. Linknbed: Multi-graph representation learning with entity linkage. In *ACL*, pages 252–262, 2018.

[42] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller. Verse: Versatile graph embeddings from similarity measures. In *WWW*, pages 539–548, 2018.

[43] K. Tu, P. Cui, X. Wang, P. S. Yu, and W. Zhu. Deep recursive network embedding with regular equivalence. In *KDD*, pages 2357–2366, 2018.

[44] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *KDD*, pages 1225–1234, 2016.

[45] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xing, and M. Guo. Graphgan: Graph representation learning with generative adversarial nets. In *AAAI*, 2018.

[46] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *KDD*, pages 839–848, 2018.

[47] Q. Wang, S. Wang, M. Gong, and Y. Wu. Feature hashing for network representation learning. In *IJCAI*, pages 2812–2818, 2018.

[48] R. Wang, S. Wang, and X. Zhou. Parallelizing approximate single-source personalized pagerank queries on shared memory. *VLDBJ*, 28(6):923–940, 2019.

[49] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li. Hubppr: Effective indexing for approximate personalized pagerank. *PVLDB*, 10(3):205–216, 2016.

[50] S. Wang, R. Yang, R. Wang, X. Xiao, Z. Wei, W. Lin, Y. Yang, and N. Tang. Efficient algorithms for approximate single-source personalized pagerank queries. *TODS*, 44(4):18, 2019.

[51] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang. FORA: simple and effective approximate single-source personalized pagerank. In *KDD*, pages 505–514, 2017.

[52] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang. Community preserving network embedding. In *AAAI*, 2017.

[53] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J. Wen. Topppr: Top-k personalized pagerank queries with precision guarantees on large graphs. In *SIGMOD*, pages 441–456, 2018.

[54] S. J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 2015.

[55] L. Y. Wu, A. Fisch, S. Chopra, K. Adams, A. Bordes, and J. Weston. Starspace: Embed all the things! In *AAAI*, 2018.

[56] C. Yang, M. Sun, Z. Liu, and C. Tu. Fast network embedding enhancement via high order proximity approximation. In *IJCAI*, pages 3894–3900, 2017.

[57] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *KAIS*, 42(1):181–213, 2015.

[58] R. Yang, J. Shi, X. Xiao, S. S. Bhowmick, and Y. Yang. Homogeneous network embedding for massive graphs via personalized pagerank. *arXiv preprint*, 2019.

[59] Y. Yin and Z. Wei. Scalable graph embeddings via sparse transpose proximities. In *KDD*, 2019.

[60] W. Yu, C. Zheng, W. Cheng, C. C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang. Learning deep network representations with adversarially regularized autoencoders. In *KDD*, pages 2663–2671, 2018.

[61] D. Zhang, J. Yin, X. Zhu, and C. Zhang. Network representation learning: A survey. *IEEE Trans. Big Data*, 2018.

[62] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding. Prone: Fast and scalable network representation learning. In *IJCAI*, pages 4278–4284, 2019.

[63] Z. Zhang, P. Cui, H. Li, X. Wang, and W. Zhu. Billion-scale network embedding with iterative random projection. In *ICDM*, pages 787–796, 2018.

[64] Z. Zhang, P. Cui, X. Wang, J. Pei, X. Yao, and W. Zhu. Arbitrary-order proximity preserved network embedding. In *KDD*, pages 2778–2786, 2018.

[65] C. Zhou, Y. Liu, X. Liu, Z. Liu, and J. Gao. Scalable graph embedding for asymmetric proximity. In *AAAI*, 2017.

[66] D. Zhu, P. Cui, D. Wang, and W. Zhu. Deep variational network embedding in wasserstein space. In *KDD*, pages 2827–2836, 2018.

[67] Z. Zhu, S. Xu, M. Qu, and J. Tang. Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *WWW*, pages 2494–2504, 2019.