No PANE, No Gain: Scaling Attributed Network Embedding in a Single Server

Renchi Yang*
National University of Singapore
renchi@nus.edu.sg

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

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

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

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

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

ABSTRACT

Given a graph G where each node is associated with a set of attributes, *attributed network embedding* (ANE) maps each node $v \in G$ to a compact vector X_v, which can be used in downstream machine learning tasks in a variety of applications. Existing ANE solutions do not scale to massive graphs due to prohibitive computation costs or generation of low-quality embeddings. This paper proposes `PANE`, an effective and scalable approach to ANE computation for massive graphs in a *single* server that achieves state-of-the-art result quality on multiple benchmark datasets for two common prediction tasks: link prediction and node classification. Under the hood, `PANE` takes inspiration from well-established data management techniques to scale up ANE in a single server. Specifically, it exploits a carefully formulated problem based on a novel random walk model, a highly efficient solver, and non-trivial parallelization by utilizing modern multi-core CPUs. Extensive experiments demonstrate that `PANE` consistently outperforms all existing methods in terms of result quality, while being orders of magnitude faster.

1. INTRODUCTION

Graphs (a.k.a networks) are ubiquitous nowadays in many application domains such as biology, social sciences, chemistry, and finance. A recent survey [12] revealed that scalability and faster graph analytics or machine learning (ML) algorithms are considered as some of the top challenges for graph processing. Although considerable efforts have been invested toward these goals in academia and industry, these issues remain tenaciously challenging to address due to high computational complexity of iterative or combinatorial graph algorithms, low parallelizability due to tight coupling between nodes and edges, and difficulty to leverage traditional graph representation (*e.g.*, adjancency matrix) for ML problems [5]. In particular, many ML techniques typically assume independent real-valued input vectors and outputs in order to learn a latent function that maps each input to an output. Unfortunately, nodes in any network data are coupled through their edges, making it challenging for using traditional network representations in ML techniques. Although in principle we can represent the nodes as their corresponding row vectors in the adjacency matrix of the network, the dimensionalilty of such simplistic representation can be prohibitively large, rendering them impractical.

Network Embedding. Network embedding [5] aims to address the aforementioned challenges of traditional network representations by learning low-dimensional, fixed-length vector representations of network nodes such that the similarity in the embedding space reflects the similarity in the network. Specifically, in the original network, relationships between nodes are captured by edges or other higher-order topological measures. In the embedding space, these relationships are captured by distances between nodes in the vector space and the topological properties of the nodes are encoded by their corresponding embedding vectors. Since each node is represented by a vector encapsulating information of interest, many iterative or combinatorial graph problems can be reframed as computing mapping functions and operations on the embedding vectors, paving the way for more efficient or scalable solutions. Furthermore, the learned embedding space enables various network inference tasks such as link prediction, node label inference, and finding "important" nodes. For example, we can input two real-valued vectors representing a pair of nodes to a machine learning algorithm to predict the existence of a link between them (*i.e.*, a binary classification problem where the output label of 0 or 1 represents absence or presence of a link, respectively). Observe that the learned embeddings for all these tasks are realizable without demanding expensive feature engineering by domain experts. All these opportunities have led to the proposal of a cornucopia of techniques in the literature for network embedding and their usage in a wide variety of applications [5].

Attributed Network Embedding. The majority of existing network embedding techniques, however, exploit only the topological connections when learning node representations. In practice, however, real-world networks often are *attributed*

*Work done when the first author was a doctoral student at the Nanyang Technological University, Singapore.

DOI: https://doi.org/10.14778/3421424.3421430 .

networks where nodes are associated with attribute-value pairs that capture important information about them. Techniques that are oblivious to such rich information associated with nodes often tend to learn poorer quality node representations, adversely impacting downstream ML tasks [14]. For example, consider a pair of users, u_1 and u_2, in a social network who are in topologically close proximity. Assume that u_1 is interested in the game of cricket whereas u_2's interest lies in canoeing. Ignoring such attribute information of u_1 and u_2 by simply considering only their neighborhoods' structural features may lead to an inferior-quality vector space representation of them. As an aftermath, a link may be predicted between u_1 and u_2 due to their topological similarity although they are individuals with highly dissimilar taste. Furthermore, the information associated with node attributes are even more useful in sparse scale-free networks where such information can complement scant topological information for learning superior embedding vectors.

Attributed network embedding (ANE) [14] aims to map *both* topological and attribute information surrounding a node to an embedding vector to facilitate superior network inference tasks. At first glance, it may seem that we can treat topology and attributes as separate features to address the ANE problem. Unfortunately, doing so loses the important information of *node-attribute affinity* [9], *i.e.*, attributes that can be reached by a node through one or more hops along the edges in the network. Hence, the key challenge to address the ANE problem is to devise efficient and scalable ways to *integrate* these information for network embedding.

Research Challenges and Gap in ANE. Effective ANE computation is a highly challenging task, especially for massive graphs. In particular, each node v in a network G could be associated with a large number of attributes, adding up to the number of dimensions. Furthermore, 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 can be broadly classified into two categories, *factorization-based* and *auto-encoder-based* approaches. Unfortunately, these solutions are prohibitively expensive and largely fail on massive networks. Factorization-based methods [14–16] are based on the idea of reducing an $n \times n$ matrix, where n is the number of nodes in G, into its smaller constituent parts so that embeddings can be discovered from the latter. In order to realize this, the $n \times n$ matrix often needs to be explicitly constructed and factorized. For a graph with 50 million nodes, storing such a matrix of double-precision values would require over 20 petabytes of memory, which is clearly infeasible. On the other hand, auto-encoder-based strategies [8, 9, 11] employ deep neural networks to extract higher-order features from nodes' connections and attributes. For a large dataset, training such a neural network incurs vast computational costs. In addition, the training process is usually done on GPUs with limited graphics memory. Consequently, for massive graphs, currently the only option is to compute ANE leveraging a large cluster, which is rather expensive, and has a significant environmental impact.

In addition, many existing ANE solutions are designed for undirected graphs. In reality, directed networks are common; as we shall see later, these methods yield suboptimal result quality on directed networks.

Gain with PANE. In this paper, we provide an affirmative answer to the following question of significance: *Can we efficiently compute effective ANE embeddings on a massive, attributed, directed graph on a single server?* To this end, we present `PANE`, a novel solution that significantly advances the state of the art in ANE computation. The key idea behind our solution is to devise techniques inspired by established ideas used in data management to speed up and scale ANE operations. Specifically, `PANE` formulates ANE as an optimization problem with the objective of approximating *normalized multi-hop node-attribute affinity* using node-attribute co-projections [9], guided by a *shifted pairwise mutual information* (SPMI) metric. The affinity between a given node-attribute pair is defined via a novel random walk model with a flexible neighborhood sampling strategy specifically adapted to attributed networks. Further, we incorporate edge direction information by defining separate *forward* and *backward* affinity, embeddings, and SPMI metrics. Solving this optimization problem is still immensely expensive with off-the-shelf algorithms, as it involves the joint factorization of two $O(n \cdot d)$-sized matrices, where n and d are the numbers of nodes and attributes in the input data, respectively. Thus, `PANE` includes a novel solver with a key module that seeds the optimizer through a highly effective greedy algorithm, which drastically reduces the number of iterations till convergence. Finally, we devise database-inspired non-trivial parallelization of the `PANE` algorithm by utilizing modern multi-core CPUs judiciously without significantly compromising result quality.

Extensive experiments demonstrate that `PANE` consistently obtains high-utility embeddings with superior prediction accuracy for link prediction and node classification, at a fraction of the cost compared to existing methods. In particular, on the largest *Microsoft Academic Knowledge Graph* (*MAG*), `PANE` is the only viable solution on a single server, whose resulting embeddings lead to 0.965 AP for link prediction and 0.57 micro-F1[1] for node classification. Notably, it obtains these results using 10 CPU cores, 1TB memory, and within 12 hours running time.

Summary of Contributions. In summary, this paper makes the following contributions: (a) We formulate the ANE task as an optimization problem with the objective of approximating multi-hop node-attribute affinity. We consider edge direction in our objective by defining forward and backward affinity matrices using the SPMI metric. (b) 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. (c) We develop non-trivial parallelization techniques of `PANE` to further boost efficiency. (d) We experimentally demonstrate the superior performance of `PANE`, in terms of efficiency and effectiveness, against 10 competitors on 4 real datasets.

2. ATTRIBUTED NETWORK EMBEDDING

In this section, we formally introduce the notion of attributed network embedding (ANE) and discuss existing efforts to address this problem.

[1]The micro-F1 score, ranging from 0 to 1, is the harmonic mean of the precision and recall, which are computed through micro averaging [18].

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 m edges E_V, each connecting two nodes in V, (iii) a set of d attributes R, 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 weight $w_{i,j}$ (*i.e.*, attribute value). For example, given a user v_i in a social network and an attribute r_j representing age, weight $w_{i,j}$ denotes the value of v_i's age. Note that for a categorical attribute, we first transform it 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) as a pair of directed edges with opposing directions: (v_i, v_j) and (v_j, v_i).

The *neighbohood* of a node v is typically generated using a graph traversal strategy such as breadth-first search, depth-first search, or a random walk. Intuitively, it represents a set of nodes that are in "close" proximity of v. Specifically, the first-order proximity indicates existence of links between a pair of nodes whereas higher-order proximity reflects the neighborhood. Note that neighborhood of different nodes can be overlapping and may be of different sizes.

Given a space budget $k \ll n$ and $k > 0$ (*i.e.*, dimensionality), a *node embedding* function $f : V \longrightarrow \mathbb{R}^k$ maps each node $v \in V$ to a length-k real-valued vector in $\mathbb{R}^k$. The broad 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 of the neighborhood of v. Following previous work [9], we also allocate a space budget $\frac{k}{2}$ (detailed in Section 3.2) for each attribute $r \in R$, and aim to compute an *attribute embedding* vector for r of length $\frac{k}{2}$.

Related Work. Existing *factorization-based* methods [14–16] mainly involve two stages: (i) build an $n \times n$ *proximity matrix* that models the proximity between nodes based on graph topology or attribute information; (ii) factorize it via techniques such as stochastic gradient descent (SGD), alternating least square (ALS), and coordinate descent. As remarked earlier, all these methods incur immense overheads in building and factorizing the proximity matrix and are designed for undirected graphs only.

An auto-encoder 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 auto-encoder-based methods for ANE [8, 9, 11] either use proximity matrices as inputs or design various neural network structures for the auto-encoder. Typically, these methods suffer from severe efficiency issues due to the expensive training process of auto-encoders; further, none of them considers edge directions.

Lastly, there exist several techniques (*e.g.,* [13, 19]) that generate embeddings without matrix factorization or auto-encoder. They employ other expensive deep learning techniques, rendering them infeasible to handle massive graphs.

3. THE PANE ALGORITHM

This section presents the proposed `PANE` algorithm. We begin by introducing notations and terminology.Then, we describe the design principles and associated challenges in realizing `PANE`. Finally, we describe the sequential and parallel versions of the algorithm.

3.1 Terminology

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.

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

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

3.2 Design Principles and Challenges

Figure 1 depicts the `PANE` framework. Intuitively, we represent the input network in a way that is conducive for subsequent computation of *affinity* between node and attribute pairs. This information is subsequently exploited to generate the embeddings for each node. We elaborate on the design of these components and associated challenges.

Extended Graph. The broad goal here is to transform the attribute information associated with nodes in G to *special* nodes and edges to create a unified framework capturing topological and attribution information. To this end, we utilize the notion of *extended graph*, denoted by $\mathcal{G}$, which we explain with an example. Figure 1(ii) shows an example extended graph $\mathcal{G}$ constructed based on an input attributed network G (Figure 1(i)) consisting of 6 nodes v_1-v_6 and 5 attributes r_1-r_5. Observe that the nodes and edges in blue show the attribute associations E_R in G. Specifically, for each attribute $r_j \in R$, we create an additional node in $\mathcal{G}$; then, for each entry in E_R, *e.g.*, $(v_3, r_2, w_{3,2})$, 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_2), with an edge weight (*e.g.*, $w_{3,2}$). Note that in this example, nodes v_1 and v_2 are not associated with any attribute.

Forward Affinity and Backward Affinity. As remarked earlier, 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*) [7], a technique that has find successful usage in data management research for finding relevance score between two nodes [1, 7]. In the sequel, 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 $\mathbf{B}$.

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 $\mathbf{R}_r[v_l, r_j]$,

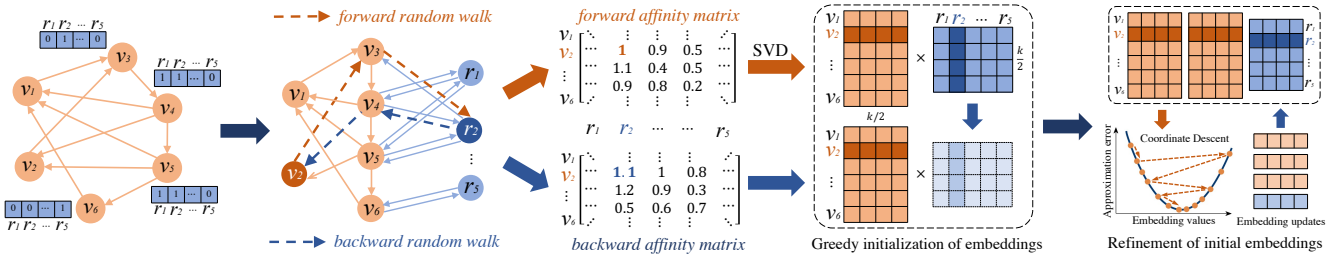

Figure 1: Overview of PANE

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 this pair to a collection $\mathcal{S}_f$.

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

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

To explain the intuition behind the above definition, note that in collection $\mathcal{S}_f$, the probabilities of observing node v_i, attribute r_j, and pair (v_i, r_j) are $\mathbb{P}(v_i) = \frac{1}{n}$, $\mathbb{P}(r_j) = \frac{\sum_{v_h \in V} p_f(v_h, r_j)}{n}$, 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)[3] [4] between node v_i and attribute r_j. Since PMI can be negative, we use the variant *shifted* PMI (SPMI), which is guaranteed to be positive. Hence, $\mathbf{F}[v_i, r_j]$ in Equation (2) is essentially SPMI(v_i, r_j).

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 and follows the aforementioned strategy. 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. Then the *backward affinity* $\mathbf{B}[v_i, r_j]$ is as follows.

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

Objective Function. The above notions of forward and backward node-attribute affinity capture the necessary information from which we can learn the embeddings of each node. Specifically, given a space budget k, our goal is to learn (i) two embedding matrices $\mathbf{X}_f$, $\mathbf{X}_b \in \mathbb{R}^{n \times \frac{k}{2}}$ for all nodes in V, whose row vectors $\mathbf{X}_f[v_i] \in \mathbb{R}^{\frac{k}{2}}$ and $\mathbf{X}_b[v_i] \in \mathbb{R}^{\frac{k}{2}}$ denote the *forward embedding vector* and *backward embedding vector* for node v_i, respectively, and (ii) an embedding matrix $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$ for all attributes in R, where each row vector $\mathbf{Y}[r_j] \in \mathbb{R}^{\frac{k}{2}}$ is the *attribute embedding vector* for attribute r_j. Mathematically, we can express this objective as to learn $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ such that the following objective is minimized:

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

Intuitively, we approximate the two affinities between node v_i and attribute r_j using the dot product of their respective embedding vectors. The objective is then to minimize the total squared error of such approximations, over all nodes and all attributes in the input data.

Challenges. A keen reader may observe that it is prohibitively expensive 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 conduct in a huge number from every node and attribute of massive graphs. Second, our objective function preserves both forward and backward affinity (*i.e.*, it takes into account 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. We tackle these challenges in the next subsections.

3.3 Seq-PANE: A Sequential Algorithm

Intuitively, PANE consists of three 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, (ii) initializing the embedding vectors with a greedy algorithm for fast convergence, and then (iii) jointly factorizing $\mathbf{F}'$ and $\mathbf{B}'$ using *cyclic coordinate descent* to efficiently obtain the embedding vectors $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$.

We first describe the single-threaded version of these three steps, referred to as Seq-PANE. The multi-threaded version that boosts efficiency further is elaborated later.

Step 1. Forward and backward affinity approximation. In order to avoid numerous random walks to compute exact node-attribute values, we transform forward and backward affinity in Equations (2) and (3) into their matrix forms and utilize iterative matrix multiplications to efficiently approximate forward and backward affinity matrices with error

[2] In the degenerate case that v_l is not associated with any attribute, *e.g.*, v_1 in Figure 1(ii), 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.

guarantee and in linear time complexity, without actually sampling random walks.

Observe that in Equations (2) and (3), the key for forward and backward affinity computation is to obtain $p_f(v_i, r_j)$ and $p_b(v_i, r_j)$ for every pair $(v_i, r_j) \in V \times R$. Recall that $p_f(v_i, r_j)$ is the probability that a forward random walk starting from node v_i picks attribute r_j, while $p_b(v_i, r_j)$ is the probability of a backward random walk from attribute r_j stopping at node v_i. Given nodes v_i and v_l, denote $\pi(v_i, v_l)$ as the probability that a random walk starting from v_i stops at v_l, *i.e.*, the random walk score of v_l with respect to v_i. By definition, $p_f(v_i, r_j) = \sum_{v_l \in V} \pi(v_i, v_l) \cdot \mathbf{R}_r[v_l, r_j]$, where $\mathbf{R}_r[v_l, r_j]$ is the probability that node v_l picks attribute r_j, according to Equation (1). Similarly, $p_b(v_i, r_j)$ is formulated as $p_b(v_i, r_j) = \sum_{v_l \in V} \mathbf{R}_c[v_l, r_j] \cdot \pi(v_l, v_i)$, where $\mathbf{R}_c[v_l, r_j]$ is the probability that attribute r_j picks node v_l from all nodes having r_j based on their attribute weights. By the definition of random walk scores in [7], we can derive the matrix form of p_f and p_b as follows.

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

where $\mathbf{P}$ is the random walk matrix (a.k.a transition matrix) of G and $\mathbf{P}^\ell[v_i, v_j]$ denotes the probability that a length-ℓ ($\ell \geq 1$) random walk from node v_i would end at node v_j. We only consider t iterations to approximate $\mathbf{P}_f$ and $\mathbf{P}_b$ in Equation (6), where t is set to $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$ and ϵ is an additive error threshold. This ensures $|\mathbf{P}_f[v_i, r_j] - \mathbf{P}_f^{(t)}[v_i, r_j]| \leq \epsilon$ and $|\mathbf{P}_b[v_i, r_j] - \mathbf{P}_b^{(t)}[v_i, r_j]| \leq \epsilon$ for every $(v_i, r_j) \in V \times R$.

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

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

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

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

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

In order to obtain the embedding vectors of all nodes and attributes, *i.e.*, $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$, we need to jointly factorize the approximate forward and backward affinity matrices $\mathbf{F}'$ and $\mathbf{B}'$. This can be done based on the *cyclic coordinate descent* (*CCD*) framework, which iteratively updates each embedding value towards optimizing the objective function in Equation (4). However, 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 (Step 2), and then design efficient techniques to refine the initial embeddings, including dynamic maintenance and partial updates of intermediate results to avoid redundant computations in CCD (Step 3), ideas that are inspired from data management techniques.

Step 2. Greedy initialization of the embeddings. In many optimization problems, all we need for efficiency is a good initialization. Thus, a key component in the joint factorization is such an initialization of embedding values, based on *singular value decomposition* (*SVD*). Note that unlike other matrix factorization problems, here SVD cannot be directly utilized to 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.

Specifically, the initialization method first employs an efficient randomized SVD algorithm [10] 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 $\mathbf{X}_f = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{Y} = \mathbf{V}$, which satisfies $\mathbf{X}_f \cdot \mathbf{Y}^\top \approx \mathbf{F}'$. In other words, this initialization immediately gains a good approximation of the forward affinity matrix.

Recall that our objective function in Equation (4) also aims to find $\mathbf{X}_b$ such that $\mathbf{X}_b \mathbf{Y}^\top \approx \mathbf{B}'$, *i.e.*, to approximate the backward affinity matrix well. We observe that matrix $\mathbf{V}$ (*i.e.*, $\mathbf{Y}$) returned by exact SVD is *unitary*, *i.e.*, $\mathbf{Y}^\top \mathbf{Y} = \mathbf{I}$, which implies that $\mathbf{X}_b \approx \mathbf{X}_b \mathbf{Y}^\top \mathbf{Y} \approx \mathbf{B}'\mathbf{Y}$. Accordingly, we seed $\mathbf{X}_b$ with $\mathbf{B}'\mathbf{Y}$. 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 CCD is drastically reduced (shown in Section 4).

Step 3. Efficient refinement of the initial embeddings. After initializing $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, we apply CCD to refine the embedding vectors according to our objective function in Equation (4). The basic idea of CCD is to cyclically iterate through all entries in $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, 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:

$$\mathbf{X}_f[v_i, l] \leftarrow \mathbf{X}_f[v_i, l] - \mu_f(v_i, l), \tag{8}$$

$$\mathbf{X}_b[v_i, l] \leftarrow \mathbf{X}_b[v_i, l] - \mu_b(v_i, l), \tag{9}$$

$$\mathbf{Y}[r_j, l] \leftarrow \mathbf{Y}[r_j, l] - \mu_y(r_j, l), \tag{10}$$

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

$$\mu_f(v_i, l) = \frac{\mathbf{S}_f[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \quad \mu_b(v_i, l) = \frac{\mathbf{S}_b[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \tag{11}$$

$$\mu_y(r_j, l) = \frac{\mathbf{X}_f^\top[l] \cdot \mathbf{S}_f[:, r_j] + \mathbf{X}_b^\top[l] \cdot \mathbf{S}_b[:, r_j]}{\mathbf{X}_f^\top[l] \cdot \mathbf{X}_f[:, l] + \mathbf{X}_b^\top[l] \cdot \mathbf{X}_b[:, l]}, \tag{12}$$

where $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$.

However, directly applying the above updating rules to learn $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ is inefficient, leading to numerous redundant matrix operations. Hence, in each iteration of CCD, we first fix $\mathbf{Y}$ and updates each row of $\mathbf{X}_f$ and $\mathbf{X}_b$, and then updates each column of $\mathbf{Y}$ with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed. According to Equations (11) and (12), $\mu_f(v_i, l)$, $\mu_b(v_i, l)$, 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, $\mathbf{S}_f$ and $\mathbf{S}_b$ need to be updated accordingly.

Directly recomputing $\mathbf{S}_f$ and $\mathbf{S}_b$ by $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$ whenever an entry in $\mathbf{X}_f, \mathbf{X}_b$ and, $\mathbf{Y}$ is updated is expensive. Instead, we dynamically maintain and partially update $\mathbf{S}_f$ and $\mathbf{S}_b$ according to Equations (13)-(15). Specifically, when $\mathbf{X}_f[v_i, l]$ and $\mathbf{X}_b[v_i, l]$ are updated, we update $\mathbf{S}_f[v_i]$ and $\mathbf{S}_b[v_i]$ respectively in $O(d)$ time by

$$\mathbf{S}_f[v_i] \leftarrow \mathbf{S}_f[v_i] - \mu_f(v_i, l) \cdot \mathbf{Y}[:, l]^\top. \tag{13}$$

$$\mathbf{S}_b[v_i] \leftarrow \mathbf{S}_b[v_i] - \mu_b(v_i, l) \cdot \mathbf{Y}[:, l]^\top. \tag{14}$$

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

$$\begin{aligned} \mathbf{S}_f[:, r_j] &\leftarrow \mathbf{S}_f[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_f[:, l], \\ \mathbf{S}_b[:, r_j] &\leftarrow \mathbf{S}_b[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_b[:, l]. \end{aligned} \tag{15}$$

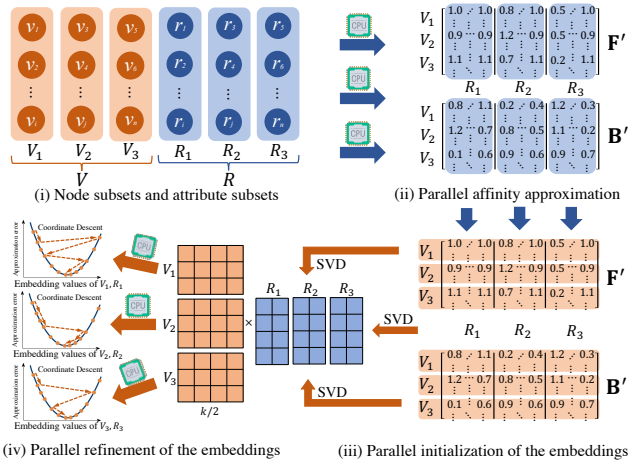

Figure 2: Overview of `Par-PANE`.

Complexity Analysis. Step 1 runs in $O(mdt) = O\left(md \cdot \log \frac{1}{\epsilon}\right)$ time. Meanwhile, given $\mathbf{F}' \in \mathbb{R}^{n \times d}$ as input, randomized SVD in Step 2 requires $O(ndkt)$ time [10]. The computation of $\mathbf{S}_f, \mathbf{S}_b$ in Step 3 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 `Seq-PANE` is $O\left((md + ndk) \cdot \log\left(\frac{1}{\epsilon}\right)\right)$.

3.4 Par-PANE: A Parallel Algorithm

Although `Seq-PANE` runs in linear time to the size of the input network, as we shall see in Section 4, it still consumes substantial amount of time in handling massive attributed networks in practice. We now present a parallel version of `PANE`, called `Par-PANE`, that draws inspiration from the exploitation of multi-core CPUs in many modern data management techniques and beyond to enhance performance.

It is challenging to develop a parallel algorithm achieving linear scalability to the number of threads on a multi-core CPU. `PANE` involves intensive matrix computation, factorization, and CCD updates, which are non-trivial to parallelize. Maintenance of the intermediate result of each thread and combining them to create the final result further aggrandize this challenge. We address these challenges in `Par-PANE`.

Figure 2 depicts an overview of `Par-PANE`. Compared to the `Seq-PANE`, `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. It parallelizes the three key steps in `Seq-PANE` as follows.

Step 1. Parallel forward and backward affinity approximation. We adopt block matrix multiplication to to estimate forward and backward affinity matrices $\mathbf{F}'$ and $\mathbf{B}'$ in a parallel manner. After obtaining $\mathbf{R}_r$ and $\mathbf{R}_c$ based on Equation (1), we divide $\mathbf{R}_r$ 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, the matrix multiplications for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ are parallelized, using n_b threads in t iterations. Then, 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. $\mathbf{P}_b{}^{(t)}$) in the main thread. Afterwards, we normalize $\widehat{\mathbf{P}}_f^{(t)}$ and $\widehat{\mathbf{P}}_b^{(t)}$, and then start n_b threads to compute $\mathbf{F}'$ and $\mathbf{B}'$ block by block in parallel, based on the definitions of forward and backward affinity.

Table 1: Datasets. (K=10^3, M=10^6)

Name	$\lvert V\rvert$	$\lvert E_V\rvert$	$\lvert R\rvert$	$\lvert E_R\rvert$	$\lvert L\rvert$	Refs
Citeseer	3.3K	4.7K	3.7K	105.2K	6	[8, 9, 11, 14, 16, 19]
Facebook	4K	88.2K	1.3K	33.3K	193	[9]
TWeibo	2.3M	50.7M	1.7K	16.8M	8	-
MAG	59.3M	978.2M	2K	434.4M	100	-

Step 2. Parallel initialization of the embeddings. To further improve the efficiency of the joint affinity matrix factorization process, we design a parallel algorithm with a split-and-merge-based parallel SVD technique for embedding vector initialization. It takes as input $\mathbf{F}'$, $\mathbf{B}'$, $\mathcal{V}$, and k. Based on $\mathcal{V}$, it 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 $V_i \in \mathcal{V}$. After obtaining $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$, we merge these matrices by concatenating $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$ into $\mathbf{V} = [\mathbf{V}_1 \ \cdots \ \mathbf{V}_{n_b}]^\top \in \mathbb{R}^{\frac{kn_b}{2} \times d}$, and then applies `RandSVD` over it to obtain $\mathbf{W} \in \mathbb{R}^{\frac{kn_b}{2} \times \frac{k}{2}}$ and $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$. Next, it 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 $\mathbf{X}_b[V_i]$, as well as computing intermediate results $\mathbf{S}_f$ and $\mathbf{S}_b$ for CCD.

Step 3. Parallel refinement of the initial embeddings. After obtaining initial embeddings $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$, we train them by CCD in parallel based on subsets $\mathcal{V}$ and $\mathcal{R}$, in t iterations. In each iteration, we first fix $\mathbf{Y}$ 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.

Note that `Par-PANE` does not return exactly the same outputs as `Seq-PANE`, as some modules (*e.g.*, the parallel version of SVD) introduce additional error. Nevertheless, as demonstrated in Section 4, the degradation of result quality in `Par-PANE` is small, but the speedup is significant.

Complexity Analysis. Based on the complexity analysis of `Seq-PANE`, it can be shown that the computational time complexity per thread in `Par-PANE` is $O\left(\frac{md+ndk}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$.

4. EXPERIMENTS

In this section, we investigate the performance of `PANE` on two tasks (link prediction and node classification). 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. The code of `PANE` is available at `https://github.com/AnryYang/PANE`.

Datasets. Table 1 lists the datasets (available at `https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html`) used in our experiments. $|V|$ and $|E_V|$ denote the number of nodes and edges in the graph, whereas $|R|$ and $|E_R|$ represent the number of attributes and node-attribute associations (*i.e.*, nonzero entries in attribute matrix $\mathbf{R}$), respectively. In addition, L is the set of *node labels* used in the node classification task. *Citeseer* and *Facebook* are benchmark datasets used in prior work. *TWeibo* is extracted from *Tencent Weibo* social network. *MAG* is extracted from the *Microsoft Academic Knowledge Graph*.

4.1 Experiments Setup

Baselines and Parameter Settings. We compare `Seq-PANE` and `Par-PANE` against 10 state-of-the-art competitors: eight

Table 2: Link prediction performance.

Method	*Citeseer*		*Facebook*		*TWeibo*		*MAG*	
	AUC	AP	AUC	AP	AUC	AP	AUC	AP
NRP	0.86	0.808	0.969	0.973	0.967	0.979	0.915	0.92
GATNE	0.687	0.767	0.961	0.954	-	-	-	-
TADW	0.895	0.868	0.752	0.793	-	-	-	-
BANE	0.899	0.873	0.796	0.795	-	-	-	-
PRRE	0.895	0.855	0.899	0.884	-	-	-	-
STNE	0.71	0.781	0.962	0.957	-	-	-	-
CAN	0.734	0.652	0.714	0.639	-	-	-	-
LQANR	0.916	0.916	0.951	0.917	-	-	-	-
Seq-PANE	0.932	0.919	0.982	0.982	0.976	0.986	0.96	0.965
Par-PANE	0.929	0.916	0.98	0.979	0.975	0.985	0.958	0.962

recent ANE methods including BANE [16], CAN [9], STNE [8], PRRE [19], TADW [14], ARGA [11], DGI [13] and LQANR [15], one state-of-the-art homogeneous network embedding method NRP [17], and one latest attributed heterogeneous network embedding algorithm GATNE [2].

The parameters of all competitors are set as suggested in their respective papers. For Seq-PANE and Par-PANE, by default we set error threshold $\epsilon = 0.015$ and random walk stopping probability $\alpha = 0.5$. We use $n_b = 10$ threads for Par-PANE. Unless otherwise specified, we set space budget $k = 128$. In our study, a method is excluded if it cannot finish training within one week.

Performance metrics. Following [9, 11], we adopt the *Area Under Curve* (AUC) and *Average Precision* (AP) metrics to measure the performance of the methods for the link prediction task. We use *Micro-F1* to measure node classification performance [9, 15]. Lastly, we use running time to measure efficiency and scalability.

4.2 Effectiveness

Link Prediction. Link prediction aims to predict the edges that are most likely to form between nodes. We first randomly remove 30% edges in input graph G, obtaining a residual graph G' and a set of the removed edges. We then randomly sample the same amount of non-existing edges as negative edges. The test set E' contains both the removed edges and the negative edges. We run PANE and all competitors on the residual graph G' to produce embedding vectors, and then evaluate the link prediction performance with E' as follows. For PANE, we calculate $p(v_i, v_j)$ as the prediction score of the directed edge (v_i, v_j):

$$p(v_i, v_j) = \sum_{r_l \in R} (\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top) \cdot (\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top) \approx \sum_{r_l \in R} \mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]. \quad (16)$$

We adopt four prediction methods (inner product, cosine similarity, Hamming distance, and edge features) over each method and report the best performance on each dataset.

Table 2 reports the AUC and AP scores of representative methods on each dataset. Observe that these scores for Seq-PANE are similar or superior to the competitors over all datasets. Furthermore, Par-PANE has comparable performance with Seq-PANE over all datasets.

Node Classification. Node classification predicts the node labels. We first run the methods on the input attributed network G to obtain their embeddings. Then we randomly sample a certain number of nodes (ranging from 10% to 90%) to train a linear support-vector machine (SVM) classifier and use the rest for testing. NRP, Seq-PANE, and Par-PANE generate a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$. So we normalize the forward and backward embeddings of each node v_i, and then concatenate them as the feature representation of v_i to be fed into the classifier. We repeat for 5 times and report the average performance.

Figure 3 depicts the Micro-F1 results when varying the percentage of nodes used for training from 10% to 90% (*i.e.*, 0.1 to 0.9). Both versions of PANE consistently outperform all competitors on all datasets, demonstrating its effectiveness in capturing the topology and attribute information of the input attributed networks. Specifically, compared with the competitors, Seq-PANE achieves a significant gain up to 17.2% on *MAG*. Over all datasets, Par-PANE has similar performance to that of Seq-PANE.

4.3 Efficiency and Scalability

Figure 4 reports the running times (in log-scale). It does not include the time for loading datasets and outputting embeddings. PANE is significantly faster than all ANE competitors, often by orders of magnitude. Specifically, on *TWeibo* and *MAG*, most existing ANE solutions cannot finish within a week, while our proposed solutions are able to handle them efficiently. Observe that Par-PANE is up to 9 times faster than Seq-PANE over all datasets. For instance, on *MAG* dataset, Par-PANE requires 11.9 hours while Seq-PANE takes about five days, emphasizing the benefits brought by our parallelization techniques in Section 3.4. Importantly, this is achieved without compromising on result quality.

Figure 5a depicts the speedup of Par-PANE over single-thread version on *TWeibo* when varying the number of threads n_b from 1 to 20. When n_b increases, Par-PANE becomes faster than single-thread PANE, demonstrating the parallel scalability of Par-PANE. Figures 5b and 5c report the running time of Par-PANE when varying space budget k and error threshold ϵ, respectively. In Figure 5b, observe that the running time remain stable and grows slowly with increasing k. In Figure 5c, the running time of Par-PANE decreases considerably with increasing ϵ, which is consistent with our analysis that PANE runs in linear to $\log(1/\epsilon)$.

5. FUTURE WORK

PANE opens up future research in multiple directions. First, state-of-the-art ANE frameworks do not provide any explanation of results of various downstream tasks such as link prediction. While a lack of explanation may not be critical for some applications (*e.g.*, friend recommendation in a social network), in others it is paramount for the acceptability and usage of an ANE framework. For instance, a biological signaling network models interactions (*i.e.*, biochemical reactions) between molecular species in a biological system. An example network is the human cancer signaling network [6] (CSN), which can be modeled as an attributed network. Predicting links between molecules (*e.g.*, proteins, genes) in a CSN without justification is unsatisfactory, as an oncologist would like to know the biological reasons behind such prediction. PANE could be the basis for a framework to build such explanation capabilities at a low cost.

Second, PANE makes remarkable progress in efficiency and scalability on CPUs. Naturally, further inroads can be made on performance by exploiting a GPU/multi-GPU environment. Further, while networks such as CSN may not evolve rapidly, many other real-world networks do (*e.g.*, social networks). These networks may also have different types of

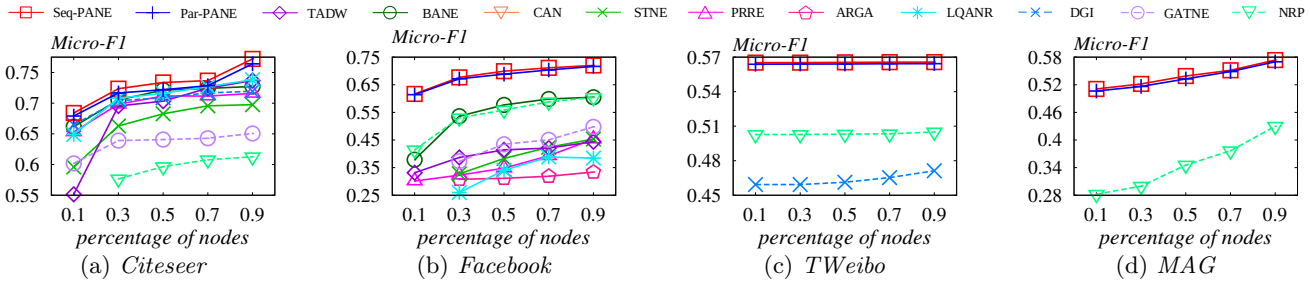

Figure 3: Node classification results (best viewed in color).

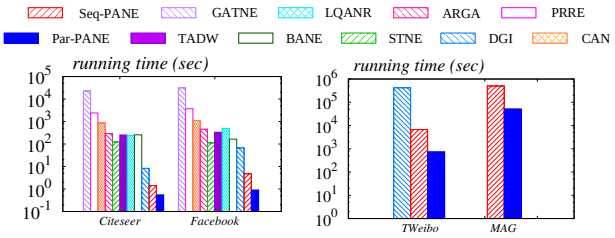

Figure 4: Running time (best viewed in color).

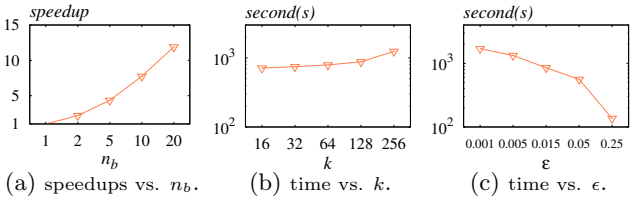

Figure 5: Efficiency with varying parameters on *TWeibo*.

links. Hence, it is also important to explore how PANE can be expanded to handle dynamic and heterogeneous networks.

Last but not the least, PANE can be a substrate to build scalable ANE-based solutions for real-world applications.For example, consider the problem of *target combination prediction* in signaling networks [3], where the goal is to predict target (nodes) combinations that can effectively modulate a set of *disease nodes*[4] in order to achieve a specific therapeutic goal (*e.g.*, reducing the activity of ERKPP protein by 50%). An *in silico* solution to this problem can aid in early rejection of unsuitable targets and guide the design of further *in vitro* and *in vivo* drug combination experiments, thereby reducing the cost and time for drug development. An effective solution needs to analyze the topology of the neighborhood of diseases nodes along with their disease-specific roles in order to capture crosstalks between pathways and their impact on targets. We are currently exploring how PANE can facilitate this by analyzing topological and disease-related attribute similarities of the neighborhood nodes encoded in their embeddings.

6. ACKNOWLEDGMENTS

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 (1-BE3T) of research project (P0033898) from the Hong Kong Polytechnic University. The findings herein reflect the work, and are solely the responsibility, of the authors.

[4]A disease node is a molecule that is either involved in some dysregulated biological processes implicated in a disease (*e.g.*, breast cancer) or is of interest due to its potential role (*e.g.*, oncogene) in the disease.

7. REFERENCES

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[2] Y. Cen, X. Zou, J. Zhang, H. Yang, J. Zhou, and J. Tang. Representation learning for attributed multiplex heterogeneous network. In *KDD*, pages 1358–1368, 2019.

[3] H. E. Chua, S. S. Bhowmick, and L. Tucker-Kellogg. Synergistic target combination prediction from curated signaling networks: Machine learning meets systems biology and pharmacology. *Methods*, 129:60–80, 2017.

[4] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Computational linguistics*, 16(1):22–29, 1990.

[5] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *TKDE*, 31(5):833–852, 2019.

[6] Q. Cui, Y. Ma, M. Jaramillo, H. Bari, A. Awan, S. Yang, S. Zhang, L. Liu, M. Lu, M. O'Connor-McCourt, et al. A map of human cancer signaling. *Molecular systems biology*, 3(1):152, 2007.

[7] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5):442–453, 2012.

[8] J. Liu, Z. He, L. Wei, and Y. Huang. Content to node: Self-translation network embedding. In *KDD*, pages 1794–1802, 2018.

[9] Z. Meng, S. Liang, H. Bao, and X. Zhang. Co-embedding attributed networks. In *WSDM*, pages 393–401, 2019.

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

[11] S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang. Adversarially regularized graph autoencoder for graph embedding. In *IJCAI*, pages 2609–2615, 2018.

[12] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[13] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm. Deep Graph Infomax. In *ICLR*, pages 1843–1852, 2019.

[14] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[15] H. Yang, S. Pan, L. Chen, C. Zhou, and P. Zhang. Low-bit quantization for attributed network representation learning. In *IJCAI*, pages 4047–4053, 2019.

[16] H. Yang, S. Pan, P. Zhang, L. Chen, D. Lian, and C. Zhang. Binarized attributed network embedding. In *ICDM*, pages 1476–1481, 2018.

[17] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick. Homogeneous network embedding for massive graphs via reweighted personalized pagerank. *PVLDB*, 13(5):670–683, 2020.

[18] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1):69–90, 1999.

[19] S. Zhou, H. Yang, X. Wang, J. Bu, M. Ester, P. Yu, J. Zhang, and C. Wang. Prre: Personalized relation ranking embedding for attributed networks. In *CIKM*, pages 823–832, 2018.