Scaling Attributed Network Embedding to Massive Graphs

Renchi Yang
Nanyang Technological University
yang0461@ntu.edu.sg

Jieming Shi
Hong Kong Polytechnic University
jieming.shi@polyu.edu.hk

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

Yin Yang
Hamad bin Khalifa University
yyang@hbku.edu.qa

Juncheng Liu
National University of Singapore
juncheng@comp.nus.edu.sg

Sourav S. Bhowmick
Nanyang Technological University
assourav@ntu.edu.sg

ABSTRACT

Given a graph G where each node is associated with a set of attributes, *attributed network embedding* (*ANE*) maps each node $v \in G$ to a compact vector X_v, which can be used in downstream machine learning tasks. Ideally, X_v should capture node v's *affinity* to each attribute, which considers not only v's own attribute associations, but also those of its connected nodes along edges in G. It is challenging to obtain high-utility embeddings that enable accurate predictions; scaling effective ANE computation to massive graphs with millions of nodes pushes the difficulty of the problem to a whole new level. Existing solutions largely fail on such graphs, leading to prohibitive costs, low-quality embeddings, or both.

This paper proposes PANE, an effective and scalable approach to ANE computation for massive graphs that achieves state-of-the-art result quality on multiple benchmark datasets, measured by the accuracy of three common prediction tasks: attribute inference, link prediction, and node classification. In particular, for the large *MAG* data with over 59 million nodes, 0.98 billion edges, and 2000 attributes, PANE is the only known viable solution that obtains effective embeddings on a single server, within 12 hours.

PANE obtains high scalability and effectiveness through three main algorithmic designs. First, it formulates the learning objective based on a novel random walk model for attributed networks. The resulting optimization task is still challenging on large graphs. Second, PANE includes a highly efficient solver for the above optimization problem, whose key module is a carefully designed initialization of the embeddings, which drastically reduces the number of iterations required to converge. Finally, PANE utilizes multi-core CPUs through non-trivial parallelization of the above solver, which achieves scalability while retaining the high quality of the resulting embeddings. Extensive experiments, comparing 10 existing approaches on 8 real datasets, demonstrate that PANE consistently outperforms all existing methods in terms of result quality, while being orders of magnitude faster.

PVLDB Reference Format:
Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S. Bhowmick. Scaling Attributed Network Embedding to Massive Graphs. PVLDB, 14(1): 37 - 49, 2021.
doi:10.14778/3421424.3421430

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:10.14778/3421424.3421430

1 INTRODUCTION

Network embedding is a fundamental task for graph analytics, which has attracted much attention from both academia (*e.g.*, [14, 31, 35]) and industry (*e.g.*, [23, 52]). Given an input graph G, network embedding converts each node $v \in G$ to a compact, fixed-length vector X_v, which captures the topological features of the graph around node v. In practice, however, graph data often comes with *attributes* associated to nodes. While we could treat graph topology and attributes as separate features, doing so loses the important information of *node-attribute affinity* [27], *i.e.*, attributes that can be reached by a node through one or more hops along the edges in G. For instance, consider a graph containing companies and board members. An important type of insights that can be gained from such a network is that one company (*e.g.*, Tesla Motors) can reach attributes of another related company (*e.g.*, SpaceX) connected via a common board member (Elon Musk). To incorporate such information, *attributed network embedding* maps both topological and attribute information surrounding a node to an embedding vector, which facilitates accurate predictions, either through the embeddings themselves or in downstream machine learning tasks.

Effective ANE computation is a highly challenging task, especially for massive graphs, *e.g.*, with millions of nodes and billions of edges. In particular, each node $v \in G$ could be associated with a large number of attributes, which corresponds to a high dimensional space; further, each attribute of v could influence not only v's own embedding, but also those of v's neighbors, neighbors' neighbors, and far-reaching connections via multiple hops along the edges in G. Existing ANE solutions are immensely expensive and largely fail on massive graphs. Specifically, as reviewed in Section 6, one class of previous methods *e.g.*, [18, 41, 44, 48], explicitly construct and factorize an $n \times n$ matrix, where n is the number of nodes in G. For a graph with 50 million nodes, storing such a matrix of double-precision values would take over 20 petabytes of memory, which is clearly infeasible. Another category of methods, *e.g.*, [8, 25, 30, 49], employ deep neural networks to extract higher-level features from nodes' connections and attributes. For a large dataset, training such a neural network incurs vast computational costs; further, the training process is usually done on GPUs with limited graphics memory, *e.g.*, 32GB on Nvidia's flagship Tesla V100 cards. Thus, for massive graphs, currently the only option is to compute ANE with a large cluster, *e.g.*, [52], which is financially costly and out of the reach of most researchers.

In addition, to our knowledge, all existing ANE solutions are designed for undirected graphs, and it is unclear how to incorporate edge direction information (*e.g.*, asymmetric transitivity [50]) into their resulting embeddings. In practice, many graphs are directed (*e.g.*, one paper citing another), and existing methods yield suboptimal result quality on such graphs, as shown in our experiments. Can we compute effective ANE embeddings on a massive, attributed, directed graph on a single server?

This paper provides a positive answer to the above question with PANE, a novel solution that significantly advances the state of the art in ANE computation. Specifically, as demonstrated in our experiments, the embeddings obtained by PANE simultaneously achieve the highest prediction accuracy compared to previous methods for 3 common graph analytics tasks: attribute inference, link prediction, and node classification, on common benchmark graph datasets. On the largest Microsoft Academic Knowledge Graph (*MAG*) dataset, PANE is the only viable solution on a single server, whose resulting embeddings lead to 0.88 average precision (AP) for attribute inference, 0.965 AP for link prediction, and 0.57 micro-F1 for node classification. PANE obtains such results using 10 CPU cores, 1TB memory, and 12 hours running time.

PANE achieves effective and scalable ANE computation through three main contributions: a well-thought-out problem formulation based on a novel random walk model, a highly efficient solver, and non-trivial parallelization of the algorithm. Specifically, As presented in Section 2.2, PANE formulates the ANE task as an optimization problem with the objective of approximating normalized multi-hop node-attribute affinity using node-attribute co-projections [27, 28], guided by a shifted pairewise mutual information (SPMI) metric that is inspired by natural language processing techniques. The affinity between a given node-attribute pair is defined via a random walk model specifically adapted to attributed networks. Further, we incorporate edge direction information by defining separate forward and backward affinity, embeddings, and SPMI metrics. Solving this optimization problem is still immensely expensive with off-the-shelf algorithms, as it involves the joint factorization of two $O(n \cdot d)$-sized matrices, where n and d are the numbers of nodes and attributes in the input data, respectively. Thus, PANE includes a novel solver with a key module that seeds the optimizer through a highly effective greedy algorithm, which drastically reduces the number of iterations till convergence. Finally, we devise non-trivial parallelization of the PANE algorithm, to utilize modern multi-core CPUs without significantly compromising result utility. Extensive experiments, using 8 real datasets and comparing against 10 existing solutions, demonstrate that PANE consistently obtains high-utility embeddings with superior prediction accuracy for attribute inference, link prediction and node classification, at a fraction of the cost compared to existing methods.

Summing up, our contributions in this paper are as follows:

- We formulate the ANE task as an optimization problem with the objective of approximating multi-hop node-attribute affinity.
- We further consider edge direction in our objective by defining forward and backward affinity matrices using the SPMI metric.
- We propose several techniques to efficiently solve the optimization problem, including efficient approximation of the affinity matrices, fast joint factorization of the affinity matrices, and a key module to greedily seed the optimizer, which drastically reduces the number of iterations till convergence.
- We develop non-trivial parallelization techniques of PANE to further boost efficiency.

Table 1: Frequently used notations.

Notation	Description
G=(V, E_V, R, E_R)	A graph G with node set V, edge set E_V, attribute set R, and node-attribute association set E_R.
n, m, d	The numbers of nodes, edges, and attributes in G, respectively.
k	The space budget of embedding vectors.
$\mathbf{A}, \mathbf{D}, \mathbf{P}, \mathbf{R}$	The adjacency, out-degree, random walk and attribute matrices of G.
$\mathbf{R}_r, \mathbf{R}_c$	The row-normalized and column-normalized attribute matrices. See Equation (1).
$\mathbf{F}, \mathbf{B}$	The forward and backward affinity matrices. See Equations (2) and (3).
$\mathbf{F}', \mathbf{B}'$	The approximate forward and backward affinity matrices. See Equation (7).
$\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}$	The forward and backward embedding vectors, and attribute embedding vectors.
α	The random walk stopping probability.
n_b	The number of threads.

- The superior performance of PANE, in terms of efficiency and effectiveness, is evaluated against 10 competitors on 8 real datasets.

The rest of the paper is organized as follows. In Section 2, we formally formulate our ANE objective by defining node-attribute affinity. We present single-thread PANE with several speedup techniques in Section 3, and further develop non-trivial parallel PANE in Section 4. The effectiveness and efficiency of our solutions are evaluated in Section 5. Related work is reviewed in Section 6. Finally, Section 7 concludes the paper. Note that proofs of lemmas and additional experiments are given in our technical report [47].

2 PROBLEM FORMULATION

2.1 Preliminaries

Let $G = (V, E_V, R, E_R)$ be an *attributed network*, consisting of (i) a node set V with cardinality n, (ii) a set of edges E_V of size m, each connecting two nodes in V, (iii) a set of attributes R with cardinality d, and (iv) a set of node-attribute associations E_R, where each element is a tuple $(v_i, r_j, w_{i,j})$ signifying that node $v_i \in V$ is directly associated with attribute $r_j \in R$ with a weight $w_{i,j}$ (*i.e.*, the attribute value). Note that for a categorical attribute such as marital status, we first apply a pre-processing step that transforms the attribute into a set of binary ones through one-hot encoding. Without loss of generality, we assume that G is a directed graph; if G is undirected, then we treat each edge (v_i, v_j) in G as a pair of directed edges with opposing directions: (v_i, v_j) and (v_j, v_i).

Given a space budget $k \ll n$, a *node embedding* maps a node $v \in V$ to a length-k vector. The general, hand-waving goal of attributed network embedding (ANE) is to compute such an embedding X_v for each node v in the input graph, such that X_v captures the graph structure and attribute information surrounding node v. In addition, following previous work [27], we also allocate a space budget $\frac{k}{2}$ (explained later in Section 2.3) for each attribute $r \in R$, and aim to compute an *attribute embedding* vector for r of length $\frac{k}{2}$.

Notations. We denote matrices in bold uppercase, *e.g.*, $\mathbf{M}$. We use $\mathbf{M}[v_i]$ to denote the v_i-th row vector of $\mathbf{M}$, and $\mathbf{M}[:, r_j]$ to denote

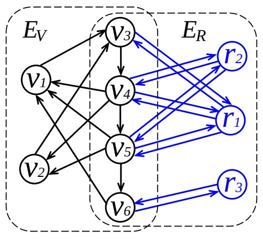


Figure 1: Extended graph $\mathcal{G}$

Table 2: Targets for $\mathbf{X}[v_i] \cdot \mathbf{Y}[r_j]^\top$.

	$\mathbf{Y}[r_1]$	$\mathbf{Y}[r_2]$	$\mathbf{Y}[r_3]$
$\mathbf{X}_f[v_1]$	1.0	0.92	0.47
$\mathbf{X}_b[v_1]$	0.93	0.88	1.17
$\mathbf{X}_f[v_2]$	1.0	0.92	0.47
$\mathbf{X}_b[v_2]$	1.11	1.08	0.8
$\mathbf{X}_f[v_3]$	1.12	1.04	0.54
$\mathbf{X}_b[v_3]$	1.06	0.95	0.99
$\mathbf{X}_f[v_5]$	0.98	1.1	1.08
$\mathbf{X}_b[v_5]$	1.09	1.22	0.61
$\mathbf{X}_f[v_6]$	0.89	0.82	2.05
$\mathbf{X}_b[v_6]$	0.53	0.61	1.6

the r_j-th column vector of $\mathbf{M}$. In addition, we use $\mathbf{M}[v_i, r_j]$ to denote the element at the v_i-th row and r_j-th column of $\mathbf{M}$. Given an index set S, we let $\mathbf{M}[S]$ (resp. $\mathbf{M}[:, S]$) be the matrix block of $\mathbf{M}$ that contains the row (resp. column) vectors of the indices in S.

Let $\mathbf{A}$ be the adjacency matrix of the input graph G, *i.e.*, $\mathbf{A}[v_i, v_j] = 1$ if $(v_i, v_j) \in E_V$, otherwise $\mathbf{A}[v_i, v_j] = 0$. Let $\mathbf{D}$ be the diagonal out-degree matrix of G, *i.e.*, $\mathbf{D}[v_i, v_i] = \sum_{v_j \in V} \mathbf{A}[v_i, v_j]$. We define the random walk matrix of G as $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$.

Furthermore, we define an attribute matrix $\mathbf{R} \in \mathbb{R}^{n \times d}$, such that $\mathbf{R}[v_i, r_j] = w_{i,j}$ is the weight associated with the entry $(v_i, r_j, w_{ij}) \in E_R$. We refer to $\mathbf{R}[v_i]$ as node v_i's *attribute vector*. Based on $\mathbf{R}$, we derive a row-normalized (resp. column-normalized) attribute matrices $\mathbf{R}_r$ (resp. $\mathbf{R}_c$) as follows:

$$\mathbf{R}_r[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{R}[v_l, r_j]}, \quad \mathbf{R}_c[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{R}[v_i, r_l]}. \quad (1)$$

Table 1 lists the frequently used notations in our paper.

Extended graph. Our solution utilizes an *extended graph* $\mathcal{G}$ that incorporates additional nodes and edges into G. To illustrate, Figure 1 shows an example extended graph $\mathcal{G}$ constructed based on an input attributed network G with 6 nodes v_1-v_6 and 3 attributes r_1-r_3. The left part of the figure (in black) shows the topology of G, *i.e.*, the edge set E_V. The right part of the figure (in blue) shows the attribute associations E_R in G. Specifically, for each attribute $r_j \in R$, we create an additional node in $\mathcal{G}$; then, For each entry in E_R, *e.g.*, $(v_3, r_1, w_{3,1})$, we include in $\mathcal{G}$ a pair of edges with opposing directions connecting the node (*e.g.*, v_3) with the corresponding attribute node (*e.g.*, r_1), with an edge weight (*e.g.*, $w_{3,1}$). Note that in this example, nodes v_1 and v_2 are not associated with any attribute.

2.2 Node-Attribute Affinity via Random Walks

As explained in Section 1, the resulting embedding of a node $v \in V$ should capture its *affinity* with attributes in R, where the affinity definition should take into account both the attributes directly associated with v in E_R, and the attributes of the nodes that v can reach via edges in E_V. To effectively model node-attribute affinity via multiple hops in $\mathcal{G}$, we employ an adaptation of the *random walks with restarts* (*RWR*) [19, 36] technique to our setting with an extended graph $\mathcal{G}$. In the following, we refer to an RWR simply as a *random walk*. Specifically, since $\mathcal{G}$ is directed, we distinguish two types of node-attribute affinity: *forward affinity*, denoted as $\mathbf{F}$, and *backward affinity*, denoted as $\mathbf{B}$.

Forward affinity. We first focus on forward affinity. Given an attributed graph G, a node v_i, and random walk stopping probability α ($0 < \alpha < 1$), a *forward random walk* on $\mathcal{G}$ starts from node v_i. At each step, assume that the walk is currently at node v_l. Then, the walk can either (i) with proabability α, terminate at v_l , or (ii) with probability $1 - \alpha$, follow an edge in E_V to a random out-neighbor of v_l. After a random walk terminates at a node v_l, we randomly follow an edge in E_R to an attribute r_j, with probability $\mathbf{R}_r[v_l, r_j]$, *i.e.*, a normalized edge weight defined in Equation (1)[1]. The forward random walk yields a *node-to-attribute pair* (v_i, r_j), and we add this pair to a collection $\mathcal{S}_f$.

Suppose that we sample n_r node-to-attribute pairs for each node v_i, the size of $\mathcal{S}_f$ is then $n_r \cdot n$, where n is the number of nodes in G. Denote $p_f(v_i, r_j)$ as the probability that a forward random walk starting from v_i yields a node-to-attribute pair (v_i, r_j). Then, the *forward affinity* $\mathbf{F}[v_i, r_j]$ between note v_i and attribute r_j is defined as follows.

$$\mathbf{F}[v_i, r_j] = \log\left(\frac{n \cdot p_f(v_i, r_j)}{\sum_{v_h \in V} p_f(v_h, r_j)} + 1\right) \quad (2)$$

To explain the intuition behind the above definition, note that in collection $\mathcal{S}_f$, the probabilities of observing node v_i, attribute r_j, and pair (v_i, r_j) are $\mathbb{P}(v_i) = \frac{1}{n}$, $\mathbb{P}(r_j) = \frac{\sum_{v_h \in V} \cdot p_f(v_h, r_j)}{n}$, and $\mathbb{P}(v_i, r_j) = \frac{p_f(v_i, r_j)}{n}$, respectively. Thus, the above definition of forward affinity is a variant of the *pointwise mutual information* (PMI) [4] between node v_i and attribute r_j. In particular, given a collection of element pairs $\mathcal{S}$, the PMI of element pair $(x, y) \in \mathcal{S}$, denoted as $\mathrm{PMI}(x, y)$, is defined as $\mathrm{PMI}(x, y) = \log\left(\frac{\mathbb{P}(x,y)}{\mathbb{P}(x) \cdot \mathbb{P}(y)}\right)$, where $\mathbb{P}(x)$ (resp. $\mathbb{P}(y)$) is the probability of observing x (resp. y) in $\mathcal{S}$ and $\mathbb{P}(x, y)$ is the probability of observing pair (x, y) in $\mathcal{S}$. The larger $\mathrm{PMI}(x, y)$ is, the more likely x and y co-occur in $\mathcal{S}$. Note that $\mathrm{PMI}(x, y)$ can be negative. To avoid this, we use an alternative: shifted PMI, defined as $\mathrm{SPMI}(x, y) = \log\left(\frac{\mathbb{P}(x,y)}{\mathbb{P}(x) \cdot \mathbb{P}(y)} + 1\right)$, which is guaranteed to be positive, while retaining the original order of values of PMI. $\mathbf{F}[v_i, r_j]$ in Equation (2) is then $\mathrm{SPMI}(v_i, r_j)$.

Another way to understand Equation (2) is through an analogy to TF/IDF [32] in natural language processing. Specifically, if we view the all forward random walks as a "document", then $n \cdot p_f(v_i, r_j)$ is akin to the term frequency of r_j, whereas the denominator in Equation (2) is similar to the inverse document frequency of r_j. Thus, the normalization penalizes common attributes, and compensates for rare attributes.

Backward affinity. Next we define backward affinity in a similar fashion. Given an attributed network G, an attribute r_j and stopping probability α, a *backward random walk* starting from r_j first randomly samples a node v_l according to probability $\mathbf{R}_c[v_l, r_j]$, defined in Equation (1). Then, the walk starts from node v_l; at each step, the walk either terminates at the current node with α probability, or randomly jumps to an out-neighbor of current node with $1 - \alpha$ probability. Suppose that the walk terminates at node v_i; then, it returns an *attribute-to-node pair* (r_j, v_i), which is added to a collection $\mathcal{S}_b$. After sampling n_r attribute-to-node pairs for each attribute, the size of $\mathcal{S}_b$ becomes $n_r \cdot d$. Let $p_b(v_i, r_j)$ be the probability that a backward random walk starting from attribute r_j stops at node v_i. In collection $\mathcal{S}_b$, the probabilities of observing attribute

[1]In the degenerate case that v_l is not associated with any attribute, *e.g.*, v_1 in Figure 1, we simply restart the random walk from the source node v_i, and repeat the process.

r_j, node v_i and pair (r_j, v_i) are $\mathbb{P}(r_j) = \frac{1}{d}$, $\mathbb{P}(v_i) = \frac{\sum_{r_h \in R} p_b(v_i, r_h)}{d}$ and $\mathbb{P}(v_i, r_j) = \frac{p_b(v_i, r_j)}{d}$, respectively. By the definition of SPMI, we define backward affinity $\mathbf{B}[v_i, r_j]$ as follows.

$$\mathbf{B}[v_i, r_j] = \log\left(\frac{d \cdot p_b(v_i, r_j)}{\sum_{r_h \in R} p_b(v_i, r_h)} + 1\right). \quad (3)$$

2.3 Objective Function

Next we define our objective function for ANE, based on the notions of forward and backward node-attribute affinity defined in Equation (2) and Equation (3), respectively. Let $\mathbf{F}[v_i, r_j]$ (resp. $\mathbf{B}[v_i, r_j]$) be the forward affinity (resp. backward affinity) between node v_i and attribute r_j. Given a space budget k, our objective is to learn (i) two embedding vectors for each node v_i, namely a *forward embedding vector*, denoted as $\mathbf{X}_f[v_i] \in \mathbb{R}^{\frac{k}{2}}$ and a *backward embedding vector*, denoted as $\mathbf{X}_b[v_i] \in \mathbb{R}^{\frac{k}{2}}$, as well as (ii) an *attribute embedding vector* $\mathbf{Y}[r_j] \in \mathbb{R}^{\frac{k}{2}}$ for each attribute r_j, such that the following objective is minimized:

$$O = \min_{\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b} \sum_{v_i \in V} \sum_{r_j \in R} \left(\mathbf{F}[v_i, r_j] - \mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top\right)^2 + \left(\mathbf{B}[v_i, r_j] - \mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top\right)^2 \quad (4)$$

Intuitively, in the above objective function, we approximate the forward node-attribute affinity $\mathbf{F}[v_i, r_j]$ between node v_i and attribute r_j using the dot product of their respective embedding vectors, *i.e.*, $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top$. Similarly, we also approximate the backward node-attribute affinity using $\mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top$. The objective is then to minimize the total squared error of such approximations, over all nodes and all attributes in the input data.

Running Example. Assume that in the extended graph $\mathcal{G}$ shown in Figure 1, all attribute weights in E_R are 1, and the random walk stopping probability α is set to 0.15 [19, 36]. Table 2 lists the target values, *i.e.*, the exact forward and backward affinity values. According to Equation (4), for the inner products of attribute embedding vectors of r_1-r_3 and that of v_1-v_6. These values are calculated based on Equations (2) and (3), using simulated random walks on $\mathcal{G}$ in Figure 1. Observe, for example, that node v_1 has high affinity values (both forward and backward) with attribute r_1, which agrees with the intuition that v_1 is connected to r_1 via many different intermediate nodes, *i.e.*, v_3, v_4, v_5. For node v_5, if only forward affinity is considered, observe that v_5 has higher forward affinity value with r_3 than that with r_1, which cannot reflect the fact that v_5 owns r_1 but not r_3, leading to wrong attribute inference. If both forward and backward affinity are considered, this issue is resolved.

3 THE PANE ALGORITHM

It is technically challenging to train embeddings of nodes and attributes that preserve our objective function in Equation (4), especially on massive attributed networks. First, node-attribute affinity values are defined by random walks, which are rather expensive to be stimulated in a huge number from every node and attribute of massive graphs, to accurately get the affinity values of all possible node-attribute pairs. Second, our objective function preserves both forward and backward affinity (*i.e.*, considering edge directions), which makes the training process hard to converge. Further, jointly preserving both forward and backward affinity involves intensive computations, severely dragging down the performance. To this end, we propose PANE that can efficiently handle large-scale data and produce high-quality ANE results. At a high level, PANE consists of two phases: (i) iteratively computing approximated versions $\mathbf{F}'$ and $\mathbf{B}'$ of the forward and backward affinity matrices with rigorous approximation error guarantees, without actually sampling random walks (Section 3.1), and (ii) initializing the embedding vectors with a greedy algorithm for fast convergence, and then jointly factorizing $\mathbf{F}'$ and $\mathbf{B}'$ using *cyclic coordinate descent* [38] to efficiently obtain the output embedding vectors $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ (Section 3.2). Given an attributed network G, space budget k, random walk stopping probability α and an error threshold ϵ as inputs, Algorithm 1 outlines the proposed PANE algorithm in the single-threaded setting. For ease of presentation, this section describes the single-threaded version of the proposed solution PANE for ANE. The full version of PANE that runs in multiple threads is explained later in Section 4.

Algorithm 1: PANE (single thread)

```
Input: Attributed network $G$, space budget $k$, random walk
       stopping probability $\alpha$, error threshold $\epsilon$.
Output: Forward and backward embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and
        attribute embedding vectors $\mathbf{Y}$.
1 $t \leftarrow \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$;
2 $\mathbf{F}', \mathbf{B}' \leftarrow$ APMI$(\mathbf{P}, \mathbf{R}, \alpha, t)$;
3 $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b \leftarrow$ SVDCCD$(\mathbf{F}', \mathbf{B}', k, t)$;
4 return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

Algorithm 2: APMI

```
Input: $\mathbf{P}, \mathbf{R}, \alpha, t$.
Output: $\mathbf{F}', \mathbf{B}'$.
1 Compute $\mathbf{R}_r$ and $\mathbf{R}_c$ by Equation (1);
2 $\mathbf{P}_f^{(0)} \leftarrow \mathbf{R}_r$, $\mathbf{P}_b^{(0)} \leftarrow \mathbf{R}_c$;
3 for $\ell \leftarrow 1$ to $t$ do
4     $\mathbf{P}_f^{(\ell)} \leftarrow (1-\alpha) \cdot \mathbf{P}\mathbf{P}_f^{(\ell-1)} + \alpha \cdot \mathbf{P}_f^{(0)}$;
5     $\mathbf{P}_b^{(\ell)} \leftarrow (1-\alpha) \cdot \mathbf{P}^\top\mathbf{P}_b^{(\ell-1)} + \alpha \cdot \mathbf{P}_b^{(0)}$;
6 Normalize $\mathbf{P}_f^{(t)}$ by columns to get $\widehat{\mathbf{P}}_f^{(t)}$;
7 Normalize $\mathbf{P}_b^{(t)}$ by rows to get $\widehat{\mathbf{P}}_b^{(t)}$;
8 $\mathbf{F}' \leftarrow \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1)$,    $\mathbf{B}' \leftarrow \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1)$;
9 return $\mathbf{F}', \mathbf{B}'$;
```

3.1 Forward and Backward Affinity Approximation

In Section 2.2, node-attribute affinity values are defined using a large number of random walks, which are expensive to simulate on a massive graph. For the purpose of efficiency, in this section, we transform forward and backward affinity in Equations (2) and (3) into their matrix forms and propose APMI in Algorithm 2, which efficiently approximates forward and backward affinity matrices with error guarantee and in linear time complexity, without actually sampling random walks.

Observe that in Equations (2) and (3), the key for forward and backward affinity computation is to obtain $p_f(v_i, r_j)$ and $p_b(v_i, r_j)$

for every pair $(v_i, r_j) \in V \times R$. Recall that $p_f(v_i, r_j)$ is the probability that a forward random walk starting from node v_i picks attribute r_j, while $p_b(v_i, r_j)$ is the probability of a backward random walk from attribute r_j stopping at node v_i. Given nodes v_i and v_l, denote $\pi(v_i, v_l)$ as the probability that a random walk starting from v_i stops at v_l, *i.e.*, the random walk score of v_l with respect to v_i. By definition, $p_f(v_i, r_j) = \sum_{v_l \in V} \pi(v_i, v_l) \cdot \mathbf{R}_r[v_l, r_j]$, where $\mathbf{R}_r[v_l, r_j]$ is the probability that node v_l picks attribute r_j, according to Equation (1). Similarly, $p_b(v_i, r_j)$ is formulated as $p_b(v_i, r_j) = \sum_{v_l \in V} \mathbf{R}_c[v_l, r_j] \cdot \pi(v_l, v_i)$, where $\mathbf{R}_c[v_l, r_j]$ is the probability that attribute r_j picks node v_l from all nodes having r_j based on their attribute weights. By the definition of random walk scores in [19, 36], we can derive the matrix form of p_f and p_b as follows.

$$\begin{aligned} \mathbf{P}_f &= \alpha \textstyle\sum_{\ell=0}^{\infty} (1-\alpha)^\ell \mathbf{P}^\ell \cdot \mathbf{R}_r, \\ \mathbf{P}_b &= \alpha \textstyle\sum_{\ell=0}^{\infty} (1-\alpha)^\ell \mathbf{P}^{\top\ell} \cdot \mathbf{R}_c. \end{aligned} \tag{5}$$

We only consider t iterations to approximate $\mathbf{P}_f$ and $\mathbf{P}_b$ in Equation (6), where t is set to $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$.

$$\mathbf{P}_f^{(t)} = \alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^\ell \cdot \mathbf{R}_r, \quad \mathbf{P}_b^{(t)} = \alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^{\top\ell} \cdot \mathbf{R}_c. \tag{6}$$

Then, we normalize $\mathbf{P}_f^{(t)}$ by columns and $\mathbf{P}_b^{(t)}$ by rows as follows.

$$\widehat{\mathbf{P}}_f^{(t)}[v_i, r_j] = \frac{\mathbf{P}_f^{(t)}[v_i,r_j]}{\sum_{v_l \in V} \mathbf{P}_f^{(t)}[v_l,r_j]}, \quad \widehat{\mathbf{P}}_b^{(t)}[v_i, r_j] = \frac{\mathbf{P}_b^{(t)}[v_i,r_j]}{\sum_{r_l \in R} \mathbf{P}_b^{(t)}[v_i,r_l]}$$

After normalization, we compute $\mathbf{F}'$ and $\mathbf{B}'$ according to the definitions of forward and backward affinity as follows.

$$\mathbf{F}' = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1), \quad \mathbf{B}' = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1) \tag{7}$$

Algorithm 2 shows the pseudo-code of APMI to compute $\mathbf{F}'$ and $\mathbf{B}'$. Specifically, APMI takes as inputs random walk matrix $\mathbf{P}$, attribute matrix $\mathbf{R}$, random walk stopping probability α and the number of iterations t. At Line 1, APMI begins by computing row-normalized attribute matrix $\mathbf{R}_r$ and column-normalized attribute matrix $\mathbf{R}_c$ according to Equation (1). Then, APMI computes $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ based on Equation (6). Note that $\mathbf{P}$ is sparse and has m non-zero entries. Thus, the computations of $\alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^\ell$ and $\alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^{\top\ell}$ in Equation (6) need $O(mnt)$ time, which is prohibitively expensive on large graphs. We avoid such expensive overheads and achieve a time cost of $O(mdt)$ for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ by an iterative process as follows. Initially, we set $\mathbf{P}_f^{(0)} = \mathbf{R}_r$ and $\mathbf{P}_b^{(0)} = \mathbf{R}_c$ (Line 2). Then, we start an iterative process from Line 3 to 5 with t iterations; at the ℓ-th iteration, we compute $\mathbf{P}_f^{(\ell)} = (1-\alpha) \cdot \mathbf{P}\mathbf{P}_f^{(\ell-1)} + \alpha \cdot \mathbf{P}_f^{(0)}$ and $\mathbf{P}_b^{(\ell)} = (1-\alpha) \cdot \mathbf{P}^\top \mathbf{P}_b^{(\ell-1)} + \alpha \cdot \mathbf{P}_b^{(0)}$. After t iterations, APMI normalizes $\mathbf{P}_f^{(t)}$ by column and $\mathbf{P}_b^{(t)}$ by row (Lines 6-7). At Line 8, APMI obtains $\mathbf{F}'$ and $\mathbf{B}'$ as the approximate forward and backward affinity matrices. The following lemma establishes the accuracy guarantee of APMI.

Lemma 3.1. *Given* $\mathbf{P}, \mathbf{R}_r, \alpha, \epsilon$ *as inputs to Algorithm 2, the returned approximate forward and backward affinity matrices* $\mathbf{F}', \mathbf{B}'$ *satisfy that, for every pair* $(v_i, r_j) \in V \times R$,

$$\frac{2^{\mathbf{F}'[v_i,r_j]}-1}{2^{\mathbf{F}[v_i,r_j]}-1} \in \left[\max\left\{0, 1 - \frac{\epsilon}{\mathbf{P}_f[v_i,r_j]}\right\}, 1 + \frac{\epsilon}{\sum_{v_l \in V} \max\{0, \mathbf{P}_f[v_l,r_j] - \epsilon\}}\right],$$

$$\frac{2^{\mathbf{B}'[v_i,r_j]}-1}{2^{\mathbf{B}[v_i,r_j]}-1} \in \left[\max\left\{0, 1 - \frac{\epsilon}{\mathbf{P}_b[v_i,r_j]}\right\}, 1 + \frac{\epsilon}{\sum_{r_l \in R} \max\{0, \mathbf{P}_b[v_i,r_l] - \epsilon\}}\right].$$

Proof. First, with $t = \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$, we have

$$\textstyle\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell = 1 - \sum_{\ell=0}^{t} \alpha(1-\alpha)^\ell = (1-\alpha)^{t+1} = \epsilon. \tag{8}$$

By the definitions of $\mathbf{P}_f, \mathbf{P}_f^{(t)}$ and $\mathbf{P}_b, \mathbf{P}_b^{(t)}$ (*i.e.*, Equation (5) and Equation (6)), for every pair $(v_i, r_j) \in V \times R$,

$$\begin{aligned} &\mathbf{P}_f[v_i, r_j] - \mathbf{P}_f^{(t)}[v_i, r_j] = \textstyle\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^\ell[v_i] \cdot \mathbf{R}_r{}^\top[r_j] \\ &= \left(\textstyle\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^\ell\right)[v_i] \cdot \mathbf{R}_r{}^\top[r_j] \le \textstyle\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell = \epsilon, \\ &\mathbf{P}_b[v_i, r_j] - \mathbf{P}_b^{(t)}[v_i, r_j] = \textstyle\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^{\top\ell}[v_i] \cdot \mathbf{R}_c^\top[r_j] \\ &\le \textstyle\sum_{v_l \in V} \left(\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell\right) \cdot \mathbf{R}_c[v_l, r_j] \le \sum_{v_l \in V} \epsilon \cdot \mathbf{R}_c[v_l, r_j] = \epsilon. \end{aligned}$$

Based on the above inequalities, for every pair $(v_i, r_j) \in V \times R$,

$$\max\{0, \mathbf{P}_f[v_i, r_j] - \epsilon\} \le \mathbf{P}_f^{(t)}[v_i, r_j] \le \mathbf{P}_f[v_i, r_j], \tag{9}$$

$$\max\{0, \mathbf{P}_b[v_i, r_j] - \epsilon\} \le \mathbf{P}_b^{(t)}[v_i, r_j] \le \mathbf{P}_b[v_i, r_j]. \tag{10}$$

According to Lines 6-9 in Algorithm 2, for every pair $(v_i, r_j) \in V \times R$,

$$\frac{2^{\mathbf{F}'[v_i,r_j]}-1}{2^{\mathbf{F}[v_i,r_j]}-1} = \frac{\widehat{\mathbf{P}}_f^{(t)}[v_i,r_j]}{\widehat{\mathbf{P}}_f[v_i,r_j]} = \frac{\mathbf{P}_f^{(t)}[v_i,r_j]}{\sum_{v_l \in V} \mathbf{P}_f^{(t)}[v_l,r_j]} \times \frac{\sum_{v_l \in V} \mathbf{P}_f[v_l,r_j]}{\mathbf{P}_f[v_i,r_j]}, \tag{11}$$

$$\frac{2^{\mathbf{B}'[v_i,r_j]}-1}{2^{\mathbf{B}[v_i,r_j]}-1} = \frac{\widehat{\mathbf{P}}_b^{(t)}[v_i,r_j]}{\widehat{\mathbf{P}}_f[v_i,r_j]} = \frac{\mathbf{P}_b^{(t)}[v_i,r_j]}{\sum_{r_l \in R} \mathbf{P}_b^{(t)}[v_i,r_l]} \times \frac{\sum_{r_l \in R} \mathbf{P}_b[v_i,r_l]}{\mathbf{P}_b[v_i,r_j]}. \tag{12}$$

Plugging Inequalities (9) and (10) into Inequalities (11) and (12) leads to the desired results, which completes our proof. □

3.2 Joint Factorization of Affinity Matrices

This section presents the proposed algorithm SVDCCD, outlined in Algorithm 4, which jointly factorizes the approximate forward and backward affinity matrices $\mathbf{F}'$ and $\mathbf{B}'$, in order to obtain the embedding vectors of all nodes and attributes, *i.e.*, $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$. As the name suggests, the proposed SVDCCD solver is based on the *cyclic coordinate descent* (*CCD*) framework, which iteratively updates each embedding value towards optimizing the objective function in Equation (4). The problem, however, is that a direct application of CCD, starting from random initial values of the embeddings, requires numerous iterations to converge, leading to prohibitive overheads. Furthermore, CCD computation itself is expensive, especially on large-scale graphs. To overcome these challenges, we firstly propose a greedy initialization method to facilitate fast convergence, and then design techniques for efficient refinement of initial embeddings, including dynamic maintenance and partial updates of intermediate results to avoid redundant computations in CCD.

Greedy initialization. In many optimization problems, all we need for efficiency is a good initialization. Thus, a key component in the proposed SVDCCD algorithm is such an initialization of embedding values, based on *singular value decomposition* (*SVD*) [10]. Note that unlike other matrix factorization problems, here SVD by itself cannot solve our problem, because the objective function in

Equation (4) requires the joint factorization of both the forward and backward affinity matrices at the same time, which cannot be directly addressed with SVD.

Algorithm 3 describes the GreedyInit module of SVDCCD, which initializes embeddings $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$. Specifically, the algorithm first employs an efficient randomized SVD algorithm [29] at Line 1 to decompose $\mathbf{F}'$ into $\mathbf{U} \in \mathbb{R}^{n\times \frac{k}{2}}, \Sigma \in \mathbb{R}^{\frac{k}{2}\times\frac{k}{2}}, \mathbf{V} \in \mathbb{R}^{d\times\frac{k}{2}}$, and then initializes $\mathbf{X}_f = \mathbf{U}\Sigma$ and $\mathbf{Y} = \mathbf{V}$ at Line 2, which satisfies $\mathbf{X}_f \cdot \mathbf{Y}^\top \approx \mathbf{F}'$. In other words, this initialization immediately gains a good approximation of the forward affinity matrix.

Recall that our objective function in Equation (4) also aims to find $\mathbf{X}_b$ such that $\mathbf{X}_b\mathbf{Y}^\top \approx \mathbf{B}'$, *i.e.*, to approximate the backward affinity matrix well. Here comes the key observation of the algorithm: that matrix $\mathbf{V}$ (*i.e.*, $\mathbf{Y}$) returned by exact SVD is *unitary*, *i.e.*, $\mathbf{Y}^\top\mathbf{Y} = \mathbf{I}$, which implies that $\mathbf{X}_b \approx \mathbf{X}_b\mathbf{Y}^\top\mathbf{Y} \approx \mathbf{B}'\mathbf{Y}$. Accordingly, we seed $\mathbf{X}_b$ with $\mathbf{B}'\mathbf{Y}$ at Line 2 of Algorithm 3. This initialization of $\mathbf{X}_b$ also leads to a relatively good approximation of the backward affinity matrix. Consequently, the number of iterations required by SVDCCD is drastically reduced, as confirmed by our experiments in Section 5.

Efficient refinement of the initial embeddings. In Algorithm 4, after initializing $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ at Line 1, we apply cyclic coordinate descent to refine the embedding vectors according to our objective function in Equation (4) from Lines 2 to 14. The basic idea of CCD is to cyclically iterate through all entries in $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, one at a time, minimizing the objective function with respect to each entry (*i.e.*, coordinate direction). Specifically, in each iteration, CCD updates each entry of $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ according to the following rules:

$$\mathbf{X}_f[v_i,l] \leftarrow \mathbf{X}_f[v_i,l] - \mu_f(v_i,l), \tag{13}$$

$$\mathbf{X}_b[v_i,l] \leftarrow \mathbf{X}_b[v_i,l] - \mu_b(v_i,l), \tag{14}$$

$$\mathbf{Y}[r_j,l] \leftarrow \mathbf{Y}[r_j,l] - \mu_y(r_j,l), \tag{15}$$

with $\mu_f(v_i,l), \mu_b(v_i,l)$ and $\mu_y(r_j,l)$ computed by:

$$\mu_f(v_i,l) = \frac{\mathbf{S}_f[v_i]\cdot\mathbf{Y}[:,l]}{\mathbf{Y}^\top[l]\cdot\mathbf{Y}[:,l]}, \quad \mu_b(v_i,l) = \frac{\mathbf{S}_b[v_i]\cdot\mathbf{Y}[:,l]}{\mathbf{Y}^\top[l]\cdot\mathbf{Y}[:,l]}, \tag{16}$$

$$\mu_y(r_j,l) = \frac{\mathbf{X}_f^\top[l]\cdot\mathbf{S}_f[:,r_j]+\mathbf{X}_b^\top[l]\cdot\mathbf{S}_b[:,r_j]}{\mathbf{X}_f^\top[l]\cdot\mathbf{X}_f[:,l]+\mathbf{X}_b^\top[l]\cdot\mathbf{X}_b[:,l]}, \tag{17}$$

where $\mathbf{S}_f = \mathbf{X}_f\mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b\mathbf{Y}^\top - \mathbf{B}'$ are obtained at Line 3 in Algorithm 3.

However, directly applying the above updating rules to learn $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ is inefficient, leading to many redundant matrix operations. Lines 2-14 in Algorithm 4 show how to efficiently apply the above updating rules by dynamically maintaining and partially updating intermediate results. Specifically, each iteration in Lines 3-14 first fixes $\mathbf{Y}$ and updates each row of $\mathbf{X}_f$ and $\mathbf{X}_b$ (Lines 3-9), and then updates each column of $\mathbf{Y}$ with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed (Lines 10-14). According to Equations (16) and (17), $\mu_f(v_i,l), \mu_b(v_i,l)$, and $\mu_y(r_j,l)$ are pertinent to $\mathbf{S}_f[v_i]$, $\mathbf{S}_b[v_i]$, and $\mathbf{S}_f[:,r_j], \mathbf{S}_b[:,r_j]$ respectively, where $\mathbf{S}_f$ and $\mathbf{S}_b$ further depend on embedding vectors $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$. Therefore, whenever $\mathbf{X}_f[v_i,l], \mathbf{X}_b[v_i,l]$, and $\mathbf{Y}[r_j,l]$ are updated in the iteration (Lines 6-7 and Line 13), $\mathbf{S}_f$ and $\mathbf{S}_b$ need to be updated accordingly. Directly recomputing $\mathbf{S}_f$ and $\mathbf{S}_b$ by $\mathbf{S}_f = \mathbf{X}_f\mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b\mathbf{Y}^\top - \mathbf{B}'$ whenever an entry in $\mathbf{X}_f, \mathbf{X}_b$ and, $\mathbf{Y}$ is updated is expensive.

Instead, we dynamically maintain and partially update $\mathbf{S}_f$ and $\mathbf{S}_b$ according to Equations (18), (19) and (20). Specifically, when

Algorithm 3: GreedyInit

```
    Input: $\mathbf{F}', \mathbf{B}', k, t$.
    Output: $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$.
1   $\mathbf{U}, \Sigma, \mathbf{V} \leftarrow$ RandSVD($\mathbf{F}', \frac{k}{2}, t$);
2   $\mathbf{Y} \leftarrow \mathbf{V},\ \mathbf{X}_f \leftarrow \mathbf{U}\Sigma,\ \mathbf{X}_b \leftarrow \mathbf{B}' \cdot \mathbf{Y}$;
3   $\mathbf{S}_f \leftarrow \mathbf{X}_f\mathbf{Y}^\top - \mathbf{F}',\ \mathbf{S}_b \leftarrow \mathbf{X}_b\mathbf{Y}^\top - \mathbf{B}'$;
4   return $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$;
```

Algorithm 4: SVDCCD

```
    Input: $\mathbf{F}', \mathbf{B}', k, t$.
    Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.
1   $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow$ GreedyInit($\mathbf{F}', \mathbf{B}', k, t$);
2   for $\ell \leftarrow 1$ to $t$ do
3       for $v_i \in V$ do
4           for $l \leftarrow 1$ to $\frac{k}{2}$ do
5               Compute $\mu_f(v_i,l), \mu_b(v_i,l)$ by Equation (16);
6               $\mathbf{X}_f[v_i,l] \leftarrow \mathbf{X}_f[v_i,l] - \mu_f(v_i,l)$;
7               $\mathbf{X}_b[v_i,l] \leftarrow \mathbf{X}_b[v_i,l] - \mu_b(v_i,l)$;
8               Update $\mathbf{S}_f[v_i]$ by Equation (18);
9               Update $\mathbf{S}_b[v_i]$ by Equation (19);
10      for $r_j \in R$ do
11          for $l \leftarrow 1$ to $\frac{k}{2}$ do
12              Compute $\mu_y(r_j,l)$ by Equation (17);
13              $\mathbf{Y}[r_j,l] \leftarrow \mathbf{Y}[r_j,l] - \mu_y(r_j,l)$;
14              Update $\mathbf{S}_f[:,r_j], \mathbf{S}_b[:,r_j]$ by Equation (20);
15  return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

$\mathbf{X}_f[v_i,l]$ and $\mathbf{X}_b[v_i,l]$ are updated (Lines 6-7), we update $\mathbf{S}_f[v_i]$ and $\mathbf{S}_b[v_i]$ respectively with $O(d)$ time at Lines 8-9 by

$$\mathbf{S}_f[v_i] \leftarrow \mathbf{S}_f[v_i] - \mu_f(v_i,l)\cdot\mathbf{Y}[:,l]^\top, \tag{18}$$

$$\mathbf{S}_b[v_i] \leftarrow \mathbf{S}_b[v_i] - \mu_b(v_i,l)\cdot\mathbf{Y}[:,l]^\top, \tag{19}$$

Whenever $\mathbf{Y}[r_j,l]$ is updated at Line 13, both $\mathbf{S}_f[:,r_j]$ and $\mathbf{S}_b[:,r_j]$ are updated in $O(n)$ time at Line 14 by

$$\begin{aligned}\mathbf{S}_f[:,r_j] &\leftarrow \mathbf{S}_f[:,r_j] - \mu_y(r_j,l)\cdot\mathbf{X}_f[:,l],\\ \mathbf{S}_b[:,r_j] &\leftarrow \mathbf{S}_b[:,r_j] - \mu_y(r_j,l)\cdot\mathbf{X}_b[:,l],\end{aligned} \tag{20}$$

3.3 Complexity Analysis

In the proposed algorithm PANE (Algorithm 1), the maximum length of random walk is $t = \frac{\log(\epsilon)}{\log(1-\alpha)} - 1 = \frac{\log(\frac{1}{\epsilon})}{\log(\frac{1}{1-\alpha})} - 1$. According to Section 3.1, Algorithm 2 runs in time $O(md\cdot t) = O\left(md\cdot\log\frac{1}{\epsilon}\right)$. Meanwhile, according to [29], given $\mathbf{F}' \in \mathbb{R}^{n\times d}$ as input, RandSVD in Algorithm 3 requires $O(ndkt)$ time, where n, d, k are the number of nodes, number of attributes, and embedding space budget, respectively. The computation of $\mathbf{S}_f, \mathbf{S}_b$ costs $O(ndk)$ time. In addition, the t iterations of CCD for updating $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ take $O(ndkt) = O(ndk\log\frac{1}{\epsilon})$ time. Therefore, the overall time complexity of Algorithm 1 is $O\left((md+ndk)\cdot\log\left(\frac{1}{\epsilon}\right)\right)$. The memory consumption of intermediate results yielded in Algorithm 1, *i.e.*, $\mathbf{F}'$,$\mathbf{B}'$, $\mathbf{U}$, Σ, $\mathbf{V}$,$\mathbf{S}_f$,$\mathbf{S}_b$ are at most $O(nd)$. Hence, the space complexity of Algorithm 1 is bounded by $O(nd+m)$.

Algorithm 5: PANE

```
Input: Attributed network $G$, space budget $k$, random walk
       stopping probability $\alpha$, error threshold $\epsilon$, the number of
       threads $n_b$.
Output: Forward and backward embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and
        attribute embedding vectors $\mathbf{Y}$.
1 Partition $V$ into $n_b$ subsets $\mathcal{V} \leftarrow \{V_1, \cdots, V_{n_b}\}$ equally;
2 Partition $R$ into $n_b$ subsets $\mathcal{R} \leftarrow \{R_1, \cdots, R_{n_b}\}$ equally;
3 $t \leftarrow \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$;
4 $\mathbf{F}', \mathbf{B}' \leftarrow$ PAPMI$(\mathbf{P}, \mathbf{R}, \alpha, t, \mathcal{V}, \mathcal{R})$;
5 $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b \leftarrow$ PSVDCCD$(\mathbf{F}', \mathbf{B}', \mathcal{V}, \mathcal{R}, k, t)$;
6 return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

4 PARALLELIZATION OF PANE

Although single-thread PANE (*i.e.*, Algorithm 1) runs in linear time to the size of the input attributed network, it still requires substantial time to handle large-scale attributed networks in practice. For instance, on *MAG* dataset that has 59.3 million nodes, PANE (single thread) takes about five days. Note that it is challenging to develop a parallel algorithm achieving such linear scalability to the number of threads on a multi-core CPU. Specifically, PANE involves various computational patterns, including intensive matrix computation, factorization, and CCD updates. Moreover, it is also challenging to maintain the intermediate result of each thread and combine them as the final result. To further boost efficiency, in this section we develop a parallel PANE (Algorithm 5), and it takes only 11.9 hours on *MAG* when using 10 threads (*i.e.*, up to 10 times speedup). In the first phase, we adopt block matrix multiplication [11] and propose PAPMI to compute forward and backward affinity matrices in a parallel manner (Section 4.1). In the second phase, we develop PSVDCCD with a split-merge-based parallel SVD technique to efficiently decompose affinity matrices, and further propose a parallel CCD technique to refine the embeddings efficiently (Section 4.2).

Algorithm 5 illustrates the pseudo-code of parallel PANE. Compared to the single-thread version, parallel PANE takes as input an additional parameter, the number of threads n_b, and randomly partitions the node set V, as well as the attribute set R, into n_b subsets with equal size, denoted as $\mathcal{V}$ and $\mathcal{R}$, respectively (Lines 1-2). PANE invokes PAPMI (Algorithm 6) at Line 4 to get $\mathbf{F}'$ and $\mathbf{B}'$, and then invokes PSVDCCD (Algorithm 8) to refine the embeddings.

Note that the parallel version of PANE does not return exactly the same outputs as the single-thread version, as some modules (*e.g.*, the parallel version of SVD) introduce additional error. Nevertheless, as the experiments in Section 5 demonstrates, the degradation of result utility in parallel PANE is small, but the speedup is significant.

4.1 Parallel Forward and Backward Affinity Approximation

We propose PAPMI in Algorithm 6 to estimate $\mathbf{F}'$ and $\mathbf{B}'$ in parallel. After obtaining $\mathbf{R}_r$ and $\mathbf{R}_c$ based on Equation (1) at Line 1, PAPMI divides $\mathbf{R}_r$ and $\mathbf{R}_c$ into matrix blocks according to two input parameters, the node subsets $\mathcal{V} = \{V_1, V_2, \cdots, V_{n_b}\}$ and attribute subsets $\mathcal{R} = \{R_1, R_2, \cdots, R_{n_b}\}$. Then, PAPMI parallelizes the matrix multiplications for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ from Line 2 to 6, using n_b threads in t iterations. Specifically, the i-th thread

Algorithm 6: PAPMI

```
Input: $\mathbf{P}, \mathbf{R}, \alpha, t, \mathcal{V}, \mathcal{R}$
Output: $\mathbf{F}', \mathbf{B}'$
1  Compute $\mathbf{R}_r$ and $\mathbf{R}_c$ by Equation (1);
2  parallel for $R_i \in \mathcal{R}$ do
3      ${\mathbf{P}_f}_i^{(0)} \leftarrow \mathbf{R}_r[:, R_i], {\mathbf{P}_b}_i^{(0)} \leftarrow \mathbf{R}_c[:, R_i]$;
4      for $\ell \leftarrow 1$ to $t$ do
5          ${\mathbf{P}_f}_i^{(\ell)} \leftarrow (1-\alpha) \cdot \mathbf{P}{\mathbf{P}_f}_i^{(\ell-1)} + \alpha \cdot {\mathbf{P}_f}_i^{(0)}$;
6          ${\mathbf{P}_b}_i^{(\ell)} \leftarrow (1-\alpha) \cdot \mathbf{P}^\top{\mathbf{P}_b}_i^{(\ell-1)} + \alpha \cdot {\mathbf{P}_b}_i^{(0)}$;
7  ${\mathbf{P}_f}^{(t)} \leftarrow [{\mathbf{P}_{f_1}}^{(t)} \cdots {\mathbf{P}_{f_{n_b}}}^{(t)}]$;
8  ${\mathbf{P}_b}^{(t)} \leftarrow [{\mathbf{P}_{b_1}}^{(t)} \cdots {\mathbf{P}_{b_{n_b}}}^{(t)}]$;
   Lines 9-10 are the same as Lines 6-7 in Algorithm 2;
11 parallel for $V_i \in \mathcal{V}$ do
12     $\mathbf{F}'[V_i] \leftarrow \log(n \cdot \widehat{\mathbf{P}}_f^{(t)}[V_i] + 1)$;
13     $\mathbf{B}'[V_i] \leftarrow \log(d \cdot \widehat{\mathbf{P}}_b^{(t)}[V_i] + 1)$;
14 return $\mathbf{F}', \mathbf{B}'$
```

Algorithm 7: SMGreedyInit

```
Input: $\mathbf{F}', \mathbf{B}', \mathcal{V}, k, t$.
Output: $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$.
1  parallel for $V_i \in \mathcal{V}$ do
2      $\mathbf{\Phi}, \mathbf{\Sigma}, \mathbf{V}_i \leftarrow$ RandSVD$(\mathbf{F}'[V_i], \frac{k}{2}, t)$;
3      $\mathbf{U}_i \leftarrow \mathbf{\Phi}\mathbf{\Sigma}$;
4  $\mathbf{V} \leftarrow \begin{bmatrix}\mathbf{V}_1 & \cdots & \mathbf{V}_{n_b}\end{bmatrix}^\top$;
5  $\mathbf{\Phi}, \mathbf{\Sigma}, \mathbf{Y} \leftarrow$ RandSVD$(\mathbf{V}, \frac{k}{2}, t)$;
6  $\mathbf{W} \leftarrow \mathbf{\Phi}\mathbf{\Sigma}$;
7  parallel for $V_i \in \mathcal{V}$ do
8      $\mathbf{X}_f[V_i] \leftarrow \mathbf{U}_i \cdot \mathbf{W}[(i-1) \cdot \frac{k}{2} : i \cdot \frac{k}{2}]$;
9      $\mathbf{X}_b[V_i] \leftarrow \mathbf{B}'[V_i] \cdot \mathbf{Y}$;
10     $\mathbf{S}_f[V_i] \leftarrow \mathbf{X}_f[V_i] \cdot \mathbf{Y}^\top - \mathbf{F}'[V_i]$;
11     $\mathbf{S}_b[V_i] \leftarrow \mathbf{B}'[V_i] - \mathbf{X}_b[V_i] \cdot \mathbf{Y}^\top$;
12 return $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$;
```

initializes ${\mathbf{P}_{f_i}}^{(0)}$ by $\mathbf{R}_r[:, R_i]$ and ${\mathbf{P}_{b_i}}^{(0)}$ by $\mathbf{R}_c[:, R_i]$ (Line 3), and then computes ${\mathbf{P}_f}_i^{(\ell)} = (1-\alpha) \cdot \mathbf{P}{\mathbf{P}_f}_i^{(\ell-1)} + \alpha \cdot {\mathbf{P}_f}_i^{(0)}$ and ${\mathbf{P}_b}_i^{(\ell)} = (1-\alpha) \cdot \mathbf{P}^\top{\mathbf{P}_b}_i^{(\ell-1)} + \alpha \cdot {\mathbf{P}_b}_i^{(0)}$ (Lines 4-6). Then, we use a main thread to aggregate the partial results of all threads at Lines 7-8. Specifically, n_b matrix blocks ${\mathbf{P}_{f_i}}^{(t)}$ (resp. ${\mathbf{P}_{b_i}}^{(t)}$) are concatenated horizontally together as ${\mathbf{P}_f}^{(t)}$ (resp. ${\mathbf{P}_b}^{(t)}$) at Line 7 (resp. Line 8). At Lines 9-10, we normalize $\widehat{\mathbf{P}}_f^{(t)}$ and $\widehat{\mathbf{P}}_b^{(t)}$ in the same way as Lines 6-7 in Algorithm 2. From Lines 11 to 13, PAPMI starts n_b threads to compute $\mathbf{F}'$ and $\mathbf{B}'$ block by block in parallel, based on the definitions of forward and backward affinity. Specifically, the i-th thread computes $\mathbf{F}'[V_i] = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)}[V_i] + 1)$ and $\mathbf{B}'[V_i] = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)}[V_i] + 1)$. Finally, PAPMI returns $\mathbf{F}'$ and $\mathbf{B}'$ as the approximate forward and backward affinity matrices (Line 14). Lemma 4.1 indicates the accuracy guarantee of PAPMI.

LEMMA 4.1. *Given same parameters* $\mathbf{P}, \mathbf{R}, \alpha$ *and* t *as inputs to Algorithm 2 and Algorithm 6, the two algorithms return the same approximate forward and backward affinity matrices* $\mathbf{F}', \mathbf{B}'$.

Algorithm 8: PSVDCCD

```
    Input: $\mathbf{F}', \mathbf{B}', \mathcal{V}, \mathcal{R}, k, t$.
    Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.
1   $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow$ SMGreedyInit$(\mathbf{F}', \mathbf{B}', \mathcal{V}, k, t)$;
2   for $\ell \leftarrow 1$ to $t$ do
3       parallel for $V_h \in \mathcal{V}$ do
4           for $v_i \in V_h$ do
                Lines 5-10 are the same as Lines 4-9 in Algorithm 4;
11      parallel for $R_h \in \mathcal{R}$ do
12          for $r_j \in R_h$ do
                Lines 13-16 are the same as Lines 11-14 in Algorithm 4;
17  return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

4.2 Parallel Joint Factorization of Affinity Matrices

This section presents the parallel algorithm PSVDCCD in Algorithm 8 to further improve the efficiency of the joint affinity matrix factorization process. At Line 1 of the algorithm, we design a parallel initialization algorithm SMGreedyInit with a split-and-merge-based parallel SVD technique for embedding vector initialization.

Algorithm 7 shows the pseudo-code of SMGreedyInit, which takes as input $\mathbf{F}'$, $\mathbf{B}'$, $\mathcal{V}$, and k. Based on $\mathcal{V}$, SMGreedyInit splits matrix $\mathbf{F}'$ into n_b blocks and launches n_b threads. Then, the i-th thread applies RandSVD to block $\mathbf{F}'[V_i]$ generated by the rows of $\mathbf{F}'$ based on node set $V_i \in \mathcal{V}$ (Line 1-3). After obtaining $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$, SMGreedyInit merges these matrices by concatenating $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$ into $\mathbf{V} = [\mathbf{V}_1 \ \cdots \ \mathbf{V}_{n_b}]^\top \in \mathbb{R}^{\frac{kn_b}{2} \times d}$, and then applies RandSVD over it to obtain $\mathbf{W} \in \mathbb{R}^{\frac{kn_b}{2} \times \frac{k}{2}}$ and $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$ (Lines 4-6). At Line 7, SMGreedyInit creates n_b threads, and uses the i-th thread to handle node subset V_i for initializing embedding vectors $\mathbf{X}_f[V_i]$ and $\mathbf{X}_b[V_i]$ at Lines 8-9, as well as computing $\mathbf{S}_f$ and $\mathbf{S}_b$ at Lines 10-11. Finally, SMGreedyInit returns initialized embedding vectors $\mathbf{Y}$, $\mathbf{X}_f$, and $\mathbf{X}_b$ as well as intermediate results $\mathbf{S}_f, \mathbf{S}_b$ at Line 12. Lemma 4.2 indicates that the initial embedding vectors produced by SMGreedyInit and GreedyInit are close.

After obtaining $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ by SMGreedyInit, Lines 2-16 in Algorithm 8 train embedding vectors by cyclic coordinate descent in parallel based on subsets $\mathcal{V}$ and $\mathcal{R}$, in t iterations. In each iteration, PSVDCCD first fixes $\mathbf{Y}$ and launches n_b threads to update $\mathbf{X}_f$ and $\mathbf{X}_b$ in parallel by blocks according to $\mathcal{V}$, and then updates $\mathbf{Y}$ using the n_b threads in parallel by blocks according to $\mathcal{R}$, with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed. Specifically, Lines 5-10 are the same as Lines 4-9 of Algorithm 4, and Lines 13-16 are the same as Lines 11-14 of Algorithm 4. Finally, Algorithm 8 returns embedding results at Line 17.

LEMMA 4.2. *Given same $\mathbf{F}', \mathbf{B}'$, k and t as inputs to Algorithm 3 and Algorithm 7, the outputs $\mathbf{X}_f, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$ returned by both algorithms satisfy that $\mathbf{X}_f \cdot \mathbf{Y}^\top = \mathbf{F}', \mathbf{Y}^\top\mathbf{Y} = \mathbf{I}$ and $\mathbf{S}_f = \mathbf{S}_b\mathbf{Y} = \mathbf{0}$, when $t = \infty$.*

4.3 Complexity Analysis

Observe that the non-parallel parts of Algorithms 6 (Lines 7-10) and 7 (Lines 4-6) take $O(nd)$ time, as each of them performs a constant number of operations on $O(nd)$ matrix entries. Meanwhile, for the parallel parts of Algorithms 6 and 8, each thread runs in $O\left(\frac{md}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$ and $O(\frac{ndkt}{n_b})$ time, respectively, since we divides the workload evenly to n_b threads. Specifically, each thread in Algorithm 6 runs in $O\left(\frac{md}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$ time. Algorithm 8 first takes $O(\frac{n}{n_b}dkt)$ time for each thread to factorize a $\frac{n}{n_b} \times d$ matrix block of $\mathbf{F}'$ (Lines 1-3 in Algorithm 7). In addition, Lines 4-6 in Algorithm 7 requires $O(n_b dk)$ time. In merge course (*i.e.*, Lines 7-11 in Algorithm 7), the matrix multiplications take $O(\frac{n}{n_b}k^2)$ time. In the t iterations of CCD (*i.e.*, Lines 2-16 in Algorithm 8), each thread spends $O(\frac{ndkt}{n_b})$ time to update. Thus, the computational time complexity per thread in Algorithm 5 is $O\left(\frac{md+ndk}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$. Algorithm 6 and Algorithm 8 require $O(m+nd)$ and $O(nd)$ space, respectively. Therefore, the space complexity of PANE is $O(m+nd)$.

5 EXPERIMENTS

We experimentally evaluate our proposed method PANE (both single-thread and parallel versions) against 10 competitors on three tasks: link prediction, attribute inference and node classification, using 8 real datasets. All experiments are conducted on a Linux machine powered by an Intel Xeon(R) E7-8880 v4@2.20GHz CPUs and 1TB RAM. The codes of all algorithms are collected from their respective authors, and all are implemented in Python, except NRP, TADW and LQANR. For fair comparison of efficiency, we re-implement TADW and LQANR in Python.

5.1 Experiments Setup

Table 3: Datasets. (K=10^3, M=10^6)

Name	$\lvert V\rvert$	$\lvert E_V\rvert$	$\lvert R\rvert$	$\lvert E_R\rvert$	$\lvert L\rvert$	Refs
Cora	2.7K	5.4K	1.4K	49.2K	7	[25, 27, 30, 41, 44, 51]
Citeseer	3.3K	4.7K	3.7K	105.2K	6	[25, 27, 30, 41, 44, 51]
Facebook	4K	88.2K	1.3K	33.3K	193	[24, 27, 45, 49]
Pubmed	19.7K	44.3K	0.5K	988K	3	[27, 30, 49, 51]
Flickr	7.6K	479.5K	12.1K	182.5K	9	[27]
Google+	107.6K	13.7M	15.9K	300.6M	468	[24, 45]
TWeibo	2.3M	50.7M	1.7K	16.8M	8	-
MAG	59.3M	978.2M	2K	434.4M	100	-

Datasets. Table 3 lists the statistics of the datasets used in our experiments. All graphs are directed except *Facebook* and *Flickr*. $|V|$ and $|E_V|$ denote the number of nodes and edges in the graph, whereas $|R|$ and $|E_R|$ represent the number of attributes and the number of node-attribute associations (*i.e.*, the number of nonzero entries in attribute matrix $\mathbf{R}$). In addition, L is the set of *node labels*, which are used in the node classification task. *Cora*[2], *Citeseer*[2], *Pubmed*[2] and *Flickr*[3] are benchmark datasets used in prior work [15, 25, 27, 30, 41, 51]. *Facebook*[4] and *Google+*[4] are social networks used in [24]. For *Facebook* and *Google+*, we treat each ego-network as a label and extract attributes from their user profiles, which is consistent with the experiments in prior work [27, 45]

To evaluate the scalability of the proposed solution, we also introduce two new datasets *TWeibo*[5] and *MAG*[6] that have not been

[2]http://linqs.soe.ucsc.edu/data (accessed September 18, 2020)
[3]https://github.com/mengzaiqiao/CAN (accessed September 18, 2020)
[4]http://snap.stanford.edu/data (accessed September 18, 2020)
[5]https://www.kaggle.com/c/kddcup2012-track1 (accessed September 18, 2020)
[6]http://ma-graph.org/rdf-dumps/ (accessed September 18, 2020)

Table 4: Attribute inference performance.

Method	*Cora*		*Citeseer*		*Facebook*		*Pubmed*		*Flickr*		*Google+*		*TWeibo*		*MAG*	
	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP
BLA	0.559	0.563	0.540	0.541	0.653	0.648	0.520	0.524	0.660	0.653	-	-	-	-	-	-
CAN	0.865	0.855	0.875	0.859	0.765	0.745	0.734	0.72	0.772	0.774	-	-	-	-	-	-
PANE (single thread)	0.913	0.925	0.903	0.916	0.828	0.84	0.871	0.874	0.825	0.832	0.972	0.973	0.774	0.837	0.876	0.888
PANE (parallel)	0.909	0.92	0.899	0.913	0.825	0.837	0.867	0.869	0.822	0.831	0.969	0.97	0.773	0.836	0.874	0.887

Table 5: Link prediction performance.

Method	*Cora*		*Citeseer*		*Pubmed*		*Facebook*		*Flickr*		*Google+*		*TWeibo*		*MAG*	
	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP	AUC	AP
NRP	0.796	0.777	0.86	0.808	0.87	0.861	0.969	0.973	0.909	0.902	0.989	0.992	0.967	0.979	0.915	0.92
GATNE	0.791	0.822	0.687	0.767	0.745	0.796	0.961	0.954	0.805	0.785	-	-	-	-	-	-
TADW	0.829	0.805	0.895	0.868	0.904	0.863	0.752	0.793	0.573	0.58	-	-	-	-	-	-
ARGA	0.64	0.485	0.637	0.484	0.623	0.474	0.71	0.636	0.676	0.656	-	-	-	-	-	-
BANE	0.875	0.823	0.899	0.873	0.919	0.847	0.796	0.795	0.64	0.605	0.56	0.533	-	-	-	-
PRRE	0.879	0.836	0.895	0.855	0.887	0.813	0.899	0.884	0.789	0.806	-	-	-	-	-	-
STNE	0.808	0.829	0.71	0.781	0.789	0.774	0.962	0.957	0.638	0.659	-	-	-	-	-	-
CAN	0.663	0.559	0.734	0.652	0.734	0.559	0.714	0.639	0.5	0.5	-	-	-	-	-	-
DGI	0.51	0.4	0.5	0.4	0.73	0.554	0.711	0.637	0.769	0.824	0.792	0.795	0.721	0.64	-	-
LQANR	0.886	0.863	0.916	0.916	0.904	0.8	0.951	0.917	0.824	0.805	-	-	-	-	-	-
PANE (single thread)	0.933	0.918	0.932	0.919	0.985	0.977	0.982	0.982	0.929	0.927	0.987	0.982	0.976	0.986	0.96	0.965
PANE (parallel)	0.929	0.914	0.929	0.916	0.985	0.976	0.98	0.979	0.927	0.924	0.984	0.98	0.975	0.985	0.958	0.962

used in previous ANE papers due to their massive sizes. *TWeibo* [21] is a social network, in which each node represents a user, and each directed edge represents a following relationship. We extract the 1657 most popular tags and keywords from its user profile data as the node attributes. The labels are generated and categorized into eight types according to the ages of users. *MAG* dataset is extracted from the well-known *Microsoft Academic Knowledge Graph* [34], where each node represents a paper and each directed edge represents a citation. We extract 2000 most frequently used distinct words from the abstract of all papers as the attribute set and regard the fields of study of each paper as its labels. We will make *TWeibo* and *MAG* datasets publicly available upon acceptance.

Baselines and Parameter Settings. We compare our methods PANE (single thread) and PANE (parallel) against 10 state-of-the-art competitors: eight recent ANE methods including BANE [44], CAN [27], STNE [25], PRRE [51], TADW [41], ARGA [30], DGI [37] and LQANR [43], one state-of-the-art homogeneous network embedding method NRP [46], and one latest attributed heterogeneous network embedding algorithm GATNE [3]. All methods except PANE (parallel) run on a single CPU core. Note that although GATNE itself is a parallel algorithm, its parallel version requires the proprietary AliGraph platform which is not available to us.

The parameters of all competitors are set as suggested in their respective papers. For PANE (single thread) and PANE (parallel), by default we set error threshold $\epsilon = 0.015$ and random walk stopping probability $\alpha = 0.5$, and we use $n_b = 10$ threads for PANE (parallel). Unless otherwise specified, we set space budget $k = 128$.

The evaluation results of our proposed methods against the competitors for attribute inference, link prediction and node classification, are reported in Sections 5.2, 5.3 and 5.4 respectively. The efficiency and scalability evaluation is reported in Section 5.5. A method will be excluded if it cannot finish training within one week. Due to space limitations, we omit the results of evaluating the impact of GreedyInit and varying the parameters of PANE here, and please see our technical report [47] if interested.

5.2 Attribute Inference

Attribute inference aims to predict the values of attributes of nodes. Note that, except for CAN [27], none of the the other competitors is capable of performing attribute inference, since they only generate embedding vectors for nodes, not attributes. Hence, we compare our solutions against CAN for attribute inference. Further, we compare against BLA, the state-of-the-art attribute inference algorithm [42]. Note that BLA is not an ANE solution.

We split the nonzero entries in the attribute matrix $\mathbf{R}$, and regard 20% as the test set $\mathbf{R}_{test}$ and the remaining 80% part as the training set $\mathbf{R}_{train}$. CAN runs over $\mathbf{R}_{train}$ to generate node embedding vector $\mathbf{X}[v_i]$ for each node $v_i \in V$ and attribute embedding vector $\mathbf{Y}[r_j]$ for each attribute $r_j \in R$. Following [27], we use the inner product of $\mathbf{X}[v_i]$ and $\mathbf{Y}[r_j]$ as the predicted score of attribute r_j with respect to node v_i. Note that PANE generates a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$, and also an attribute embedding vector $\mathbf{Y}[r_j]$ for each attribute $r_j \in R$. Based on objective function in Equation (4), $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top$ is expected to preserve forward affinity value $\mathbf{F}[v_i, r_j]$, and $\mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top$ is expected to preserve backward

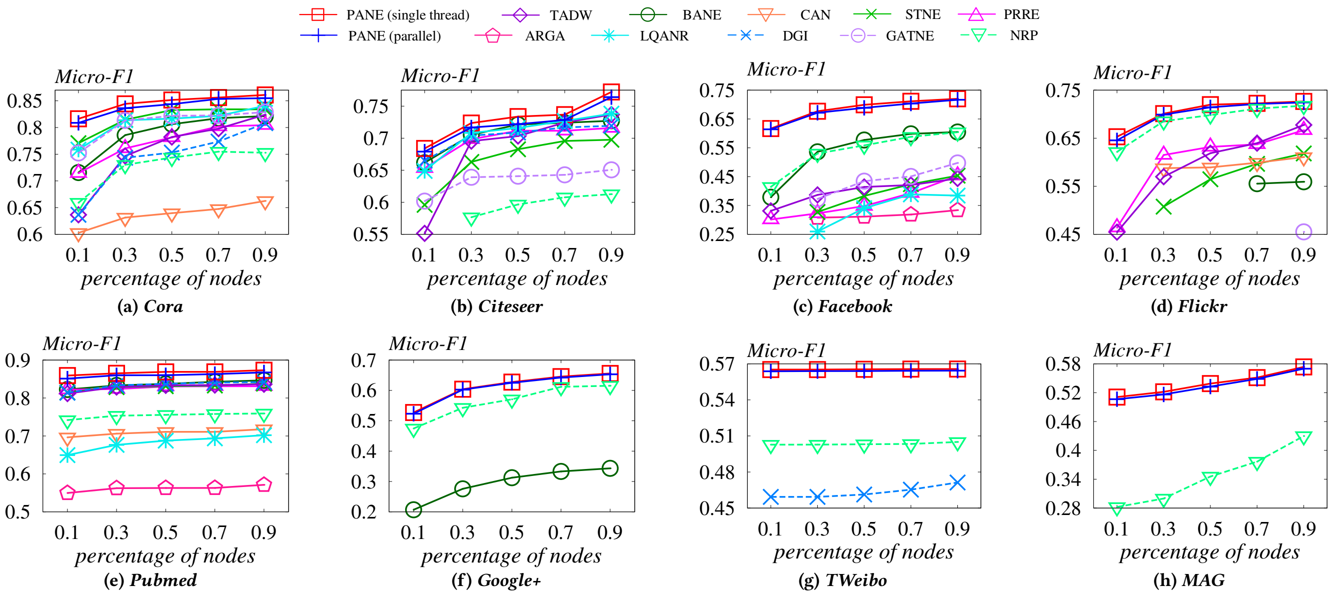


Figure 2: Node classification results (best viewed in color).

affinity value $\mathbf{B}[v_i, r_j]$. Thus, we predict the score between v_i and r_j through the affinity between node v_i and attribute r_j, including both forward affinity and backward affinity, denoted as $p(v_i, r_j)$, by utilizing their embedding vectors as follows.

$$\begin{aligned} p(v_i, r_j) &= \mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top + \mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top \\ &\approx \mathbf{F}[v_i, r_j] + \mathbf{B}[v_i, r_j]. \end{aligned} \quad (21)$$

Following prior work [27], we adopt the *Area Under Curve* (AUC) and *Average Precision* (AP) metrics to measure the performance.

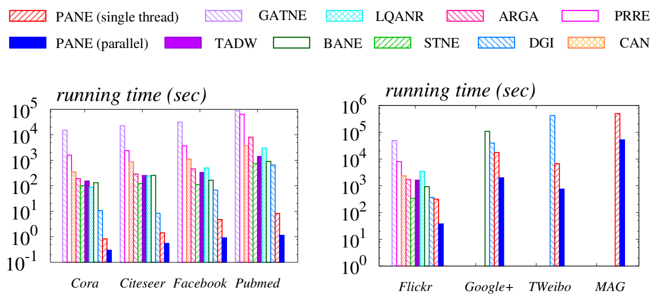
Table 4 presents the attribute inference performance of PANE (single thread), PANE (parallel), CAN and BLA. Observe that PANE (single thread) consistently achieves the best performance on all datasets and significantly outperforms existing solutions by a large margin, demonstrating the power of forward affinity and backward affinity that are preserved in embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$, to capture the affinity between nodes and attributes in attributed networks. For instance, on *Pubmed*, PANE (single thread) has high accuracy 0.871 AUC and 0.874 AP, while that of CAN are only 0.734 and 0.72 respectively. Further, CAN and BLA fail to process large attributed networks *Google+*, *TWeibo* and *MAG* in one week, and, thus are not reported. Observe that parallel PANE has close performance (*i.e.*, AUC and AP) to that of PANE (single thread). For instance, on *Pubmed*, the difference of AUC between PANE (single thread) and PANE (parallel) is just 0.004. This negligible difference is introduced by the split-merge-based parallel SVD technique SMGreedyInit for matrix decomposition. As shown in Section 5.5, parallel PANE is considerably faster than PANE (single thread) by up to 9 times, while obtaining almost the same accuracy performance.

5.3 Link Prediction

Link prediction aims to predict the edges that are most likely to form between nodes. We first randomly remove 30% edges in input graph G, obtaining a residual graph G' and a set of the removed edges. We then randomly sample the same amount of non-existing edges as negative edges. The test set E' contains both the removed edges and the negative edges. We run PANE and all competitors on the residual graph G' to produce embedding vectors, and then evaluate the link prediction performance with E' as follows. PANE produces the a forward embedding $\mathbf{X}_f[v_i]$ and a backward embedding $\mathbf{X}_b[v_i]$ for each node $v_i \in V$, as well as an attribute embedding $\mathbf{Y}[r_l]$ for each attribute $r_l \in R$. As explained, $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top$ preserves $\mathbf{F}[v_i, r_l]$, and $\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top$ preserves $\mathbf{B}[v_j, r_l]$. Recall that $\mathbf{F}[v_i, r_l]$ measures the affinity from v_i to r_l over the attributed network; similarly given node v_j and attribute r_l, $\mathbf{B}[v_j, r_l]$ measures the affinity from r_l to v_j over the network. Intuitively, $\mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]$ represents the affinity from node v_i to node v_j based on attribute r_l. The affinity between nodes v_i and v_j, denoted as $p(v_i, v_j)$, can be evaluated by summing up the affinity between the two nodes over all attributes in R, which can be computed as follows and indicates the possibility of forming an edge from v_i to v_j.

$$\begin{aligned} p(v_i, v_j) &= \textstyle\sum_{r_l \in R} (\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top) \cdot (\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top) \\ &\approx \textstyle\sum_{r_l \in R} \mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]. \end{aligned} \quad (22)$$

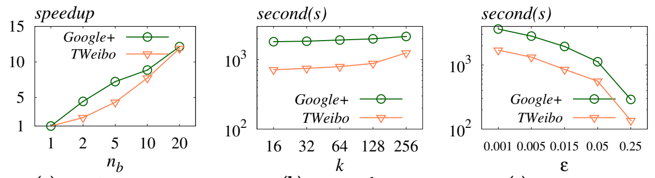
Therefore, for PANE, we can calculate $p(v_i, v_j)$ as the prediction score of the directed edge (v_i, v_j). NRP generates a forward embedding $\mathbf{X}_f[v_i]$ and a backward embedding $\mathbf{X}_b[v_i]$ for each node v_i and uses $p(v_i, v_j) = \mathbf{X}_f[v_i] \cdot \mathbf{X}_b[v_i]^\top$ as the prediction score for the directed edge (v_i, v_j) [46]. For undirected graphs, PANE (single thread), PANE (parallel) and NRP utilize $p(v_i, v_j) + p(v_j, v_i)$ as the prediction score for the undirected edge between v_i and v_j. In terms of the remaining competitors that only work for undirected graphs, they learn one embedding $\mathbf{X}[v_i]$ for each node v_i. In literature, there are four ways to calculate the link prediction score $p(v_i, v_j)$, including *inner product* method used in CAN and ARGA, *cosine similarity* method used in PRRE and ANRL, *Hamming distance* method used in BANE, as well as *edge feature* method used in [14, 26]. We adopt all these four prediction methods over each competitor and report



(a) small graphs.

(b) large graphs.

Figure 3: Running time (best viewed in color).



(a) speedups vs. n_b. **(b) time vs. k.** **(c) time vs. ϵ.**

Figure 4: Efficiency with varying parameters.

the competitor's best performance on each dataset. Following previous work [27, 30], we use *Area Under Curve* (AUC) and *Average Precision* (AP) to evaluate link prediction accuracy.

Table 5 reports the AUC and AP scores of each method on each dataset. PANE (single thread) consistently outperforms all competitors over all datasets except NRP on *Google+*, by a substantial margin of up to 6.6% for AUC and up to 13% for AP. For large attributed networks including *Google+, TWeibo* and *MAG*, most existing solutions fail to finish processing within a week and thus are not reported. The superiority of PANE (single thread) over competitors is achieved by (i) learning a forward embedding vector and a backward embedding vector for each node to capture the asymmetric transitivity (*i.e.*, edge direction) in directed graphs, and (ii) combining both node embedding vectors and attribute embedding vectors together for link prediction in Equation (22), with the consideration of both topological and attribute features. On *Google+*, NRP is slightly better than PANE (single thread), since *Google+* has more than 15 thousand attributes (see Table 3), leading to some accuracy loss when factorizing forward and backward affinity matrices into low dimensionality $k = 128$ by PANE (single thread). As shown in Table 5, our parallel PANE also outperforms all competitors significantly except NRP on *Google+*, and parallel PANE has comparable performance with PANE (single thread) over all datasets. As reported later in Section 5.5, parallel PANE is significantly faster than PANE (single thread) by up to 9 times, with almost the same accuracy performance for link prediction.

5.4 Node Classification

Node classification predicts the node labels. Note that *Facebook*, *Google+* and *MAG* are multi-labelled, meaning that each node can have multiple labels. We first run PANE (single thread), PANE and the competitors on the input attributed network G to obtain their embeddings. Then we randomly sample a certain number of nodes (ranging from 10% to 90%) to train a linear support-vector machine (SVM) classifier [6] and use the rest for testing. NRP, PANE (single thread), and PANE generate a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$. So we normalize the forward and backward embeddings of each node v_i, and then concatenate them as the feature representation of v_i to be fed into the classifier. Akin to prior work [17, 27, 43], we use Micro-F1 and Macro-F1 to measure node classification performance. We repeat for 5 times and report the average performance.

Figure 2 shows the Micro-F1 results when varying the percentage of nodes used for training from 10% to 90% (*i.e.*, 0.1 to 0.9). The results of Macro-F1 are similar and thus omitted for brevity. Both versions of PANE consistently outperform all competitors on all datasets, which demonstrates that our proposed solutions effectively capture the topology and attribute information of the input attributed networks. Specifically, compared with the competitors, PANE (single thread) achieves a remarkable improvement, up to 3.7% on *Cora*, *Citeseer*, *Pubmed* and *Flickr*, and up to 11.6% on *Facebook*. On the large graphs *Google+*, *TWeibo* and *MAG*, most existing solutions fail to finish within a week and thus their results are omitted. Furthermore, PANE (single thread) outperforms NRP by at least 3.4% and 6% on *Google+* and *TWeibo* as displayed in Figures 2f and 2g, respectively. In addition, PANE (single thread) and PANE (parallel) are superior to NRP with a significant gain up to 17.2% on *MAG*. Over all datasets, PANE (parallel) has similar performance to that of PANE (single thread), while as shown in Section 5.5, PANE (parallel) is significantly faster than PANE (single thread).

5.5 Efficiency and Scalability

Figure 3a and Figure 3b together report the running time required by each method on all datasets. The y-axis is the running time (seconds) in log-scale. The reported running time does not include the time for loading datasets and outputting embedding vectors. We omit any methods with processing time exceeding one week.

Both versions of PANE are significantly faster than all ANE competitors, often by orders of magnitude. For instance, on *Pubmed* in Figure 3a, PANE takes 1.1 seconds and PANE (single thread) requires 8.2 seconds, while the fastest ANE competitor TADW consumes 405.3 seconds, meaning that PANE (single thread) (resp. PANE) is 49× (resp. 368×) faster. On large attributed networks including *Google+*, *TWeibo*, and *MAG*, most existing ANE solutions cannot finish within a week, while our proposed solutions PANE (single thread) and PANE are able to handle such large-scale networks efficiently. PANE is up to 9 times faster than PANE (single thread) over all datasets. For instance, on *MAG* dataset that has 59.3 million nodes, when using 10 threads, PANE requires 11.9 hours while PANE (single thread) costs about five days, which indicates the power of our parallel techniques in Section 4.

Figure 4a displays the speedups of parallel PANE over single-thread version on *Google+* and *TWeibo* when varying the number of threads n_b from 1 to 20. When n_b increases, parallel PANE becomes much faster than single-thread PANE, demonstrating the parallel scalability of PANE with respect to the n_b. Figure 4b and Figure 4c illustrate the running time of PANE when varying space budget k from 16 to 256 and error threshold ϵ from 0.001 to 0.25, respectively. In Figure 4b, when k is increased from 16 to 256, the running time is quite stable and goes up slowly, showing the efficiency robustness of our solution. In Figure 4c, the running time of PANE decreases considerably when increasing ϵ in $\{0.001, 0.005, 0.015, 0.05, 0.25\}$. When ϵ increases from 0.001 to 0.25, the running time on *Google+* and *TWeibo* reduces by about 10 times, which is consistent with our analysis that PANE runs in linear to $\log(1/\epsilon)$ in Section 4.

6 RELATED WORK

Factorization-based methods. Given an attributed network G with n nodes, existing factorization-based methods mainly involve two stages: (i) build a proximity matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ that models the proximity between nodes based on graph topology or attribute information; (ii) factorize $\mathbf{M}$ via techniques such as SGD [2], ALS [5], and coordinate descent [38]. Specifically, TADW [41] constructs a second-order proximity matrix $\mathbf{M}$ based on the adjacency matrix of G, and aims to reconstruct $\mathbf{M}$ by the product of the learned embedding matrix and the attribute matrix. HSCA [48] ensures that the learned embeddings of connected nodes are close in the embedding space. AANE [18] constructs a proximity matrix $\mathbf{M}$ using the cosine similarities between the attribute vectors of nodes. BANE [44] learns a binary embedding vector per node, *i.e.*, $\{-1, 1\}^k$, by minimizing the reconstruction loss of a unified matrix that incorporates both graph topology and attribute information. BANE reduces space overheads at the cost of accuracy. To further balance the trade-off between space cost and representation accuracy, LQANR [43] learns embeddings $\in \{-2^b, \cdots, -1, 0, 1, \cdots, 2^b\}^k$, where b is the bit-width. All these factorization-based methods incur immense overheads in building and factorizing the $n \times n$ proximity matrix. Further, these methods are designed for undirected graphs only.

Auto-encoder-based methods. An auto-encoder [12] is a neural network model consisting of an encoder that compresses the input data to obtain embeddings and a decoder that reconstructs the input data from the embeddings, with the goal to minimize the reconstruction loss. Existing methods either use different proximity matrices as inputs or design various neural network structures for the auto-encoder. Specifically, ANRL [49] combines auto-encoder with SkipGram model to learn embeddings. DANE [8] designs two auto-encoders to reconstruct the high-order proximity matrix and the attribute matrix respectively. ARGA [30] integrates auto-encoder with graph convolutional networks [22] and generative adversarial networks [13] together. STNE [25] samples nodes via random walks and feeds the attribute vectors of the sampled nodes into a LSTM-based auto-encoder [16]. NetVAE [20] compresses the graph structures and node attributes with a shared encoder for transfer learning and information integration. CAN [27] embeds both nodes and attributes into two Gaussian distributions using a graph convolutional network and a dense encoder. None of these auto-encoder-based methods considers edge directions. Further, they suffer from severe efficiency issues due to the expensive training process of auto-encoders. SAGE2VEC [33] proposes an enhanced auto-encoder model that preserves global graph structure and meanwhile handles the non-linearity and sparsity of both graph structures and attributes. AdONE [1] designs an auto-encoder model for detecting and minimizing the effect of community outliers while generating embeddings.

Other methods. PRRE [51] categorizes node relationships into positive, ambiguous and negative types, according to the graph and attribute proximities between nodes, and then employs Expectation Maximization (EM) [7] to learn embeddings. SAGE [15] samples and aggregates features from a node's local neighborhood and learns embeddings by LSTM and pooling. NetHash [40] builds a rooted tree for each node by expanding along the neighborhood of the node, and then recursively sketches the rooted tree to get a summarized attribute list as the embedding vector of the node. PGE [17] groups nodes into clusters based on their attributes, and then trains neural networks with biased neighborhood samples in clusters to generate embeddings. ProGAN [9] adopts generative adversarial networks to generate node proximities, followed by neural networks to learn node embeddings from the generated node proximities. DGI [37] derives embeddings via graph convolutional networks, such that the mutual information between the embeddings for nodes and the embedding vector for the whole graph is maximized. MARINE [39] preserves the long-range spatial dependencies between nodes into embeddings by minimizing the information discrepancy in a Reproducing Kernel Hilbert Space.

Recently, there are embedding studies on attributed heterogeneous networks that consist of not only graph topology and node attributes, but also node types and edge types. When there are only one type of node and one type of edge, these methods effectively work on attributed networks. For instance, Alibaba proposed GATNE [3], to process attributed heterogeneous network embedding. For each node on every edge type, it learns an embedding vector, by using SkipGram model and random walks over the attributed heterogeneous network. Then it obtains the overall embedding vector for each node by concatenating the embeddings of the node over all edge types. GATNE incurs expensive training overheads and highly relies on the power of distributed systems. In addition, there are many studies on homogeneous network embedding, which purely focuses on graph topology without considering attributes, as reviewed in our technical report [47].

7 CONCLUSION

This paper presents PANE, an effective solution for ANE computation that scales to massive graphs with tens of millions of nodes, while obtaining state-of-the art result utility. The high scalability and effectiveness of PANE are mainly due to a novel problem formulation based on a random walk model, a highly efficient and sophisticated solver, and non-trivial parallelization. Extensive experiments show that PANE achieves substantial performance enhancements over state-of-the-arts in terms of both efficiency and result utility. Regarding future work, we plan to further develop GPU / multi-GPU versions of PANE, and adapt PANE to heteogeneous graphs, as well as time-varying graphs where attributes and node connections change over time.

ACKNOWLEDGMENTS

This work is supported by the National University of Singapore under SUG grant R-252-000-686-133, by NPRP grant #NPRP10-0208-170408 from the Qatar National Research Fund (a member of Qatar Foundation). Jieming Shi is supported by the financial support (1-BE3T) of research project (P0033898) from the Hong Kong Polytechnic University. The findings herein reflect the work, and are solely the responsibility, of the authors.

REFERENCES

[1] Sambaran Bandyopadhyay, Saley Vishal Vivek, and MN Murty. 2020. Outlier Resistant Unsupervised Deep Architectures for Attributed Network Embedding. In *WSDM*. 25–33.

[2] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*. 177–186.

[3] Yukuo Cen, Xu Zou, Jianwei Zhang, Hongxia Yang, Jingren Zhou, and Jie Tang. 2019. Representation learning for attributed multiplex heterogeneous network. In *SIGKDD*. 1358–1368.

[4] Kenneth Ward Church and Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational linguistics* (1990), 22–29.

[5] Pierre Comon, Xavier Luciani, and André LF De Almeida. 2009. Tensor decompositions, alternating least squares and other tales. *J. Chemom.* (2009), 393–405.

[6] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[7] AP Dempster, NM Laird, and DB Rubin. 1977. Maximum Likelihood from Incomplete Data via the *EM* Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)* 39, 1 (1977), 1–38.

[8] Hongchang Gao and Heng Huang. 2018. Deep Attributed Network Embedding.. In *IJCAI*.

[9] Hongchang Gao, Jian Pei, and Heng Huang. 2019. ProGAN: Network Embedding via Proximity Generative Adversarial Network. In *KDD*.

[10] Gene H Golub and Christian Reinsch. 1971. Singular value decomposition and least squares solutions. *Linear Algebra* (1971), 134–151.

[11] Gene H Golub and Charles F Van Loan. 1996. Matrix computations. 1996. *Johns Hopkins University, Press, Baltimore, MD, USA* (1996).

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *NeurIPS*. 2672–2680.

[14] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* (1997), 1735–1780.

[17] Yifan Hou, Hongzhi Chen, Changji Li, James Cheng, and Ming-Chang Yang. 2019. A Representation Learning Framework for Property Graphs. In *KDD*.

[18] Xiao Huang, Jundong Li, and Xia Hu. 2017. Accelerated attributed network embedding. In *SDM*.

[19] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *WWW*. 271–279.

[20] Di Jin, Bingyi Li, Pengfei Jiao, Dongxiao He, and Weixiong Zhang. 2019. Network-Specific Variational Auto-Encoder for Embedding in Attribute Networks. In *IJCAI*.

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

[22] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *ICLR* (2016).

[23] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *SysML*.

[24] Jure Leskovec and Julian J Mcauley. 2012. Learning to discover social circles in ego networks. In *NeurIPS*. 539–547.

[25] Jie Liu, Zhicheng He, Lai Wei, and Yalou Huang. 2018. Content to node: Self-translation network embedding. In *KDD*.

[26] Jianxin Ma, Peng Cui, Xiao Wang, and Wenwu Zhu. 2018. Hierarchical taxonomy aware network embedding. In *KDD*.

[27] Zaiqiao Meng, Shangsong Liang, Hongyan Bao, and Xiangliang Zhang. 2019. Co-embedding Attributed Networks. In *WSDM*.

[28] Zaiqiao Meng, Shangsong Liang, Xiangliang Zhang, Richard McCreadie, and Iadh Ounis. 2020. Jointly Learning Representations of Nodes and Attributes for Attributed Networks. *TOIS* (2020), 1–32.

[29] Cameron Musco and Christopher Musco. 2015. Randomized block krylov methods for stronger and faster approximate singular value decomposition. In *NeurIPS*.

[30] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. 2018. Adversarially Regularized Graph Autoencoder for Graph Embedding.. In *IJCAI*.

[31] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *SIGKDD*. 701–710.

[32] Gerard Salton and Michael J McGill. 1986. Introduction to Modern Information Retrieval.

[33] Nasrullah Sheikh, Zekarias T Kefato, and Alberto Montresor. 2019. A Simple Approach to Attributed Graph Embedding via Enhanced Autoencoder. In *Complex Networks*. Springer, 797–809.

[34] 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*. 243–246.

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

[36] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *ICDM*. IEEE, 613–622.

[37] Petar Veličković, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2019. Deep Graph Infomax. In *ICLR*.

[38] Stephen J Wright. 2015. Coordinate descent algorithms. *Mathematical Programming* (2015), 3–34.

[39] Jun Wu and Jingrui He. 2019. Scalable manifold-regularized attributed network embedding via maximum mean discrepancy. In *CIKM*. 2101–2104.

[40] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. 2018. Efficient Attributed Network Embedding via Recursive Randomized Hashing.. In *IJCAI*.

[41] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Chang. 2015. Network representation learning with rich text information. In *IJCAI*.

[42] Carl Yang, Lin Zhong, Li-Jia Li, and Luo Jie. 2017. Bi-directional joint inference for user links and attributes on large social graphs. In *WWW*.

[43] Hong Yang, Shirui Pan, Ling Chen, Chuan Zhou, and Peng Zhang. 2019. Low-Bit Quantization for Attributed Network Representation Learning. In *IJCAI*.

[44] Hong Yang, Shirui Pan, Peng Zhang, Ling Chen, Defu Lian, and Chengqi Zhang. 2018. Binarized attributed network embedding. In *ICDM*.

[45] Jaewon Yang, Julian McAuley, and Jure Leskovec. 2013. Community detection in networks with node attributes. In *ICDM*.

[46] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. In *PVLDB*. 670–683.

[47] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S. Bhowmick. 2020. Scaling Attributed Network Embedding to Massive Graphs. *arXiv preprint* (2020).

[48] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. 2016. Homophily, structure, and content augmented network representation learning. In *ICDM*.

[49] Zhen Zhang, Hongxia Yang, Jiajun Bu, Sheng Zhou, Pinggang Yu, Jianwei Zhang, Martin Ester, and Can Wang. 2018. ANRL: Attributed Network Representation Learning via Deep Neural Networks.. In *IJCAI*.

[50] Chang Zhou, Yuqiong Liu, Xiaofei Liu, Zhongyi Liu, and Jun Gao. 2017. Scalable graph embedding for asymmetric proximity. In *AAAI*.

[51] Sheng Zhou, Hongxia Yang, Xin Wang, Jiajun Bu, Martin Ester, Pinggang Yu, Jianwei Zhang, and Can Wang. 2018. PRRE: Personalized Relation Ranking Embedding for Attributed Networks. In *CIKM*.

[52] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *PVLDB* (2019), 2094–2105.