Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs

XINYU DU, The Chinese University of Hong Kong, China
XINGYI ZHANG, The Chinese University of Hong Kong, China
SIBO WANG*, The Chinese University of Hong Kong, China
ZENGFENG HUANG, Fudan University, China

Subset embedding is the task to learn low-dimensional representations for a subset of nodes according to the graph topology. It has applications when we focus on a subset of users, e.g., young adults, and aim to make better recommendations for these target users. In real-world scenarios, graphs are dynamically changing. Thus, it is more desirable to dynamically maintain the subset embeddings to reflect graph updates. The state-of-the-art methods, e.g., DynPPE, still adopt a hashing-based method, while hashing-based solutions are shown to be less effective than matrix factorization (MF)-based methods in existing studies. At the same time, MF-based methods in the literature are too expensive to update the embedding when the graph changes, making them inapplicable on dynamic graphs.

Motivated by this, we present Tree-SVD, an efficient and effective MF-based method for dynamic subset embedding. If we simply maintain the whole proximity matrix, then we need to re-do the MF, e.g., truncated Singular Value Decomposition (SVD), on the whole matrix after graph updates, which is prohibitive. To tackle this issue, our main idea is to do hierarchical SVD (HSVD) on the proximity matrix of the given subset, which vertically divides the proximity matrix into multiple sub-matrices, and then repeatedly do SVD on sub-matrices and merge the intermediate results to obtain the final embedding. We first present Tree-SVD, which combines a sparse randomized SVD with an HSVD. Our theoretical analysis shows that our Tree-SVD gains the efficiency of sparse randomized SVD and the flexibility of the HSVD with theoretical guarantees. To further reduce update costs, we present a lazy-update strategy. In this strategy, we only update sub-matrices that changes remarkably in terms of the Frobenius norm. We present theoretical analysis to show the guarantees with our lazy-update strategy. Extensive experiments show the efficiency and effectiveness of Tree-SVD on node classification and link prediction tasks.

CCS Concepts: • **Information systems → Data mining**; • **Computing methodologies → Learning latent representations**; • **Mathematics of computing → Computations on matrices**.

Additional Key Words and Phrases: Subset Node Embedding; Tree-SVD; Proximity Matrix Factorization; Dynamic Graph

ACM Reference Format:
Xinyu Du, Xingyi Zhang, Sibo Wang, and Zengfeng Huang. 2023. Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs. *Proc. ACM Manag. Data* 1, 1, Article 96 (May 2023), 26 pages. https://doi.org/10.1145/3588950

*Sibo Wang is the corresponding author.

Authors' addresses: Xinyu Du, The Chinese University of Hong Kong, Hong Kong SAR, China, xydu@se.cuhk.edu.hk; Xingyi Zhang, The Chinese University of Hong Kong, Hong Kong SAR, China, xyzhang@se.cuhk.edu.hk; Sibo Wang, The Chinese University of Hong Kong, Hong Kong SAR, China, swang@se.cuhk.edu.hk; Zengfeng Huang, Fudan University, Shanghai, China, huangzf@fudan.edu.cn.

2836-6573/2023/5-ART96 $15.00
https://doi.org/10.1145/3588950

1 INTRODUCTION

Given an input graph G with n nodes, node embedding aims to learn a low-dimensional vector for each node to preserve the graph topology. Due to its wide applications in graph mining tasks like node classification, graph clustering, and graph reconstruction, it has attracted a plethora of research works, e.g., [7, 25, 27, 28, 38, 39, 42], to devise efficient and effective node embeddings. Although *graph neural networks (GNNs)* with feature information have achieved great success in many graph mining tasks, node embedding, which only uses the graph topology, is still irreplaceable. Firstly, as shown in existing research work [3, 10], the rich node feature information is not always available for downstream tasks like recommendations. In such scenarios, graph topology is the only information available. Besides, by focusing on the graph topology, node embedding provides a structure feature for each node, which is independent of downstream tasks. In contrast, existing GNNs, e.g., GCN [16], are typically end-to-end and needs different training process for different tasks. This may cause a much higher running cost since the graph is usually used to handle many tasks in real-world scenarios. Thus, node embedding provides a trade-off between the accuracy of downstream tasks and the training cost. Besides, we may integrate the derived node embedding for each node and the node feature together as the input for the training process. It is shown in a recent study [30] that such a solution can also gain satisfying performance. Thus, node embedding still has its wide application scenarios.

In the literature, most existing studies focus on deriving the embedding for the entire node set in the graph. However, subset embedding, which finds the embedding for a subset of nodes according to the *whole input graph* also finds important applications. For example, an IT company *(i)* may have a set of VIP users and want to provide better recommendations for these VIP users; *(ii)* may want to recommend to a targeted age group e.g., young adults; *(iii)* may want to recommend to users in the same city. By focusing on such groups, we only need to compute the rows in the proximity matrix of the subset S, and allow more memory to contain the non-zero entries for the proximity matrix of subset S. This allows the subset embedding to gain better performance for downstream tasks like node classification and link prediction than their global embedding counterparts. For example, given a randomly selected subset S, a state-of-the-art *matrix-factorization (MF)*-based node embedding method STRAP [39], named Global-STRAP, gains inferior classification micro-F1 score as shown in Table 1. However, when we only focus on the subset S and allow more computational/memory resources, e.g., deriving a more accurate proximity matrix for the subset S, the performance can be significantly improved. For example, Subset-STRAP is an extension of STRAP, where we only need to compute the rows in the proximity matrix of the subset S, and allow more memory to contain the non-zero entries for the proximity matrix of subset S. As shown in Table 1, it gains significant improvement, taking a lead by up to 35% on tested datasets. This demonstrates the huge potential of subset embeddings.

In real-world scenarios, graphs are usually dynamically changing. Thus, it is more desirable to dynamically maintain the subset embeddings to reflect graph changes. The state-of-the-art solution DynPPE proposed by Guo et al. [8] is the first research work focusing on dynamic subset embeddings. In DynPPE, they explore the classic proximity measure *personalized PageRank (PPR)* [26] that is widely adopted in many embedding frameworks [39, 42] to generate the final embedding. Given a source node s, the PPR $\pi_s(u)$ of a node u with respect to s is the probability that an α-decay random walk (aka. random walk with restart) from s terminates at node u. Here, an α-decay random walk starts from the source node s, and at each step, it either stops at the current node v (initially $v = s$) with probability α or randomly jumps to one of its out-neighbors with $1 - \alpha$ probability. The personalized PageRank score $\pi_s(u)$ indicates the importance of u from the viewpoint of s. In DynPPE, it first computes the personalized PageRank vector for each node s in the subset S

Table 1. Micro-F1 score (%) of node classification on subset embedding v.s. global embedding with 50% training ratio.

Method	Patent	Mag-authors	Wikipedia
Global-STRAP	37.67	34.73	48.67
Subset-STRAP	72.40	61.53	76.93
DynPPE	64.27	51.27	73.67

and then hashes the PPR vector to a d-dimensional vector. Such a method is more friendly to graph updates since we can incrementally update the PPR vector using existing dynamic PPR algorithms, e.g., [24, 41], and then re-hash the PPR vector according to the affected entries in each PPR vector to do the update. For PPR entries that are not changed, it simply avoids the update cost. However, hashing-based methods suffer from degraded performance in terms of effectiveness and are shown to be outperformed by MF methods [32]. This is further verified in our experiment as shown in Table 1 where DynPPE is outperformed by Subset-STRAP by a large margin in terms of effectiveness, where Subset-STRAP adopts truncated SVD to generate the embedding.

Existing state-of-the-art node embedding methods are mostly MF-based. However, global MF-based methods like NetSMF [28], NPR [38], Lemane [42] and STRAP [39] are difficult to adapt to the dynamic subset embedding setting. To explain, for methods that decompose the adjacency matrix, like NetSMF and NRP, it is difficult to keep only the proximity matrix of subset S. Instead, they need to involve the whole adjacency matrix to update the subset embedding. For Lemane [42], it is unclear how to update the proximity measure for subset S efficiently without re-computing the proximity matrix for S and how to update the stopping probabilities at each step after the graph update. For STRAP [39], there exist efficient algorithms to update the PPR matrix of the subset S, e.g., [24, 41], as shown in DynPPE [8]. However, it still needs to re-compute the truncated SVD when the input graph changes, resulting in unnecessarily high computational costs. There also exist several GNN-based methods [19, 21, 31] that employ long-short term memory [11] to capture the temporal information in the updates. However, as pinpointed in the state-of-the-art subset embedding algorithm DynPPE [8], these methods either need to have features as input or cannot be applied to large-scale dynamic graphs.

Motivated by the limitations of existing solutions on large dynamic graphs, we present an efficient and effective MF-based framework for subset embedding on large dynamic graphs. Following existing research work, e.g., DynPPE [8], STRAP [39], and NRP [38], we take PPR as the proximity measure. When the graph changes, the proximity matrix $\boldsymbol{M}_S$ for subset S is then efficiently updated with the existing dynamic PPR algorithm [41]. However, as we have mentioned above, existing state-of-the-art solutions with MF as the backbone are generally challenging to handle dynamic graphs because any update to $\boldsymbol{M}_S$ will lead to a recalculation of MF, e.g., truncated SVD, to $\boldsymbol{M}_S$, which is too time-consuming.

Our key idea is to divide the proximity matrix into sub-matrices, then do a *truncated SVD* (Ref. to Section 2.2 for its definition) for each one, and finally, merge the truncated SVD results hierarchically. We will see shortly why such a hierarchical structure is important to improve the update efficiency. In this paper, we first propose Tree-SVD, which constructs a hierarchical tree structure, makes a combination of sparse randomized SVD [4] and *hierarchical SVD (HSVD)* [14], and inherits both the efficiency of the sparse randomized SVD and the flexibility of the HSVD. We present theoretical analysis and show that Tree-SVD achieves an approximation guarantee comparable to HSVD and at the same time gains an improved time complexity over HSVD. As we will see in Section 6, Tree-SVD is far more efficient than HSVD and is even faster than the state-of-the-art sparse randomized SVD algorithm FRPCA [6] without sacrificing the embedding effectiveness in our empirical evaluation.

We note that the proposed Tree-SVD is not limited to subset embedding and can be used to speed up the SVD computation for any rectangular matrix $\boldsymbol{M}$ with c rows, n columns, and $c \ll n$.

The main advantage of such a tree structure is that we can now track the changes of different sub-matrices with graph evolution. If sub-matrices do not change, then we can simply use corresponding cached representations without any re-calculation. Based on this insight, we present dynamic algorithms for our Tree-SVD which only updates affected sub-matrices, saving computational costs. However, in practical applications, edges may get updated in batches and with such batch updates, it is more likely that many of the sub-matrices in the proximity matrix $\boldsymbol{M}_S$ are changed unevenly. Thus we try to design a measure to find out locally highly influenced sub-matrices with graph evolution for eager updates, whereas cache the past representation in the hierarchical structure and put off the update of sub-matrices with negligible differences to a future snapshot with truly meaningful changes, which forms our lazy update framework. As to be explained later, we resort to a measure in terms of the Frobenius norm with theoretical analysis and guarantees. To summarize, our key contributions are as follows.

- We show the importance of subset embedding, where we observe that subset embedding can help gain improvement (sometimes significant) on tasks like node classification and link prediction.
- We present Tree-SVD, an efficient and effective SVD framework, which provides an improved time complexity over HSVD [14] and at the same time provides an identical approximation guarantee. Our Tree-SVD is not limited to subset embedding and can be used to speed up SVD computation for rectangular matrices.
- We present our dynamic scheme for Tree-SVD, which only updates the affected sub-matrices to reduce the update cost. We further present a lazy-update strategy to reduce the update cost without sacrificing empirical accuracy for downstream tasks.
- Extensive experiments show that Tree-SVD outperforms existing non-MF-based methods by a large margin in terms of effectiveness. Besides, Tree-SVD is also far more efficient than existing MF-based methods without sacrificing accuracy, showing a better trade-off between update efficiency and accuracy.

2 PRELIMINARIES

2.1 Background

Problem definition. In this paper, we use snapshot to denote a meaningful timestamp at which we take out the graph and need to generate the node embedding for the graph at the current snapshot. For dynamic graphs, we model it as different graph snapshots $\mathcal{G}^0, \mathcal{G}^1, \cdots, \mathcal{G}^t, \cdots$, where there exists one or multiple updates between two snapshots. For ease of exposition, we assume graph updates are only edge insertions or edge deletions as a node insertion/deletion can be mapped to a set of edge insertions or deletions. The graph $\mathcal{G}^t = (E^t, V^t, \Delta^t)$ at snapshot t consists of the node set V^t that exists at snapshot t, the edge set E^t that exists at snapshot t, and the set Δ^t of edge events that occurred between snapshot $t-1$ and snapshot t. Following [15], the dynamic graph model is defined as follows to simulate the graph evolution process over time.

Definition 2.1 (Dynamic graph model [15]). A dynamic graph model is defined as an ordered set of snapshots $\mathcal{G} = \{\mathcal{G}^0, \mathcal{G}^1, \ldots, \mathcal{G}^\tau\}$ where $\mathcal{G}^0$ is empty and $\mathcal{G}^1$ is the initial graph. Let n_t and m_t be the number of nodes and edges at snapshot t, respectively. Define $\Delta^t = \left\{e_1^t, e_2^t, \ldots, e_{m^t}^t\right\}$ as the set of edge events from snapshot $t-1$ to t. For each edge event $e_i^t = \langle u, v, \text{event}\rangle$, the *event* has two types $\{Insert, Delete\}$, indicating that the edge is inserted or deleted.

Definition 2.2 (Dynamic subset embedding [8]). Given a dynamic graph $\left\{\mathcal{G}^0, \mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^\tau\right\}$ in Definition 2.1 and a subset $S = \left\{v_1, v_2, \ldots, v_{|S|}\right\}$, the dynamic subset embedding problem is

Table 2. Frequently used notations.

Notations	Descriptions
$G = (V, E)$	Graph with node set and edge set
n, m	Number of nodes and edges
$\mathcal{G}^t = (V^t, E^t, \Delta^t)$	Graph at snapshot t with edge set change
$\boldsymbol{\pi}_u(v)$, $d_{\text{out}}(v)$	PPR of v w.r.t. u and out-degree of v
α, r_{max}	Decay factor and threshold of PPR
$\boldsymbol{r}_s^t[u]$, $\boldsymbol{p}_s^t[u]$	Residue, estimation of u w.r.t. s at snapshot t
S, $\boldsymbol{M}_S$	Subset of nodes and proximity matrix of S
$\boldsymbol{X}^t$, d	Embedding at snapshot t and its dimension
$\boldsymbol{B}_{l,j}$	The j-th sub-matrix in level l of Tree-SVD
$\boldsymbol{U}_{l,j}\Sigma_{l,j}\boldsymbol{V}_{l,j}$	The three matrices of the SVD result to $\boldsymbol{B}_{l,j}$
Z, k	The updated index and branching factor of Tree-SVD
q, $b = k^{q-1}$	Level of Tree-SVD and the number of sub-matrices in the first level of Tree-SVD

to dynamically maintain embeddings along with each snapshot of the graph for subset S where $|S| \ll n$. Given any snapshot t, for $\forall v_i \in S$, let $\boldsymbol{x}_{v_i}^t$ denotes the embedding vector of node v at snapshot t, we denote the embedding matrix of all embedding vectors as follows:

$$\boldsymbol{X}^t := \left[\boldsymbol{x}_{v_1}^t, \boldsymbol{x}_{v_2}^t, \ldots, \boldsymbol{x}_{v_{|S|}}^t\right]^\top, v_i \in S \text{ and } \boldsymbol{x}_{v_i}^t \in \mathbb{R}^d.$$

At snapshot $t+1$, the task of dynamic subset embedding is to update the embedding matrix from $\boldsymbol{X}^t$ to $\boldsymbol{X}^{t+1}$.

Table 2 lists the notations that are frequently used in this paper.

Personalized PageRank (PPR). Consider graph $\mathcal{G}^t = (E^t, V^t, \Delta^t)$ at snapshot t. Given a source s and a target node v, recap that the PPR score $\pi_s(v)$ is the probability that an α-decay random walk from s stops at node v. Computing the exact PPR score is expensive on large graphs. Following DynPPE [8], we adopt the classic *Forward-Push* algorithm proposed by Anderson et al. [2] to derive the PPR scores for each node v with respect to a given source s. The pseudo-code of the Forward-Push algorithm [2] is shown in Algorithm 1. In particular, given a source node $s \in V$, it maintains two vectors: the estimation vector $\boldsymbol{p}_s$ and the residue vector $\boldsymbol{r}_s$. Initially, the estimation vector $\boldsymbol{p}_s[v]$ is set to zero for each node $v \in V$; the residue vector $\boldsymbol{r}_s$ is initialized as a one-hot vector where only the position of s is 1 and all other positions are zero. Whenever there exists a node whose residue over its out-degree is larger than threshold r_{max}, it performs a push operation (Algorithm 1 Lines 5-8). It terminates when no such node exists. After any number of push operations, the following invariant holds for any u [2, 12, 13, 18, 33–37]:

$$\boldsymbol{\pi}_s(u) = \boldsymbol{p}_s(u) + \sum_{v \in V} \boldsymbol{r}_s(v) \cdot \boldsymbol{\pi}_v(u).$$

By the above equation, we can see that $\boldsymbol{p}_s(u)$ stands as an estimation of $\boldsymbol{\pi}_s(u)$. However, there exists no approximation guarantee on the estimation on directed graphs, as mentioned in [41]. Recall that there exists a threshold r_{max}, which controls the trade-off between accuracy and running cost. With a smaller r_{max}, the more accurate the estimation is, the higher running costs it incurs since Forward-Push algorithm runs in $O(1/r_{max})$ cost. We tune r_{max} so that further reducing r_{max} will not improve the performance.

```
Algorithm 1: Forward-Push($G, \alpha, s, r_{max}, \boldsymbol{p}_s, \boldsymbol{r}_s$)
  Input: $G$, $\alpha$, source $s$, threshold $r_{max}$
  Output: residual vector $\boldsymbol{r}_s$, estimation vector $\boldsymbol{p}_s$
1 $\boldsymbol{p}_s = \boldsymbol{0}, \boldsymbol{r}_s = \boldsymbol{1}_s$;
2 while $\exists u \in V s.t. \frac{\boldsymbol{r}_s[u]}{d_{out}(u)} > r_{max}$ do
3     PUSH($u, G, \alpha, \boldsymbol{p}_s, \boldsymbol{r}_s$)
4 return $\boldsymbol{p}_s, \boldsymbol{r}_s$;
5 procedure PUSH($u, \alpha, G, \boldsymbol{p}_s, \boldsymbol{r}_s$):
6     for each out-neighbor $v$ of $u$ do
7         $\boldsymbol{r}_s[v] += (1-\alpha) \cdot \boldsymbol{r}_s[u] / d_{out}(u)$
8     $\boldsymbol{p}_s[u] += \alpha \cdot \boldsymbol{r}_s[u], \boldsymbol{r}_s[u] = 0$
```

Given the residue vector $\boldsymbol{r}_s^t$ and estimation vector $\boldsymbol{p}_s^t$ for each node $s \in S$ at snapshot t, if the graph has changed to snapshot $(t+1)$, then a straightforward solution is to re-run the Forward-Push algorithm for each $s \in S$ on $\mathcal{G}^{t+1}$, which might be too expensive. Zhang et al. [41] present a dynamic Forward-Push algorithm to incrementally update $\boldsymbol{r}_s^t$ and $\boldsymbol{p}_s^t$ to $\boldsymbol{r}_s^{t+1}$ and $\boldsymbol{p}_s^{t+1}$, respectively. The pseudo-code is shown in Algorithm 2. According to [41], it takes $O(|\Delta^{t+1}| + 1/r_{max})$ cost to incrementally update $\boldsymbol{r}_s^t$ (resp. $\boldsymbol{p}_s^t$) to $\boldsymbol{r}_s^{t+1}$ (resp. $\boldsymbol{p}_s^{t+1}$) for a uniformly chosen source node s.

2.2 Main Competitors

DynPPE. The state-of-the-art dynamic subset embedding method is DynPPE proposed by Guo et al. [8]. Given a subset S, DynPPE first derives an approximate PPR vector $\hat{\boldsymbol{\pi}}_s$ for each node $s \in S$ by invoking the Forward-Push algorithm (Algorithm 1). The proximity matrix of size $|S| \times n$ consists of $|S|$ PPR vectors. In DynPPE, they adopt hashing to generate the embedding for each node $s \in S$. In particular, they adopt a hash function $h : \mathbb{R}^n \longrightarrow \mathbb{R}^d$ to map each n-dimensional vector $\hat{\boldsymbol{\pi}}_s$ into a d-dimensional representation vector. When the graph changes from one snapshot to another, DynPPE uses Algorithm 2 to update the PPR vector for each node $s \in S$ and then re-hashes PPR vectors to the d-dimensional embedding space.

Subset-STRAP. As our main idea is to do truncated SVD on the proximity matrix and then incrementally update on the SVD results, another baseline of our Tree-SVD is to do truncated SVD on the proximity matrix of subset S from scratch at each snapshot. As we discussed in Section 1, most existing state-of-the-art matrix-factorization-based node embedding methods are difficult to be adapted to subset setting except STRAP [39]. Thus, we extend STRAP to the subset embedding setting and denote this extension as Subset-STRAP. Subset-STRAP follows STRAP, which imposes a threshold r_{max} and returns at most $O\left(\frac{1}{r_{max}}\right)$ proximity scores no smaller than r_{max} for each node, making a proximity matrix with non-zero entries of size $O\left(\frac{|S|}{r_{max}}\right)$. Since Subset-STRAP explicitly derives the proximity matrix, it allows taking non-linear operations on the proximity matrix, improving the representation powers. In particular, given the PPR matrix $\boldsymbol{M}_S$ at snapshot t for the subset S, the baseline method first takes a non-linear operation, e.g., log or sigmoid, on $\boldsymbol{M}_S$. Next, a *truncated SVD* is applied on $\boldsymbol{M}_S$, which decomposes $\boldsymbol{M}_S$ into three matrices: $\boldsymbol{U} \in \mathbb{R}^{|S|\times d}$ of the d left singular vectors corresponding to the d largest singular values of $\boldsymbol{M}_s$, $\boldsymbol{V} \in \mathbb{R}^{d\times n}$ of the d right singular vectors corresponding to the d largest singular values of $\boldsymbol{M}_s$, and a diagonal matrix $\Sigma \in \mathbb{R}^{d\times d}$ of d largest singular values of $\boldsymbol{M}_s$. In STRAP [39], they invoke a randomized SVD algorithm [6] to derive the truncated SVD result. The returned three matrices via the randomized

Algorithm 2: Dynamic Forward-Push

```
Input: $\mathcal{G}^t$, $\mathcal{G}^{t+1}$, $\Delta^{t+1}$, $\alpha$, node $s$, threshold $r_{max}$, residue vector $\boldsymbol{r}_s^t$, estimate vector $\boldsymbol{p}_s^t$
1  for each $\langle u, v, event \rangle \in \Delta^{t+1}$ do
2      if $event == INSERT$ then
3          $\boldsymbol{p}_s^t[u] \times = \frac{d_{out}(u)}{d_{out}(u)-1}$
4          $\boldsymbol{r}_s^t[u] - = \frac{\boldsymbol{p}_s^t[u]}{d_{out}(u)} \times \frac{1}{\alpha}$; $\boldsymbol{r}_s^t[v] + = \frac{(1-\alpha)\times \boldsymbol{p}_s^t[u]}{d_{out}(u)} \times \frac{1}{\alpha}$
5      else
6          $\boldsymbol{p}_s^t[u] \times = \frac{d_{out}(u)}{d_{out}(u)+1}$
7          $\boldsymbol{r}_s^t[u] + = \frac{\boldsymbol{p}_s^t[u]}{d_{out}(u)} \times \frac{1}{\alpha}$; $\boldsymbol{r}_s^t[v] - = \frac{(1-\alpha)\times \boldsymbol{p}_s^t[u]}{d_{out}(u)} \times \frac{1}{\alpha}$
8  while $\exists w \in V$ s.t. $\frac{\boldsymbol{r}_s^t[w]}{d_{out}(w)} > r_{max}$ do
9      PUSH($w, \alpha, \mathcal{G}^{t+1}, r_{max}, \boldsymbol{p}_s^t, \boldsymbol{r}_s^t$)
10 while $\exists w \in V$ s.t. $\frac{r_s[w]}{d_{out}(w)} < -r_{max}$ do
11     PUSH($w, \alpha, \mathcal{G}^{t+1}, r_{max}, \boldsymbol{p}_s^t, \boldsymbol{r}_s^t$)
12 return ($\boldsymbol{p}_s^{t+1} \leftarrow \boldsymbol{p}_s^t, \boldsymbol{r}_s^{t+1} \leftarrow \boldsymbol{r}_s^t$);
```

SVD algorithm have the following guarantee in terms of the Frobenius norm:

$$||\boldsymbol{U}\Sigma\boldsymbol{V} - \boldsymbol{M}_S||_F = (1+\epsilon) \min_{\mathbf{rank}(B)\le d} ||(\boldsymbol{B})_d - \boldsymbol{M}_S||_F. \quad (1)$$

Following [39], the embedding of Subset-STRAP is $\boldsymbol{X} = \boldsymbol{U}\sqrt{\Sigma}$. Note that taking $\boldsymbol{X} = \boldsymbol{U}\sqrt{\Sigma}$ is a classic choice in node embedding in existing works [38, 39] that gains good empirical results. Thus, we follow such a choice to be consistent with existing studies. The reason behind the choice is that the singular vectors corresponding to the top-d largest singular values keep the d most important dimensions. By scaling according to singular values on each of these d dimensions, the singular vectors with larger singular values tend to make more contributions to the final embedding. By taking the square root to the matrix Σ of the singular values, it imposes equal weights on both the matrix $\boldsymbol{U}$ of the left singular vectors and the matrix $\boldsymbol{V}$ of the right singular vectors.

FREDE. Another competitor, FREDE [32], adopts a matrix-sketching algorithm to factorize the PPR proximity matrix. Different from Tree-SVD, which maintains multiple intermediate SVD results and merges these results group by group, FREDE only maintains one SVD result. It first recursively reads in $2d$ rows of the proximity matrix and compresses them into d rows. Then it repeatedly merges d rows new vectors with existing compressed d rows to form a $2d$ rows matrix. After that, these $2d$ vectors are compressed to d-dimension matrix by SVD again. This process terminates until all rows are compressed to get the final d-dimension representation. Thus, FREDE works in a streaming way and can be extended to the subset embedding. However, it provides no guarantee in terms of the Frobenius norm and does not support dynamic updates. Moreover, as we will show in Section 6, it has inferior performance on tasks like node classification and link prediction.

RandNE. The last competitor is RandNE [43], which designs a proximity matrix of high-order adjacency matrices and adopts a Gaussian random projection approach rather than classic SVD operations for better efficiency. In the Gaussian random projection process, RandNE designs an iterative projection procedure, thus avoids the explicit calculation of high-order proximity matrices and further increase its efficiency. Again, such a solution can be extended to subset embedding. However, it also provide inferior performance on downstream tasks as we will see in the experiment.

Algorithm 3: Tree-SVD Node Embedding

```
Input: Proximity matrix $M_S = [M_{1,1}|M_{1,2}|\cdots|M_{1,b}]$, embedding dimension $d$
Output: Embedding matrix $X$
1 Set $B_{1,j} = M_{1,j}$ for $j = 1, 2, \cdots, b$
2 for $l = 1, 2..., q-1$ do
3     for $j = 1, 2, ..., b/k^{l-1}$ do
4         Compute the $d$-rank randomized SVD (resp. $d$-rank truncated SVDs) of $B_{l,j}$ if $j = 1$
          (resp. $j > 1$); keep the first $d$ singular vectors and values to get $(U_{l,j})_d$, $(\Sigma_{l,j})_d$ and
          form $(U_{l,j})_d \cdot (\Sigma_{l,j})_d$
5     for $j = 1, 2, ..., b/k^l$ do
6         $B_{l+1,j} := \left[(U_{l,(j-1)k+1})_d(\Sigma_{l,(j-1)k+1})_d, |\cdots|(U_{l,jk})_d(\Sigma_{l,jk})_d\right]$.
7 Compute the truncated SVD of $B_{q,1}$ for the first $d$ singular vectors and values to get
  $(U_{q,1})_d$, $(\Sigma_{q,1})_d$.
8 Set $X = (U_{q,1})_d\sqrt{(\Sigma_{q,1})_d}$
```

3 OUR SOLUTION: TREE-SVD

Recap that a major deficiency of existing SVD-based node embedding algorithms, e.g., STRAP [39], NRP [38], NetSMF [28], is that when the proximity matrix changes, the whole SVD results need to be re-computed from scratch. To avoid re-computations, our main idea is to vertically divide the proximity matrix M_S into multiple sub-matrices and do truncated SVD of dimension d on each sub-matrix as shown in Figure 1, where we divide the proximity matrix into k^2 sub-matrices $M_{1,1}, M_{1,2}, \cdots, M_{1,k\times k}$. Then, we maintain the intermediate SVD results for these k^2 sub-matrices. If a sub-matrix, say, $M_{1,1}$ gets affected, we do not need to re-compute the truncated SVD results for $M_{1,2}$ to $M_{1,k\times k}$, which significantly reduces the update cost. It provides more flexibility as now we are able to control which part of the matrix M_S to re-compute and which part of the matrix M_S to use past values. Given the first-level truncated SVD results, we concatenate these factorized sub-matrices in groups to form the intermediate matrices and feed them into the next level. Then, a truncated SVD is invoked on each intermediate matrix at the next level. Given the intermediate results, we repeatedly merge them into groups and do SVD on newly concatenated matrices. Each time, the column dimension of the matrix calculated by SVD is fixed to d. By these SVD operations, the overall dimension (as we maintain an $|S| \times d$ matrix for each sub-matrix) is reduced level by level. We will repeat this process in a hierarchical manner until there is only one matrix left. Then, our Tree-SVD terminates and outputs compressed representations calculated by the last truncated SVD as our final embedding. The level q of Tree-SVD is defined as the largest number of SVDs performed along with the path from the root to any leaf. Figure 1 shows the architecture of a 3-level Tree-SVD, where the bottom is the root.

Our Tree-SVD is a combination of sparse randomized SVD [4] and hierarchical SVD (HSVD) [14]. Concretely, given the original proximity matrix, we can easily partition it as a group of sub-matrices in the first level. With these sub-matrices in the first level of the hierarchical structure, we apply a truncated sparse randomized SVD to quickly embed each of these sub-matrices to a $|S| \times d$ matrix formed by $(U)_d(\Sigma)_d$. Note that the HSVD in [14] requires that the first level is an exact SVD to provide theoretical guarantees. We show that with the approximation ratio $(1+\epsilon)$ of the sparse randomized SVD at the first level, we still provide a comparable theoretical guarantee for Tree-SVD as shown in Theorem 3.2.

Note that this sparse randomized SVD is quite crucial because we tremendously reduce the column dimension from $O(n)$ to $O(d)$ in the first level, which is originally the time bottleneck of the HSVD with a slow exact SVD algorithm at the first level. For example, given a matrix with 6 million columns, we may reduce the column dimension from 6 million to $64 * 128$ by this sparse randomized SVD in our case, where 64 is the number of sub-matrices in the first level and 128 is the dimension. This significantly accelerates our computation. Then we can conduct exact SVD in the next few levels with very small column dimensions efficiently. Therefore, this two-step Tree-SVD scheme, as a combination of sparse randomized SVD and HSVD, inherits both the efficiency of the sparse randomized SVD and the flexibility of the HSVD.

Although we tremendously accelerate the computation and alleviate the original time bottleneck by bringing in randomized SVD in the first level, we note that the major computational cost of Tree-SVD still comes from the first level. Actually, this cost is still much higher than that of the exact SVD in the next few levels, since the dimension of the original proximity matrix may be larger by two orders of magnitude compared with that of the second-level matrices. Based on this observation, we may avoid the re-computation of SVD under dynamic settings by reducing the number of randomized SVD computations in the first level, which is the major bottleneck of Tree-SVD. Our strategy is that we maintain intermediate results, which are much smaller than the input proximity matrix. Then, during the update, if a sub-matrix is not updated, we can reuse the cached intermediate results without recalculating SVD on the sub-matrix from scratch. However, such a solution may still become ineffective if a batch update occurs, in which many sub-matrices may get changed. To tackle this issue, we present a lazy update framework to dynamically monitor the updates of each sub-matrix. If the matrix has changed by a large portion in terms of the Frobenius norm, then a re-calculation of SVD on the sub-matrix is performed. Next, we first show how our Tree-SVD works without any updates in Section 3.1. Then, we elaborate on our dynamic algorithm and lazy update strategy in Section 3.2.

3.1 Tree-SVD on Static Graphs

In this section, we present how Tree-SVD works on static graphs. Firstly, we derive the proximity matrix $\boldsymbol{M}_S$. For each node $s \in S$, we perform a Forward-Push on graph $\mathcal{G}$ and derive the estimation vector $\boldsymbol{p}_s$. Then, a Forward-Push is also performed for each node $s \in S$ on the reverse graph $\mathcal{G}^\top$ by reversing the direction of each edge. For the proximity matrix $\boldsymbol{M}_S$ of size $|S| \times n$, we have:

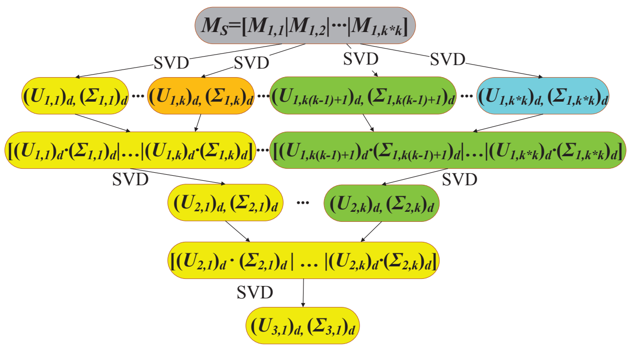
$$\boldsymbol{M}_S(s,v) = \log\left(\frac{\boldsymbol{p}_s(v)}{r_{max}} + \frac{\boldsymbol{p}_s^\top(v)}{r_{max}}\right).$$

Notice that in the above equation, dividing by r_{max} is to scale the values while taking the logarithm is to perform a non-linear transformation to improve the representation power, both of which are widely adopted in the literature [39, 42]. Given the proximity matrix $\boldsymbol{M}_S$, we divide the matrix into b different sub-matrices:

$$\boldsymbol{M}_S = [\boldsymbol{M}_{1,1}|\boldsymbol{M}_{1,2}|\cdots|\boldsymbol{M}_{1,b}],$$

where $\boldsymbol{M}_{1,j}$ denotes the j-th sub-matrix in the first level.

In the rest of the paper, we define $(\boldsymbol{M}_S)_d$ as the best d-rank approximation of $\boldsymbol{M}_S$, i.e., $||(\boldsymbol{M}_S)_d - \boldsymbol{M}_S||_F = \min_{\mathbf{rank}(A)\le d} ||(\boldsymbol{A})_d - \boldsymbol{M}_S||_F$. For a matrix $\boldsymbol{B}$, let $\boldsymbol{U}\Sigma\boldsymbol{V}$ be the truncated SVD of matrix $\boldsymbol{B}$, i.e., $\boldsymbol{B} = \boldsymbol{U}\Sigma\boldsymbol{V}$, where $\boldsymbol{U}$ is the left singular vector matrix, Σ is the diagonal singular value matrix, and $\boldsymbol{V}$ is the right singular vector matrix. We define $\bar{\boldsymbol{B}} = \boldsymbol{B}\boldsymbol{V}^* = \boldsymbol{U}\Sigma$. It is important to note that if $\boldsymbol{B}$ is rank deficient and/or has repeated singular values, $\bar{\boldsymbol{B}}$ is not necessarily uniquely determined by $\boldsymbol{B}$. Similarly, we define $\bar{\boldsymbol{B}} = \bar{\boldsymbol{C}}$, meaning that $\boldsymbol{B}$ and $\boldsymbol{C}$ are equivalent up to multiplication by a unitary matrix $\boldsymbol{E}$ on the right, i.e. $\boldsymbol{B} = \boldsymbol{C}\boldsymbol{E}$ or $\boldsymbol{B}\boldsymbol{E} = \boldsymbol{C}$.


M_S=[$M_{1,1}$|$M_{1,2}$|···|$M_{1,k*k}$]
SVD
SVD
SVD
SVD
$(U_{1,1})_d, (\Sigma_{1,1})_d$ ··· $(U_{1,k})_d, (\Sigma_{1,k})_d$ ··· $(U_{1,k(k-1)+1})_d, (\Sigma_{1,k(k-1)+1})_d$ ··· $(U_{1,k*k})_d, (\Sigma_{1,k*k})_d$
$[(U_{1,1})_d\cdot(\Sigma_{1,1})_d|\ldots|(U_{1,k})_d\cdot(\Sigma_{1,k})_d]$ ··· $[(U_{1,k(k-1)+1})_d\cdot(\Sigma_{1,k(k-1)+1})_d|\ldots|(U_{1,k*k})_d\cdot(\Sigma_{1,k*k})_d]$
SVD
SVD
$(U_{2,1})_d, (\Sigma_{2,1})_d$ ··· $(U_{2,k})_d, (\Sigma_{2,k})_d$
$[(U_{2,1})_d\cdot(\Sigma_{2,1})_d| \ldots |(U_{2,k})_d\cdot(\Sigma_{2,k})_d]$
SVD
$(U_{3,1})_d, (\Sigma_{3,1})_d$
Fig. 1. Tree-SVD with 3 levels

We denote $\boldsymbol{B}_{l,j}$ as the j-th sub-matrix that we factorize in the l-th level of Tree-SVD. Let $\boldsymbol{B}_{1,j} = \boldsymbol{M}_{1,j}$, i.e., the j-th sub-matrix of the input matrix $\boldsymbol{M}_S$ at the first level. Recap that for a sub-matrix $\boldsymbol{B}_{1,j}$ at the first level, we apply a randomized SVD to derive the best d-rank approximation of $\boldsymbol{B}_{1,j}$. Let $(\boldsymbol{B}_{1,j})_d = (\boldsymbol{U}_{1,j})_d(\Sigma_{1,j})_d(\boldsymbol{V}_{1,j})_d$ be the d-rank ϵ-approximation of $\boldsymbol{B}_{l,j}$ (Ref. to Eqn. 1) derived by the randomized SVD. For the second level, $\boldsymbol{B}_{2,j}$ is defined as follows:

$$\begin{aligned}\boldsymbol{B}_{2,j} &:= \left[\overline{(\boldsymbol{B}_{1,(j-1)\cdot k+1})_d} | \cdots | \overline{(\boldsymbol{B}_{1,j\cdot k})_d}\right] \\ &= \left[\boldsymbol{U}_{1,(j-1)\cdot k+1} \cdot \Sigma_{1,(j-1)\cdot k+1} \mid \cdots \mid \boldsymbol{U}_{1,j\cdot k}\Sigma_{l,j\cdot k}\right],\end{aligned}$$

i.e., we merge the SVD results of the first k consecutive sub-matrices ($\boldsymbol{B}_{1,1}, \boldsymbol{B}_{1,2}, \cdots, \boldsymbol{B}_{1,k}$) at the first level and produce $\boldsymbol{B}_{2,1}$, the first sub-matrix at level two; the SVD results of the next k consecutive sub-matrices ($\boldsymbol{B}_{1,k+1}, \boldsymbol{B}_{1,k+2}, \cdots, \boldsymbol{B}_{1,2k}$) produces $\boldsymbol{B}_{2,2}$, the second sub-matrix at level two, and etc.

More generally, for $l > 1$, given the sub-matrix $\boldsymbol{B}_{i,j} = \boldsymbol{U}_{l,j}\Sigma_{l,j}\boldsymbol{V}_{l,j}$, where $\boldsymbol{U}_{l,j}$, $\Sigma_{l,j}$, and $\boldsymbol{V}_{l,j}$ are the matrix of the left singular vectors, the diagonal matrix of the singular values, and the matrix of the right singular vectors of $\boldsymbol{B}_{l,j}$, respectively. Then, we take the d-rank truncated SVD of $\boldsymbol{B}_{l,j}$ by keeping the first d columns of $\boldsymbol{U}_{l,j}$, $\Sigma_{l,j}$, and $\boldsymbol{V}_{l,j}$, i.e., $(\boldsymbol{B}_{l,j})_d = (\boldsymbol{U}_{l,j})_d \cdot (\Sigma_{l,j})_d \cdot (\boldsymbol{V}_{l,j})_d$. Then, for the $(l+1)$-th level ($l > 1$), we define $\boldsymbol{B}_{\boldsymbol{l+1,j}}$ as follows:

$$\begin{aligned}\boldsymbol{B}_{l+1,j} &:= \left[(\overline{\boldsymbol{B}_{l,(j-1)k+1}})_d | \cdots | (\overline{\boldsymbol{B}_{l,jk}})_d\right] \\ &= \left[(\boldsymbol{U}_{l,(j-1)\cdot k+1})_d \cdot (\Sigma_{l,(j-1)\cdot k+1})_d \mid \cdots \mid (\boldsymbol{U}_{l,j\cdot k})_d \cdot (\Sigma_{l,j\cdot k})_d\right].\end{aligned}$$

Finally, on the last level q, there is only one matrix $\boldsymbol{B}_{q,1}$. We derive the d-rank truncated SVD of $\boldsymbol{B}_{q,1}$ and output $(\boldsymbol{U}_{q,1})_d$ and $(\Sigma_{q,1})_d$ as the final truncated SVD results. For the subset embedding, it returns $(\boldsymbol{U}_{q,1})_d\sqrt{(\Sigma_{q,1})_d}$ as the embedding of vertex set S. Here, we assume that k sub-matrices are aggregated together in each level, which is adopted in our implementation. We also call k here as the *branching factor* of Tree-SVD. In our implementation, given b sub-matrices at the first level, then there are b/k sub-matrices at the second level, b/k^2 sub-matrices at the third level, and finally 1 matrix at the last level q. Clearly, the number b of sub-matrices in the first level satisfies that $b = k^{q-1}$. Here, we further assume that n can be divided evenly by b and each matrix has the same size. Notice that, we make the above assumption for the ease of exposition and there are no such restrictions to Tree-SVD, where different levels may aggregate different numbers of sub-matrices and n does not need to be evenly divided by b.

Example 3.1. Figure 1 shows a 3-level Tree-SVD with a branching factor of k. If we set the branching factor $k = 8$, then 8 consecutive sub-matrices are aggregated together in each level and the number b of sub-matrices at the first level satisfies that $b = k^2 = 64$. Thus, we derive $\boldsymbol{B}_{2,1}$ by $\left[(\boldsymbol{U}_{1,1})_d \cdot (\Sigma_{1,1})_d \mid \cdots \mid (\boldsymbol{U}_{1,8})_d \cdot (\Sigma_{1,8})_d\right]$, derive $\boldsymbol{B}_{2,2}$ by $\left[(\boldsymbol{U}_{1,9})_d \cdot (\Sigma_{1,9})_d \mid \cdots \mid (\boldsymbol{U}_{1,16})_d \cdot (\Sigma_{1,16})_d\right]$, and etc. At the second level, we maintain 8 sub-matrices $(\boldsymbol{B}_{2,1}, \cdots \boldsymbol{B}_{2,8})$. We calculate the d-rank truncated SVD of each sub-matrix at the second level and merge the SVD results together to get the only matrix at the third level, i.e., $\boldsymbol{B}_{3,1}$. Finally, we compute the d-rank truncated SVD of $\boldsymbol{B}_{3,1}$ to get the final output embedding $(\boldsymbol{U}_{3,1})_d\sqrt{(\Sigma_{3,1})_d}$. □

Algorithm 3 shows the pseudo-code of Tree-SVD for static subset node embedding. The steps are self-explanatory with the above discussions. Theorem 3.2 shows the quality guarantee of Tree-SVD.

THEOREM 3.2. *Given the input matrix $\boldsymbol{M}_S$ and level $q \geq 2$, let $(\boldsymbol{U}_{q,1})_d, (\Sigma_{q,1})_d$ be the d-rank truncated SVD of the last level of Tree-SVD. Assume that the randomized SVD in the first level achieves $(1+\epsilon)$-approximation ratio. Further let the approximate matrix of the right singular matrices be restored by $(\tilde{\boldsymbol{V}}_{q,1})_d = (\Sigma_{q,1})_d^{-1}(\boldsymbol{U}_{q,1})_d^{\top}\boldsymbol{M}_S$. Then, there exists a unitary matrix $\boldsymbol{W}$ such that Algorithm 3 is guaranteed to recover an $\tilde{\boldsymbol{M}}_S = (\boldsymbol{U}_{q,1})_d(\Sigma_{q,1})_d(\tilde{\boldsymbol{V}}_{q,1})_d \in \mathbb{R}^{|S|\times n}$ with approximation guarantee. In particular, let $\overline{\tilde{\boldsymbol{M}}_S} = \overline{(\boldsymbol{B}_{q,1})_d} = (\boldsymbol{M}_S)\boldsymbol{W} + \Psi$. Then, we have that:*

$$\|\Psi\|_F \leq \left((2+\epsilon)(1+\sqrt{2})^{q-1} - 1\right)\|\boldsymbol{M}_S - (\boldsymbol{M}_S)_d\|_F .$$

All proofs are deferred to Section 4. As we can observe from Theorem 3.2 that provides the final error bound of static Tree-SVD, the smaller the number of levels we have, the better the approximation guarantee we achieve. Thus, we set $q = 3$, $k = 8$, and $d = 128$ in our implementation to generate static subset node embedding. The parameters of our Tree-SVD in dynamic graphs will be the same.

Complexity analysis. With the first level as the randomized SVD, we further reduce the time complexity of our Tree-SVD compared to the original hierarchical SVD in [14]. Assume that $nnz(\boldsymbol{M})$ indicates the number of non-zero elements in a matrix $\boldsymbol{M}$. The following theorem shows the time complexity of Tree-SVD.

THEOREM 3.3. *Let $b = k^{q-1}$ be the number of sub-matrices in the first level of Tree SVD, where k is the branching factor of Tree-SVD and q is the level of Tree-SVD. Let d be the dimension of the embedding we output and let ϵ be the error parameter of the first level of randomized SVD. Then, Tree-SVD, as shown in Algorithm 3, takes $O(nnz(\boldsymbol{M}_S) + \frac{|S|\cdot d^2\cdot b}{\epsilon^4})$.*

For the hierarchical SVD proposed in [14], the time complexity is $O(|S|^2 \cdot n + |S|^3)$. Note that, $nnz(\boldsymbol{M}_S)$ is bounded by $|S| \cdot n$ and the term $\frac{d^2\cdot b}{\epsilon^4}$ can be regarded as a constant and is dominated by n as well. Thus, our Tree-SVD reduces the time complexity by a factor of $|S|$ compared to the hierarchical SVD as proposed in [14].

3.2 Tree-SVD on Dynamic Graphs

In this subsection, we introduce how to use our hierarchical structure to update subset embeddings from snapshot $t-i, i \in [1, t-1]$ to snapshot t. We inherit the notations from Section 3.1, except that we use t as the superscript to denote snapshot t, e.g., $\boldsymbol{M}_S^t$.

As we mentioned at the beginning of Section 3, the update with Tree-SVD becomes more efficient than global SVD methods since we maintain all the intermediate results (which are much smaller than the input proximity matrix) and only need to update the sub-matrices that are changed. However, in real-life scenarios, node embeddings are usually updated periodically after a few days or weeks. In such scenarios, when only a few edge events occur compared to the existing mature graph topology, the PPR proximity sub-matrices often change unevenly. We have also observed experimentally that with a large sub-matrix partition size b, the PPR entries often concentrate on some local sub-matrices. This motivates us to find out sub-matrices that have changed noticeably with graph evolution for eager updates. Specifically, we cache past representations in the hierarchical

```
Algorithm 4: Tree-SVD-Lazy-Update
   Input: Level $q$, branching factor $k$, dimension $d$, submatrices $(\boldsymbol{B}^{t-i}_{1,j})_d$, $\boldsymbol{B}^t_{1,j}$
 1 Initialize $Z \leftarrow \emptyset$
 2 Run Algorithm 2 to update the proximity matrix
 3 if $\left\|\left(\boldsymbol{B}^{t-i}_{1,j}\right)_d - \left(\boldsymbol{B}^{t-i}_{1,j}\right)\right\|_F + \|\boldsymbol{D}_j\|_F > \sqrt{2}\delta \left\|\boldsymbol{B}^t_{1,j}\right\|_F$ then
 4     $Z \leftarrow Z \cup \{j\}$
 5     Update the corresponding block from $\boldsymbol{B}^{t-i}_{1,j}$ to $\boldsymbol{B}^t_{1,j}$
 6 for $l = 1$ to $q-1$ do
 7     Initialize $Z_{parent} \leftarrow \emptyset$
 8     for $j \in Z$ do
 9         Compute truncated SVDs (resp. randomized SVD) of $(\boldsymbol{B}^t_{l,j})_d$ at the $l$-th level with
           $l > 1$ (resp. $l = 1$) to get $(\boldsymbol{U}^t_{l,j})_d$, $(\Sigma^t_{l,j})_d$. Let the parent of $\boldsymbol{B}_{l,j}$ be $\boldsymbol{B}_{l+1,x}$.
           $Z_{parent} \leftarrow Z_{parent} \cup \{x\}$.
10     for $j \in Z_{parent}$ do
11         $\boldsymbol{B}_{l+1,j} := \left[(\boldsymbol{U}^t_{l,(j-1)k+1})_d(\Sigma^t_{l,(j-1)k+1})_d, |\cdots|(\boldsymbol{U}^t_{l,jk})_d(\Sigma^t_{l,jk})_d,\right]$
12     $Z \leftarrow Z_{parent}$
13 Compute truncated SVD of $\boldsymbol{B}^t_{q,1}$ to get $(\boldsymbol{U}^t_{q,1})_d$, $(\Sigma^t_{q,1})_d$.
14 Set $\boldsymbol{X}^t = (\boldsymbol{U}^t_{q,1})_d\sqrt{(\Sigma^t_{q,1})_d}$
```

structure and put off the update of sub-matrices with negligible differences to a future snapshot. Such a strategy tends not to impact the quality of final embeddings.

Then the main task is to design a good measure for our lazy update scheme to monitor the changes of sub-matrices. Although it is heuristic, efficient, and effective to track the number of non-zeros or 1-norm of each sub-matrix to monitor the changes of sub-matrices, such measures could not provide any theoretical guarantee to bound the error of the embedding. To overcome such limitations, we investigate and model the correlation between the proximity matrix differences in consecutive snapshots and verify if the cached representation could be a good approximation in terms of the Frobenius norm, i.e., $\left\|\left(\boldsymbol{B}^{t-i}_{1,j}\right)_d - \left(\boldsymbol{B}^t_{1,j}\right)\right\|_F$. We further bound these terms by the original desired error $\left\|\boldsymbol{M}^t_{1,j} - (\boldsymbol{M}^t_{1,j})_d\right\|_F$. However, $\left\|\boldsymbol{M}^t_{1,j} - (\boldsymbol{M}^t_{1,j})_d\right\|_F$ could not be obtained without the SVD computation in the first level, which contradicts to our motivation that aims to efficiently find out sub-matrices that are changed noticeably. In our solution, we aim to avoid the unnecessary heavy SVD computation of sightly modified sub-matrices in the first level. Thus, we resort to find a measure correlated to $\left\|\boldsymbol{M}^t_{1,j}\right\|_F$ as a replacement, which is summarized in Lemma 3.4.

Lemma 3.4. *Let $j \in \{1, 2, \cdots, b\}$. Let $\boldsymbol{D}_j \in \mathbb{R}^{|S| \times n_j}$ be a matrix such that $\boldsymbol{B}^t_{1,j} = \boldsymbol{B}^{t-i}_{1,j} + \boldsymbol{D}_j$ holds for $\boldsymbol{B}^t_{1,j}$, $\boldsymbol{B}^{t-i}_{1,j}$, and $i \in [1, t-1]$. If $\boldsymbol{B}^t_{1,j}$, $\boldsymbol{B}^{t-1}_{1,j}$, and $\boldsymbol{D}_j$ satisfy that:*

$$\left\|\left(\boldsymbol{B}^{t-i}_{1,j}\right)_d - \left(\boldsymbol{B}^{t-i}_{1,j}\right)\right\|_F + \|\boldsymbol{D}_j\|_F \le \sqrt{2}\delta \left\|\boldsymbol{B}^t_{1,j}\right\|_F. \quad (2)$$

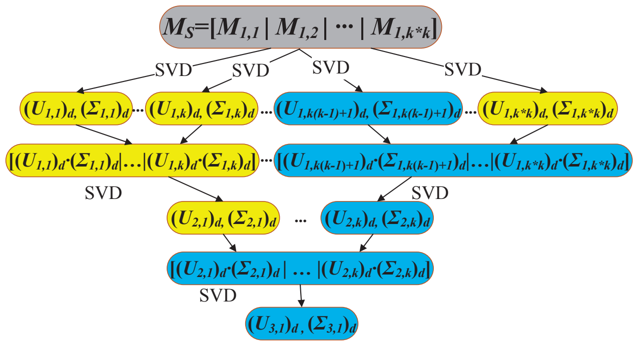

$M_S=[M_{1,1} \mid M_{1,2} \mid \cdots \mid M_{1,k*k}]$
SVD
SVD
SVD
SVD
$(U_{1,1})_d, (\Sigma_{1,1})_d$
$(U_{1,k})_d, (\Sigma_{1,k})_d$
$(U_{1,k(k-1)+1})_d, (\Sigma_{1,k(k-1)+1})_d$
$(U_{1,k*k})_d, (\Sigma_{1,k*k})_d$
$[(U_{1,1})_d \cdot (\Sigma_{1,1})_d \mid \ldots \mid (U_{1,k})_d \cdot (\Sigma_{1,k})_d]$
$[(U_{1,k(k-1)+1})_d \cdot (\Sigma_{1,k(k-1)+1})_d \mid \ldots \mid (U_{1,k*k})_d \cdot (\Sigma_{1,k*k})_d]$
SVD
SVD
$(U_{2,1})_d, (\Sigma_{2,1})_d$
$(U_{2,k})_d, (\Sigma_{2,k})_d$
$[(U_{2,1})_d \cdot (\Sigma_{2,1})_d \mid \ldots \mid (U_{2,k})_d \cdot (\Sigma_{2,k})_d]$
SVD
$(U_{3,1})_d, (\Sigma_{3,1})_d$

Fig. 2. Lazy update with only blue color blocks updated.

Then, there exists a unitary matrix $\boldsymbol{W}^{1,j} \in \mathbb{R}^{n_j \times n_j}$ *such that:*

$$\left\| \overline{\left(\boldsymbol{B}_{1,j}^{t-i}\right)_d} - \boldsymbol{B}_{1,j}^{t} \boldsymbol{W}_{1,j} \right\|_F \leq \sqrt{2}\delta \left\| \boldsymbol{B}_{1,j}^{t} \right\|_F .$$

Our lazy-update strategy is inspired by Lemma 3.4. For the first level sub-matrices, we only need to re-compute the truncated SVD results for the sub-matrices that have changed significantly, i.e., the sub-matrices that violate Eqn. 2. We call a sub-matrix as *affected* sub-matrices if it violates Eqn. 2 or it has at least one first-level matrix descendent that violates Eqn. 2. For example, in Figure 2, assume that a first level sub-matrix $\boldsymbol{M}_{1,k\cdot(k-1)+1}$ violates Eqn. 2. Then, $\boldsymbol{M}_{1,k\cdot(k-1)+1}$ is an affected sub-matrix. In addition, the second and third level sub-matrices that have $\boldsymbol{M}_{1,k\cdot(k-1)+1}$ as descendent are all affected sub-matrices (shown in blue). Then, with our lazy-update strategy, we only need to re-compute SVD results for the affected sub-matrices, significantly reducing computational costs, which mainly come from SVD computations for first-level sub-matrices.

Next, we elaborate on the details of our update algorithm.

Update algorithm. Algorithm 4 shows the pseudo-code of the update algorithm. Firstly, we update the proximity matrix $\boldsymbol{M}_S$ using the dynamic Forward-Push algorithm (Algorithm 2). Then, we can easily derive $\boldsymbol{D}_j$ for each sub-matrix $\boldsymbol{B}_{1,j}^t$, where $1 \leq j \leq b = k^{q-1}$. Besides, $(\boldsymbol{B}_{1,j}^{t-i})_d$ has been computed at timestamp $t-i$ as it needs to be fed to the next level, where $t-i$ is the timestamp of the last time when the SVD of $\boldsymbol{B}_{1,j}$ is computed. Thus, at timestamp t, $(\boldsymbol{B}_{1,j}^{t-i})_d$ is already available, which means from timestamp $t-i$ to $t-1$, it satisfies Eqn. 2 and thus the SVD result computed at timestamp $t-i$ is reused after the updates from $t-i$ to $t-1$. After the update at timestamp t, we dynamically track if any sub-matrix in the first level violates the condition in Eqn. 2. If this is the case, we add the index of such sub-matrix to Z and update $\boldsymbol{B}_{1,j}^{t-i}$ to $\boldsymbol{B}_{1,j}^{t}$ (Lines 3-5). Next, we update the affected sub-matrices with indices in Z in a bottom-up fashion (Lines 6-12). In particular, we retrieve the index of the affected sub-matrices at the l-th level (initially $l=1$). Here, we assume that the bottom level is 1 and the root level is q. Then, we re-compute the truncated SVD (resp. randomized SVD) for level l with $l>1$ (resp. $l=1$). Given the updated sub-matrix $\boldsymbol{B}_{l,j}$, it further retrieves the index of its parent at level $l+1$. Assume that the index is x. Then, the index x of its parent is added to set Z_{parent} (Line 9). Next, the sub-matrix $\boldsymbol{B}_{l+1,x}$ will also need to be updated and hence is affected. Thus, we update the sub-matrix for level $l+1$ based on the updated SVD results of the affected children at level l (Lines 10-11). After re-computing the SVD at level l and updating the affected matrices at level $l+1$, we proceed to the next level $l+1$ by setting $Z = Z_{parent}$ (Line 12). The updates proceed until we reach level q where we compute the truncated SVD of the only matrix $\boldsymbol{B}_{q,1}$ (Line 13) and return the updated embedding (Line 14).

Example 3.5. Assume that $q = 3$, $k = 8$, and $\boldsymbol{B}_{1,57}$ is the only sub-matrix at the first level that violates Eqn. 2. Then its parent, $\boldsymbol{B}_{2,8}$, is affected and formed by the updated $(\boldsymbol{U}_{1,57})_d(\Sigma_{1,57})_d$ and cached $(\boldsymbol{U}_{1,58})_d(\Sigma_{1,58})_d$ to $(\boldsymbol{U}_{1,64})_d(\Sigma_{1,64})_d$. Finally, the grandparent of $\boldsymbol{B}_{1,57}$, $\boldsymbol{B}_{3,1}$, is also affected and should be updated to get the final embedding. For other unaffected sub-matrices, no update is required and the cached intermediate results can be reused. □

Notice that the running cost at level l ($l > 1$) is much smaller than that at the first level. In addition, most of the sub-matrices are not updated, saving a lot of computational costs compared to a re-computation of SVD from scratch. The following theorem shows that the lazy-update strategy still provides approximation guarantees for the final SVD result.

Theorem 3.6. *Let $\boldsymbol{M}_S^t \in \mathbb{R}^{|S|\times n}$ be the input matrix and $q \geq 2$. Algorithm 4 is guaranteed to recover an $(\tilde{\boldsymbol{M}}_S)_{q,1}^t \in \mathbb{R}^{|S|\times n}$ such that $\overline{\left((\tilde{\boldsymbol{M}}_S)_{q,1}^t\right)_d} = (\boldsymbol{M}_S)^t\boldsymbol{W} + \Psi$, where $\boldsymbol{W}$ is a unitary matrix, and*

$$\|\Psi\|_F \leq \left((1+\delta\sqrt{2})(1+\sqrt{2})^{q-1} - 1\right)\left\|\boldsymbol{M}_S^t\right\|_F.$$

There are three cases of updates to discuss: *(i)* when all sub-matrices, or dubbed as blocks, at the first level are error-bounded as Lemma 3.4 and thus no block is updated in the current snapshot t, we get error bound as Theorem 3.6; *(ii)* when blocks are updated partially in the current snapshot t, we have the same error bound but with $\delta \leq \frac{1+\epsilon}{\sqrt{2}}$ as the updating threshold, and set $\delta = \frac{1+\epsilon}{\sqrt{2}}$ for the worst case theoretical guarantee to bound both updated blocks and cached blocks; *(iii)* when all blocks are updated in the current snapshot t, we have a theoretical bound the same as static Tree-SVD reconstruction, which is shown in Theorem 3.2.

With the lazy update strategy, for every snapshot t, we just need to update a small portion of blocks, which overcomes the existing efficiency bottleneck of the MF-based approach, at the same time achieving accuracy on downstream tasks comparable to the static counterparts. Theorem 3.7 shows the update cost of Algorithm 4.

Theorem 3.7. *Given a batch of τ edge updates, assume that b' sub-matrices at the first level violate Eqn. 2 and the number of non-zero entries in b' sub-matrices is nnz', Algorithm 4 has a time complexity of $O\left(\min\{\tau + 1/r_{max}, |S|/r_{max}\} + nnz' + \frac{|S|\cdot d^2\cdot k^2\cdot b'}{\epsilon^4}\right)$.*

Besides, it is easy to derive that the cached intermediate matrices in our lazy update have a space cost of $O(|S|\cdot d\cdot b)$, where d is the dimension of the embedding and b is the number of first-level sub-matrices. Since $d\cdot b$ is far smaller than n, this space cost is negligible to the input proximity matrix with a space of $O(|S|\cdot n)$.

4 THEORETICAL ANALYSIS

Due to limited space, we only include the proof of Theorem 3.2. Other omitted proofs can be found in our technical report [1].

Proof of Theorem 3.2. Let $\boldsymbol{M}_S$ denote the original matrix with block components, i.e., $\boldsymbol{M}_S = [\boldsymbol{M}_{1,1}|\boldsymbol{M}_{1,2}|\cdots|\boldsymbol{M}_{1,b}]$, where $b = k^{q-1}$, and $\boldsymbol{M}_{l,j}$ denote the error-free block of the original matrix $\boldsymbol{M}_S$ whose entries correspond to the entries included in $\boldsymbol{B}_{l,j}^t$. Thus, $\boldsymbol{M}_S = \left[\boldsymbol{M}_{l,1}\left|\boldsymbol{M}_{l,2}\right|\cdots\mid \boldsymbol{M}_{l,b/k^{(l-1)}}\right]$ holds for all $l \in [1,q]$, where

$$\boldsymbol{M}_{l+1,j} := \left[\boldsymbol{M}_{l,(j-1)k+1}|\cdots|\boldsymbol{M}_{l,jk}\right], \tag{3}$$

for all $l \in [1, q-1]$, and $j \in [1, b/k^l]$. Our goal is to bound the final matrix $\overline{(\boldsymbol{B}_{q,1})_d}$ with respect to the original matrix $\boldsymbol{M}_S$. Next, we will prove this by induction on level l. We will prove that:

(1) $\overline{\left(\boldsymbol{B}_{l,j}\right)_d} = \boldsymbol{M}_{l,j}\boldsymbol{W}_{l,j} + \boldsymbol{\Psi}_{l,j}$;
(2) $\boldsymbol{W}_{l,j}$ is always a unitary matrix;
(3) $\|\boldsymbol{\Psi}_{l,j}\|_{\mathrm{F}} \leq \left((2+\epsilon)(1+\sqrt{2})^{l-1}-1\right)||(\boldsymbol{M}_{l,j})_d - \boldsymbol{M}_{l,j}||_F$.
Here for any classic Randomized-SVD, $\forall \boldsymbol{M}_{1,j}$, we have

$$||\boldsymbol{U}'\Sigma'\boldsymbol{V}' - \boldsymbol{M}_{1,j}||_F \leq (1+\epsilon)||(\boldsymbol{M}_{1,j})_d - \boldsymbol{M}_{1,j}||_F.$$

Then we apply any classic Randomized-SVD to the first level sub-matrices, we could get the following results:

$$||(\boldsymbol{B}_{1,j})_d - \boldsymbol{M}_{1,j}||_F \leq (1+\epsilon)||(\boldsymbol{M}_{1,j})_d - \boldsymbol{M}_{1,j}||_F.$$

For $l = 1$, There exists a unitary matrix $\boldsymbol{W}_{1,j}$, $j \in [1, b]$, such that:

$$\overline{\left(\boldsymbol{B}_{1,j}\right)_d} = \boldsymbol{M}_{1,j}\boldsymbol{W}_{1,j} + \left[\left(\boldsymbol{B}_{1,j}\right)_d - \boldsymbol{M}_{1,j}\right]\boldsymbol{W}_{1,j},$$

where $\boldsymbol{\Psi}_{1,j} := \left[\left(\boldsymbol{B}_{1,j}\right)_d - \boldsymbol{M}_{1,j}\right]\boldsymbol{W}_{1,j}$. Then we have

$$\begin{aligned}\left\|\boldsymbol{\Psi}_{1,j}\right\|_{\mathrm{F}} =& \left\|\overline{\left(\boldsymbol{B}_{1,j}\right)_d} - \boldsymbol{M}_{1,j}\boldsymbol{W}_{1,j}\right\|_{\mathrm{F}} = \left\|\left(\boldsymbol{B}_{1,j}\right)_d - \boldsymbol{M}_{1,j}\right\|_{\mathrm{F}} \\ \leq& (1+\epsilon)||(\boldsymbol{M}_{1,j})_d - \boldsymbol{M}_{1,j}||_F = \left((2+\epsilon)(1+\sqrt{2})^{1-1}-1\right)\left\|\boldsymbol{M}_{l+1,j} - \left(\boldsymbol{M}_{l+1,j}\right)_d\right\|_{\mathrm{F}}.\end{aligned}$$

Now, Conditions 1-3 hold for $l = 1$. Suppose Conditions 1-3 hold for $l \in [1, q-1]$. Then, from Condition 1, we have:

$$\begin{aligned}\boldsymbol{B}_{l+1,j} :=& \left[\overline{\left(\boldsymbol{B}_{l,(j-1)k+1}\right)_d}|\cdots|\overline{\left(\boldsymbol{B}_{l,jk}\right)_d}\right] \\ =& \left[\boldsymbol{M}_{l,(j-1)k+1}\boldsymbol{W}_{l,(j-1)k+1} + \boldsymbol{\Psi}_{l,(j-1)k+1}|\cdots|\boldsymbol{M}_{l,jk}\boldsymbol{W}_{l,jk} + \boldsymbol{\Psi}_{l,jk}\right] \\ =& \left[\boldsymbol{M}_{l,(j-1)k+1}\boldsymbol{W}_{l,(j-1)k+1}|\cdots|\boldsymbol{M}_{l,jk}\boldsymbol{W}_{l,jk}\right] + \left[\boldsymbol{\Psi}_{l,(j-1)k+1}|\cdots|\boldsymbol{\Psi}_{l,jk}\right] \\ =& \left[\boldsymbol{M}_{l,(j-1)k+1}|\cdots|\boldsymbol{M}_{l,jk}\right]\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{\Psi}},\end{aligned}$$

where $\tilde{\boldsymbol{W}} := diag(\boldsymbol{W}_{l,(j-1)k+1}, \boldsymbol{W}_{l,(j-1)k+2}, ..., \boldsymbol{W}_{l,jk})$, and $\tilde{\boldsymbol{\Psi}} := \left[\boldsymbol{\Psi}_{l,(j-1)k+1}|\cdots|\boldsymbol{\Psi}_{l,jk}\right]$. Note that $\tilde{\boldsymbol{W}}$ is unitary since its diagonal blocks are all unitary by Condition 2. Therefore, we have $\boldsymbol{B}_{l+1,j} = \boldsymbol{M}_{l+1,j}\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{\Psi}}$. Then, we can derive the following bound:

$$\begin{aligned}\left\|\left(\boldsymbol{B}_{l+1,j}\right)_d - \boldsymbol{M}_{l+1,j}\tilde{\boldsymbol{W}}\right\|_F \leq& \left\|\left(\boldsymbol{B}_{l+1,j}\right)_d - \boldsymbol{B}_{l+1,j}\right\|_F + \left\|\boldsymbol{B}_{l+1,j} - \boldsymbol{M}_{l+1,j}\tilde{\boldsymbol{W}}\right\|_F \\ =& \sqrt{\sum_{j=d+1}^{D}\sigma_j^2\left(\boldsymbol{M}_{l+1,j}\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{\Psi}}\right)} + \|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}} \\ \leq& \sqrt{\sum_{j=d+1}^{D}2\sigma_j^2\left(\boldsymbol{M}_{l+1,j}\tilde{\boldsymbol{W}}\right)} + \sqrt{\sum_{j=1}^{D}2\sigma_j^2(\tilde{\boldsymbol{\Psi}})} + \|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}} \\ =& \sqrt{2}\left\|\boldsymbol{M}_{l+1,j} - \left(\boldsymbol{M}_{l+1,j}\right)_d\right\|_{\mathrm{F}} + (1+\sqrt{2})\|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}}.\end{aligned} \tag{4}$$

In addition, by Condition 3, we know that:

$$
\begin{aligned}
\|\tilde{\Psi}\|_F &= \sum_{i=1}^{k} \left\|\Psi_{l,(j-1)k+i}\right\|_F \\
&\leq \sum_{i=1}^{k} \left((2+\epsilon)(1+\sqrt{2})^{l-1}-1\right) \left\|M_{l,(j-1)k+i} - \left(M_{l,(j-1)k+i}\right)_d\right\|_F \\
&\leq \sum_{i=1}^{k} \left((2+\epsilon)(1+\sqrt{2})^{l-1}-1\right) \left\|M_{l,(j-1)k+i} - \left(M_{l+1,j}\right)_d^i\right\|_F \\
&= \left((2+\epsilon)(1+\sqrt{2})^{l-1}-1\right) \left\|M_{l+1,j} - \left(M_{l+1,j}\right)_d\right\|_F,
\end{aligned}
\tag{5}
$$

where $\left(M_{l+1,j}\right)_d^i$ denotes the block of $\left(M_{l+1,j}\right)_d$ corresponding to $M_{l,(j-1)k+i}$. Combining Equations 4 and 5, we have:

$$\left\|\left(B_{l+1,j}\right)_d - M_{l+1,j}\tilde{W}\right\|_F \leq \left((2+\epsilon)(1+\sqrt{2})^{l}-1\right) \left\|M_{l+1,j} - \left(M_{l+1,j}\right)_d\right\|_F.$$

Moreover, note that:

$$
\begin{aligned}
\left\|\left(B_{l+1,j}\right)_d - M_{l+1,j}\tilde{W}\right\|_F &= \left\|\left(B_{l+1,j}\right)_d \hat{W}_{l+1}^j - M_{l+1,j}\tilde{W}\hat{W}_{l+1}^j\right\|_F \\
&= \left\|\overline{\left(B_{l+1,j}\right)_d} - M_{l+1,j}W_{l+1,j}\right\|_F,
\end{aligned}
$$

where $\hat{W}_{l+1}^j$ is the unitary matrix that represents the transformation, and $W_{l+1,j} = \tilde{W}\hat{W}_{l+1}^j$ is the unitary matrix in Condition 1. Hence, Conditions 1-3 hold for $l+1$ with $\Psi_{l+1,j} := \overline{\left(B_{l+1,j}\right)_d} - M_{l+1,j}W_{l+1,j}$. Since on the last level, we derive $B_{q,1}$ and $M_{q,1} = M_S$ according to the definition. Applying these three conditions, we get the claimed result in Theorem 3.2, which finishes the proof.

5 OTHER RELATED WORK

Static node embedding. There are numerous works dealing with static node embedding. Traditional random walk based [7, 27] approaches are inspired by skip-gram model [22]. They focus on preserving the co-occurrence probability of the nodes on the random walks. MF-baseds methods [25, 29] are also successful attempts for static node embedding. They first define a proximity matrix and then apply matrix factorization algorithms to get the node embeddings. In general, the state-of-the-art MF-based methods [39, 42] factorize a PPR matrix. Recent progress on deep learning provides alternative solutions [9, 16] for static node embedding. They compute node embeddings by training graph neural networks (GNNs). However, GNNs train models in a supervised manner and they all need features as input, which is explained thoroughly in [3, 8]. Different from GNNs, we focus on designing unsupervised embedding methods on topology-only graphs.

Dynamic node embedding. CTDNE [23], DNE [5], and dynnode2vec [20] are random walk-based methods for dynamic node embedding task. The main idea of these solutions is to update the random walks according to the graph changes. Such solutions are generally outperformed by hashing-based methods as shown in [32]. Another line of research work focuses on designing dynamic algorithms for MF-based methods. LIST [40] incorporates temporal information into the learned embeddings and predicts the edges for the next snapshot by solving a least squares optimization problem. However, the recalculation of the matrix decomposition at each snapshot is computationally prohibitive. To tackle this issue, TIMERS [44] proposes an incremental eigenvalue decomposition-based method that sets a tolerated error threshold for the restart time to reduce the

Table 3. Statistics of datasets.

Dataset	Type	n	m	$\|C\|$	τ
Patent (PT)	Citation	2.7M	14.0M	6	25
Mag-authors (MA)	Co-authorship	5.8M	27.7M	19	9
Wikipedia (WK)	Web-link	6.2M	178M	10	20
YouTube (YT)	Socialnet	3.2M	9.4M	-	8
Flickr (FK)	Socialnet	2.3M	33.1M	-	6

error accumulation. However, TIMERS generates embeddings by eigenvalue decomposition on the whole square proximity matrix, which is not suitable for subset node embeddings.

6 EXPERIMENT

In this section, We experimentally evaluate our Tree-SVD against competitors on link prediction and node classification tasks. All experiments are conducted on a Linux machine with 2 CPUs (2.30GHz), 32 cores (64 threads), and 416 GB memory. Our source code is publicly available at [1].

6.1 Experimental Settings

Dynamic graph datasets. Following DynPPE [8], we use three large dynamic graph datasets, Patent, Mag-Authors, and Wikipedia for node classification tasks. We also test on link prediction tasks, which are broadly applied to social network analysis. Thus, we select two large social network datasets, YouTube and Flickr, together with Mag-authors, as three link prediction datasets. The statistics of these datasets are shown in Table 3, where $|C|$ denotes the number of classes and τ indicates the number of snapshots.

Competitors. The main competitors are listed as follows:

- DynPPE [8], the state-of-the-art subset node embedding method;
- STRAP [39], dubbed as Global-STRAP, one of the state-of-the-art static node embedding methods;
- Subset-STRAP, the subset version of STRAP (Ref. to Section. 2.2).
- RandNE [43] and FREDE [32], two efficient embedding methods.

All competitors are in parallel using 64 threads. We obtain their implementations from Github and use default settings suggested by their authors. For our methods, we use Tree-SVD to indicate the algorithm on dynamic graphs and Tree-SVD-S (the solution in Section 3.1) to indicate the static version of Tree-SVD.

Parameter settings. Following DynPPE [8], we set $|S| = 3000$ for all datasets. We randomly sample 3000 nodes from the graph topology at the first snapshot to form the subset S, which is the same as DynPPE [8]. In addition, for our Tree-SVD, we set the number of sub-matrices $b = 64$ and the level $q = 3$. For our lazy update strategy, we set $\delta = 0.65$ (Ref. to Section 3.2) as it achieves a good trade-off between the running time and accuracy. For Global-STRAP, Subset-STRAP, Tree-SVD, we tune r_{max} (Ref. to Section 2.1) so that their performance does not further improve in affordable time. The r_{max} for Tree-SVD on Wikipedia, Flickr, Mag-author, Patent and Youtube are 10^{-5}, 10^{-7}, 10^{-8}, 10^{-8}, and 10^{-8}, respectively.

Task settings. For the classification task, we follow the same setting as DynPPE [8] to conduct single-label classification. For the link prediction task, given the subset S, we predict the links from S to V. Let E_S be the set of outgoing edges of $s \in S$. We first randomly sample 70% of all edges from each snapshot as training edges. Then for the rest 30% of edges in each snapshot, we select relevant edges which belong to E_S, add these edges as positive pairs to the test set and discard the remaining irrelevant edges. Next, we randomly generate the same number of negative pairs

Table 4. Precision score on LP task. (Exp. 1)

Method	YouTube	Flickr	Mag-authors
Global-STRAP	79.97	89.97	87.80
Subset-STRAP	82.34	91.35	89.34
FREDE	47.59	49.45	45.26
RandNE	64.08	86.62	72.42
Tree-SVD-S	**82.40**	**92.68**	**90.42**

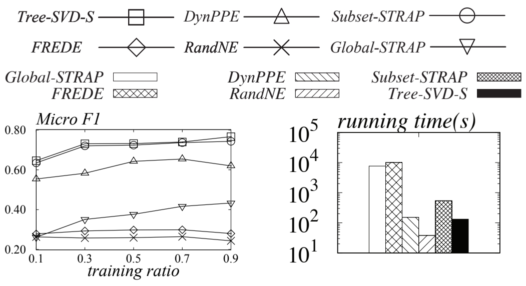
as positive pairs. In particular, we randomly generate node pairs from arbitrary node $s \in S$ to an arbitrary node $v \in V$ and assure that such node pairs are not edges. Such negative edges are also added to the test set. Finally, We remove all positive edges from the graph and generate subset embeddings on the graph with the remaining edges.

6.2 Static Subset Embedding

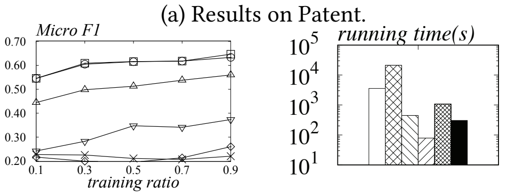
Exp1: Global vs. subset embedding methods. We first evaluate subset embeddings generated by global and subset methods on static graphs to show the importance of subset embedding methods. The last snapshot of each graph is used to generate subset embeddings.

We test all solutions on node classification tasks using three datasets that contain node labels, i.e., Patent, Mag-authors, and Wikipedia. Figure 3 reports the Micro-F1 scores and the embedding time of each method. As we can observe, compared to the global embedding method Global-STRAP, Subset-STRAP achieves much better Micro-F1 scores in all settings, which demonstrates the potential to adopt subset embedding methods for better effectiveness. Meanwhile, our Tree-SVD-S consistently achieves the best results on all datasets while taking comparable running time to RandNE. Notice that DynPPE adopts a smaller r_{max} to get a more accurate proximity matrix, resulting in a high running cost. On the other hand, Tree-SVD-S and Subset-STRAP all achieve better performances than DynPPE, which demonstrates the effectiveness of MF-based methods. Finally, compared with Subset-STRAP, Tree-SVD-S achieves similar performance on Patent and Mag-authors datasets with much less time cost, while taking a lead by at least 5% on the Wikipedia dataset. This demonstrates that our Tree-SVD-S achieves a better trade-off between the efficiency and effectiveness on static subset embeddings.

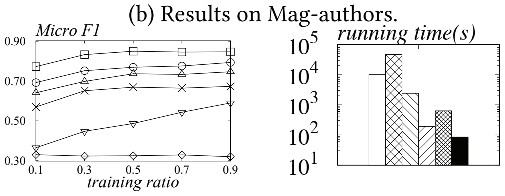
We then evaluate each method on the link prediction task. As explained earlier, link prediction is meaningful on social networks. Therefore, we conduct experiments on two social networks YouTube and Flickr, together with a co-authorship graph Mag-authors. We select the same competitors as the node classification task excluding DynPPE due to efficiency. To explain, for the link prediction task, we need both embeddings of the start node $s \in S$ (left embedding matrix) and the target node $t \in V$ (right embedding matrix). For MF-based methods, the right embedding matrix is generated naturally with the left one without additional time costs. Although DynPPE generates embedding for S with a similar time as MF-based methods, compared to the left embedding matrix, it needs $n/|S|$ times more time to generate the right embedding matrix. Therefore, DynPPE does not work in subset link prediction and thus is omitted. Table 4 reports the precision score and Figure 4 reports the embedding time for each method. As we can observe, compared with global embedding methods, both Subset-STRAP and Tree-SVD-S achieve better results on all datasets, which again demonstrates the potential to design subset embedding for better effectiveness. Furthermore, our Tree-SVD-S achieves similar performance on Youtube as Subset-STRAP, while taking the lead by more than 1% on Flickr and Mag-authors. At the same time, compared with other effective methods, Tree-SVD-S takes much less running time to generate embeddings from scratch on each dataset.



(a) Results on Patent.



(b) Results on Mag-authors.



(c) Results on Wikipedia.

Fig. 3. Results on NC task. (Exp. 1)

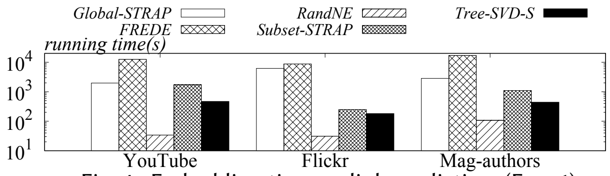


Fig. 4. Embedding time on link prediction. (Exp. 1)

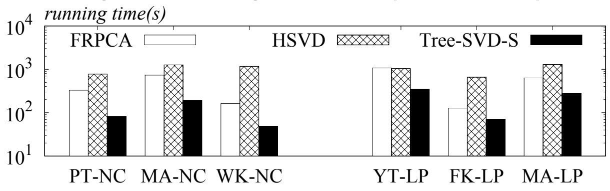


Fig. 5. Embedding time of SVD methods. (Exp. 2)

Table 5. Micro-F1(%) of 50% training ratio. (Exp. 2)

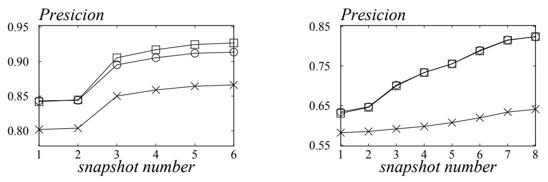
Method	Patent	Mag-authors	Wikipedia
FRPCA	72.19	61.60	82.93
HSVD	73.40	61.47	84.40
Tree-SVD-S	73.20	61.60	85.00

This again demonstrates that our Tree-SVD-S gains a better trade-off between the running time and embedding quality on static subset embedding.

Exp2: SVD comparison. In the second set of experiments, we will compare our Tree-SVD-S against two SVD alternatives, FRPCA and hierarchical SVD (HSVD), to evaluate different SVD

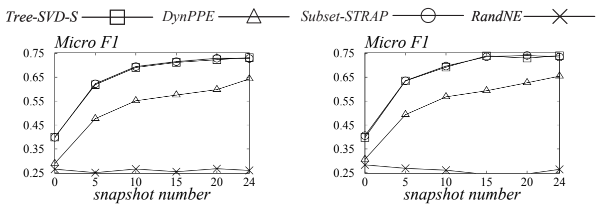
Table 6. Precision on link prediction (LP). (Exp.2)

Method	YouTube	Flickr	Mag-authors
FRPCA	82.11	92.54	90.01
HSVD	82.27	92.67	90.29
Tree-SVD	82.40	92.68	90.42



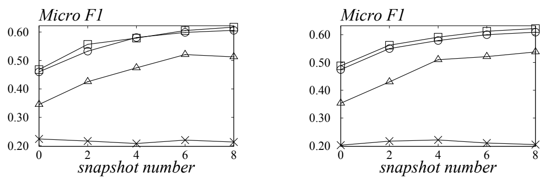
(a) Flickr (b) YouTube

Fig. 6. Link prediction on different snapshots. (Exp. 3)



(a) 50% nodes for training (b) 70% nodes for training

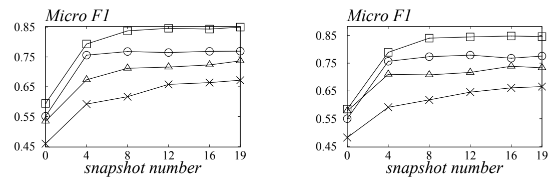
Fig. 7. Node classification on Patent. (Exp. 3)



(a) 50% nodes for training (b) 70% nodes for training

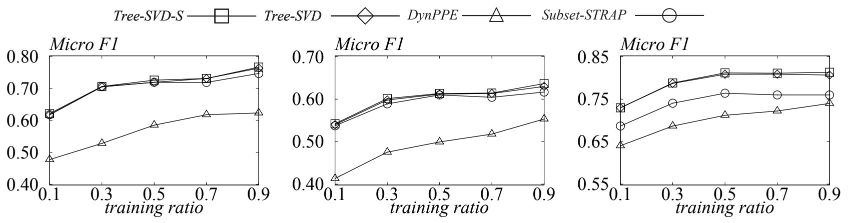
Fig. 8. Node classification on Mag-authors. (Exp. 3)

frameworks. To make a fair comparison, the proximity matrices of FRPCA and HSVD are the same as ours, i.e., computing the PPR on the reverse graph and taking the log operation as shown in Section 3.1. For HSVD, we set the number of sub-matrices to be the same as Tree-SVD-S, i.e., $b = 64$. We further evaluate HSVD and Tree-SVD-S with varying b in Section 6.4. In Exp. 2, we generate subset embeddings on the last snapshot of each graph. Figure 5 reports the embedding time of each SVD method on both node classification (NC) and link prediction (LP) tasks. Table 5 and 6 show the experimental results of subset embeddings generated by different SVD methods. As we can observe, HSVD and Tree-SVD achieve similar results on both node classification tasks and link prediction tasks. Meanwhile, our Tree-SVD-S is up to an order of magnitude faster than HSVD and up to 3.9x faster than FRPCA. If we examine the Micro-F1 scores in Table 5 and precision scores in Table 6, we can find that our Tree-SVD consistently achieves better results than FRPCA, and even take the lead by 2% on Wikipedia. This demonstrates that compared with other SVD frameworks, our Tree-SVD enables us to significantly speed up the SVD computation without sacrificing the embedding effectiveness on downstream tasks.



(a) 50% nodes for training (b) 70% nodes for training

Fig. 9. Node classification on Wikipedia. (Exp. 3)



(a) Patent. (b) Mag-authors. (c) Wikipedia.

Fig. 10. Classification results after 10^6 edge events. (Exp. 4)

Table 7. Precision on LP after 10^6 edge events. (Exp.4)

Method	YouTube	Flickr	Mag-authors
Subset-STRAP	82.33	90.21	85.62
Tree-SVD	81.88	90.78	86.13
Tree-SVD-S	82.31	91.25	86.29

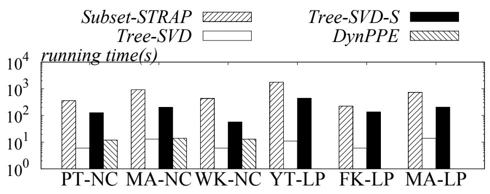


Fig. 11. Average update time. (Exp. 4)

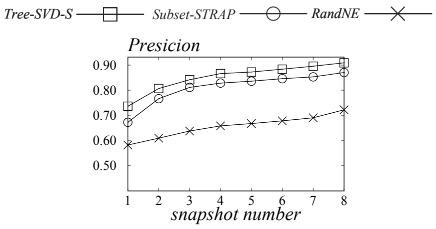
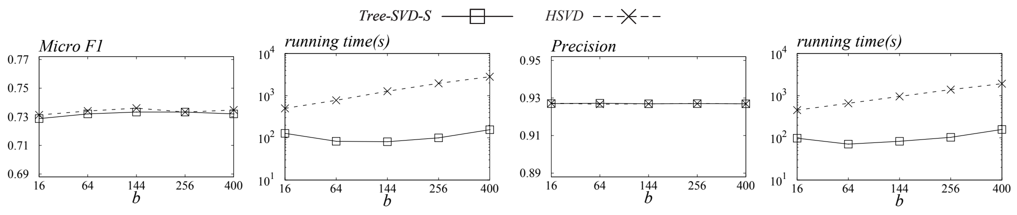


Fig. 12. LP results on Twitter. (Exp. 5)

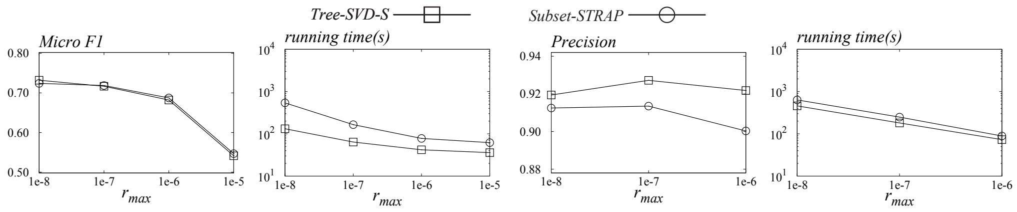
6.3 Dynamic Subset Embedding

Exp. 3: Impact of dynamic updates. Next, we generate node embeddings for each snapshot to show that dynamically updating the embedding along with the time significantly affects the embedding quality. Since the numbers of snapshots are very small in all five datasets, there exist a huge number of edge events in two consecutive snapshots. For every two consecutive snapshots, all methods actually re-compute the subset embedding from scratch. We first examine how the micro-F1 score of the node classification task changes along with snapshots on each dataset. We omit the results of Global-STRAP as it shows inferior performance on subset embedding in Section 6.2. Since we re-construct the embeddings, Tree-SVD is the same as Tree-SVD-S. Figures 7-9 show the micro-F1 score of our Tree-SVD, three competitors RandNE, DynPPE and Subset-STRAP, with 50% and 70% training ratios on three datasets. As we can observe, with the change of the graph, almost all subset embedding methods gain better Micro-F1 scores with the update of the model along with the snapshots in most scenarios. This demonstrates the importance of updating the subset embeddings when the graph changes. Moreover, our Tree-SVD consistently achieves the best performance in all settings.



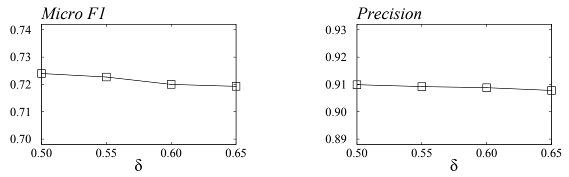
(a) Node classification on Patent. (b) Link prediction on Flickr.

Fig. 13. Experimental results of Tree-SVD-S and HSVD with varying b



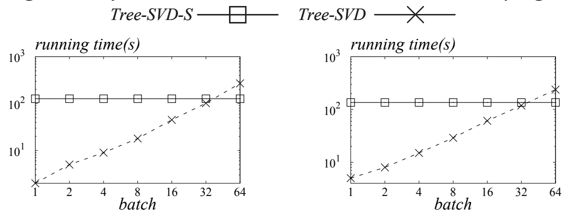
(a) Node classification on Patent. (b) Link prediction on Flickr.

Fig. 14. Experimental results of Tree-SVD-S and Subset-STRAP with varying r_{max}



(a) Node classification on Patent. (b) Link prediction on Flickr.

Fig. 15. Experimental results of Tree-SVD with varying δ.



(a) Node classification on Patent. (b) Link prediction on Flickr.

Fig. 16. Impact of update size to Tree-SVD.

We further examine how the precision changes along with snapshots for the link prediction task on each dataset. Figures 6 show the precision score of our Tree-SVD-S, the competitors RandNE and Subset-STRAP on Flickr and YouTube. The results on Mag-authors are similar and thus are omitted due to the interest of space and can be found in the attached full version technical report. As we can observe, all methods are able to improve the results by updating the model on the next snapshot. This again reflects the importance to update the subset embeddings when the graph gets changed. Moreover, both Tree-SVD and Subset-STRAP achieve more significantly improved precision scores. After several embedding updates, our Tree-SVD achieves the best performance.

Exp. 4: Batch updates. In real-world applications, the models are usually updated daily or weekly depending on the efficiency of their update algorithms. We simulate such a scenario by updating the embeddings after every batch update, i.e., 10,000 edge events for all methods. In this set of experiments, we start from a middle snapshot and then proceed 1,000,000 edge events. Thus, we trigger 100 batch updates. We invoke our dynamic Tree-SVD after every 10,000 edge events to

Table 8. Precision (%) on link prediction after 10^6 edge events. (Exp. 5)

Method	Twitter	Average Time (seconds)
Subset-STRAP	75.71	1631
Tree-SVD	79.44	54
Tree-SVD-S	79.67	586

update subset embeddings. For Subset-STRAP and Tree-SVD-S, we simply re-run the algorithm on the updated proximity matrix after every 10,000 edge events.

Figure 10 shows the Micro-F1 scores after 100 batch updates and Figure 11 shows the average update time. As we can see, our Tree-SVD is up to an order of magnitude faster than Tree-SVD-S and up to 71x faster than Subset-STRAP on node classification tasks. DynPPE is as efficient as our Tree-SVD to handle these batch updates. Moreover, we can find that our Tree-SVD is consistently achieving almost identical results as Tree-SVD-S, and leads DynPPE by up to 17%. This demonstrates that compared with other competitors, our Tree-SVD gains a better trade-off between the update efficiency and embedding effectiveness. As for link prediction, Table 7 further reports the precisions after these updates. As we can observe, Our Tree-SVD is still an order of magnitude faster than Tree-SVD-S and up to 160x faster than Subset-STRAP. Meanwhile, our Tree-SVD provides similar results as Subset-STRAP and Tree-SVD-S, which again shows that our Tree-SVD achieves a better trade-off between update efficiency and embedding effectiveness.

Exp. 5: Scalability evaluation. To further verify the scalability of our algorithm, we test our solutions on a public graph Twitter [17], with 41.6 million nodes and 1.5 billion edges, for the link prediction task. We randomly split the graph into 8 snapshots where each snapshot includes the same number of edges. We note that such a random split might not be as meaningful as previous dynamic graphs with real timestamp information as it does not capture the real evolving process. In this setting, our experimental results still share the same conclusions as that of Exp. 3 and Exp. 4.

In this set of experiments, we follow the setting in Exp. 3 that reconstructs the embedding at each snapshot. Note that FREDE and Global-Strap cannot generate embeddings within 24 hours or run out of memory and thus are omitted. Figure 12 shows the precision score along with snapshots on Twitter. All methods are able to improve the results by updating the model on every next snapshot. This reflects the importance to update the subset embeddings when the graph gets changed. Besides, our static Tree-SVD-S consistently achieves the best performance on all snapshots.

We then conduct experiments following Exp. 4 to examine the scalability of our dynamic Tree-SVD. Table 8 reports the link prediction results on Twitter after 1,000,000 edge updates. The average running time of dynamic Tree-SVD is still an order of magnitude faster than static Tree-SVD-S and 30x faster than Subset-STRAP. Meanwhile, dynamic Tree-SVD achieves comparable results to Static Tree-SVD-S. These results show that our dynamic Tree-SVD is efficient and effective, and scales to billion-edge scale graphs.

6.4 Parameter Analysis

In this subsection, we analyze the impact of different parameters on Patent and Flickr datasets. The results on other datasets can be found in the attached full version technical report [1].

Parameter b. We compare our Tree-SVD-S against Hierarchical SVD (HSVD) using a different number of b sub-matrices at the first level during the SVD computation (Ref. to Section 3.1). We use the last snapshot to generate embeddings. Figure 13 reports experimental results and running time of HSVD and Tree-SVD-S. As we can observe, our Tree-SVD-S achieves comparable results to HSVD while speeding up the embedding process by up to an order of magnitude. Meanwhile, since parameter b controls the number of sub-matrices in the SVD computation, as b increases, the SVD architecture becomes more complex. Thus, the cost of HSVD increases significantly. However, our Tree-SVD-S is not sensitive to b, demonstrating the superiority of our novel SVD architecture.

Threshold r_{max}. We use the last snapshot of each graph to generate embeddings. Figure 14 reports experimental results and running time of Subset-STRAP and Tree-SVD-S with varying r_{max}. As we can observe, our Tree-SVD-S achieves comparable results to Subset-STRAP on both tasks. On the other hand, our Tree-SVD-S is consistently faster than Subset-STRAP on all datasets. Meanwhile, as parameter r_{max} controls the accuracy of PPR estimations (Ref. to Section 2.1), when r_{max} increases, the proximity matrix becomes sparser, speeding up the embedding process. However, the performances of both Tree-SVD-S and Subset-STRAP degrade in most scenarios. This shows that the quality of the proximity matrix has a significant impact on the embedding quality. Compared with the competitor, our Tree-SVD-S gains a better trade-off between embedding effectiveness and computation efficiency.

Parameter δ. In this set of experiments, we generate dynamic embeddings for each graph. Figure 15 reports experimental results of our Tree-SVD on different datasets with varying δ. As we can observe, since δ controls the error bound in our lazy-update strategy (Ref. to Section 3.2), smaller δ leads to slightly improved results on all datasets.

Impact of update size. We vary the number of edge updates to examine the cut-off points where dynamic Tree-SVD is more efficient than the static Tree-SVD-S. Figure 16 shows the running time of Tree-SVD-S and Tree-SVD with varying update size, where there exist 10^4 edge updates per batch. Our Tree-SVD still updates the subset embeddings after each batch update. As we can observe, Tree-SVD is still beneficial after up to 32×10^4 edge updates, which accounts for up to 10% of the edges on the tested datasets. This shows the high effectiveness of our dynamic Tree-SVD since in real-world scenarios, the update is less frequent than that is tested in our experiment, where 10% edges get changed in a short period.

7 CONCLUSION

In this paper, we present Tree-SVD, an efficient and effective framework for dynamic subset embedding. Experiments show that our Tree-SVD is far more efficient than existing static and dynamic solutions while providing identical effectiveness. In this paper, we improve the effectiveness of subset embedding via spending more computational resources on a small subset. We note that if we focus on a subset of users with similar properties, e.g., in the same age group or same city, the performance of subset embedding also tends to improve over global counterparts. We plan to investigate this direction in our future work.

ACKNOWLEDGMENTS

Sibo Wang is supported by the Hong Kong RGC ECS grant (No. 24203419), RGC GRF grant (No. 14217322), RGC CRF grant (No. C4158-20G), Hong Kong ITC ITF grant (No. MRP/071/20X), NSFC grant (No. U1936205), and a gift from Huawei. Zengfeng Huang is supported by the National Natural Science Foundation of China No. 62276066, No. U2241212.

REFERENCES

[1] 2023. Technical report and source code. https://github.com/DoThingYo/Tree-Embedding.
[2] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.
[3] Xu Chen, Siheng Chen, Jiangchao Yao, Huangjie Zheng, Ya Zhang, and Ivor W Tsang. 2022. Learning on attribute-missing graphs. *TPAMI* 44, 2 (2022), 740–757.
[4] Kenneth L Clarkson and David P Woodruff. 2017. Low-rank approximation and regression in input sparsity time. *JACM* 63, 6 (2017), 1–45.

[5] Lun Du, Yun Wang, Guojie Song, Zhicong Lu, and Junshan Wang. 2018. Dynamic Network Embedding : An Extended Approach for Skip-gram based Network Embedding. In *IJCAI*. 2086–2092.
[6] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. 2018. Fast Randomized PCA for Sparse Data. In *ACML*. 710–725.
[7] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. 855–864.
[8] Xingzhi Guo, Baojian Zhou, and Steven Skiena. 2021. Subset Node Representation Learning over Large Dynamic Graphs. In *SIGKDD*. 516–526.
[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*.
[10] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, YongDong Zhang, and Meng Wang. 2020. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *SIGIR*. 639–648.
[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.
[12] Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. 2021. Massively parallel algorithms for personalized pagerank. *PVLDB* 14, 9 (2021), 1668–1680.
[13] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *PACMMOD* 1, 1 (2023), 25:1–25:26.
[14] Mark A Iwen and BW Ong. 2016. A distributed and incremental SVD algorithm for agglomerative data analysis on large networks. *SIMAX* 37, 4 (2016), 1699–1718.
[15] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation Learning for Dynamic Graphs: A Survey. *JMLR* 21, 70 (2020), 1–73.
[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.
[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*. 591–600.
[18] Peter Lofgren. 2015. *Efficient algorithms for personalized pagerank.* Stanford University.
[19] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *SIGIR*. 719–728.
[20] Sedigheh Mahdavi, Shima Khoshraftar, and Aijun An. 2018. dynnode2vec: Scalable Dynamic Network Embedding. In *ICBD*. 3762–3765.
[21] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020).
[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeurIPS*. 3111–3119.
[23] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *WWW Companion*. 969–976.
[24] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. *Efficient PageRank Tracking in Evolving Networks.* 875–884.
[25] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *SIGKDD*. 1105–1114.
[26] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).
[27] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *SIGKDD*. 701–710.
[28] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Chi Wang, Kuansan Wang, and Jie Tang. 2019. NetSMF: Large-Scale Network Embedding As Sparse Matrix Factorization. In *WWW*. 1509–1520.
[29] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and Node2vec. In *WSDM*. 459–467.
[30] Mingyue Tang, Pan Li, and Carl Yang. 2022. Graph Auto-Encoder via Neighborhood Wasserstein Reconstruction. In *ICLR*.
[31] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.
[32] Anton Tsitsulin, Marina Munkhoeva, Davide Mottin, Panagiotis Karras, Ivan Oseledets, and Emmanuel Müller. 2021. FREDE: Anytime Graph Embeddings. *PVLDB* 14, 6 (2021), 1102–1110.
[33] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. 2020. Personalized pagerank to a target node, revisited. In *SIGKDD*. 657–667.
[34] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. Hubppr: effective indexing for approximate personalized pagerank. *PVLDB* 10, 3 (2016), 205–216.
[35] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient algorithms for approximate single-source personalized pagerank queries. *TODS* 44, 4 (2019), 1–37.

[36] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and Effective Approximate Single-Source Personalized PageRank. In *SIGKDD*. 505–514.

[37] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. 2018. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In *SIGMOD*. 441–456.

[38] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *PVLDB* 13, 5 (2020), 670–683.

[39] Yuan Yin and Zhewei Wei. 2019. Scalable graph embeddings via sparse transpose proximities. In *SIGKDD*. 1429–1437.

[40] Wenchao Yu, Wei Cheng, Charu C Aggarwal, Haifeng Chen, and Wei Wang. 2017. Link Prediction with Spatial and Temporal Consistency in Dynamic Networks. In *IJCAI*. 3343–3349.

[41] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *SIGKDD*. 1315–1324.

[42] Xingyi Zhang, Kun Xie, Sibo Wang, and Zengfeng Huang. 2021. Learning Based Proximity Matrix Factorization for Node Embedding. In *SIGKDD*. 2243–2253.

[43] Ziwei Zhang, Peng Cui, Haoyang Li, Xiao Wang, and Wenwu Zhu. 2018. Billion-scale network embedding with iterative random projection. In *ICDM*. 787–796.

[44] Ziwei Zhang, Peng Cui, Jian Pei, Xiao Wang, and Wenwu Zhu. 2018. TIMERS: Error-Bounded SVD Restart on Dynamic Networks. In *AAAI*. 224–231.

Received July 2022; revised October 2022; accepted November 2022