The VLDB Journal (2023) 32:1237–1262
https://doi.org/10.1007/s00778-023-00790-4

SPECIAL ISSUE PAPER

PANE: scalable and effective attributed network embedding

Renchi Yang[1] · Jieming Shi[2] · Xiaokui Xiao[3] · Yin Yang[4] · Sourav S. Bhowmick[5] · Juncheng Liu[3]

Received: 1 April 2022 / Revised: 22 February 2023 / Accepted: 2 March 2023 / Published online: 24 March 2023

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. 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. The performance of PANE depends upon the number of attributes in the input network. To handle large networks with numerous attributes, we further extend PANE to PANE^{++}, which employs an effective attribute clustering technique. Extensive experiments, comparing 10 existing approaches on 8 real datasets, demonstrate that PANE and PANE^{++} consistently outperform all existing methods in terms of result quality, while being orders of magnitude faster.

Keywords Network embedding · Attributed graph · Random walk · Matrix factorization · Scalability

1 Introduction

Network embedding is a fundamental task for graph analytics, which has attracted much attention from both academia (e.g., [26,58,69]) and industry (e.g., [41,105]). Given an input graph or network 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

Work was done when the first author was a doctoral student at NTU, Singapore, and a research fellow at NUS.

✉ Sourav S. Bhowmick
assourav@ntu.edu.sg

Renchi Yang
renchi@hkbu.edu.hk

Jieming Shi
jieming.shi@polyu.edu.hk

Xiaokui Xiao
xkxiao@nus.edu.sg

Yin Yang
yyang@hbku.edu.qa

Juncheng Liu
juncheng@comp.nus.edu.sg

1 Hong Kong Baptist University, Kowloon Tong, Hong Kong

2 The Hong Kong Polytechnic University, Hung Hom, Hong Kong

3 National University of Singapore, Singapore, Singapore

4 Hamad Bin Khalifa University , Al Rayyan, Qatar

5 Nanyang Technological University (NTU) , Singapore, Singapore

attributes as separate features, doing so loses the important information of *node-attribute affinity* [53], 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) 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* (ANE) 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 correspond 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 Sect. 8, one class of previous methods, e.g., [34,84,89,98], explicitly constructs and factorizes 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 in practice. Another category of methods, e.g., [18,47,57,101], employs 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., 80GB on Nvidia's flagship H100 cards. Thus, for massive graphs, currently the only practical option is to compute ANE leveraging a large cluster, e.g., [105], which is not only expensive but may also have significant environmental impact.

In addition, to the best of our knowledge, all existing ANE solutions are designed for undirected graphs. In particular, it is unclear how to incorporate edge direction information (e.g., asymmetric transitivity [103]) into their resulting embeddings. In practice, many graphs are directed and existing methods yield suboptimal result quality on such graphs as shown later in our experimental study. *Can we compute effective embeddings of a massive, attributed, directed graph on a single server?*

This paper provides an affirmative answer to the above question by presenting a novel framework coined PANE.[1] It incorporates several variants: a single-thread version Seq-PANE, a parallel version Par-PANE optimized for multi-core CPUs, and a version called PANE^{++} designed to handle networks with numerous attributes. PANE significantly advances the state of the art in ANE computation. Specifically, as demonstrated in our experiments in Sect. 7, the embeddings obtained by PANE simultaneously achieve the highest prediction accuracy compared to existing methods for three common graph analytics tasks, namely attribute inference, link prediction, and node classification, on common benchmark graph datasets. On the largest *Microsoft Academic Knowledge Graph* (*MAG*) dataset involving tens of millions of nodes, a billion edges, millions of distinct attributes (in the *MAG-SC* variant of the dataset), and over a billion node-attribute associations, PANE is the only viable solution on a single server (10 CPU cores, 1TB memory) whose resulting embeddings lead to superior prediction accuracy for all tasks.

PANE achieves effective and scalable ANE computation through four main contributions: (i) a well-thought-out problem formulation based on a novel random walk model, (ii) a highly efficient solver, (iii) non-trivial parallelization of the algorithm (i.e., Par-PANE), and (iv) an effective attribute clustering technique capable of handling networks with a large attribute set (i.e., PANE^{++}). Specifically, as presented in Sect. 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 [53], guided by a *shifted* pairwise mutual information (SPMI) metric. 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. Further, we devise a non-trivial parallelization of the PANE algorithm that utilizes modern multi-core CPUs without significantly compromising the result utility.

For networks with numerous attributes and/or node-attribute associations, the PANE^{++} variant of the proposed solution exploits an effective attribute clustering algorithm that groups similar attributes into *super attributes* to significantly reduce the computational overhead while retaining the high result quality of the obtained embeddings. 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

[1] The work reported here is an extended version of [92,93].

classification, at a fraction of the cost compared to existing methods.

In summary, our main contributions 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 a non-trivial parallelization technique of PANE (*i.e.*, Par-PANE) to further boost efficiency on multi-core CPUs.
- We augment PANE to PANE^{++} with an effective attribute clustering technique that scales well to a massive attribute set associated with the input network.
- We experimentally demonstrate the superior performance of PANE and PANE^{++}, in terms of both effectiveness and efficiency, against 10 competitors on 8 real datasets.

The rest of the paper is organized as follows. We formally define the ANE problem addressed in this paper in Sect. 2. We introduce the sequential and parallel versions of PANE in Sects. 3 and 4, respectively. Section 4 presents the extension of PANE to effectively handle large attribute set associated with an input network. We discuss how the embeddings generated by our proposed techniques can be exploited by representative machine learning tasks in Sect. 6. We report exhaustive performance study of our proposed techniques in Sect. 7. Related work is discussed in Sect. 8. The last section concludes the paper.

2 Problem formulation

In this section, we formally define the problem addressed in this paper. We begin by introducing the notations and terminology used in this work. Next, we introduce the notion of *node-attribute affinity*. Finally, we formally define the ANE problem by exploiting the notion of node-attribute affinity.

2.1 Notations and terminology

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 with cardinality 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 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 [53], we also allocate a space budget $\frac{k}{2}$ (explained later in Sect. 2.3) for each attribute $r \in R$, and aim to compute an *attribute embedding* vector for r of length $\frac{k}{2}$.

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 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:

$$\begin{aligned}\mathbf{R}_r[v_i, r_j] &= \frac{\mathbf{R}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{R}[v_i, r_l]},\\ \mathbf{R}_c[v_i, r_j] &= \frac{\mathbf{R}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{R}[v_l, r_j]}.\end{aligned} \tag{1}$$

Table 1 lists the frequently used notations in our paper.

Our solution utilizes an *extended graph* $\mathcal{G}$ that incorporates additional nodes and edges into G. To illustrate, Fig. 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 $\mathcal{G}$;

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 (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 (6)
$\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}$	The forward and backward embedding vectors, and attribute embedding vectors
α	The random walk stopping probability
n_b	The number of threads
κ	The number of super attributes

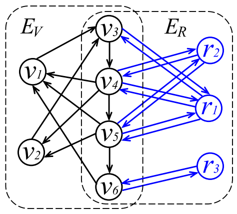


Fig. 1 Extended graph $\mathcal{G}$

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 Sect. 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*) [36,70] 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. 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 probability 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),.[2] The forward random walk yields a *node-to-attribute pair* (v_i, r_j) and we add it 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.

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

To explain the intuition behind the above definition, note that in $\mathcal{S}_f$, 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) [8] between node v_i and attribute r_j.[3] 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 nonnegative,

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

[3] The PMI quantifies how much more or less likely we are to see the two events co-occur, given their individual probabilities, and relative to the case where they are completely independent.

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 [62] in natural language processing. Specifically, if we view all the 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 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.

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

2.3 Objective function

Lastly, 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:

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

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

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 Fig. 1, all attribute weights in E_R are 1, and the random walk stopping probability α is set to 0.15 [36,70]. Table 2 lists the inner products of attribute embedding vectors of r_1-r_3 and embedding vectors of node v_1-v_6, which are the forward and backward affinity values preserved in these embedding vectors. These values are calculated based on Equations (2) and (3), using simulated random walks on $\mathcal{G}$ in Fig. 1. Observe, for example, that the 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. Meanwhile, regarding node v_5, if we only consider forward affinity, then, observe that v_5 has a higher forward affinity value with r_3 than that with r_1. Such an affinity value fails to capture the fact that v_5 is associated with r_1 but not with r_3, which may lead to incorrect attribute inference. This issue is resolved when we consider both forward and backward affinity.

3 `Seq-PANE`: single-thread `PANE`

In this section, we describe a single-thread version of `PANE`, called `Seq-PANE`. Further improved versions of `PANE` are presented in subsequent sections. Observe that it is a challenging task 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 undertake in a large number from every node and attribute

Algorithm 1: Seq-PANE

```
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 $\mathbf{F}', \mathbf{B}'$;
```

in order to accurately obtain 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 address these challenges, we propose Seq-PANE that can efficiently handle large-scale data and produce high-quality ANE results.

At a high level, Seq-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 (Sect. 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* [79] to efficiently obtain the output embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$ (Sect. 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 Seq-PANE algorithm. We elaborate on these steps now.

3.1 Forward and backward affinity approximation

In Sect. 2.2, node-attribute affinity values are defined using a large number of random walks, which are expensive to simulate on a massive graph. Hence, we transform the 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 selects 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 [36,70], we derive the matrix form of p_f and p_b as follows.

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

We only consider t iterations to approximate $\mathbf{P}_f$ and $\mathbf{P}_b$ in Eq. 5, where t is set to $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$.

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

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

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

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

$$\mathbf{F}' = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1), \quad \mathbf{B}' = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1). \tag{6}$$

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 Eq. 1. Then, APMI computes $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ based on Eq. 5. Note that $\mathbf{P}$ is sparse and has m nonzero 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 Eq. 5 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 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* $(v_i,r_j)\in V\times R$,

$$\max\left\{0,1-\frac{\epsilon}{\mathbf{P}_f[v_i,r_j]}\right\}\le\frac{2^{\mathbf{F}'[v_i,r_j]}-1}{2^{\mathbf{F}[v_i,r_j]}-1},$$
$$\max\left\{0,1-\frac{\epsilon}{\mathbf{P}_b[v_i,r_j]}\right\}\le\frac{2^{\mathbf{B}'[v_i,r_j]}-1}{2^{\mathbf{B}[v_i,r_j]}-1},$$

and

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

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

$$\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell=1-\sum_{\ell=0}^{t}\alpha(1-\alpha)^\ell=(1-\alpha)^{t+1}=\epsilon. \tag{7}$$

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

$$\mathbf{P}_f[v_i,r_j]-\mathbf{P}_f^{(t)}[v_i,r_j]=\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell\mathbf{P}^\ell[v_i]\cdot\mathbf{R}_r^\top[r_j]$$
$$=\left(\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell\mathbf{P}^\ell\right)[v_i]\cdot\mathbf{R}_r^\top[r_j]$$
$$\le\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell=\epsilon,$$

$$\mathbf{P}_b[v_i,r_j]-\mathbf{P}_b^{(t)}[v_i,r_j]$$
$$=\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell\mathbf{P}^{\top\ell}[v_i]\cdot\mathbf{R}_c^\top[r_j]$$
$$\le\sum_{v_l\in V}\sum_{\ell=t+1}^{\infty}\alpha(1-\alpha)^\ell\cdot\mathbf{R}_c[v_l,r_j]$$
$$\le\sum_{v_l\in V}\epsilon\cdot\mathbf{R}_c[v_l,r_j]=\epsilon.$$

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

$$\max\{0,\mathbf{P}_f[v_i,r_j]-\epsilon\}\le\mathbf{P}_f^{(t)}[v_i,r_j]\le\mathbf{P}_f[v_i,r_j], \tag{8}$$
$$\max\{0,\mathbf{P}_b[v_i,r_j]-\epsilon\}\le\mathbf{P}_b^{(t)}[v_i,r_j]\le\mathbf{P}_b[v_i,r_j]. \tag{9}$$

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

$$\frac{2^{\mathbf{F}'[v_i,r_j]}-1}{2^{\mathbf{F}[v_i,r_j]}-1}=\frac{\widehat{\mathbf{P}}_f^{(t)}[v_i,r_j]}{\widehat{\mathbf{P}}_f[v_i,r_j]}=\frac{\mathbf{P}_f^{(t)}[v_i,r_j]\cdot\sum_{v_l\in V}\mathbf{P}_f[v_l,r_j]}{\sum_{v_l\in V}\mathbf{P}_f^{(t)}[v_l,r_j]\cdot\mathbf{P}_f[v_i,r_j]}$$
$$\frac{2^{\mathbf{B}'[v_i,r_j]}-1}{2^{\mathbf{B}[v_i,r_j]}-1}=\frac{\widehat{\mathbf{P}}_f^{(t)}[v_i,r_j]}{\widehat{\mathbf{P}}_f[v_i,r_j]}=\frac{\mathbf{P}_b^{(t)}[v_i,r_j]\cdot\sum_{r_l\in R}\mathbf{P}_b[v_i,r_l]}{\sum_{r_l\in R}\mathbf{P}_b^{(t)}[v_i,r_l]\cdot\mathbf{P}_b[v_i,r_j]}$$

Plugging in the above equations 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 $\mathbf{Y}$. Specifically, the proposed SVDCCD solver is based on the *cyclic coordinate descent* (*CCD*) framework, which iteratively updates each embedding value toward optimizing the objective function in Equation (4). Unfortunately, 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*) [20]. 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 GInit 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 [56] at Line 1 to decompose $\mathbf{F}'$ into $\mathbf{U} \in \mathbb{R}^{n\times\frac{k}{2}}$, $\mathbf{\Sigma} \in \mathbb{R}^{\frac{k}{2}\times\frac{k}{2}}$, $\mathbf{V} \in \mathbb{R}^{d\times\frac{k}{2}}$, and then initializes $\mathbf{X}_f = \mathbf{U6}$ 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 Eq. 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. We observe that the 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 Sect. 7.

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 Eq. 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:

$$\mathbf{X}_f[v_i,l] \leftarrow \mathbf{X}_f[v_i,l] - \mu_f(v_i,l), \tag{10}$$
$$\mathbf{X}_b[v_i,l] \leftarrow \mathbf{X}_b[v_i,l] - \mu_b(v_i,l), \tag{11}$$
$$\mathbf{Y}[r_j,l] \leftarrow \mathbf{Y}[r_j,l] - \mu_y(r_j,l), \tag{12}$$

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

$$\mu_f(v_i,l) = \frac{\mathbf{S}_f[v_i]\cdot\mathbf{Y}[:,l]}{\mathbf{Y}^\top[l]\cdot\mathbf{Y}[:,l]},\ \mu_b(v_i,l) = \frac{\mathbf{S}_b[v_i]\cdot\mathbf{Y}[:,l]}{\mathbf{Y}^\top[l]\cdot\mathbf{Y}[:,l]}, \tag{13}$$

$$\mu_y(r_j,l) = \frac{\mathbf{X}_f^\top[l]\cdot\mathbf{S}_f[:,r_j] + \mathbf{X}_b^\top[l]\cdot\mathbf{S}_b[:,r_j]}{\mathbf{X}_f^\top[l]\cdot\mathbf{X}_f[:,l] + \mathbf{X}_b^\top[l]\cdot\mathbf{X}_b[:,l]}, \tag{14}$$

Algorithm 3: GInit

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

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 (13) and (14), $\mu_f(v_i,l)$, $\mu_b(v_i,l)$, 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. It would be expensive if we directly recompute $\mathbf{S}_f$ 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.

Instead, we dynamically maintain and partially update $\mathbf{S}_f$ and $\mathbf{S}_b$ according to Equations (15), (16) and (17). Specifically, when $\mathbf{X}_f[v_i,l]$ 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

$$\mathbf{S}_f[v_i] \leftarrow \mathbf{S}_f[v_i] - \mu_f(v_i,l)\cdot\mathbf{Y}[:,l]^\top, \tag{15}$$
$$\mathbf{S}_b[v_i] \leftarrow \mathbf{S}_b[v_i] - \mu_b(v_i,l)\cdot\mathbf{Y}[:,l]^\top, \tag{16}$$

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

$$\begin{aligned}\mathbf{S}_f[:,r_j] &\leftarrow \mathbf{S}_f[:,r_j] - \mu_y(r_j,l)\cdot\mathbf{X}_f[:,l],\\ \mathbf{S}_b[:,r_j] &\leftarrow \mathbf{S}_b[:,r_j] - \mu_y(r_j,l)\cdot\mathbf{X}_b[:,l].\end{aligned} \tag{17}$$

3.3 Complexity analysis

In the proposed algorithm Seq-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 Sect. 3.1, Algorithm 2 runs in time $O(md\cdot t) = O\left(md\cdot\log\frac{1}{\epsilon}\right)$. Meanwhile, according to [56], given $\mathbf{F}' \in \mathbb{R}^{n\times d}$ as input, RandSVD in Algorithm 3 requires

Algorithm 4: SVDCCD

```
Input: $\mathbf{F}', \mathbf{B}', k, t$.
Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.
1  $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow$ GInit($\mathbf{F}', \mathbf{B}', k, t$);
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 13;
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 15;
9              Update $\mathbf{S}_b[v_i]$ by Equation 16;
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 14;
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 17;
15 return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

Algorithm 5: Par-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 $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

$O(ndkt)$ 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.*, $\mathbf{F}'$,$\mathbf{B}'$, $\mathbf{U}$, $\mathbf{\Sigma}$, $\mathbf{V}$,$\mathbf{S}_f$,$\mathbf{S}_b$ are at most $O(nd)$. Hence, the space complexity of Algorithm 1 is bounded by $O(nd + m)$.

4 Par-PANE: parallel PANE

Although single-thread PANE (*i.e.*, Seq-PANE in 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, Seq-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, Seq-PANE involves various complex computational steps, 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 (Par-PANE in Algorithm 5), which takes only 11.9 hours on *MAG* when using 10 threads (*i.e.*, up to 10 times speedup with respect to Seq-PANE). In the first phase, we adopt block matrix multiplication [21] and propose PAPMI to compute forward and backward affinity matrices in a parallel manner (Sect. 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 (Sect. 4.2).

Algorithm 5 illustrates the pseudo-code of parallel Par-PANE. Compared to the single-thread version, parallel Par-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). Par-PANE invokes PAPMI (Algorithm 6) at Line 4 to get $\mathbf{F}'$ and $\mathbf{B}'$, and then invokes PSVDCCD (Algorithm 7) to refine the embeddings.

Note that the parallel version of Seq-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 Sect. 7 demonstrate, the degradation of result utility in parallel Seq-PANE is small but the speedup is significant.

4.1 Parallel forward and backward affinity approximation

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 $\mathbf{F}', \mathbf{B}'$
```

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 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 2 indicates the accuracy guarantee of PAPMI.

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

Proof According to Line 3 in Algorithm 6, we have

$$\mathbf{R}_r = \left[\mathbf{P}_{f}{}_1^{(0)}\ \mathbf{P}_{f}{}_2^{(0)} \cdots \mathbf{P}_{f}{}_{n_b}^{(0)}\right],$$

where $\mathbf{P}_{f}{}_1^{(0)}, \cdots, \mathbf{P}_{f}{}_{n_b-1}^{(0)} \in \mathbb{R}^{n\times\frac{d}{n_b}}$ and $\mathbf{P}_{f}{}_{n_b}^{(0)} \in \mathbb{R}^{n\times(d\%n_b)}$ ($d\%n_b$ denotes the remainder of integer d divded by n_b), and

$$\mathbf{R}_c = \left[\mathbf{P}_{b}{}_1^{(0)}\ \mathbf{P}_{b}{}_2^{(0)} \cdots \mathbf{P}_{b}{}_{n_b}^{(0)}\right],$$

where $\mathbf{P}_{b}{}_1^{(0)}, \cdots, \mathbf{P}_{b}{}_{n_b-1}^{(0)} \in \mathbb{R}^{n\times\frac{d}{n_b}}$ and $\mathbf{P}_{b}{}_{n_b}^{(0)} \in \mathbb{R}^{n\times(d\%n_b)}$. After t iterations, by Lines 4–6 in Algorithm 6, we have

$$\mathbf{P}_{f}{}_i^{(t)} = \alpha\sum_{\ell=0}^{t}(1-\alpha)^\ell\mathbf{P}^\ell\mathbf{P}_{f}{}_i^{(0)} \text{ and}$$

$$\mathbf{P}_{b_i}^{(t)} = \alpha\sum_{\ell=0}^{t}(1-\alpha)^\ell\mathbf{P}^{\top\ell}\mathbf{P}_{b_i}^{(0)}.$$

Thus, we can derive that

$$\mathbf{P}_f^{(t)} = \left[\mathbf{P}_{f}{}_1^{(t)} \cdots \mathbf{P}_{f}{}_{n_b}^{(t)}\right] = \alpha\sum_{\ell=0}^{t}(1-\alpha)^\ell\mathbf{P}^\ell\mathbf{R}_r,$$

$$\mathbf{P}_b^{(t)} = \left[\mathbf{P}_{b}{}_1^{(t)} \cdots \mathbf{P}_{b}{}_{n_b}^{(t)}\right] = \alpha\sum_{\ell=0}^{t}(1-\alpha)^\ell\mathbf{P}^\ell\mathbf{R}_c.$$

According to Inequality 8 and Inequality 9, for every pair $(v_i, r_j) \in V \times R$,

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

By Lines 9–10 in Algorithm 6, for i-th block and every pair $(v_l, r_j) \in V \times R_i$,

$$\widehat{\mathbf{P}}_f^{(t)}[v_l, r_j] = \frac{\mathbf{P}_{f}{}_i^{(t)}[v_l, r_j]}{\sum_{v_h\in V}\mathbf{P}_{f}{}_i^{(t)}[v_h, r_j]} = \frac{\mathbf{P}_f^{(t)}[v_l, r_j]}{\sum_{v_h\in V}\mathbf{P}_f^{(t)}[v_h, r_j]},$$

$$\widehat{\mathbf{P}}_b^{(t)}[v_l, r_j] = \frac{\mathbf{P}_{b_i}^{(t)}[v_l, r_j]}{\sum_{R_i\in\mathcal{R}}\sum_{r_h\in R_i}\mathbf{P}_{b_i}^{(t)}[v_l, r_h]} = \frac{\mathbf{P}_b^{(t)}[v_l, r_j]}{\sum_{r_h\in R}\mathbf{P}_b^{(t)}[v_l, r_h]}.$$

By Lines 11–13 in Algorithm 6, the results in in Lemma 2 are now at hand. □

Algorithm 7: PSVDCCD

```
Input: $\mathbf{F}', \mathbf{B}', \mathcal{V}, \mathcal{R}, k, t$.
Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.
1  $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow$ SMGInit($\mathbf{F}', \mathbf{B}', \mathcal{V}, k, t$);
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 7 to further improve the efficiency of the joint affinity matrix factorization process. At Line 1 of the algorithm, we design a parallel initialization algorithm SMGInit with a split-and-merge-based parallel SVD technique for embedding vector initialization.

Algorithm 8 displays the pseudo-code of SMGInit, which takes as input $\mathbf{F}', \mathbf{B}', \mathcal{V}$, and k. Based on $\mathcal{V}$, SMGInit

Algorithm 8: `SMGInit`

```
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  $\mathbf{V} \leftarrow [\mathbf{V}_1 \cdots \mathbf{V}_{n_b}]^\top$;
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 $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$;
```

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 $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$, `SMGInit` merges them 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, `SMGInit` 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, `SMGInit` 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 3 indicates that the initial embedding vectors produced by `SMGInit` and `GInit` are close.

After obtaining $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ by `SMGInit`, Lines 2–16 in Algorithm 7 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 7 returns embedding results at Line 17.

Lemma 3 *Given the same $\mathbf{F}', \mathbf{B}', k$ and t as inputs to Algorithm 3 and Algorithm 8, 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 $t = \infty$.*

Proof Let the output of Algorithm 3 be $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f$, and $\mathbf{S}_b$, and the results returned by Algorithm 8 be $\widehat{\mathbf{X}}_f, \widehat{\mathbf{X}}_b, \widehat{\mathbf{Y}}$ and $\widehat{\mathbf{S}}_f, \widehat{\mathbf{S}}_b$. According to [56], $t = \infty$ implies that RandSVD produces the same factorized results as that returned by exact SVD. Therefore, $\mathbf{X}_f \cdot \mathbf{Y}^\top = \mathbf{F}', \mathbf{S}_f = \mathbf{0}, \mathbf{X}_b = \mathbf{B}'\mathbf{Y}$ and $\mathbf{Y}$ is unitary, *i.e.*, $\mathbf{Y}^\top\mathbf{Y} = \mathbf{I}$. This leads to $\mathbf{S}_b\mathbf{Y} = (\mathbf{X}_b\mathbf{Y}^\top - \mathbf{B}')\mathbf{Y} = \mathbf{0}$.

On the other hand, consider Algorithm 8. Based on Lines 2–3, we have $\mathbf{U}_i\mathbf{V}_i^\top = \mathbf{F}'[V_i]$,

$$\mathbf{F}' = \begin{bmatrix} \mathbf{F}'[V_1] \\ \mathbf{F}'[V_1] \\ \vdots \\ \mathbf{F}'[V_{n_b}] \end{bmatrix} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{U}_{n_b} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{V}_1^\top \\ \mathbf{V}_2^\top \\ \vdots \\ \mathbf{V}_{n_b}^\top \end{bmatrix}.$$

By Lines 5–6, $\mathbf{W}\widehat{\mathbf{Y}}^\top = \mathbf{V}$ and $\widehat{\mathbf{Y}}$ is a unitary matrix, *i.e.*, $\widehat{\mathbf{Y}}^\top\widehat{\mathbf{Y}} = \mathbf{I}$. Then by Line 8 and Line 10, we derive that

$$\mathbf{F}' = \begin{bmatrix} \mathbf{U}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{U}_{n_b} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \vdots \\ \mathbf{W}_{n_b} \end{bmatrix} \cdot \widehat{\mathbf{Y}}^\top = \begin{bmatrix} \widehat{\mathbf{X}}_f[V_1] \\ \widehat{\mathbf{X}}_f[V_2] \\ \vdots \\ \widehat{\mathbf{X}}_f[V_{n_b}] \end{bmatrix} \cdot \widehat{\mathbf{Y}}^\top$$
$$= \widehat{\mathbf{X}}_f \cdot \widehat{\mathbf{Y}}^\top,$$

and thus $\widehat{\mathbf{S}}_f = \mathbf{0}$. In addition, according to Line 9 and Line 11, we have $\widehat{\mathbf{X}}_b = \mathbf{B}'\widehat{\mathbf{Y}}$ and $\widehat{\mathbf{S}}_b\widehat{\mathbf{Y}} = (\widehat{\mathbf{X}}_b\widehat{\mathbf{Y}}^\top - \mathbf{B}')\widehat{\mathbf{Y}} = \mathbf{0}$. The proof is complete. □

4.3 Complexity analysis

Observe that the non-parallel parts of Algorithms 6 (Lines 7–10) and 8 (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 7, 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 divide 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 7 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 8). In addition, Lines 4–6 in Algorithm 8 require $O(n_b dk)$ time. In merge course (*i.e.*, Lines 7–11 in Algorithm 8), 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 7), 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 7 require $O(m+nd)$ and $O(nd)$ space, respectively. Therefore, the space complexity of `Par-PANE` is $O(m+nd)$.

5 `PANE`$^{++}$: scaling to large attribute set

The aforementioned algorithms of `PANE` run in time linear to the number d of attributes in G, as shown in Sects. 3.3 and 4.3. Hence, when d is large (*e.g.*, millions of attributes, as is the case in the *MAG-SC* dataset in our experiments in Sect. 7), `PANE` (both `Seq-PANE` and `Par-PANE`) may still incur rather high computational costs. To overcome this problem, this section introduces PANE^{++}, which significantly improves over PANE in terms of efficiency in the presence of a large attribute set, while retaining the high result quality of PANE. Specifically, PANE^{++} first compresses the attribute matrix $\mathbf{R} \in \mathbb{R}^{n\times d}$ into a lower-dimensional one $\widetilde{\mathbf{R}} \in \mathbb{R}^{n\times\kappa}$. This is achieved by clustering d attributes into κ *super attributes*, where $\kappa \ll d$. Then, PANE^{++} proceeds with the PANE algorithm, with $\widetilde{\mathbf{R}}$ replacing $\mathbf{R}$. After obtaining the node embeddings $\mathbf{X}_f, \mathbf{X}_b$, and the super attribute embeddings $\widetilde{\mathbf{Y}} \in \mathbb{R}^{\kappa\times\frac{k}{2}}$ from PANE, PANE^{++} reconstructs attribute embeddings $\mathbf{Y} \in \mathbb{R}^{d\times\frac{k}{2}}$, based on the cluster information obtained in the first step.

In the following, we present an effective attribute clustering technique in Sect. 5.1, and describe the complete PANE^{++} algorithm and its analysis in Sects. 5.2 and 5.3, respectively.

5.1 Attribute clustering

As explained above, a key step in PANE^{++} is to cluster the input d attributes into κ super attributes, denoted as $\{c_1, c_2, \cdots, c_\kappa\}$, where each super attribute c_l corresponds to an attribute cluster $R_{c_l} \subset R$ consisting of multiple similar attributes. Our clustering ensures (i) that $R_{c_1} \cup R_{c_1} \cdots \cup R_{c_\kappa} = R$, and (ii) that no two attribute clusters have overlapping members. To preserve the information in the original attribute space, we need an appropriate attribute similarity measure, as well as an effective objective function for the attribute clustering, described in the following.

Attribute Similarity. Let $\mathbf{S}_R[r_i, r_j]$ denote the similarity between two attributes $r_i, r_j \in R$. Recall that $\mathbf{R}$ is the attribute matrix of the input graph G, where each row vector denotes an attribute vector of a node. Here, we focus on attributes (*i.e.*, columns of $\mathbf{R}$) instead of nodes (rows of $\mathbf{R}$). In particular, we view each node as a *feature*; then, each column of $\mathbf{R}$, say, $\mathbf{R}[:, r_i]$ corresponding to attribute r_i, can be regarded as a feature vector of r_i. Accordingly, we define the similarity $\mathbf{S}_R[r_i, r_j]$ between attributes r_i, r_j as the cosine similarity of their feature vectors:

$$\begin{aligned}\mathbf{S}_R[r_i, r_j] &= cosine(\mathbf{R}[:, r_i]^\top, \mathbf{R}[:, r_j]) \\ &= \frac{\mathbf{R}[:, r_i]^\top \cdot \mathbf{R}[:, r_j]}{\|\mathbf{R}[:, r_i]\| \cdot \|\mathbf{R}[:, r_j]\|} \\ &= \mathbf{R}_s[:, r_i]^\top \cdot \mathbf{R}_s[:, r_j], \end{aligned} \tag{18}$$

$$where\ \ \mathbf{R}_s[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sqrt{\sum_{v_l \in V} \mathbf{R}[v_l, r_j]^2}} = \frac{\mathbf{R}[v_i, r_j]}{\|\mathbf{R}[:, r_j]\|}. \tag{19}$$

Equation 18 indicates that we can simply calculate $\mathbf{S}_R[r_i, r_j]$ using the dot product of the normalized feature vectors of attributes r_i, r_j defined in Eq. 19.

Objective Function. Let c_l be a super attribute and R_{c_l} is its corresponding attribute cluster. Intuitively, a good attribute cluster R_{c_l} should satisfy that attributes within R_{c_l} are similar to each other and dissimilar to those outside R_{c_l}. Inspired by the RatioCut algorithm [28,74], we partition the attribute set R into κ disjoint subsets $R_{c_1}, R_{c_2}, \cdots, R_{c_\kappa}$ by solving the mincut problem, formulated as the following optimization problem:

$$\min_{R_{c_1}, R_{c_2}, \cdots, R_{c_\kappa}} \sum_{l=1}^{\kappa} \Phi(R_{c_\kappa}), \tag{20}$$

where $\Phi(R_{c_l})$ represents the *attribute cut* of R_{c_l}, defined as follows.

$$\Phi(R_{c_l}) = \sum_{r_i \in R_{c_l}, r_j \in R\setminus R_{c_l}} \frac{\mathbf{S}_R[r_i, r_j]}{|R_{c_l}|} \tag{21}$$

In the above formulation, $\Phi(R_{c_l})$ measures the averaged similarity between an attribute in R_{c_l} and another outside R_{c_l}; intuitively, a good attribute cluster R_{c_l} should have a low $\Phi(R_{c_l})$. As such, our objective in Eq. 20 is to find κ partitions $R_{c_1}, R_{c_2}, \cdots, R_{c_\kappa}$ of R such that the averaged similarities of attributes crossing different attribute clusters are minimized.

Lemma 4 *Given κ disjoint subsets $\{R_{c_1}, R_{c_2}, \cdots, R_{c_\kappa}\}$ of attribute set R and a clustering indicator matrix $\mathbf{C} \in \mathbb{1}^{d\times\kappa}$ such that for each entry with index r_j, c_l,*

$$\mathbf{C}[r_j, c_l] = \begin{cases} 1 & r_j \in R_{c_l}, \\ 0 & r_j \in R \setminus R_{c_l}, \end{cases} \tag{22}$$

the objective in Eq. 20 is equivalent to minimizing the following:

$$\sum_{l=1}^{\kappa} \Phi(R_{c_l}) = \mathrm{Tr}\left(\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1} \mathbf{C}^\top(\mathbf{I} - \mathbf{S}_R)\mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right), \tag{23}$$

where Tr *denotes the trace of a matrix.*

Proof According to the definitions of $\Phi(R_{c_l})$ and $\mathbf{C}$ in Eq. 21 and Eq. 22, respectively, we have

$$\begin{aligned}\Phi(R_{c_l}) &= \sum_{r_i \in R_{c_l}, r_j \in R\setminus R_{c_l}} \frac{\mathbf{S}_R[r_i, r_j]}{|R_{c_l}|} \\ &= \sum_{r_i, r_j \in R} \mathbf{S}_R[r_i, r_j] \cdot \left(\frac{\mathbf{C}[r_i, c_l]}{\sqrt{|R_{c_l}|}} - \frac{\mathbf{C}[r_j, c_l]}{\sqrt{|R_{c_l}|}}\right)^2\end{aligned}$$

$$= \frac{\mathbf{C}[:, c_l]^\top}{\sqrt{|R_{c_l}|}} \cdot (\mathbf{I} - \mathbf{S}_R) \cdot \frac{\mathbf{C}[:, c_l]}{\sqrt{|R_{c_l}|}}.$$

Note that $\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}$ is a $\kappa \times \kappa$ diagonal matrix, whose (c_l, c_l) entry is equal to $\frac{1}{\sqrt{|R_{c_l}|}}$. Therefore,

$$\sum_{l=1}^{\kappa} \Phi(R_{c_l}) = \sum_{l=1}^{\kappa} \frac{\mathbf{C}[:, c_l]^\top}{\sqrt{|R_{c_l}|}} \cdot (\mathbf{I} - \mathbf{S}_R) \cdot \frac{\mathbf{C}[:, c_l]}{\sqrt{|R_{c_l}|}}$$
$$= \mathrm{Tr}\left(\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\mathbf{C}^\top \cdot (\mathbf{I} - \mathbf{S}_R) \cdot \mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right),$$

which finishes the proof. □

Using Lemma 4, the optimization objective in Eq. 20 can be transformed into the following:

$$\max_{\mathbf{C}\in\mathbb{R}^{d\times\kappa}} \mathrm{Tr}\left(\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\mathbf{C}^\top \cdot \mathbf{S}_R \cdot \mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right). \tag{24}$$

Therefore, the problem of finding κ super attributes becomes computing a clustering indicator matrix (CIM) $\mathbf{C}$ defined in Eq. 22 such that Eq. 24 is optimized.

Computing CIM C. Since CIM $\mathbf{C}$ satisfies Eq. 22, we have $\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\mathbf{C}^\top \cdot \mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1} = \mathbf{I}$. Further, according to [63] and the Rayleigh-Ritz theorem (Section 5.5.2 of [50]),

$$\mathrm{Tr}\left(\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\mathbf{C}^\top \cdot \mathbf{S}_R \cdot \mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right) \le \mathrm{Tr}(\mathbf{U}^\top\mathbf{S}_R\mathbf{U}), \tag{25}$$

where the columns in the matrix $\mathbf{U} \in \mathbb{R}^{d\times\kappa}$ are the κ eigenvectors corresponding to the κ largest eigenvalues of $\mathbf{S}_R$. Inequality 25 suggests that if we can find a CIM $\mathbf{C}$ that minimizes the difference between $\mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}$ and $\mathbf{U}$ as follows:

$$\|\mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1} - \mathbf{U}\|_F, \tag{26}$$

then, our objective in Eq. 24 can be roughly optimized. Recall that CIM $\mathbf{C}$ satisfies Eq. 22. Hence, for each attribute r_i that belongs to subset R_{c_l}, we have $\left(\mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right)[r_i, c_l] = \frac{1}{\sqrt{|R_{c_l}|}}$ and

$$\left(\mathbf{C}\sqrt{\mathbf{C}^\top\mathbf{C}}^{-1}\right)[r_i, c_j] = 0\ \forall c_j \in \{c_1, \cdots, c_\kappa\} \setminus c_l.$$

This implies that to minimize Eq. 26, for each attribute r_i and its corresponding super attribute c_l, we can simply ensure that $\mathbf{U}[r_i, c_l]$ is the maximum entry in row vector $\mathbf{U}[r_i]$. In other words, we choose the super attribute c_l such that $c_l = \arg\max_{c_j\in\{c_1,\cdots,c_\kappa\}} \mathbf{U}[r_i, c_l]$ and assign r_i to the attribute cluster R_{c_l}.

Algorithm 9: PANE^{++}

```
Input: $G, \mathbf{R}, k, \alpha, \kappa$
Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$
1  Compute $\mathbf{R}_s$ by Equation 19;
2  Let $\mathbf{U}$ be the approximate top-$\kappa$ left singular vectors returned by
   RandSVD($\mathbf{R}_s^\top, \kappa, \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$);
3  Initialize $\mathbf{C} \leftarrow \mathbf{0} \in \mathbb{R}^{d\times\kappa}$;
4  for $r_i \in R$ do
5      $c_l \leftarrow \arg\max_{c_j\in\{c_1,\cdots,c_\kappa\}} \mathbf{U}[r_i, c_j]$;
6      $\mathbf{C}[r_i, c_l] \leftarrow 1$
7  $\widetilde{\mathbf{R}} \leftarrow \mathbf{RC}$;
8  Invoke Seq-PANE with $\widetilde{\mathbf{R}}$;
9  Let $\mathbf{X}_f, \widetilde{\mathbf{Y}}, \mathbf{X}_b$ be the output of Seq-PANE;
10 $\mathbf{Y} \leftarrow \mathbf{C}\widetilde{\mathbf{Y}}$;
11 return $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
```

Now, the optimization problem in Eq. 24 is transformed to finding the top-κ eigenvectors $\mathbf{U}$ of $\mathbf{S}_R$. However, by Eq. 18, the construction of $\mathbf{S}_R$ incurs $O(nd^2)$ time and $O(d^2)$ space, which is prohibitively expensive when d is large. Observe that in Eq. 18, $\mathbf{S}_R$ is the dot product of $\mathbf{R}_s$ and its transpose. Suppose the exact SVD of $\mathbf{R}_s^\top \in \mathbb{R}^{d\times n}$ is $\mathbf{R}_s^\top = \widehat{\mathbf{U}}\widehat{\mathbf{\Sigma}}\widehat{\mathbf{V}}^\top$, where $\widehat{\mathbf{U}} \in \mathbb{R}^{d\times d}$ contains the full left singular vectors of $\mathbf{R}_s^\top$ and the diagonal entries in $\widehat{\mathbf{\Sigma}}$ are the singular values of $\mathbf{R}_s^\top$. According to [67], the columns in $\widehat{\mathbf{U}}$ are the eigenvectors of matrix $\mathbf{R}_s^\top\mathbf{R}_s = \mathbf{S}_R$ and the diagonal entries in $\widehat{\mathbf{\Sigma}}^2$ are the eigenvalues of $\mathbf{S}_R$. Since all singular values are nonnegative, the i-th largest eigenvalue of $\mathbf{S}_R$ is equal to the square of the i-th largest singular values of $\mathbf{R}_s^\top$. Therefore, the κ largest eigenvectors of $\mathbf{S}_R$ are equal to the top-κ left singular vectors of $\mathbf{R}_s^\top$, *i.e.*, the κ left singular vectors corresponding to the κ largest singular values of $\mathbf{R}_s^\top$. Thus, the problem is transformed to computing the top-κ left singular vectors of $\mathbf{R}_s$, which eliminates the need to construct and materialize $\mathbf{S}_R$ explicitly.

5.2 Complete PANE^{++} algorithm

The pseudo-code of PANE^{++} is displayed in Algorithm 9. Compared to Seq-PANE, PANE^{++} takes as input an additional parameter κ, *i.e.*, the number of super attributes. Overall, PANE^{++} consists two phases: (i) constructing CIM $\mathbf{C}$ and the super attribute matrix $\widetilde{\mathbf{R}}$ (Lines 1–7); and (ii) invoking Seq-PANE to obtain node embeddings $\mathbf{X}_f, \mathbf{X}_b$ and attribute embeddings $\mathbf{Y}$ (Lines 8–11). In the first phase, PANE^{++} starts by normalizing attribute matrix $\mathbf{R}$ as $\mathbf{R}_s$ according to Eq. 19 (Line 1). After that, PANE^{++} obtains an approximate top-κ left singular vectors $\mathbf{U}$ by utilizing the efficient randomized SVD algorithm [56] at Line 2, with dimensionality κ and the number of iterations $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$. Next, Algorithm 9 proceeds to constructing CIM $\mathbf{C}$ (Lines

3–6). More specifically, for each attribute $r_i \in R$, we find the super attribute c_l such that $\mathbf{U}[r_i, c_l]$ is maximized among all super attributes, and then set $\mathbf{C}[r_i, c_l] = 1$ (Lines 4–6). Accordingly, we obtain $\widetilde{\mathbf{R}} = \mathbf{RC}$ (Line 7). That is, for each node v_i, its attribute value on super attribute c_l is computed by aggregating the values of all attributes in the attribute cluster R_{c_l} that super attribute c_l corresponds to, *i.e.*, $\widetilde{\mathbf{R}}[v_i, c_l] = \sum_{r_j \in R_{c_l}} \mathbf{R}[v_i, r_j]$. $\texttt{PANE}^{++}$ then invokes `Seq-PANE` with $\widetilde{\mathbf{R}}$ as the attribute matrix and obtains the returned forward embedding matrix $\mathbf{X}_f$, backward embedding matrix $\mathbf{X}_b$, as well as the embedding matrix $\widetilde{\mathbf{Y}} \in \mathbb{R}^{\kappa \times \frac{k}{2}}$ for κ super attributes $\{c_1, c_2, \cdots, c_\kappa\}$ (Lines 8–9). Finally, $\texttt{PANE}^{++}$ computes $\mathbf{Y} = \mathbf{C}\widetilde{\mathbf{Y}}$ as the attribute embeddings and return $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$ as the output embeddings.

5.3 Complexity analysis

First, Line 1 in Algorithm 9 needs to process every nonzero entry in $\mathbf{R}$, and, thus, takes $O(|E_R|)$ time. Given $\mathbf{R}_s$ as input, `RandSVD` [56] at Line 2 requires $O\left((|E_R| + d\kappa) \cdot \kappa \log\left(\frac{1}{\epsilon}\right)\right)$ time. Recall that in Lines 4–6, we need to find the largest value among κ entries for each $r_i \in R$. This time cost can be bounded by $O(d\kappa)$. The sparse matrix multiplications at Line 7 and Line 10 can be implemented with $O(|E_R| \cdot \kappa)$ and $O(d\kappa)$ time, respectively. According to Sect. 3.3, the invocation of `Seq-PANE` (Lines 8–9) in Algorithm 9 takes $O((m + nk) \cdot \kappa \log(\frac{1}{\epsilon}))$ time and $O(n\kappa + m)$ space. Overall, the total time complexity of $\texttt{PANE}^{++}$ is $O((m + nk + |E_R| + d\kappa) \cdot \kappa \log(\frac{1}{\epsilon}))$. Regarding space complexity, $\mathbf{R}$ and $\widetilde{\mathbf{R}}$ require $O(|E_R|)$ and $O(d\kappa)$ space, respectively. Hence, the space overhead incurred by $\texttt{PANE}^{++}$ is $O(n\kappa + d\kappa + m + |E_R|)$.

When to use $\texttt{PANE}^{++}$. Given a space budget b (*e.g.*, total available RAM), we employ $\texttt{PANE}^{++}$ instead of `Seq-PANE`/`Par-PANE` when d is very large (*e.g.*, $d \geq 10^4$) or $2nd + \frac{k}{2} \cdot (2n + d) \geq b$, where term $2nd$ represents the space overhead incurred by constructing $\mathbf{F}$ and $\mathbf{B}$, and $\frac{k}{2} \cdot (2n + d)$ is the total space cost for embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$ in `PANE`. The rationale is that in the first condition (i.e., d is large), factorizing large $n \times d$ affinity matrices $\mathbf{F}$ and $\mathbf{B}$ will severely impede the efficacy of `PANE`. As for the second condition, $2nd + \frac{k}{2} \cdot (2n + d) \geq b$ means that other variants of `PANE` would run out of space due to the large size of the input network and the intermediate structures.

6 Usage of embeddings

Given an input attributed network G, our ANE methods return two embedding vectors $\mathbf{X}_f[v_i]$ and $\mathbf{X}_b[v_i]$ for each node v_i, and an embedding vector $\mathbf{Y}[r_j]$ for each attribute r_j. In this section, we explain how to use the embeddings to achieve high performance in downstream tasks, including node classification, attribute inference, and link prediction.

Node Classification. We apply L2-normalization over the forward embedding vector $\mathbf{X}_f[v_i]$ and backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$, and concatenate them as the feature representation of v_i, which is then used as input to train or evaluate node classifiers, such as a linear support-vector machine (SVM) classifier [10] in Sect. 7.4.

Attribute Inference. Attribute inference is a supervised task that aims to predict the existence of an attribute r_j associated to a given node v_i. We leverage the following three heuristics for attribute inference using the obtained embeddings. First, if the forward affinity $\mathbf{F}[v_i, r_j]$ from node v_i to attribute r_j defined in Eq. 2 and the backward affinity $\mathbf{B}[v_i, r_j]$ from attribute r_j to v_i defined in Eq. 3 are both high, then, intuitively, node v_i is likely to be associated with attribute r_j. Second, if an attribute r_j appears frequently in different nodes of a training set, it tends to exist in the nodes of the test set. In other words, if the number of nonzero entries in column $\mathbf{R}[:, r_j]$ is large, r_j should be popular in many nodes, where $\mathbf{R}$ is the attribute matrix. Lastly, if a node v_i has many attributes (*i.e.*, there are many nonzero entries in row $\mathbf{R}[v_i]$), v_i is more likely to be associated with attribute r_j.

Based on the above three heuristics, we derive the following indicator value for predicting whether node v_i is associated with attribute r_j:

$$\mathbf{F}[v_i, r_j] + \mathbf{B}[v_i, r_j] + \log(\gamma_{v_i} + 1) + \log(\gamma_{r_j} + 1) \tag{27}$$

$$= \log\left(\left(\frac{n \cdot p_f(v_i, r_j)}{\sum_{v_h \in V} p_f(v_h, r_j)} + 1\right) \cdot \sqrt{(\gamma_{v_i} + 1) \cdot (\gamma_{r_j} + 1)}\right) + \log\left(\left(\frac{d \cdot p_b(v_i, r_j)}{\sum_{r_h \in R} p_b(v_i, r_h)} + 1\right) \cdot \sqrt{(\gamma_{v_i} + 1) \cdot (\gamma_{r_j} + 1)}\right). \tag{28}$$

Specifically, if Eq. 27 has a large value, then, node v_i is likely to have attribute r_j. Particularly, in Eq. 27, in addition to the forward and backward affinities $\mathbf{F}[v_i, r_j]$ and $\mathbf{B}[v_i, r_j]$, γ_{v_i} and γ_{r_j} are the numbers of nonzero entries in $\mathbf{R}[v_i]$ and $\mathbf{R}[:, r_j]$ for node v_i and attribute r_j, respectively, which serve as scaling factors to give more weights if v_i and r_j are popular in attribute matrix $\mathbf{R}$. Since forward and backward affinities $\mathbf{F}[v_i, r_j]$ and $\mathbf{B}[v_i, r_j]$ are based on shifted PMI (SPMI) explained in Sect. 2.2, we also apply logarithm operation over γ_{v_i} and γ_{r_j} shifted by 1, to make sure the factors are positive. Then Eq. 27 is rewritten into Eq. 28, based on Eq. 2 and Eq. 3. Equation 28 provides a detailed interpretation about how the two scaling factors γ_{v_i} and γ_{r_j} work together as a factor $\sqrt{(\gamma_{v_i} + 1) \cdot (\gamma_{r_j} + 1)}$ to affect the forward and backward affinity computations.

Recall that the embedding vectors are expected to satisfy that $\mathbf{X}_f[v_i]\cdot\mathbf{Y}[r_j]^\top$ preserves $\mathbf{F}[v_i,r_j]$, and $\mathbf{X}_b[v_i]\cdot\mathbf{Y}[r_j]^\top$ preserves $\mathbf{B}[v_i,r_j]$, according to objective function in Eq. 4. Therefore, after obtaining the embeddings, we use $p(v_i,r_j)$ in Eq. 29 to infer if attribute r_j is associated to node v_i.

$$\begin{aligned} p(v_i,r_j) =& \mathbf{X}_f[v_i]\cdot\mathbf{Y}[r_j]^\top+\mathbf{X}_b[v_i]\cdot\mathbf{Y}[r_j]^\top \\ &+\log(\gamma_{v_i}+1)+\log(\gamma_{r_j}+1), \end{aligned} \tag{29}$$

where γ_{v_i} and γ_{r_j} are the numbers of nonzero entries in $\mathbf{R}[v_i]$ and $\mathbf{R}[:,r_j]$, respectively.

Link Prediction. Given two nodes v_i and v_j that are not directly connected, link prediction aims to predict if there will be an edge from v_i to v_j. Intuitively, if the affinity between v_i to v_j is strong, the probability of forming an edge from v_i to v_j is high. We propose to evaluate the affinity by combining forward affinities from v_i and backward affinities to v_j over the input attributed network, with the consideration of both graph topology and attributes. Specifically, given nodes v_i and v_j and attribute r_l, $\mathbf{F}[v_i,r_l]$ measures the affinity from v_i to r_l, $\mathbf{B}[v_j,r_l]$ evaluates the affinity from r_l to v_j, and consequently $\mathbf{F}[v_i,r_l]\times\mathbf{B}[v_j,r_l]$ represents the affinity from node v_i to node v_j via attribute r_l over the attributed network. However, note that $\mathbf{F}[v_i,r_l]\times\mathbf{B}[v_j,r_l]$ does not consider the degrees of v_i and v_j for link prediction, while node degrees have been shown to be crucial in improving the performance of link prediction [91]. Intuitively, if node v_i (resp. v_j) has large out-edges (resp. in-edges), v_i (resp. v_j) tends to connect to (resp. be connected to by) other nodes. Therefore, we further use the out-degree $d_{out}(v_i)$ of v_i and in-degree $d_{in}(v_j)$ of v_j as weights for the node affinity values. In particular, the following equation is used to evaluate the weighted affinity between v_i and v_j, by summing up all possible $\mathbf{F}[v_i,r_l]\times\mathbf{B}[v_j,r_l]$ for any $r_l\in R$, with weights $\sqrt{d_{out}(v_i)+1}$ and $\sqrt{d_{out}(v_j)+1}$:

$$\sum_{r_l\in R}\sqrt{d_{out}(v_i)+1}\cdot\mathbf{F}[v_i,r_l]\times\mathbf{B}[v_j,r_l]\cdot\sqrt{d_{in}(v_j)+1}.$$

As explained above, embedding vectors $\mathbf{X}_f[v_i]\cdot\mathbf{Y}[r_l]^\top$ preserves $\mathbf{F}[v_i,r_l]$, and $\mathbf{X}_b[v_j]\cdot\mathbf{Y}[r_l]^\top$ preserves $\mathbf{B}[v_j,r_l]$. Therefore, the node affinity $p(v_i,v_j)$ between v_i and v_j is estimated by Eq. 30. For undirected graphs, we use $p(v_i,v_j)+p(v_j,v_i)$ as the score for predicting the edge between v_i and v_j.

$$\begin{aligned} p(v_i,v_j)=\sum_{r_l\in R}\Bigg(&\left(\sqrt{d_{out}(v_i)+1}\cdot\mathbf{X}_f[v_i]\cdot\mathbf{Y}[r_l]^\top\right) \\ &\cdot\left(\sqrt{d_{in}(v_j)+1}\cdot\mathbf{X}_b[v_j]\cdot\mathbf{Y}[r_l]^\top\right)\Bigg). \end{aligned} \tag{30}$$

In the next section, we adopt the above methods to utilize the embedding results for node classification, link prediction, and attribute inference over real-world attributed networks.

7 Experiments

This section experimentally evaluates our proposed PANE and PANE^{++} against 10 competitors on three tasks: node classification, link prediction, and attribute inference, over 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.

7.1 Experiments setup

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. *Citeseer*[4] and *Flickr*[5] are benchmark datasets used in prior work [29,47,53,57,84,104]. *Facebook*[6] and *Google*+footnote 6 are social networks used in [42]. 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 [53,90].

To evaluate the scalability of the proposed solutions, we also include three large datasets, namely *TWeibo*,[7] *MAG*[8] and *MAG-SC*.[9] These datasets have not been used in existing ANE work due to their massive size. Specifically, *TWeibo* [38] 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* [66], where each node represents a paper and each directed edge repre-

4 http://linqs.soe.ucsc.edu/data

5 https://github.com/mengzaiqiao/CAN

6 http://snap.stanford.edu/data

7 https://www.kaggle.com/c/kddcup2012-track1

8 http://ma-graph.org/rdf-dumps/

9 https://figshare.com/articles/dataset/mag_scholar/12696653

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

Name	$\|V\| = n$	$\|E_V\| = m$	$\|R\| = d$	$\|E_R\|$	$\|L\|$	Type	References
Citeseer	3.3K	4.7K	3.7K	105.2K	6	Directed	[47,53,57,84,89,104]
Pubmed	19.7K	44.3K	0.5K	988K	3	Directed	[53,57,101,104]
Facebook	4K	88.2K	1.3K	33.3K	193	Undirected	[42,53,90,101]
Flickr	7.6K	479.5K	12.1K	182.5K	9	Undirected	[53]
Google+	107.6K	13.7 M	15.9K	300.6 M	468	Directed	[42,90]
TWeibo	2.3 M	50.7 M	1.7K	16.8 M	8	Directed	–
MAG	59.3 M	978.2 M	2K	434.4 M	100	Directed	–
MAG-SC	10.5 M	265.2 M	2.78 M	1.1B	8	Directed	[4,94]

sents a citation. We extract 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. *MAG-SC* is also a citation graph extracted from Microsoft Academic Knowledge Graph by [4]. In *MAG-SC*, the attributes of a node are the bag-of-words representation of the respective paper abstract. In total, there are 2.78 million distinct attributes in *MAG-SC*.

Note that *Flickr*, *Google+*, and *MAG-SC* involve large values of d, *i.e.*, number of attributes, while the other datasets have relatively small d. Hence, PANE^{++} is evaluated on these three datasets to validate its ability to handle large attribute sets effectively.

Baselines and Parameter Settings. We compare our methods Seq-PANE (single-thread PANE), Par-PANE (parallel PANE), and PANE^{++} against 10 state-of-the-art competitors: eight recent ANE methods including BANE [89], CAN [53], STNE [47], PRRE [104], TADW [84], ARGA [57], DGI [73] and LQANR [88], one state-of-the-art homogeneous network embedding method NRP [91], and one latest attributed heterogeneous network embedding algorithm GATNE [6]. All methods except Par-PANE run on a single CPU core. Note that although GATNE itself is a parallel algorithm, its parallel version requires the proprietary AliGraph platform.

The parameters of all competitors are set as suggested in their respective papers. For Seq-PANE, Par-PANE, and PANE^{++}, 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 Par-PANE and $\kappa = 1024$ for PANE^{++}. Unless otherwise specified, we set space budget $k = 128$.

The efficiency evaluation results of all methods are presented in Sect. 7.2. We report the evaluation results of all methods for link prediction, node classification, and attribute inference, in Sects. 7.3, 7.4, and 7.5, respectively. A method is excluded in our study if it cannot finish training within one week.

7.2 Efficiency of ANE methods

Figure 2 and Fig. 2 report the running time required by each ANE method on datasets with small or large d, respectively. The y-axis is the running time (seconds) in log-scale. The reported time does not include the time for loading datasets and outputting embedding vectors. We omit any methods with time exceeding one week.

As shown in Fig. 2, both Seq-PANE and Par-PANE are significantly faster than all ANE competitors, often by orders of magnitude. For instance, on *Pubmed* in Fig. 2, Par-PANE takes 1.1 s and Seq-PANE requires 8.2 s, while the fastest ANE competitor TADW consumes 405.3 s, demonstrating that Seq-PANE (resp. Par-PANE) is 49× (resp. 368×) faster. On large attributed networks including *TWeibo* and *MAG*, most existing ANE solutions cannot finish within a week, while Seq-PANE and Par-PANE are able to handle such large-scale networks efficiently. Par-PANE is up to 9 times faster than Seq-PANE over all datasets. For instance, on *MAG* dataset that has 59.3 million nodes, when using 10 threads, Par-PANE requires 11.9 hours while Seq-PANE takes about five days, which demonstrates the power of our parallel techniques in Sect. 4. Note that PANE^{++} is not reported in Fig. 2 since these datasets have small d values whereas PANE^{++} is designed for datasets with a large d.

As shown in Fig. 2, on datasets with large d, PANE^{++} is significantly faster than Seq-PANE (both are single-threaded), validating the efficiency of techniques proposed in Sect. 5 for handling a large number of attributes. Seq-PANE is slower than PANE^{++} on *Flickr* and *Google+* since it needs to construct, materialize, and decompose two high dimensional dense affinity matrices in $n \times d$ dimensions, while PANE^{++} works on matrices in $n \times \kappa$ dimensions to obtain embeddings, where $\kappa \ll d$. All competitors are slower than our methods. Moreover, as reported in Fig. 2, PANE^{++} is the only method that can efficiently handle *MAG-SC* with 2.78 M attributes, while all other methods including Seq-PANE and Par-PANE run out of memory or time. Further, as we show shortly in Sects. 7.3, 7.4, and 7.5, compared to Seq-PANE, PANE^{++} achieves comparable and sometimes even superior accuracy for link prediction, node classification, and attribute inference, which validates the efficiency and effectiveness of PANE^{++} on attributed networks with numerous attributes.

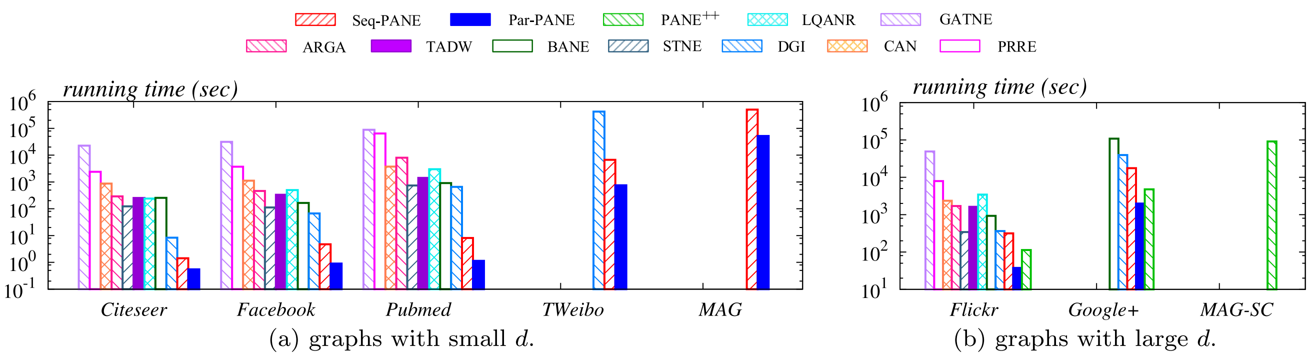


Fig. 2 Running time (best viewed in color)

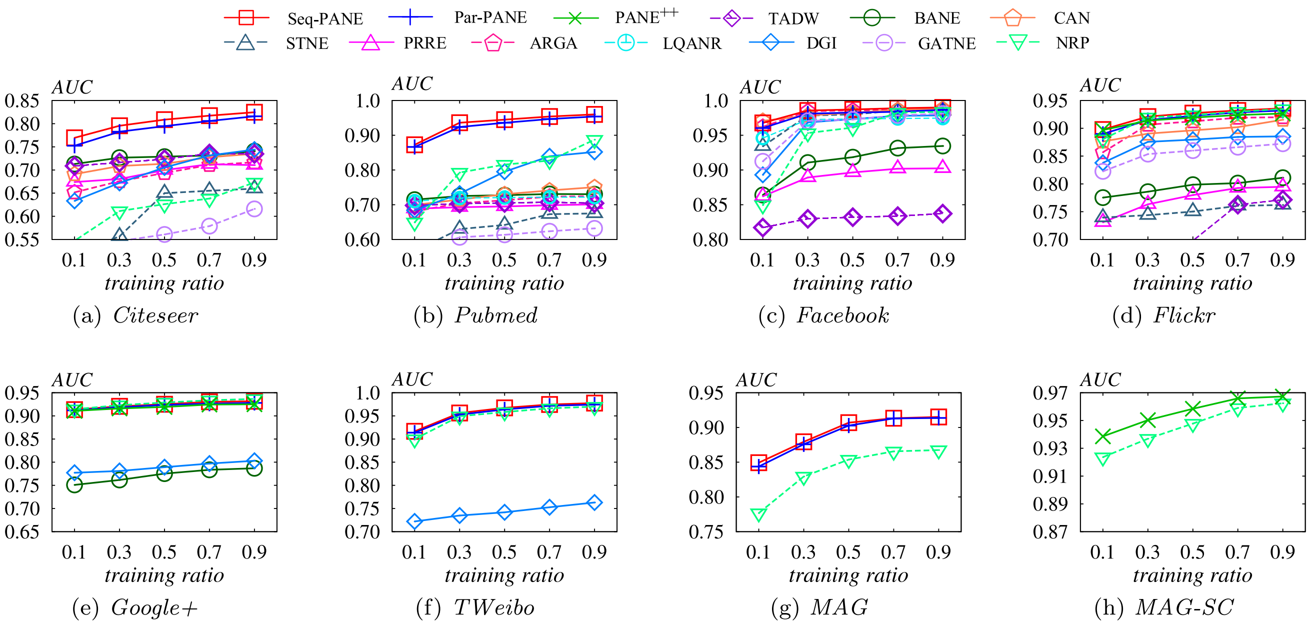


Fig. 3 Link prediction results on graphs (best viewed in color)

7.3 Link prediction

Recall from Sect. 6 that the link prediction task aims to predict the edges that are most likely to form between nodes. In this set of experiments, we first randomly remove a certain number of edges E_{rm} (ranging from 10% to 90%) in input graph G, obtaining a residual graph G'. On undirected graphs, we then randomly sample the same amount of pairs of nodes without edges connecting each other as negative edges E_{neg} (non-existing edges). On directed graphs, a node pair (u, v) is ordered, and link prediction predicts whether there is a directed edge from u to v. Hence, for directed graphs, E_{neg} contains $|E_{rm}|/2$ non-existing edges obtained by reversing $|E_{rm}|/2$ edges picked from E_{rm} and $|E_{rm}|/2$ non-existing edges that are randomly sampled. The test set E_{test} contains both the removed edges E_{rm} and the negative edges E_{neg}.

We run all methods on the residual graph G' to produce embedding vectors, and then evaluate the link prediction performance with E_{test} as follows. Recall that our methods produce 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, given a node pair (v_i, v_j), we use $p(v_i, v_j)$ in Eq. 30 for our methods to predict links on directed graphs, and use $p(v_i, v_j) + p(v_j, v_i)$ on undirected graphs. Competitor 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 $\mathbf{X}_f[v_i]\cdot\mathbf{X}_b[v_j]^\top$ as the prediction score for the directed edge (v_i, v_j) [91], and $\mathbf{X}_f[v_i]\cdot\mathbf{X}_b[v_j]^\top + \mathbf{X}_f[v_j]\cdot\mathbf{X}_b[v_i]^\top$ for undirected edges. 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 the 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 [26,51]. We

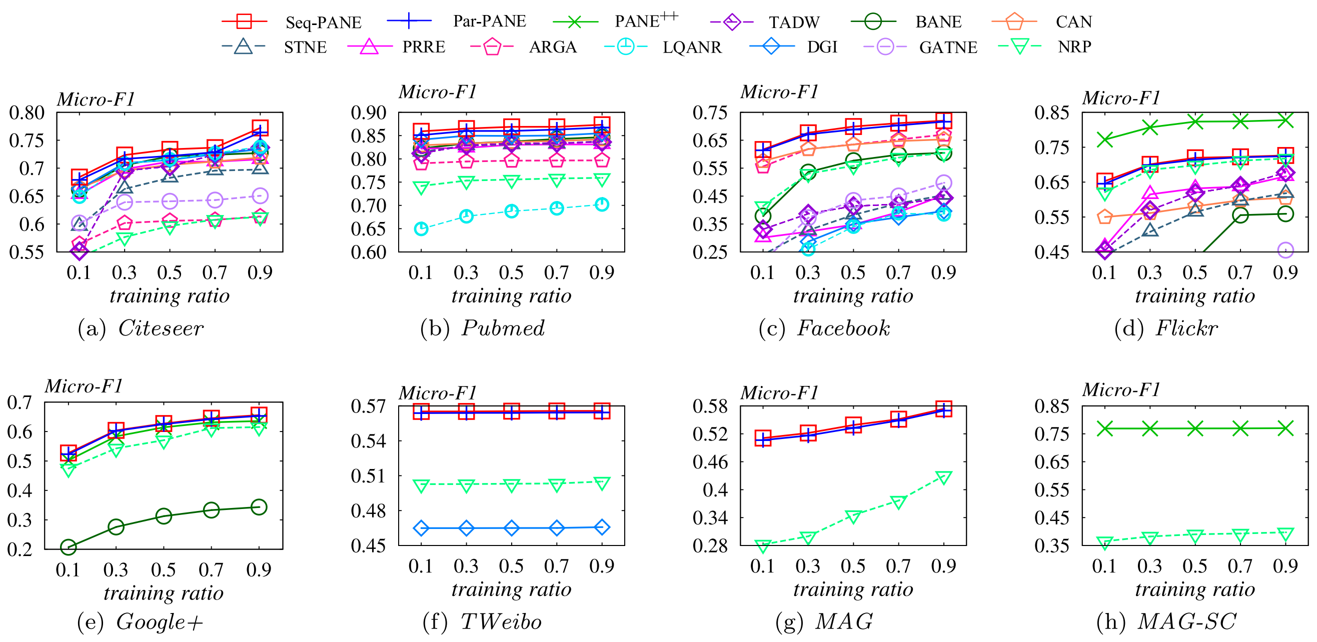


Fig. 4 Node classification results on graphs (best viewed in color)

use all these four prediction methods for each method, and report the method's best performance. Following previous work [53,57], we use *Area under the ROC Curve* (AUC) to evaluate link prediction accuracy.

Figure 3 reports the AUC scores of each method on each dataset. On undirected graphs including *Facebook* and *Flickr* in Figs. 3 and 3, our methods Seq-PANE, Par-PANE and PANE^{++} achieve superior or comparable performance to the best competitors. Moreover, Seq-PANE, Par-PANE and PANE^{++} consistently outperform all competitors over all directed graphs except NRP on *Google+*, by a large margin in terms of AUC.

For large attributed networks including *Google+*, *TWeibo*, *MAG*, and *MAG-SC*, most existing solutions cannot finish processing within a week and thus are not reported. The superiority of our methods 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 Eq. 30, with the consideration of both topological and attribute features. On *Google+*, NRP is slightly better than Seq-PANE 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 Seq-PANE. As shown in Fig. 3, our Par-PANE also outperforms all competitors significantly except NRP on *Google+*. Par-PANE also has comparable performance with Seq-PANE over all datasets. As reported in Sect. 7.2, Par-PANE is significantly faster than Seq-PANE by up to 9 times, with almost the same accuracy performance for link prediction. For datasets with a large d, *i.e.*, *Flickr*, *Google+*, and *MAG-SC*, we can observe that the extended version of Seq-PANE, *i.e.*, PANE^{++} also yields competitive performance. In particular, PANE^{++} is the only viable ANE solution and achieves a considerable gain over the best competitor NRP.

7.4 Node classification

Node classification predicts the node labels. Note that *Facebook*, *Google+* and *MAG* are multi-labeled. That is, each node can have multiple labels. We first run Seq-PANE, Par-PANE, PANE^{++}, and the competitors on the input attributed network G to obtain their embeddings. Then, we randomly sample a certain number of labeled nodes (ranging from 10% to 90%) to train a linear support-vector machine (SVM) classifier [10] and use the rest for testing. NRP, Seq-PANE, Par-PANE, 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 $v_i \in V$. As explained in Sect. 6, 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 [32,53,88], we use Micro-F1 and Macro-F1 to measure node classification performance. We repeat for 5 times and report the average performance.

Figure 4 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. For graphs with small d, including

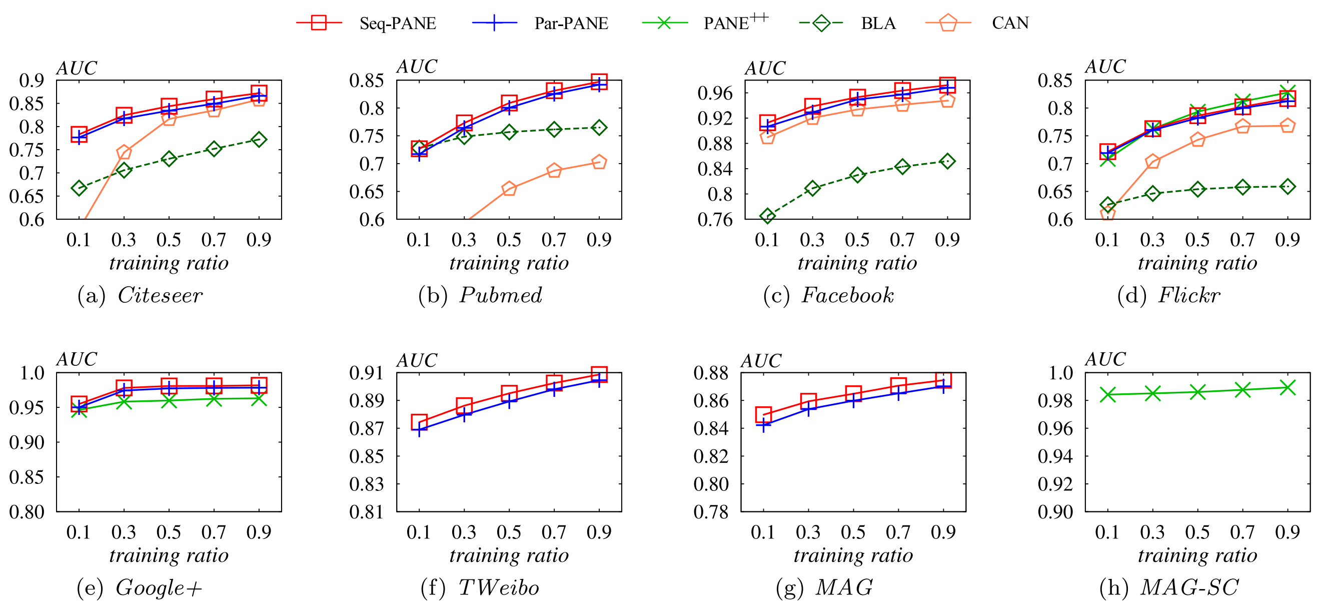


Fig. 5 Attribute inference results on graphs (best viewed in color)

Citeseer, Pubmed, Facebook, TWeibo, and *MAG* in Fig. 4, our methods Seq-PANE, Par-PANE and PANE^{++} consistently outperform all competitors, which demonstrates that our proposed solutions effectively capture the topology and attribute information of the input attributed networks. Specifically, compared with the competitors, Seq-PANE achieves a remarkable improvement on *Citeseer*, *Pubmed*, and *Facebook*. On the large graphs *TWeibo* and *MAG*, most existing solutions fail to finish within a week and thus their results are omitted. Furthermore, Seq-PANE outperforms NRP by a notable margin on *TWeibo* as displayed in Fig. 4. In addition, Seq-PANE and Par-PANE are superior to NRP with a significant gain on *MAG*. Over all datasets, Par-PANE has similar performance to that of Seq-PANE, while as shown in Sect. 7.2, Par-PANE is significantly faster than Seq-PANE.

As for the three datasets with large *d*, *i.e.*, *Flickr*, *Google+*, and *MAG-SC* in Fig. 4, Seq-PANE, Par-PANE, and PANE^{++} still outperform all competitors. Further, we can observe that PANE^{++} is significantly better than Seq-PANE and Par-PANE on *Flickr*, and comparable to them on *Google+*. In particular, on *Flickr*, PANE^{++} outperforms Seq-PANE by a significant margin of at least 10% in terms of Micro-F1. On *MAG-SC* with millions of attributes in Fig. 4, PANE^{++} can efficiently obtain effective embeddings for classification, while Seq-PANE and Par-PANE run out of memory, as reported in Sect. 7.2. On *MAG-SC*, PANE^{++} is far better than the only competitor NRP, which is designed for homogeneous networks without considering the attributes in *MAG-SC*. The superior performance of PANE^{++} over Seq-PANE, Par-PANE, and all competitors on graphs with a large number of attributes validates the effectiveness of the proposed techniques in Sect. 5.1, which boosts efficiency and also improves effectiveness in obtaining high-quality embeddings.

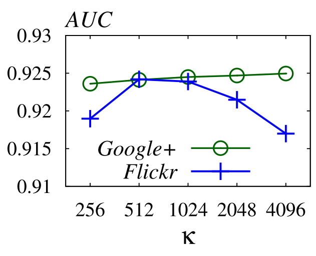


Fig. 7 Varying κ in PANE^{++}

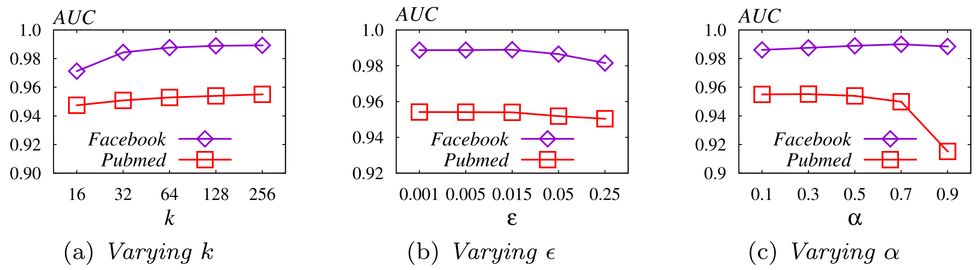


Fig. 6 Varying parameters in Seq-PANE

Another observation we can make from Fig. 4 is that on *TWeibo* and *MAG-SC* datasets, the performance of all methods remain stable when increasing the training ratio from 0.1 to 0.9, whereas their performance on other datasets goes up notably. The reason is as follows. Note that both *TWeibo* and *MAG-SC* datasets have millions of nodes and most nodes are associated with only 2 to 4 dominant labels. As such, even with 10% training data, we can learn a classifier with accuracy comparable to that with 90% training data. In contrast, other datasets either have a small number of nodes or balanced label distributions, making the classifiers learned on them more sensitive to the training ratio.

7.5 Attribute inference

Attribute inference aims to predict the existence of an attribute in a node. Note that, except for CAN [53], none of the other competitors is capable of performing attribute inference since they only generate embedding vectors for nodes and 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 [87]. Note that BLA is not an ANE solution.

We split the node-attribute associations E_R, and regard a certain number of these associations (ranging from 10% to 90%) as the test set E_R^{te} and the remaining part as the training set E_R^{tr}. In E_R^{te}, we also added an equal number of node-attribute pairs that are not in the original E_R as negative samples. CAN runs over E_R^{tr} 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 $r_j \in R$, and uses 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. For our methods, we use Eq. 29 to calculate the predicted score of an attribute r_j to a node v_i. Following prior work [53], we adopt the *Area under the ROC Curve* (AUC) to measure the performance.

In Fig. 5, we present the AUC scores of Seq-PANE, Par-PANE, PANE^{++}, CAN, and BLA in terms of attribute inference when varying the percentage of node-attribute associations (*i.e.*, the entries in $\mathbf{R}$) used for training from 10% to 90% (*i.e.*, 0.1 to 0.9). Observe from Fig. 5 that our methods consistently outperform existing solutions often by a large margin, demonstrating the power of the learned embedding

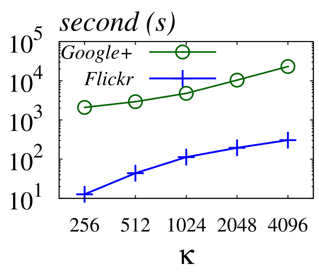


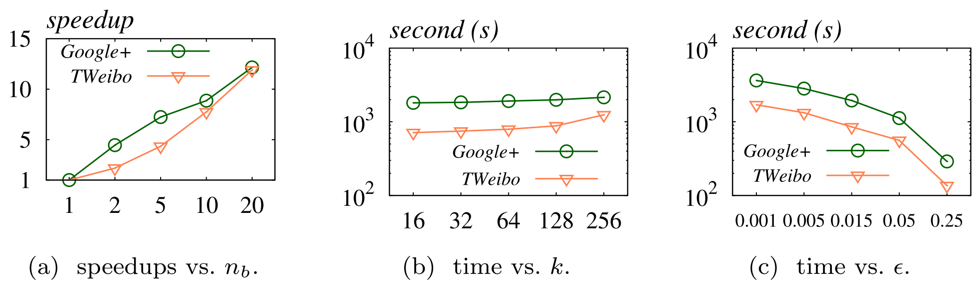
Fig. 9 Varying κ in PANE^{++}

vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$, which capture the affinity between nodes and attributes in attributed networks. In particular, on graphs with small d, including *Citeseer, Pubmed, Facebook, TWeibo*, and *MAG* in Fig. 5, Seq-PANE always obtains the highest AUC scores among all methods, while Par-PANE and PANE^{++} have comparable performance. On *Pubmed*, the difference of AUC between Seq-PANE and Par-PANE is very small. This negligible difference is introduced by the split-merge-based parallel SVD technique SMGInit for matrix decomposition. As shown in Sect. 7.2, parallel Par-PANE is considerably faster than Seq-PANE by up to 9 times, while obtaining almost the same accuracy performance. Moreover, on graphs with large d (*Flickr, Google+*, and *MAG-SC*), PANE^{++} is the only method able to handle *MAG-SC* with millions of attributes (Fig. 5) and also is slightly better than Seq-PANE and Par-PANE on *Flickr* (Fig. 5). It also has comparable performance on *Google+* as depicted in Fig. 5. Moreover, on *Flickr*, all our methods outperform competitors CAN and BLA by a remarkable absolute improvement in terms of AUC. In addition, CAN and BLA fail to process large attributed networks, including *TWeibo*, *Google+*, *MAG* and *MAG-SC* in one week. Thus, their results are omitted in Fig. 5. Considering the high efficiency of PANE^{++} demonstrated in in Sect. 7.2, we conclude that the techniques in Sect. 5.1 are effective and efficient to learn ANE embeddings on graphs with many attributes.

7.6 Parameter analysis

In this section, we evaluate the performance of our methods with varying parameter values, to study the effects of these parameters. First, we vary the embedding dimensionality k,

Fig. 8 Varying parameters in Seq-PANE/Par-PANE



error threshold ϵ and random walk stopping probability α in `Seq-PANE` for link prediction on *Facebook* and *Pubmed*. The AUC results are reported in Fig. 6. Specifically, Fig. 6 displays the link prediction AUC scores of `Seq-PANE` on *Facebook* and *Pubmed*, with varying values of the embedding size k in $\{16, 32, 64, 128, 256\}$. Observe that the AUC scores grow notably when k increases from 16 to 256, indicating that a large embedding dimensionality generally leads to more effective embedding vectors. Figure 6 reports the AUC scores of `Seq-PANE` when varying ϵ from 0.001 to 0.25. Observe that the link prediction performance remains relatively stable when increasing ϵ from 0.001 to 0.015, and decreases slightly when ϵ increases from 0.015 to 0.25. Note that when $\alpha = 0.5$, varying ϵ from 0.001 to 0.25 is equivalent to varying the number of iterations t from 9 down to 1. We then vary α from 0.1 to 0.9, and report the AUC scores of `Seq-PANE` on link prediction in Fig. 6. Observe that when α increases, the performance of `Seq-PANE` on *Facebook* is relatively stable, while that on *Pubmed* decreases significantly when $\alpha > 0.5$. Therefore, we choose to set $\alpha = 0.5$ by default. We speculate that the different behaviors of `Seq-PANE` are due to the different underlying graph properties of *Facebook* and *Pubmed*.

Next, we vary κ from 256 to 4096 in `PANE`$^{++}$ on *Flickr* and *Google+*, and report the link prediction performance in Fig. 7. When κ increases from 256 to 4096, the AUC performance on *Google+* goes up slightly, while the performance on *Flickr* increases first and then decreases after $\kappa > 1024$. Therefore, we choose to set $\kappa = 1024$ by default. The different behaviors of `PANE`$^{++}$ on *Flickr* and *Google+* are probably due to their different graph properties.

Fig. 8 displays the speedups of `Par-PANE` over `Seq-PANE` on *Google+* and *TWeibo* when varying the number of threads n_b from 1 to 20. When n_b increases, `Par-PANE` becomes much faster than `Seq-PANE`, demonstrating the parallel scalability of `Par-PANE` with respect to n_b. Figure 8 and Fig. 8 illustrate the running time of `Seq-PANE` with varying space budget k from 16 to 256 and error threshold ϵ from 0.001 to 0.25, respectively. In Fig. 8, when k is increased from 16 to 256, the running time is quite stable and goes up slowly, indicating the robust efficiency of our solution. In Fig. 8, the running time of `Seq-PANE` decreases considerably when error threshold ϵ is increased 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 `Seq-PANE` runs in linear to $\log(1/\epsilon)$ in Sect. 4. Figure 9 presents the running time of `PANE`$^{++}$ when varying the number of attribute clusters κ from 256 to 4096 on *Flickr* and *Google+*. As κ increases, the running time of `PANE`$^{++}$ increases, which is consistent with the time complexity analysis of `PANE`$^{++}$ in Sect. 5.3.

8 Related work

The work reported in this paper is an extended version of [92,93]. It differs from these earlier versions in the following ways. First, this work introduces the `PANE`$^{++}$ algorithm for handling an input network with a large attribute set. Second, it presents a detailed description on how the obtained embeddings are exploited for downstream machine learning tasks, in particular, node classification, attribute inference, and link prediction. Lastly, our experimental study is extended by incorporating a new dataset, *MAG-SC*, with millions of attributes and billions of node-attribute associations, as well as an extensive parameter analysis.

In the following, we review related work in the literature.

8.1 Network embedding

Network embedding (NE) [58] is to learn low-dimensional, fixed-length vector representations of network nodes such that the similarity in the embedding space reflects the similarity in the network. A pioneering effort is `DeepWalk` [58], which adopts the SkipGram model and random walks to capture the graph structure surrounding a node and map it into a low-dimensional embedding vector. Several studies [26,69,71,103] aim to improve the performance over `DeepWalk` by exploiting different random walk schemes. These random-walk-based solutions suffer from severe efficiency issues as they need to sample a large number of random walks and conduct expensive training processes. To address these challenges, massively parallel network embedding systems, including `PBG` [41], `Graphy` [106] and `LightNE` [59], are developed to utilize a large system with multiple processing units, including CPUs and GPUs. However, these systems consume immense amounts of computational resources that are financially expensive. Qiu et al. proved that the aforementioned random-walk-based methods have their equivalent matrix factorization forms and proposed an efficient factorization-based UNE solution [60]. In the literature, there are many factorization-based UNE solutions exhibiting superior efficiency and effectiveness, such as `RandNE` [99], `AROPE` [100], `STRAP` [96], `NRP` [91], and `FREDE` [72]. In addition to preserving the affinities between nodes, a number of studies [16,22,77,95] propose to incorporate community structures into network embedding. However, all NE solutions ignore attributes associated with nodes, limiting their utility in real-world attributed networks.

8.2 Attributed network embedding

Factorization-based methods: Given an attributed network G with n nodes, existing factorization-based methods mainly involve two stages: (i) building 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) factorizing **M** via techniques such as SGD [5], ALS [9], and coordinate descent [79]. Specifically, TADW [84] constructs a second-order proximity matrix **M** based on the adjacency matrix of G, and aims to reconstruct **M** by the product of the learned embedding matrix and the attribute matrix. HSCA [98] ensures that the learned embeddings of connected nodes are close in the embedding space. AANE [34] constructs a proximity matrix **M** using the cosine similarities between the attribute vectors of nodes. BANE [89] 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 [88] learns embeddings $\in \{-2^b, \cdots, -1, 0, 1, \cdots, 2^b\}^k$, where b is the bit-width. GAGE [39] formulates the ANE problem based on multi-dimensional scaling [11] and employs the tensor factorization over distance matrices to produce node embeddings. ANEM [43] utilizes *nonnegative matrix factorization* [1] to jointly decompose the low-order proximity matrix and community membership strength matrix to obtain node embeddings. 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 [23] 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 of minimizing the reconstruction loss. Existing methods either use different proximity matrices as inputs or design various neural network structures for the auto-encoder. Specifically, ANRL [101] combines auto-encoder with the SkipGram model [55] to learn embeddings. DANE [18] designs two auto-encoders to reconstruct the high-order proximity matrix and the attribute matrix, respectively. ARGA [57] integrates auto-encoder with graph convolutional networks [40] and generative adversarial networks [24] together. STNE [47] samples nodes via random walks and feeds the attribute vectors of the sampled nodes into a LSTM-based auto-encoder [30]. NetVAE [37] compresses the graph structures and node attributes with a shared encoder for transfer learning and information integration. CAN [53] embeds both nodes and attributes into two Gaussian distributions using a graph convolutional network and a dense encoder. None of these methods based on auto-encoders considers edge directions. Further, they suffer from severe efficiency issues due to the expensive training process of auto-encoders.

SAGE2VEC [64] proposes an enhanced auto-encoder model that preserves global graph structure and meanwhile handles the nonlinearity and sparsity of both graph structures and attributes. AdONE [2] designs an auto-encoder model for detecting and minimizing the effect of community outliers while generating embeddings. SAGES [76] first samples subgraphs containing highly relevant nodes with the consideration of node connections and attributes, and then learn the node embeddings by applying an unbiased graph auto-encoder on the sampled subgraphs with the guide of structure, content and community loss.

Other methods: PRRE [104] categorizes node relationships into positive, ambiguous, and negative types, according to the graph and attribute proximities between nodes, and then employs Expectation Maximization [12] to learn embeddings. SAGE [29] samples and aggregates features from a node's local neighborhood and learns embeddings by LSTM and pooling. NetHash [81] 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. In contrast to learning-based algorithms, NetHash [81] expands each node along with its neighboring nodes into a rooted tree and then recursively sketches the rooted tree to get a summarized attribute list as the embedding vector. PGE [32] groups nodes into clusters based on their attributes, and then trains neural networks with biased neighborhood samples in clusters to generate embeddings. ProGAN [19] adopts generative adversarial networks to generate node proximities, followed by neural networks to learn node embeddings from the generated node proximities. DGI [73] 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. [46] proposes a generic framework ASNE for embedding social networks, which captures the structural proximity and attribute proximity using a deep neural network architecture model. MUSAE [61] extends the SkipGram model with negative sampling used in homogeneous network embedding [26,58] for attributed networks and show their method implicitly factorizes a matrix of pointwise mutual information. SANE [75] trains embeddings via a united approach which combines the attention network with CBOW model [54] to learn the similarity of the graph structure and attributes simultaneously. MARINE [80] preserves the long-range spatial dependencies between nodes into embeddings by minimizing the information discrepancy in a Reproducing Kernel Hilbert Space. BiANE [33] jointly models the attribute proximity and the structure proximity through latent correlation training to embed bipartite attributed networks. MTSN [49] learns dynamic node embeddings by simultaneously modeling both local high-order structural proximities and temporal dynamics for dynamic attributed networks. Inspired by [29], InfomaxANE [45] leverages feature aggregation for the combination of topological features and node attributes, and then trains global and local embeddings based on mutual information estimation. ANGM [48] focuses on embedding networks with multipartite, hubs, and hybrid structures by combining neural networks and the *stochastic block model* [31].

8.3 Other related work

Heterogeneous networks contain nodes and edges of different types. A series of studies focus on embedding heterogeneous networks as surveyed in [14,82,86]. We review several representative studies as follows. Inspired by LINE [69] for NE, [68] and [65] preserve first/second-order proximities into the embeddings with the consideration of edge types. To incorporate high-order proximities, metapath2ec [13], HIN2Vec [17], and JUST [35] exploit different random walk models that are guided by pre-defined meta-paths to sample node context for representation learning. Li et al. [44] propose a biased correlated random walk model to capture node proximity inside each view (i.e., node type) without user-specified meta-paths and, further, a cross-view algorithm to transfer information across views. Without the assumption that different meta-paths share the same semantic space, SAHE [102] measures the relative proximities on each meta-path in its own semantic space and then aggregates them to obtain the final node proximity for embedding generation.

Recently, there are substantial embedding studies [7,78, 85,97] 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 [6], to process attributed heterogeneous network embedding. For each node on every edge type, it learns an embedding vector, by using the 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.

To cope with dynamic graphs that evolve over time, increasing research efforts [83] have been invested in *dynamic network embedding* (DNE) in recent years. For instance, [15] generalizes the Skip-gram model to DNE through a decomposable objective equivalent to that of LINE [69] and a carefully designed mechanism to select the greatly affected nodes that need to be updated. DynGEM [25] represents dynamic graphs as a collection of snapshots and incrementally updates the embeddings based on the ones from the previous snapshot via deep auto-encoders. Building on node2vec [26], dynnode2vec [52] keeps updating the list of random walks for evolving nodes and utilizes the dynamic Skip-gram model to generate updated embeddings. In lieu of embedding the entire graph, [27] learns embeddings for a subset of interesting nodes in large graphs with a dynamic algorithm for PPR computation. Bielak et al. [3] develop a general framework FILDNE for DNE, which integrates embeddings from any existing NE methods by an incremental updating scheme with batched data and an alignment mechanism.

9 Conclusions

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. Further, we extend PANE to PANE^{++} with an effective attribute clustering algorithm to efficiently handle large attributed networks with numerous attributes. Extensive experiments show that PANE and PANE^{++} achieve substantial performance advantages over the previous state-of-the-art in terms of both efficiency and result utility. Regarding future work, we plan to further develop GPU / multi-GPU versions of PANE and extend PANE to heterogeneous networks and dynamic graphs.

Acknowledgements This work is supported by the National University of Singapore SUG grant R-252-000-686-133, Singapore Government AcRF Tier-2 Grant MOE2019-T2-1-029, NPRP grant #NPRP10-0208-170408 from the Qatar National Research Fund (Qatar Foundation), and the financial support of Hong Kong RGC ECS (No. 25201221) and Start-up Fund (P0033898) by PolyU. The findings herein reflect the work, and are solely the responsibility of the authors.

References

1. Arora, S., Ge, R., Kannan, R., Moitra, A.: Computing a nonnegative matrix factorization-provably. STOC, pp. 145–161 (2012)
2. Bandyopadhyay, S., Vivek, S.V., Murty, M.: Outlier resistant unsupervised deep architectures for attributed network embedding. WSDM, pp. 25–33 (2020). https://doi.org/10.1145/3336191.3371788
3. Bielak, P., Tagowski, K., Falkiewicz, M., Kajdanowicz, T., Chawla, N..V.: FILDNE: A framework for incremental learning of dynamic networks embeddings. Knowl. Based Syst **236**, 107–453 (2022). https://doi.org/10.1016/j.knosys.2021.107453
4. Bojchevski, A., Klicpera, J., Perozzi, B., Kapoor, A., Blais, M., Rózemberczki, B., Lukasik, M., Günnemann, S.: Scaling graph neural networks with approximate pagerank. In: KDD, pp. 2464–2473 (2020). https://doi.org/10.1145/3394486.3403296
5. Bottou, L.: Large-scale machine learning with stochastic gradient descent. COMPSTAT pp. 177–186 (2010). https://doi.org/10.1007/978-3-7908-2604-3_16
6. Cen, Y., Zou, X., Zhang, J., Yang, H., Zhou, J., Tang, J.: Representation learning for attributed multiplex heterogeneous network.

KDD pp. 1358–1368 (2019). https://doi.org/10.1145/3292500.3330964
7. Chang, S., Han, W., Tang, J., Qi, G.J., Aggarwal, C.C., Huang, T.S.: Heterogeneous network embedding via deep architectures. KDD pp. 119–128 (2015). https://doi.org/10.1145/2783258.2783296
8. Church, K.W., Hanks, P.: Word association norms, mutual information, and lexicography. Comput. Linguist., pp. 22–29 (1990)
9. Comon, P., Luciani, X., De Almeida, A.L.: Tensor decompositions, alternating least squares and other tales. J. Chemom., pp. 393–405 (2009)
10. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
11. Davison, M.L.: Introduction to Multidimensional Scaling (1983)
12. Dempster, A., Laird, N., Rubin, D.: Maximum likelihood from incomplete data via the em algorithm. J. R. Stat. Soc. **39**(1), 1–38 (1977)
13. Dong, Y., Chawla, N.V., Swami, A.: metapath2vec: scalable representation learning for heterogeneous networks. In: KDD, pp. 135–144. ACM (2017). https://doi.org/10.1145/3097983.3098036
14. Dong, Y., Hu, Z., Wang, K., Sun, Y., Tang, J.: Heterogeneous network representation learning. In: C. Bessiere (ed.) IJCAI, pp. 4861–4867. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/677
15. Du, L., Wang, Y., Song, G., Lu, Z., Wang, J.: Dynamic network embedding : An extended approach for skip-gram based network embedding. In: J. Lang (ed.) IJCAI, pp. 2086–2092. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/288
16. Duan, Z., Sun, X., Zhao, S., Chen, J., Zhang, Y., Tang, J.: Hierarchical community structure preserving approach for network embedding. Inf. Sci. **546**, 1084–1096 (2021). https://doi.org/10.1016/j.ins.2020.09.053
17. Fu, T., Lee, W., Lei, Z.: Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning. In: E. Lim, M. Winslett, M. Sanderson, A.W. Fu, J. Sun, J.S. Culpepper, E. Lo, J.C. Ho, D. Donato, R. Agrawal, Y. Zheng, C. Castillo, A. Sun, V.S. Tseng, C. Li (eds.) CIKM, pp. 1797–1806. ACM (2017). https://doi.org/10.1145/3132847.3132953
18. Gao, H., Huang, H.: Deep attributed network embedding. IJCAI pp. 3364–3370 (2018). https://doi.org/10.24963/ijcai.2018/467
19. Gao, H., Pei, J., Huang, H.: Progan: Network embedding via proximity generative adversarial network. KDD pp. 1308–1316 (2019). https://doi.org/10.1145/3292500.3330866
20. Golub, G.H., Reinsch, C.: Singular value decomposition and least squares solutions. Linear Algebra, pp. 134–151 (1971)
21. Golub, G.H., Van Loan, C.F.: Matrix Computations, 1996. Johns Hopkins University, Press, Baltimore, MD, USA (1996)
22. Gong, M., Chen, C., Xie, Y., Wang, S.: Community preserving network embedding based on memetic algorithm. TETCI **4**(2), 108–118 (2020). https://doi.org/10.1109/TETCI.2018.2866239
23. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT press (2016)
24. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. NeurIPS pp. 2672–2680 (2014)
25. Goyal, P., Kamra, N., He, X., Liu, Y.: Dyngem: Deep embedding method for dynamic graphs. CoRR **abs/1805.11273** (2018). http://arxiv.org/abs/1805.11273
26. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
27. Guo, X., Zhou, B., Skiena, S.: Subset node representation learning over large dynamic graphs. In: F. Zhu, B.C. Ooi, C. Miao (eds.) KDD, pp. 516–526. ACM (2021). https://doi.org/10.1145/3447548.3467393
28. Hagen, L., Kahng, A..B.: New spectral methods for ratio cut partitioning and clustering. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst **11**(9), 1074–1085 (1992). https://doi.org/10.1109/43.159993
29. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. NeurIPS, pp. 1025–1035 (2017)
30. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput., pp. 1735–1780 (1997)
31. Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic blockmodels: first steps. Soc. Netw **5**(2), 109–137 (1983)
32. Hou, Y., Chen, H., Li, C., Cheng, J., Yang, M.C.: A representation learning framework for property graphs. KDD pp. 65–73 (2019). https://doi.org/10.1145/3292500.3330948
33. Huang, W., Li, Y., Fang, Y., Fan, J., Yang, H.: Biane: Bipartite attributed network embedding. In: SIGIR, pp. 149–158 (2020). https://doi.org/10.1145/3397271.3401068
34. Huang, X., Li, J., Hu, X.: Accelerated attributed network embedding. SDM, pp. 633–641 (2017). https://doi.org/10.1137/1.9781611974973.71
35. Hussein, R., Yang, D., Cudré-Mauroux, P.: Are meta-paths necessary?: Revisiting heterogeneous graph embeddings. In: A. Cuzzocrea, J. Allan, N.W. Paton, D. Srivastava, R. Agrawal, A.Z. Broder, M.J. Zaki, K.S. Candan, A. Labrinidis, A. Schuster, H. Wang (eds.) CIKM, pp. 437–446. ACM (2018). https://doi.org/10.1145/3269206.3271777
36. Jeh, G., Widom, J.: Scaling personalized web search. TheWebConf, pp. 271–279 (2003). https://doi.org/10.1145/775152.775191
37. Jin, D., Li, B., Jiao, P., He, D., Zhang, W.: Network-specific variational auto-encoder for embedding in attribute networks. IJCAI, pp. 2663–2669 (2019). https://doi.org/10.24963/ijcai.2019/370
38. Kaggle: Kdd cup (2012). https://www.kaggle.com/c/kddcup2012-track1
39. Kanatsoulis, C.I., Sidiropoulos, N.D.: Gage: Geometry preserving attributed graph embeddings. In: WSDM, pp. 439–448 (2022). https://doi.org/10.1145/3488560.3498467
40. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. ICLR (2016)
41. Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., Peysakhovich, A.: PyTorch-BigGraph: a large-scale graph embedding system. SysML, pp. 120–131 (2019)
42. Leskovec, J., Mcauley, J.J.: Learning to discover social circles in ego networks. NeurIPS, pp. 539–547 (2012)
43. Li, J., Huang, L., Wang, C., Huang, D., Lai, J., Chen, P.: Attributed network embedding with micro-meso structure. TKDD **15**(4), 72:1-72:26 (2021). https://doi.org/10.1145/3441486
44. Li, Z., Zheng, W., Lin, X., Zhao, Z., Wang, Z., Wang, Y., Jian, X., Chen, L., Yan, Q., Mao, T.: Transn: Heterogeneous network representation learning by translating node embeddings. In: ICDE, pp. 589–600. IEEE (2020). https://doi.org/10.1109/ICDE48307.2020.00057
45. Liang, X., Li, D., Madden, A.: Attributed network embedding based on mutual information estimation. In: M. d'Aquin, S. Dietze, C. Hauff, E. Curry, P. Cudré-Mauroux (eds.) CIKM, pp. 835–844. ACM (2020). https://doi.org/10.1145/3340531.3412008
46. Liao, L., He, X., Zhang, H., Chua, T..S.: Attributed social network embedding. TKDE **30**(12), 2257–2270 (2018). https://doi.org/10.1109/TKDE.2018.2819980
47. Liu, J., He, Z., Wei, L., Huang, Y.: Content to node: Self-translation network embedding. KDD, pp. 1794–1802 (2018). https://doi.org/10.1145/3219819.3219988
48. Liu, X., Yang, B., Song, W., Musial, K., Zuo, W., Chen, H., Yin, H.: A block-based generative model for attributed network embedding. World Wide Web **24**(5), 1439–1464 (2021). https://doi.org/10.1007/s11280-021-00918-y

49. Liu, Z., Huang, C., Yu, Y., Dong, J.: Motif-preserving dynamic attributed network embedding. In: TheWebConf, pp. 1629–1638 (2021)
50. Lutkepohl, H.: Handbook of matrices. Comput. Stat. Data Anal. **2**(25), 243 (1997)
51. Ma, J., Cui, P., Wang, X., Zhu, W.: Hierarchical taxonomy aware network embedding. KDD, pp. 1920–1929 (2018). https://doi.org/10.1145/3219819.3220062
52. Mahdavi, S., Khoshraftar, S., An, A.: dynnode2vec: Scalable dynamic network embedding. In: N. Abe, H. Liu, C. Pu, X. Hu, N.K. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, J.S. Saltz (eds.) IEEE BigData, pp. 3762–3765. IEEE (2018). https://doi.org/10.1109/BigData.2018.8621910
53. Meng, Z., Liang, S., Bao, H., Zhang, X.: Co-embedding attributed networks. WSDM, pp. 393–401 (2019). https://doi.org/10.1145/3289600.3291015
54. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
55. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. NeurIPS, pp. 3111–3119 (2013)
56. Musco, C., Musco, C.: Randomized block krylov methods for stronger and faster approximate singular value decomposition. NeurIPS, pp. 1396–1404 (2015)
57. Pan, S., Hu, R., Long, G., Jiang, J., Yao, L., Zhang, C.: Adversarially regularized graph autoencoder for graph embedding. IJCAI, pp. 2609–2615 (2018). https://doi.org/10.24963/ijcai.2018/362
58. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. KDD, pp. 701–710 (2014). https://doi.org/10.1145/2623330.2623732
59. Qiu, J., Dhulipala, L., Tang, J., Peng, R., Wang, C.: Lightne: a lightweight graph processing system for network embedding. In: SIGMOD, pp. 2281–2289 (2021). https://doi.org/10.1145/3448016.3457329
60. Qiu, J., Dong, Y., Ma, H., Li, J., Wang, K., Tang, J.: Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. WSDM, pp. 459–467 (2018). https://doi.org/10.1145/3159652.3159706
61. Rozemberczki, B., Allen, C., Sarkar, R.: Multi-scale attributed node embedding. J. Complex Netw. **9**(1), 1–22 (2021). https://doi.org/10.1093/comnet/cnab014
62. Salton, G., McGill, M.J.: Introduction to modern information retrieval (1986)
63. Sameh, A.H., Wisniewski, J.A.: A trace minimization algorithm for the generalized eigenvalue problem. J. Numer. Anal. **19**(6), 1243–1259 (1982)
64. Sheikh, N., Kefato, Z.T., Montresor, A.: A simple approach to attributed graph embedding via enhanced autoencoder. Complex Netw., pp. 797–809 (2019). https://doi.org/10.1007/978-3-030-36687-2_66
65. Shi, Y., Zhu, Q., Guo, F., Zhang, C., Han, J.: Easing embedding learning by comprehensive transcription of heterogeneous information networks. In: Y. Guo, F. Farooq (eds.) KDD, pp. 2190–2199. ACM (2018). https://doi.org/10.1145/3219819.3220006
66. Sinha, A., Shen, Z., Song, Y., Ma, H., Eide, D., Hsu, B.J., Wang, K.: An overview of microsoft academic service (mas) and applications. TheWebConf, pp. 243–246 (2015). https://doi.org/10.1145/2740908.2742839
67. Strang, G., Strang, G., Strang, G., Strang, G.: Introduction to Linear Algebra, vol. 3. Wellesley-Cambridge Press, Cambridge (1993)
68. Tang, J., Qu, M., Mei, Q.: PTE: predictive text embedding through large-scale heterogeneous text networks. In: L. Cao, C. Zhang, T. Joachims, G.I. Webb, D.D. Margineantu, G. Williams (eds.) KDD, pp. 1165–1174. ACM (2015). https://doi.org/10.1145/2783258.2783307
69. Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., Mei, Q.: Line: Large-scale information network embedding. TheWebConf, pp. 1067–1077 (2015). https://doi.org/10.1145/2736277.2741093
70. Tong, H., Faloutsos, C., Pan, J.Y.: Fast random walk with restart and its applications. ICDM, pp. 613–622 (2006). https://doi.org/10.1109/ICDM.2006.70
71. Tsitsulin, A., Mottin, D., Karras, P., Müller, E.: Verse: Versatile graph embeddings from similarity measures. TheWebConf, pp. 539–548 (2018). https://doi.org/10.1145/3178876.3186120
72. Tsitsulin, A., Munkhoeva, M., Mottin, D., Karras, P., Oseledets, I., Müller, E.: Frede: anytime graph embeddings. PVLDB **14**(6), 1102–1110 (2021). https://doi.org/10.14778/3447689.3447713
73. Veličković, P., Fedus, W., Hamilton, W.L., Liò, P., Bengio, Y., Hjelm, R.D.: Deep graph infomax. ICLR (2019)
74. Von Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**(4), 395–416 (2007)
75. Wang, H., Chen, E., Liu, Q., Xu, T., Du, D., Su, W., Zhang, X.: A united approach to learning sparse attributed network embedding. ICDM, pp. 557–566 (2018). https://doi.org/10.1109/ICDM.2018.00071
76. Wang, J., Qu, X., Bai, J., Li, Z., Zhang, J., Gao, J.: Sages: Scalable attributed graph embedding with sampling for unsupervised learning. TKDE, (01), 1–1 (2022). https://doi.org/10.1109/TKDE.2022.3148272
77. Wang, X., Cui, P., Wang, J., Pei, J., Zhu, W., Yang, S.: Community preserving network embedding. In: S. Singh, S. Markovitch (eds.) AAAI, pp. 203–209. AAAI Press (2017). http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14589
78. Wang, Y., Duan, Z., Liao, B., Wu, F., Zhuang, Y.: Heterogeneous attributed network embedding with graph convolutional networks. In: AAAI, pp. 10,061–10,062 (2019)
79. Wright, S.J.: Coordinate descent algorithms. Math. Program., pp. 3–34 (2015)
80. Wu, J., He, J.: Scalable manifold-regularized attributed network embedding via maximum mean discrepancy. CIKM, pp. 2101–2104 (2019). https://doi.org/10.1145/3357384.3358091
81. Wu, W., Li, B., Chen, L., Zhang, C.: Efficient attributed network embedding via recursive randomized hashing. IJCAI, pp. 2861–2867 (2018). https://doi.org/10.24963/ijcai.2018/397
82. Xie, Y., Yu, B., Lv, S., Zhang, C., Wang, G., Gong, M.: A survey on heterogeneous network representation learning. Pattern Recognit. **116**, 107–936 (2021). https://doi.org/10.1016/j.patcog.2021.107936
83. Xue, G., Zhong, M., Li, J., Chen, J., Zhai, C.: Dynamic network embedding survey. Neurocomputing **472**, 212–223 (2022). https://doi.org/10.1016/j.neucom.2021.03.138
84. Yang, C., Liu, Z., Zhao, D., Sun, M., Chang, E.: Network representation learning with rich text information. IJCAI, pp. 2111–2117 (2015)
85. Yang, C., Xiao, Y., Zhang, Y., Sun, Y., Han, J.: Heterogeneous network representation learning: a unified framework with survey and benchmark. TKDE (2020)
86. Yang, C., Xiao, Y., Zhang, Y., Sun, Y., Han, J.: Heterogeneous network representation learning: a unified framework with survey and benchmark. TKDE **34**(10), 4854–4873 (2022). https://doi.org/10.1109/TKDE.2020.3045924
87. Yang, C., Zhong, L., Li, L.J., Jie, L.: Bi-directional joint inference for user links and attributes on large social graphs. TheWebConf, pp. 564–573 (2017). https://doi.org/10.1145/3041021.3054181
88. Yang, H., Pan, S., Chen, L., Zhou, C., Zhang, P.: Low-bit quantization for attributed network representation learning. IJCAI, pp. 4047–4053 (2019). https://doi.org/10.24963/ijcai.2019/562

89. Yang, H., Pan, S., Zhang, P., Chen, L., Lian, D., Zhang, C.: Binarized attributed network embedding. ICDM, pp. 1476–1481 (2018). https://doi.org/10.1109/ICDM.2018.8626170
90. Yang, J., McAuley, J., Leskovec, J.: Community detection in networks with node attributes. ICDM, pp. 1151–1156 (2013). https://doi.org/10.1109/ICDM.2013.167
91. Yang, R., Shi, J., Xiao, X., Yang, Y., Bhowmick, S..S.: Homogeneous network embedding for massive graphs via reweighted personalized pagerank. PVLDB **13**(5), 670–683 (2020). https://doi.org/10.14778/3377369.3377376
92. Yang, R., Shi, J., Xiao, X., Yang, Y., Bhowmick, S.S., Liu, J.: No pane, no gain: Scaling attributed network embedding in a single server. ACM SIGMOD Record **51**(1), 42–49 (2022)
93. Yang, R., Shi, J., Xiao, X., Yang, Y., Liu, J., Bhowmick, S..S.: Scaling attributed network embedding to massive graphs. Proc. VLDB Endow. **14**(1), 37–49 (2020). https://doi.org/10.14778/3421424.3421430
94. Yang, R., Shi, J., Yang, Y., Huang, K., Zhang, S., Xiao, X.: Effective and scalable clustering on massive attributed graphs. In: TheWebConf, pp. 3675–3687 (2021). https://doi.org/10.1145/3442381.3449875
95. Ye, D., Jiang, H., Jiang, Y., Wang, Q., Hu, Y.: Community preserving mapping for network hyperbolic embedding. Knowl. Based Syst. **246**, 108–699 (2022). https://doi.org/10.1016/j.knosys.2022.108699
96. Yin, Y., Wei, Z.: Scalable graph embeddings via sparse transpose proximities. KDD, pp. 1429–1437 (2019). https://doi.org/10.1145/3292500.3330860
97. Zhang, C., Swami, A., Chawla, N.V.: Shne: Representation learning for semantic-associated heterogeneous networks. In: WSDM, pp. 690–698 (2019). https://doi.org/10.1145/3289600.3291001
98. Zhang, D., Yin, J., Zhu, X., Zhang, C.: Homophily, structure, and content augmented network representation learning. ICDM, pp. 609–618 (2016). https://doi.org/10.1109/ICDM.2016.0072
99. Zhang, Z., Cui, P., Li, H., Wang, X., Zhu, W.: Billion-scale network embedding with iterative random projection. ICDM, pp. 787–796 (2018). https://doi.org/10.1109/ICDM.2018.00094
100. Zhang, Z., Cui, P., Wang, X., Pei, J., Yao, X., Zhu, W.: Arbitrary-order proximity preserved network embedding. KDD, pp. 2778–2786 (2018). https://doi.org/10.1145/3219819.3219969
101. Zhang, Z., Yang, H., Bu, J., Zhou, S., Yu, P., Zhang, J., Ester, M., Wang, C.: Anrl: Attributed network representation learning via deep neural networks. IJCAI, pp. 3155–3161 (2018). https://doi.org/10.24963/ijcai.2018/438
102. Zheng, S., Guan, D., Yuan, W.: Semantic-aware heterogeneous information network embedding with incompatible meta-paths. WWW **25**(1), 1–21 (2022). https://doi.org/10.1007/s11280-021-00903-5
103. Zhou, C., Liu, Y., Liu, X., Liu, Z., Gao, J.: Scalable graph embedding for asymmetric proximity. AAAI, pp. 2942–2948 (2017)
104. Zhou, S., Yang, H., Wang, X., Bu, J., Ester, M., Yu, P., Zhang, J., Wang, C.: Prre: Personalized relation ranking embedding for attributed networks. CIKM, pp. 823–832 (2018). https://doi.org/10.1145/3269206.3271741
105. Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., Zhou, J.: Aligraph: a comprehensive graph neural network platform. PVLDB **12**(12), 2094–2105 (2019). https://doi.org/10.14778/3352063.3352127
106. Zhu, Z., Xu, S., Tang, J., Qu, M.: Graphvite: A high-performance cpu-gpu hybrid system for node embedding. TheWebConf, pp. 2494–2504 (2019). https://doi.org/10.1145/3308558.3313508