# Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs: Technical Report

Anonymous Author(s)

## ABSTRACT

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., 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.

**ACM Reference Format:**

Anonymous Author(s). 2022. Efficient Tree-SVD for Subset Node Embedding over Large Dynamic Graphs: Technical Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 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, 21, 23, 24, 29, 32], to devise efficient and effective node embeddings. 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 [29], 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)* [22] that is widely adopted in many embedding frameworks [29, 32] 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 $\alpha$-decay random walk (aka. random walk with restart) from $s$ terminates at node $u$. Here, an $\alpha$-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 $\alpha$, 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 $S$ 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., [20, 31], and then re-hash the PPR vector according

**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 |

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 [27]. 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 SVD to generate the embedding.

Existing state-of-the-art node embedding methods are mostly MF-based. However, global MF-based methods like NetSMF [24], NPR [28], Lemane [32] and STRAP [29] are difficult to adapt to the dynamic subset embedding setting. To explain, for methods that decompose on 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 [32], 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 [29], there exist efficient algorithms to update the PPR matrix of the subset $S$, e.g., [20, 31], as shown in DynPPE [8]. However, it still needs to re-do the SVD when the input graph changes, resulting in unnecessarily high computational costs. There also exist several *graph neural network (GNN)*-based methods [15, 17, 26] that employ long-short term memory [10] 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 [29], and NPR [28], 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 [31]. 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., SVD, to $\boldsymbol{M}_S$, which is too time-consuming.

Our key idea is to divide the proximity matrix into sub-matrices, then do SVD for each one, and finally merge the 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)* [11], 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. 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 the existing hierarchical SVD framework [11] 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 compromising the accuracy of downstream tasks.
- Extensive experiments show that Tree-SVD outperforms existing non-MF-based methods by a large margin in terms of effectiveness. In addition, 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 the 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 $t$,

```
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     $\boldsymbol{p}_s[u] += \alpha \cdot \boldsymbol{r}_s[u], \boldsymbol{r}_s[u] = 0$
```

and the set $\Delta^t$ of edge events that occurred between snapshot $t-1$ and snapshot $t$. Following [12], the dynamic graph model is defined as follows to simulate the graph evolution process over time.

*Definition 2.1 (Dynamic graph model [12]).* 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 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 $\boldsymbol{X}^{t+1}$.

**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 $\alpha$-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. [1] to derive the PPR scores for each node $v$ with respect to a given source $s$. The pseudo-code of the Forward-Push algorithm [1] 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 $u$ [1, 14]:

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

```
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. $\frac{\boldsymbol{r}_s[w]}{d_{out}(w)} < -r_{max}$ do
11     PUSH($w, \alpha, \mathcal{G}^{t+1}, r_{max}, \boldsymbol{p}_s^t, \boldsymbol{r}_s^t$)
12 return ($\boldsymbol{p}_s^{t+1} \leftarrow \boldsymbol{p}_s^t, \boldsymbol{r}_s^{t+1} \leftarrow \boldsymbol{r}_s^t$);
```

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 [31]. 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.

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. [31] 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 [31], 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 for dynamic subset embedding 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 $h : \mathbb{R}^n \to \mathbb{R}^d$ 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 SVD on the proximity matrix and then incrementally update on the SVD results, another baseline of our Tree-SVD is to do SVD on the proximity matrix of the subset $S$ from scratch at each snapshot. As we discussed

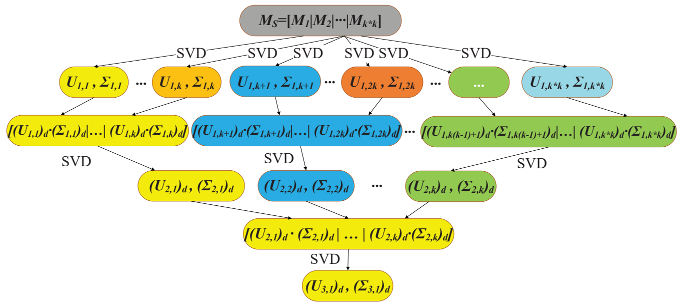

$M_S=[M_1|M_2|\cdots|M_{k*k}]$
SVD
$U_{1,1}, \Sigma_{1,1}$
$U_{1,k}, \Sigma_{1,k}$
$U_{1,k+1}, \Sigma_{1,k+1}$
$U_{1,2k}, \Sigma_{1,2k}$
$U_{1,k*k}, \Sigma_{1,k*k}$
$[(U_{1,1})_d\cdot(\Sigma_{1,1})_d|\ldots|(U_{1,k})_d\cdot(\Sigma_{1,k})_d]$
$[(U_{1,k+1})_d\cdot(\Sigma_{1,k+1})_d|\ldots|(U_{1,2k})_d\cdot(\Sigma_{1,2k})_d]$
$[(U_{1,k(k-1)+1})_d\cdot(\Sigma_{1,k(k-1)+1})_d|\ldots|(U_{1,k*k})_d\cdot(\Sigma_{1,k*k})_d]$
$(U_{2,1})_d, (\Sigma_{2,1})_d$
$(U_{2,2})_d, (\Sigma_{2,2})_d$
$(U_{2,k})_d, (\Sigma_{2,k})_d$
$[(U_{2,1})_d\cdot(\Sigma_{2,1})_d|\ldots|(U_{2,k})_d\cdot(\Sigma_{2,k})_d]$
$(U_{3,1})_d, (\Sigma_{3,1})_d$

**Figure 1: Tree-SVD with 3 levels**

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 [29]. Thus, we extend STRAP to the subset embedding setting, and denote this extension as Subset-STRAP. Subset-STRAP follows STRAP, which imposes a threshold $\delta$ and returns at most $O\left(\frac{1}{\delta}\right)$ proximity scores no smaller than $\delta$ for each node, making a proximity matrix with non-zero entries of size $O\left(\frac{|S|}{\delta}\right)$. 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 randomized SVD [6] is applied on $\boldsymbol{M}_S$ to decompose $\boldsymbol{M}_S$ into three matrices $\boldsymbol{U} \in \mathbb{R}^{|S|\times d}$, $\boldsymbol{V} \in \mathbb{R}^{d\times n}$, and $\Sigma \in \mathbb{R}^{d\times d}$. The returned three matrices have the following guarantee in terms of Frobenius norm:

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

Following [29], the embedding of Subset-STRAP is $\boldsymbol{X} = \boldsymbol{U}\sqrt{\Sigma}$.

**FREDE.** Another competitor is FREDE [27], which 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 [33], 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.

## 3 OUR SOLUTION: TREE-SVD

The main idea of our solution is to vertically divide the proximity matrix into multiple sub-matrices and do truncated SVD on each sub-matrix. We concatenate factorized sub-matrices in groups to form the intermediate matrices. Then, truncated SVD is invoked on each intermediate matrix. 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 SVD as our final embedding. The level $q$ of Tree-SVD is defined as the largest number of SVDs proceeded 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 solution is a combination of sparse randomized SVD [4] and hierarchical SVD [11]. Concretely, given the original proximity matrix, we could 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 $|S| * d$ matrix formed by $(\boldsymbol{U})_{\boldsymbol{d}}(\Sigma)_{\boldsymbol{d}}$. Note that the hierarchical SVD in [11] 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 3.1.

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 hierarchial SVD 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 extremely accelerates our computation. Then we could conduct exact SVD in the next few levels with very small column dimension efficiently. Therefore, this two-step Tree-SVD scheme, as a combination of sparse randomized SVD and hierarchical SVD, inherits both the efficiency of the sparse randomized SVD and the flexibility of the hierarchical SVD.

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 re-use the intermediate results without re-doing SVD on the sub-matrix from scratch. However, such a solution may still become ineffective if a

**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         $B_{l+1,j} :=$
          $[(U_{l,(j-1)k+1})_d(\Sigma_{l,(j-1)k+1})_d, |\cdots|(U_{l,jk})_d(\Sigma_{l,jk})_d]$.
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 $X = (U_{q,1})_d\sqrt{(\Sigma_{q,1})_d}$
```

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 proceeded. 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 $M_S$. For each node $s \in S$, we perform a Forward-Push on graph $G$ and derive the estimation vector $p_s$. Then, a Forward-Push is also performed for each node $s \in S$ on the reverse graph $G^\top$ by reversing the direction of each edge. For the proximity matrix $M_S$ of size $|S| \times n$, $M_S(s, v)$ is:

$$M_S(s,v) = \log\left(\frac{p_s(v)}{r_{max}} + \frac{p_s^\top(v)}{r_{max}}\right).$$

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, e.g., [29, 32]. Given the proximity matrix $M_S$, we divide the matrix into $b$ different sub-matrices:

$$M_S = [M_{1,1}|M_{1,2}|\cdots|M_{1,b}].$$

where 1 in $M_{1,j}$ denotes the sub-matrix is in the first level, $j$ denotes the $j$-th sub-matrix in the first level.

In the rest of the paper, we define $(M_S)_d$ as the best $d$-rank approximation of $M_S$, i.e., $||(M_S)_d - M_S||_F = \min_{\mathbf{rank}(A)\le d} ||(A)_d - M_S||_F$. For a matrix $B$, let $U\Sigma V$ be the SVD of matrix $B$, i.e., $B = U\Sigma V$, where $U$ is the left singular vector matrix, $\Sigma$ is the diagonal singular value matrix, and $V$ is the right singular vector matrix. We define $\bar{B}$ to be $BV^*$, i.e., $\bar{B} = BV^*$, which is equal to $U\Sigma$. It is important to note that $\bar{B}$ is not necessarily uniquely determined by $B$, e.g., if $B$ is rank deficient and/or has repeated singular values.

Similarly, we define a statement of the form $\bar{B} = \bar{C}$ as meaning that $B$ and $C$ are equivalent up to multiplication by a unitary matrix $E$ on the right, i.e. $B = CE$ or $BE = C$.

We define $B_{l,j}$ as the $j$-th actual sub-matrix that we factorize in the $l$-th level of Tree-SVD. Let $B_{1,j} = M_{1,j}$, i.e., the $j$-th sub-matrix of the input matrix $M_S$ at the first level. Recap that for a sub-matrix $B_{1,j}$ at the first level, we apply a randomized SVD to derive the best $d$-rank approximation of $B_{1,j}$. Let $(B_{1,j})_d = (U_{1,j})_d(\Sigma_{1,j})_d(V_{1,j})_d$ be the $d$-rank $\epsilon$-approximation of $B_{l,j}$ (Ref. to Equation 1) derived by the randomized SVD. For the second level, we define $B_{2,j}$ as:

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

i.e., we merge the SVD results of the first $k$ consecutive sub-matrices at the first level ($B_{1,1}, B_{1,2}, \cdots, B_{1,k}$) and produce $B_{2,1}$, the first sub-matrix at level two; the SVD results of the next $k$ consecutive sub-matrices ($B_{1,k+1}, B_{1,k+2}, \cdots, B_{1,2k}$) produces $B_{2,2}$, the second sub-matrix at level two, and etc. More generally, for $l > 1$, given the sub-matrix $B_{l,j}$, let $U_{l,j}$, $\Sigma_{l,j}$, and $V_{l,j}$ be the SVD of $B_{l,j}$ (hence $B_{i,j} = U_{l,j}\Sigma_{l,j}V_{l,j}$), where $U_{l,j}$, $\Sigma_{l,j}$, and $V_{l,j}$ are the matrix of the left singular vectors, the diagonal matrix of the singular values, and the matrix of the right singular vectors of $B_{l,j}$. Then, we take the $d$-rank truncated SVD of $B_{l,j}$ by keeping the first $d$ columns of $U_{l,j}$, first $d$ columns of $\Sigma_{l,j}$, first $d$ columns of $V_{l,j}$, and then taking their product, i.e., $(B_{l,j})_d = (U_{l,j})_d \cdot (\Sigma_{l,j})_d \cdot (V_{l,j})_d$. Then, for the $(l+1)$-th level ($l > 1$), we define $B_{l+1,j}$ as:

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

Finally, on the last level $q$, there is only one matrix $B_{q,1}$. We derive the $d$-rank truncated SVD of $B_{q,1}$ and output $(U_{q,1})_d$ and $(\Sigma_{q,1})_d$ as the final SVD result. For the subset embedding, it returns $(U_{q,1})_d\sqrt{(\Sigma_{q,1})_d}$ as the embedding of vertex set $S$. Here, we assume that $k$ sub-matrices are aggregated together in each level, which is the setting 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 $b = k^{q-1}$. For example, the number $b$ of sub-matrices at the first level of a 3-level Tree-SVD as shown in Figure 1 satisfies that $b = k^2$. 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 $b$. Algorithm 3 shows the pseudo-code of Tree-SVD. The steps are self-explanatory according to above discussions. Theorem 3.1 shows the quality guarantee of Tree-SVD.

Theorem 3.1. *Given the input matrix $M_S$ and level $q \ge 2$, let $(U_{q,1})_d$, $(\Sigma_{q,1})_d$ 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 $(\tilde{V}_{q,1})_d = (\Sigma_{q,1})_d^{-1}(U_{q,1})_d^\top M_S$. Then, there exists a unitary matrix $W$ such that Algorithm 3 is* *guaranteed to recover an* $\tilde{M}_S = (U_{q,1})_d(\Sigma_{q,1})_d(\tilde{V}_{q,1})_d \in \mathbb{R}^{|S|\times n}$ *with approximation guarantee. In particular, let* $\tilde{M}_S = \overline{(B_{q,1})_d} = (M_S)W + \Psi$*. Then, we have that:*

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

All proofs are deferred to Section 4. As we can observe from Theorem 3.1, which is 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 $q = 3$ in our implementation and $k = 8$, $d = 128$, $B_{1,j} = M_{1,j}$ as initial $j$-th horizontal block at level 1, to run Algorithm 3 to construct static 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 [11]. Assume that $nnz(M)$ indicates the number of non-zero elements in a matrix $M$. The following theorem shows the time complexity of Tree-SVD.

Theorem 3.2. *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* $\epsilon$ *be the error-parameter of the first level of randomized SVD. Then, Tree-SVD, as shown in Algorithm 3, takes* $O(nnz(M_S) + \frac{|S|\cdot d^2 \cdot b}{\epsilon^4})$.

For the hierarchical SVD proposed in [11], the time complexity is $O(|S|^2 \cdot n + |S|^3)$. Note that, $nnz(M_S)$ 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 [11].

## 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., $M_S^t$.

With Tree-SVD, the update becomes more efficient than global SVD methods as we only need to update the sub-matrices if they are changed, thus saving a lot of time. 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, whereas cache past representations in the hierarchical structure and put off the update of sub-matrices with negligible differences to a future snapshot that tends not to impact the quality of final embeddings.

Then the main task for us is to design a good measure for our lazy update scheme to monitor the changes of sub-matrices. Although it is quick, heuristic, 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 give us any theoretical guarantee to bound the error of the embedding. To overcome such

**Algorithm 4:** Update Tree-SVD Node Embedding( $q, k, d, (B_{l,j}^{t-i})_d, B_{1,j}^t$ ) , from snapshot $t-1$ to $t$.

```
Input: $q$, $k$, $d$, cached from snapshot $t-i$ as $(B_{l,j}^{t-i})_d$, $B_{1,j}^t$
1  Initialize an empty set $Z$ to record blocks updated.
2  Run Algorithm 2 to update the proximity matrix; for each
     block $B_{1,j}^t$, update the corresponding block from $B_{1,j}^{t-i}$ to
     $B_{1,j}^t$, and insert $j$'s index in $Z$ if
     $\left\|\left(B_{1,j}^{t-i}\right)_d - \left(B_{1,j}^{t-i}\right)\right\|_F + \|D_j\|_F > \sqrt{2}\delta \left\|B_{1,j}^t\right\|_F$.
3  for $l = 1, 2, ..., q-1$ do
4      Initialize a new empty set $Z_{parent}$ to record the blocks
         that needs to be updated in the next level.
5      for $j \in Z$ do
6          Compute truncated SVDs (resp. randomized SVD) of
             $(B_{l,j}^t)_d$ at the $l$-th level with $l > 1$ (resp. $l = 1$) to
             get $(U_{l,j}^t)_d$, $(\Sigma_{l,j}^t)_d$. Let the parent of $B_{l,j}$ be $B_{l+1,x}$.
             Insert parent index $x$ into $Z_{parent}$.
7      for $j \in Z_{parent}$ do
8          $B_{l+1,j} :=$
             $\left[(U_{l,(j-1)k+1}^t)_d(\Sigma_{l,(j-1)k+1}^t)_d, | \cdots |(U_{l,jk}^t)_d(\Sigma_{l,jk}^t)_d,\right]$
9      $Z = Z_{parent}$
10 Compute truncated SVD of $B_{q,1}^t$ to get $(U_{q,1}^t)_d$, $(\Sigma_{q,1}^t)_d$.
11 Set $X^t = (U_{q,1}^t)_d\sqrt{(\Sigma_{q,1}^t)_d}$
```

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., $\left\|\left(B_{1,j}^{t-i}\right)_d - \left(B_{1,j}^t\right)\right\|_F$. We further bound these term by the original desired error $\left\|M_{1,j}^t - (M_{1,j}^t)_d\right\|_F$. However, $\left\|M_{1,j}^t - (M_{1,j}^t)_d\right\|_F$ 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 $\left\|M_{1,j}^t\right\|_F$ as a replacement, which is summarized in Lemma 3.3.

Lemma 3.3. *Let* $j \in \{1, 2, \cdots, b\}$. *Let* $D_j \in \mathbb{R}^{|S|\times n_j}$ *be a matrix such that* $B_{1,j}^t = B_{1,j}^{t-i} + D_j$ *holds for* $B_{1,j}^t, B_{1,j}^{t-i}$*, and* $i \in [1, t-1]$. *If* $B_{1,j}^t, B_{1,j}^{t-1}$*, and* $D_j$ *satisfy that:*

$$\left\|\left(B_{1,j}^{t-i}\right)_d - \left(B_{1,j}^{t-i}\right)\right\|_F + \|D_j\|_F \leq \sqrt{2}\delta \left\|B_{1,j}^t\right\|_F. \qquad (2)$$

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

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

Our lazy-update strategy is inspired by Lemma 3.3, where we only re-compute the first level SVD for sub-matrices that violates Equation 2. 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 $\boldsymbol{M}_S$ by the dynamic Forward-Push algorithm (Algorithm 2). Then, we can easily derive $\boldsymbol{D}_j$ for each sub-matrix $\boldsymbol{B}^t_{1,j}$, where $1 \le j \le b = k^{q-1}$. Besides, in the computation steps of Tree-SVD, $(\boldsymbol{B}_{1,j})_d$ has been computed as it needs to be fed to the next level. Thus, at timestamp $t$, $(\boldsymbol{B}^{t-i}_{1,j})_d$ is already available, where $t-i$ is the timestamp of the last time when the SVD is computed for $\boldsymbol{B}_{1,j}$. That means from timestamp $t-i$ to $t-1$, it satisfies Equation 2 and thus the SVD results 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 will violate the condition in Equation 2. If this is the case, we then record the index of such a sub-matrix and add it to $Z$ (Line 2). Next, we update the affected sub-matrices in a bottom-up fashion. 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. $l = 1$) (Lines 5-6). 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 6). 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 7-8). 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 9). The updates are proceeded until we reach level $q$ where we compute the truncated SVD of the only matrix $\boldsymbol{B}_{q,1}$ at level $q$ (Line 10) and return the updated embedding (Line 11).

The following theorem shows that the lazy-update strategy still provides approximation guarantee for the final SVD result.

Theorem 3.4. *Let $\boldsymbol{M}^t_S \in \mathbb{R}^{|S| \times n}$ be the input matrix and $q \ge 2$. Algorithm 4 is guaranteed to recover an $(\tilde{\boldsymbol{M}}_S)^t_{q,1} \in \mathbb{R}^{|S| \times n}$ such that $\overline{\left((\tilde{\boldsymbol{M}}_S)^t_{q,1}\right)_d} = (\boldsymbol{M}_S)^t \boldsymbol{W} + \Psi$, where $\boldsymbol{W}$ is a unitary matrix, and*

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

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 3.3 and thus no block is updated in the current snapshot $t$, we get error bound as Theorem 3.4. *(ii)* When blocks are updated partially in the current snapshot $t$, we have the same error bound but with $\delta \le \frac{1+\epsilon}{\sqrt{2}}$ 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 3.1.

Figure 2 shows an example of the lazy update algorithm. The yellow blocks indicate that the SVD results can be still reused. The blue color indicates that $\boldsymbol{B}_{1,k+1} = \boldsymbol{M}_{1,k+1}$ in the first level does not satisfy Equation 2 at the current timestamp. Then, a sparse randomized SVD is redo on $\boldsymbol{B}_{1,k+1}$. Then, all its parents at level 2 and level 3 also need to be updated. However, the running cost at level 2 and level 3 are much smaller than that at level one. In

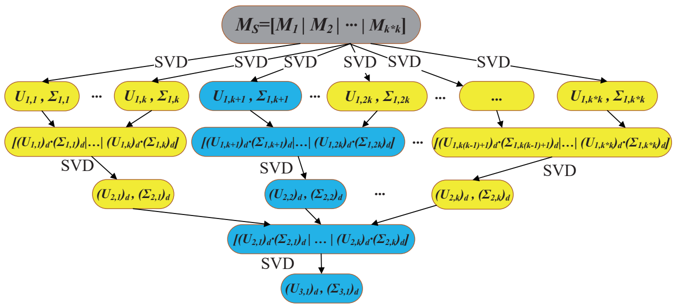

$M_S = [M_1 \mid M_2 \mid \cdots \mid M_{k*k}]$
SVD
SVD
SVD
SVD
SVD
SVD
$U_{1,1}, \Sigma_{1,1}$
$U_{1,k}, \Sigma_{1,k}$
$U_{1,k+1}, \Sigma_{1,k+1}$
$U_{1,2k}, \Sigma_{1,2k}$
$U_{1,k*k}, \Sigma_{1,k*k}$
$[(U_{1,1})_d \cdot (\Sigma_{1,1})_d \mid \ldots \mid (U_{1,k})_d \cdot (\Sigma_{1,k})_d]$
$[(U_{1,k+1})_d \cdot (\Sigma_{1,k+1})_d \mid \ldots \mid (U_{1,2k})_d \cdot (\Sigma_{1,2k})_d]$
$[(U_{1,k(k-1)+1})_d \cdot (\Sigma_{1,k(k-1)+1})_d \mid \ldots \mid (U_{1,k*k})_d \cdot (\Sigma_{1,k*k})_d]$
SVD
SVD
SVD
$(U_{2,1})_d, (\Sigma_{2,1})_d$
$(U_{2,2})_d, (\Sigma_{2,2})_d$
$(U_{2,k})_d, (\Sigma_{2,k})_d$
$[(U_{2,1})_d \cdot (\Sigma_{2,1})_d \mid \ldots \mid (U_{2,k})_d \cdot (\Sigma_{2,k})_d]$
SVD
$(U_{3,1})_d, (\Sigma_{3,1})_d$

**Figure 2: Lazy update with only blue color blocks updated.**

addition, most of the sub-matrices (yellow ones) are not updated, saving a lot of computational costs compared to a re-computation of SVD from scratch. With this lazy update strategy, for every snapshot $t$, we just need to update a small portion of blocks, which overcomes the existing efficiency bottleneck of MF-based approach, at the same time achieve accuracy on downstream tasks comparable to the best static counterparts. We have the following theorem to show the update cost of Algorithm 4.

Theorem 3.5. *Given a batch of $\tau$ edge updates, assume that $b'$ sub-matrices at the first level violates Equation 2 and the number of non-zero entries in $b'$ sub-matrices is $nnz'$, Algorithm 4 has a time complexity of $O\left(\min\{\tau + 1/r_{max}, |S|/r_{max}\} + nnz' + \frac{|S| \cdot d^2 \cdot k^2 \cdot b'}{\epsilon^4}\right)$.*

## 4 THEORETICAL ANALYSIS

In this section, we present proofs of all theorems and lemmas.

**Proof of Theorem 3.1.** 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 $\boldsymbol{B}^t_{l,j}$. 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) $\overline{\left(\boldsymbol{B}_{l,j}\right)_d} = \boldsymbol{M}_{l,j} \boldsymbol{W}_{l,j} + \Psi_{l,j}$;
(2) $\boldsymbol{W}_{l,j}$ is always a unitary matrix;
(3) $\|\Psi_{l,j}\|_F \le \left((2+\epsilon)(1+\sqrt{2})^{l-1} - 1\right) ||(\boldsymbol{M}_{l,j})_d - \boldsymbol{M}_{l,j}||_F$.

Here for any classic Randomized-SVD, $\forall \boldsymbol{M}_{1,j}$, we have

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

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

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

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

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

where $\Psi_{1,j} := \left[\left(B_{1,j}\right)_d - M_{1,j}\right] W_{1,j}$. Then we have

$$\begin{aligned}\|\Psi_{1,j}\|_F &= \left\|\overline{\left(B_{1,j}\right)_d} - M_{1,j}W_{1,j}\right\|_F \\ &= \left\|\left(B_{1,j}\right)_d - M_{1,j}\right\|_F \le (1+\epsilon)\|(M_{1,j})_d - M_{1,j}\|_F. \\ &= \left((2+\epsilon)(1+\sqrt{2})^{1-1} - 1\right)\left\|M_{l+1,j} - \left(M_{l+1,j}\right)_d\right\|_F.\end{aligned}$$

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

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

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

$$\begin{aligned}&\left\|\left(B_{l+1,j}\right)_d - M_{l+1,j}\tilde{W}\right\|_F \\ &\le \left\|\left(B_{l+1,j}\right)_d - B_{l+1,j}\right\|_F + \left\|B_{l+1,j} - M_{l+1,j}\tilde{W}\right\|_F \\ &= \sqrt{\sum_{j=d+1}^{D} \sigma_j^2\left(M_{l+1,j}\tilde{W} + \tilde{\Psi}\right)} + \|\tilde{\Psi}\|_F \\ &\le \sqrt{\sum_{j=d+1}^{D} 2\sigma_j^2\left(M_{l+1,j}\tilde{W}\right)} + \sqrt{\sum_{j=1}^{D} 2\sigma_j^2(\tilde{\Psi})} + \|\tilde{\Psi}\|_F \\ &= \sqrt{2}\left\|M_{l+1,j} - \left(M_{l+1,j}\right)_d\right\|_F + (1+\sqrt{2})\|\tilde{\Psi}\|_F.\end{aligned} \tag{4}$$

In addition, by Condition 3, we know that:

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

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:

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

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 the proved three conditions, we get the claimed result in Theorem 3.1, which finishes the proof.

**Proof of Theorem 3.2.** Firstly, consider the first level sparse randomized SVDs in Tree-SVD, which is the dominating cost of Tree-SVD. For sparse randomized SVD, given a matrix $M$ of size $|S| \times n$, it is shown in [3] that it takes $O\left(nnz(M) + \frac{|S|\cdot d^2}{\epsilon^4}\right)$ computational cost to provide $\epsilon$-approximation. Since the number of non-zero entries of all sub-matrices $B_{1,1}, B_{1,2} \cdots, B_{1,b}$ is exactly $nnz(M_S)$, and by the fact that we have $b$ sub-matrices, we derive that the computational cost of the first level is: $O(nnz(M_S) + \frac{|S|\cdot d^2 \cdot b}{\epsilon^4})$.

Then, for the second level, there are in total $b/k$ sub-matrices of size $|S| \times (k \cdot d)$. The time complexity of an exact SVD to a matrix of size $x \times y$ is bounded by $O(\min\{xy^2, x^2y\})$. Thus, the time complexity of the second level is obviously bounded by $O(|S| \cdot k^2 \cdot d^2 \cdot k^{q-2}) = O(|S| \cdot b \cdot d^2/k)$. We can derive the similar complexity in the third level as $O\left(|S|b \cdot d^2\right)$ and so on. Adding the total cost from the second level to the last level $q$, we can bound the cost to be $O(|S| \cdot d^2 \cdot b \cdot k)$. As we can treat $k$ as a constant. Thus, the total time complexity of Tree-SVD is bounded by $O(nnz(M_S) + \frac{|S|\cdot d^2 \cdot b}{\epsilon^4})$.

**Proof of Lemma 3.3.** We have $\left(B_{1,j}^{t-i}\right)_d = B_{1,j}^t + \left[\left(B_{1,j}^{t-i}\right)_d - B_{1,j}^t\right]$, for $i \in [1, t-1]$, such that there exists unitary matrix $W^{1,j}$ with

$$\overline{\left(B_{1,j}^{t-i}\right)_d} = B_{1,j}^t W_{1,j} + \left[\left(B_{1,i}^{t-i}\right)_d - B_{1,j}^t\right] W_{1,j}.$$

Thus, we can derive that:

$$\begin{aligned}\|\Psi_{1,j}\|_F &= \left\|\overline{\left(B_{1,j}^{t-i}\right)_d} - B_{1,j}^t W_{1,j}\right\|_F = \left\|\left(B_{1,j}^{t-i}\right)_d - B_{1,j}^t\right\|_F \\ &\le \left\|\left(B_{1,j}^{t-i}\right)_d - B_{1,j}^{t-i}\right\|_F + \left\|B_{1,j}^{t-i} - B_{1,j}^t\right\|_F \\ &= \left\|\left(B_{1,j}^{t-i}\right)_d - B_{1,j}^{t-i}\right\|_F + \|D_j\|_F \le \sqrt{2}\delta\left\|B_{1,j}^t\right\|_F.\end{aligned}$$

**Proof of Theorem 3.4.** We first prove the case with all blocks are error-bounded, in which no update is conducted. Let $(M_S)^t$ denote the original matrix with block components, i.e.,

$$(M_S)^t = [M_{1,1}^t | M_{1,2}^t | \cdots | M_{1,b}^t],$$

where $b = k^{q-1}$. Furthermore, let $M_{l,j}^t$ denote the error-free block of the original matrix $(M_S)^t$ whose entries correspond to the entries included in $B_{l,j}^t$. Thus,

$$(M_S)^t = \left[M_{l,1}^t \middle| M_{l,2}^t \middle| \cdots \middle| M_{l,b/k^{(l-1)}}^t\right]$$

holds for all $l \in [1, q]$, where $M_{l+1,j}^t := \left[M_{l,(j-1)k+1}^t\right| \cdots \left|M_{l,jk}^t\right]$, for all $l \in [1, q-1]$, and $j \in [1, b/k^l]$.

Suppose the cached representation of snapshot $t-i$, i.e. $\overline{(B_{1,j}^{t-i})}_d$, $i \in [1, t-1]$, is a good approximation to $B_{1,j}^t$, then we directly set $\overline{(B_{1,j}^t)}_d = \overline{(B_{1,j}^{t-i})}_d$, for $i \in [1, t-1]$. As our goal is to bound the final matrix $\overline{(\boldsymbol{B}^{t-i}_{q+1,1})_d}$ with respect to the original matrix $(\boldsymbol{M}_S)^t$. Next, we will prove this by induction on level $l$. More specifically, we will prove that:

(1) $\overline{\left(\boldsymbol{B}^{t-i}_{l,j}\right)_d} = \boldsymbol{M}^t_{l,j} W_{l,j} + \boldsymbol{\Psi}_{l,j}$;
(2) $\boldsymbol{W}_{l,j}$ is always a unitary matrix;
(3) $\|\boldsymbol{\Psi}^t_{l,j}\|_{\mathrm{F}} \le \left((1+\delta\sqrt{2})(1+\sqrt{2})^{l-1}-1\right)\|(\boldsymbol{M}_S)^t\|_{\mathrm{F}}$.

By Lemma 3.3, there exist unitary $\boldsymbol{W}_{1,j}$ for all $j \in [1,b]$ such that:

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

where $\boldsymbol{\Psi}_{1,j} := \left[\left(\boldsymbol{B}^{t-i}_{1,j}\right)_d - \boldsymbol{B}^t_{1,j}\right]\boldsymbol{W}_{1,j}$ satisfies

$$\left\|\overline{\left(\boldsymbol{B}^{t-i}_{1,j}\right)_{\boldsymbol{d}}} - \boldsymbol{B}^t_{1,j}\boldsymbol{W}_{1,j}\right\|_{\mathrm{F}} \le \sqrt{2}\delta\left\|\boldsymbol{B}^t_{1,j}\right\|_{\mathrm{F}}.$$

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

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

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 $\boldsymbol{B}^{t-i}_{l+1,j} = \boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{\Psi}}$. Then we can derive the following bound:

$$\begin{aligned}
&\left\|\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d - \boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}}\right\|_F\\
&\le \left\|\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d - \boldsymbol{B}^{t-i}_{l+1,j}\right\|_F + \left\|\boldsymbol{B}^{t-i}_{l+1,j} - \boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}}\right\|_F\\
&= \sqrt{\sum_{j=d+1}^{D}\sigma_j^2\left(\boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}} + \tilde{\boldsymbol{\Psi}}\right)} + \|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}}\\
&\le \sqrt{\sum_{j=d+1}^{D}2\sigma_j^2\left(\boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}}\right)} + \sqrt{\sum_{j=1}^{D}2\sigma_j^2(\tilde{\boldsymbol{\Psi}})} + \|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}}\\
&= \sqrt{2}\left\|\boldsymbol{M}^t_{l+1,j} - \left(\boldsymbol{M}^t_{l+1,j}\right)_d\right\|_{\mathrm{F}} + (1+\sqrt{2})\|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}}\\
&\le \sqrt{2}\left\|\boldsymbol{M}^t_{l+1,j}\right\|_{\mathrm{F}} + (1+\sqrt{2})\|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}}.
\end{aligned} \tag{6}$$

In addition, we know that:

$$\begin{aligned}
\|\tilde{\boldsymbol{\Psi}}\|_{\mathrm{F}} &= \sum_{i=1}^{k}\left\|\boldsymbol{\Psi}_{l,(j-1)k+i}\right\|_{\mathrm{F}}\\
&\le \sum_{i=1}^{k}\left((1+\delta\sqrt{2})(1+\sqrt{2})^{l-1}-1\right)\left\|\boldsymbol{M}^t_{l,(j-1)k+i}\right\|_{\mathrm{F}}\\
&= \left((1+\delta\sqrt{2})(1+\sqrt{2})^{l-1}-1\right)\left\|\boldsymbol{M}^t_{l+1,j}\right\|_{\mathrm{F}}.
\end{aligned} \tag{7}$$

Combining (6) and (7), we have:

$$\left\|\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d - \boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}}\right\|_F \le \left((1+\delta\sqrt{2})(1+\sqrt{2})^{l}-1\right)\left\|\boldsymbol{M}^t_{l+1,j}\right\|_{\mathrm{F}}.$$

Moreover, note that:

$$\left\|\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d - \boldsymbol{M}^t_{l+1,j}\tilde{\boldsymbol{W}}\right\|_{\mathrm{F}} = \left\|\overline{\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d} - \boldsymbol{M}^t_{l+1,j}W_{l+1,j}\right\|_{\mathrm{F}},$$

where $\boldsymbol{W}_{l+1,j}$ is unitary. Hence, Conditions 1-3 hold for $l+1$ with $\boldsymbol{\Psi}_{l+1,j} := \overline{\left(\boldsymbol{B}^{t-i}_{l+1,j}\right)_d - \boldsymbol{M}^t_{l+1,j}\boldsymbol{W}_{l+1,j}}$.

For the case we need to update blocks partially, although we can set a smaller $\delta$ to update more blocks, here we just set $\delta = \frac{1+\epsilon}{\sqrt{2}}$ for the worst case theoretical guarantee. At the first level, we have:

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

where $\boldsymbol{\Psi}_{1,j} := \left[\left(\boldsymbol{B}^t_{1,j}\right)_d - \boldsymbol{B}^t_{1,j}\right]\boldsymbol{W}_{1,j}$ satisfies that:

$$\left\|\overline{\left(\boldsymbol{B}^t_{1,j}\right)_d} - \boldsymbol{B}^t_{1,j}\boldsymbol{W}_{1,j}\right\|_{\mathrm{F}} \le (1+\epsilon)\left\|\boldsymbol{B}^t_{1,j}\right\|_{\mathrm{F}} = \sqrt{2}\delta\left\|\boldsymbol{B}^t_{1,j}\right\|_{\mathrm{F}}.$$

Then the same bound still holds for the final level.

**Proof of Theorem 3.5.** Based on the results derived by Zhang et al. [31], given $\tau$ edge updates, the total update cost of Algorithm 2 can be bounded by $O(\tau + 1/r_{max})$. When $\tau$ is large, it may actually dominate the cost and thus a re-computation of the $|S|$ PPR vectors turns out to be more efficient, which takes $O(|S|/r_{max}$ cost. Thus, the total cost of the forward push takes $O(\min\{\tau + 1/r_{max}, |S|/r_{max}\})$. Then, given $b'$ updated sub-matrices with in total $nnz'$ number of non-zero entries, then the randomized SVD at level 1 can be bounded by $O(nnz' + \frac{|S|\cdot k^2\cdot d^2\cdot b'}{\epsilon^4})$. We update constant levels and at each level the number of updated sub-matrices is no more than $b'$. Thus, the total update cost can be bounded by:

$$O\left(\min\{\tau + 1/r_{max}, |S|/r_{max}\} + nnz' + \frac{|S|\cdot d^2\cdot k^2\cdot b'}{\epsilon^4}\right),$$

which finises the proof.

## 5 OTHER RELATED WORK

**Static node embedding.** There are numerous works dealing with static node embedding. Traditional random walk based [7, 23] approaches are inspired by skip-gram model [18]. They focus on preserving the co-occurrence probability of the nodes on the random walks. MF-baseds methods [21, 25] 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 [29, 32] factorize a PPR matrix. Recent progress on deep learning provides alternative solutions [9, 13] for static node embedding. They compute node embeddings by training graph neural networks (GNNs). However, GNNs trains models in a supervised manner and they all need features as input, which is explained thoroughly in [2, 8]. Different from GNNs, we focus on designing unsupervised embedding methods on topology-only graphs.

**Dynamic node embedding.** CTDNE [19], DNE [5], and dynnode2vec [16] are random walk-based methods for dynamic node embedding task. The main idea of these solutions is to update the

**Table 2: Statistics of datasets.**

| Dataset | Type | $n$ | $m$ | $\|C\|$ | $\tau$ |
|---|---|---|---|---|---|
| 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 |

random walks according to the graph changes. Such solutions are generally outperformed by hashing-based methods as shown in [27]. Another line of research work focuses on designing dynamic algorithms for MF-based methods. LIST [30] 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 [34] proposes an incremental eigenvalue decomposition-based method that sets a tolerated error threshold for the restart time to reduce the 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.

### 6.1 Experimental Settings

**Datasets.** Following DynPPE [8], we use three large dynamic graph datasets, Patent, Mag-Authors, and Wikipedia for the node classification task. We also conduct experiments on link prediction task, which is broadly applied to social network analysis. Thus, we select two large social network datasets, YouTube and Flickr, together with Mag-authors dataset, as three link prediction datasets. The statistics of these datasets are shown in Table 2, where $|C|$ denotes the number of classes and $\tau$ 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 [29], 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 [33] and FREDE [27], two efficient embedding methods.

All competitors and their types are listed in Table 3. 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.

**Remark.** Note that GNNs **are not** our baselines. To explain, firstly, GNNs require features as input, while we focus on topology-only graphs where no feature vector is available [2, 8]; secondly, node embedding methods are unsupervised models while GNNs require label information on classification tasks.

**Parameter settings.** Following DynPPE [8], we set $|S| = 3000$ for all datasets. We generate the subset $S$ by repeating the following

**Table 3: Setting of all tested methods.**

| Type | Method | Threads |
|---|---|---|
| Static | FREDE | 64 |
| | RandNE | 64 |
| | Global-STRAP | 64 |
| | Subset-STRAP | 64 |
| | Tree-SVD-S | 64 |
| Dynamic | DynPPE | 64 |
| | Tree-SVD | 64 |

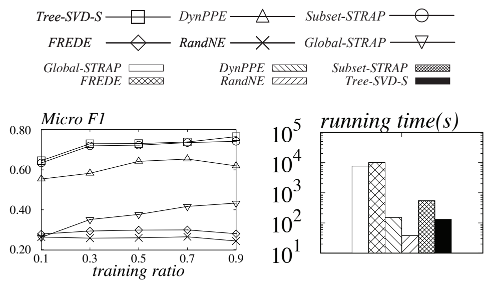
process: randomly sample 3000 nodes from the graph topology of the first snapshot to form subset $S$, which is the same as DynPPE [8]. In addition, for our Tree-SVD, we set the number of sub-matrices $b = 64$ and the level $q = 3$. For our lazy update strategy of Tree-SVD, we set $\delta = 0.65$ (Ref. 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 tuned $r_{max}$ 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 out-going 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 remaining irrelevant edges. Next, we randomly generate the same number of negative pairs 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 remaining edges.

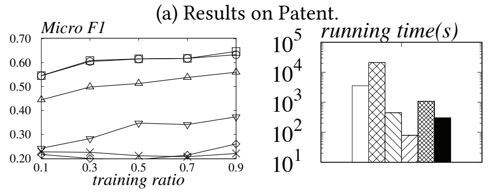
### 6.2 Static Subset Embedding

**Exp1: global vs. subset embedding methods.** We firstly 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.

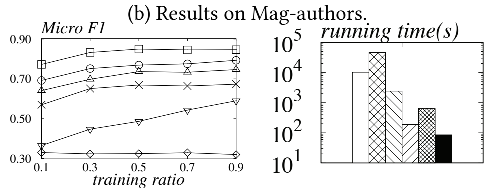
We first conduct experiments on the node classification task using three datasets that contain node labels, i.e., Patent, Mag-authors, and Wikipedia. Figure 3 reports the Micro-F1 scores and the embedding time of each method. As we can observe, compared to global embedding method Global-STRAP, Subset-STRAP achieve 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 as RandNE. Notice that DynPPE adopts a smaller $r_{max}$ to get a more accurate proximity matrix, resulting in 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



(a) Results on Patent.



(b) Results on Mag-authors.



(c) Results on Wikipedia.

**Figure 3: Exp. 1: node classification on Patent.**

datasets with much less time cost, while taking a lead by at least 5% on 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 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 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. Futhermore, 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 embedding vectors from scratch on each dataset. This again demonstrates that our Tree-SVD-S gains a better trade-off between the running time and embedding quality on static subset embedding.

**Table 4: Exp. 1: Precision score on LP task.**

| 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-Static | **82.40** | **92.68** | **90.42** |

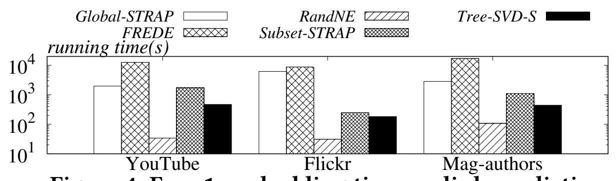


**Figure 4: Exp. 1: embedding time on link prediction.**

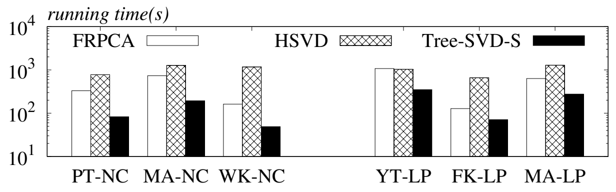


**Figure 5: Exp.2: embedding time of SVD methods.**

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

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

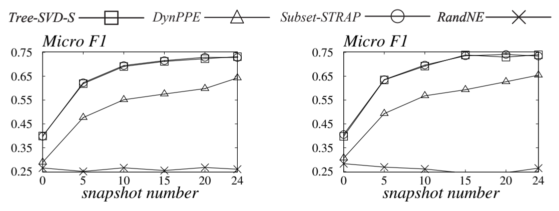
**Exp2: SVD comparison.** In the second set of experiments, we will compare our Tree-SVD-S against two SVD alternatives, FRPCA and hierarchical SVD (HSVD), to evaluate different SVD frameworks. To make a fair comparison, the proximity matrices of FRPCA and HSVD are the same as ours, i.e., computing the 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 task and link prediction task. 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.

**Table 6: Exp.2: precision on link prediction.**

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

**Table 7: Exp.4 :link prediction precision after $10^6$ edge events.**

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



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

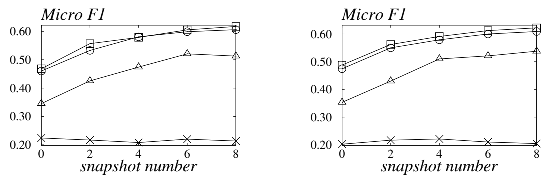
**Figure 6: Exp. 3: node classification on Patent.**

## 6.3 Dynamic Subset Embedding

**Exp. 3: impact of dynamic updates.** In the third set of experiments, 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.

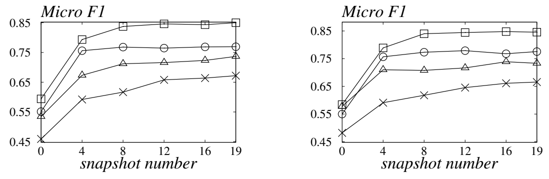
We first examine how the micro-F1 score of the node classification task changes along with snapshots on each dataset. We omit results of Global-STRAP as it shows inferior performance on subset embedding in Section 6.2. Since we re-construct the embeddings, Tree-SVD is the same as Tree-SVD-S. Figures 6-8 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.

Next, we examine how the precision changes along with snapshots for the link prediction task on each dataset. Figures 9 show the precision score of our Tree-SVD-S, the competitors RandNE and Subset-STRAP on Flickr and YouTube. 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 performances.



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

**Figure 7: Exp. 3: node classification on Mag-authors.**



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

**Figure 8: Exp. 3: node classification on Wikipedia.**

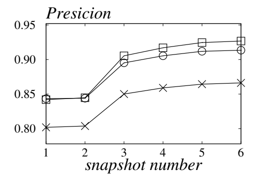
**Exp. 4: Batch updates.** In real-world applications, the models are usually updated daily or weekly depending on the efficiency of their update algorithms. We simulate such a scenario by updating the embeddings after every batch update, i.e., 10,000 edge events for all methods. In this set of experiments, 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 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 average update time for these 100 rounds and the Micro-F1 scores after the batch updates. 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 task. 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.

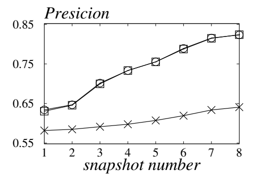
## 6.4 Parameter Analysis

In this subsection, we analyze the impact of different parameters on all datasets.

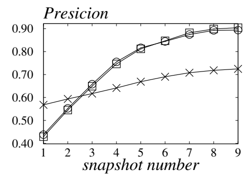
**Parameter $b$.** We compare our Tree-SVD-S against Hierarchical SVD (HSVD) using different number of $b$ sub-matrices at the first level during the SVD computation (Ref. to Section 3.1). We use the last snapshot to generate embeddings. Figure 11 reports experimental results and running time of HSVD and Tree-SVD-S. As we can observe, our Tree-SVD-S achieves comparable results as 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



(a) Flickr

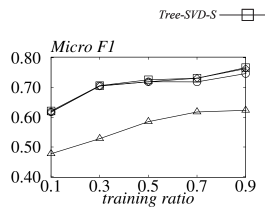


(b) YouTube

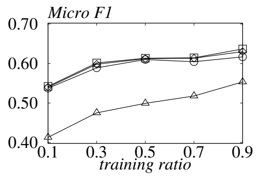


(c) Mag-authors

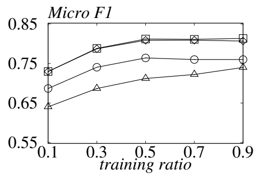
**Figure 9: Exp. 3: link prediction on different snapshots.**



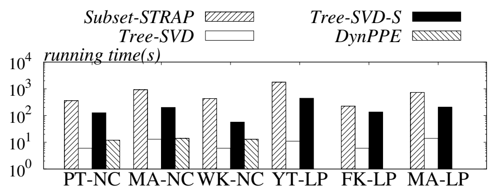
(a) Patent.



(b) Mag-authors.

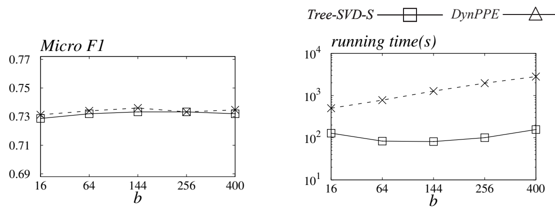


(c) Wikipedia.

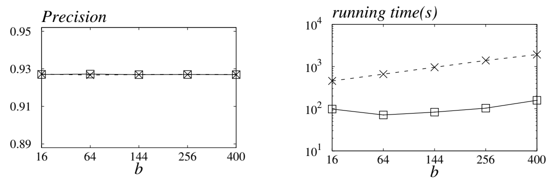


(d) Total embedding time.

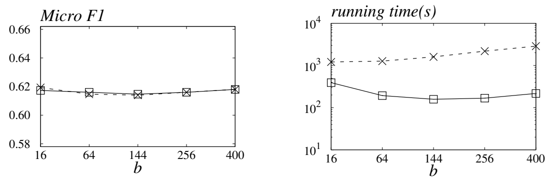
**Figure 10: Exp. 4: experimental results after $10^6$ edge events**



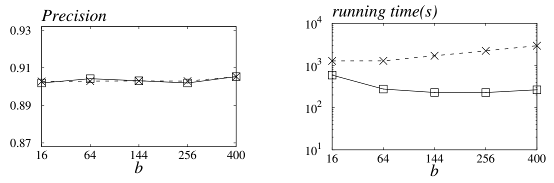
(a) Node classification on Patent.



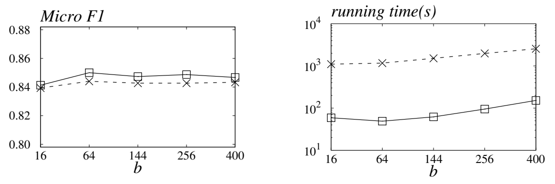
(d) Link prediction on Flickr.



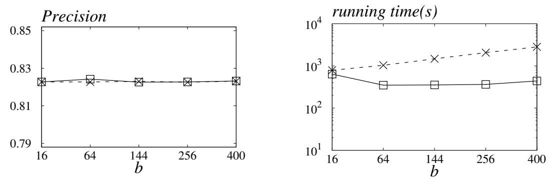
(b) Node classification on Mag-authors.



(e) Link prediction on Mag-authors.



(c) Node classification on Wikipedia.



(f) Link prediction on YouTube.

**Figure 11: Experimental results of Tree-SVD-S and HSVD with varying $b$**

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.

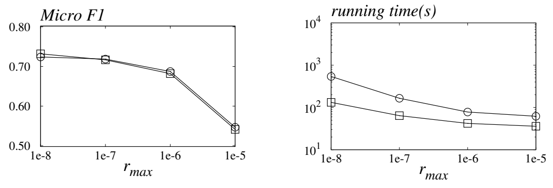
**Threshold $r_{max}$.** We use the last snapshot of each graph to generate embeddings. Figure 12 reports experimental results and running time of Subset-STRAP and Tree-SVD-S with varying $r_{max}$. As we can observe, our Tree-SVD-S achieves comparable results as Subset-STRAP on both tasks. On the other 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 $\delta$.** In this set of experiments, we generate dynamic embeddings for each graph. Figure 13 reports experimental results and running time of our Tree-SVD on different datasets with varying $\delta$. As we can observe, since $\delta$ controls the error bound in our lazy-update strategy (Ref. to Section 3.2), smaller $\delta$ leads to slightly improved results on all datasets.

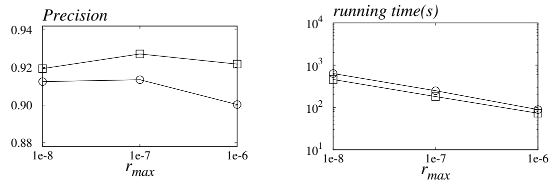
## 7 CONCLUSION

In this paper, we present Tree-SVD, an efficient and effective framework for dynamic subset embedding. Experiments show that our

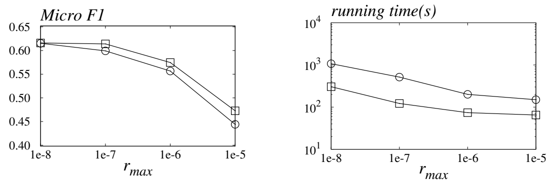
Tree-SVD-S
Subset-STRAP



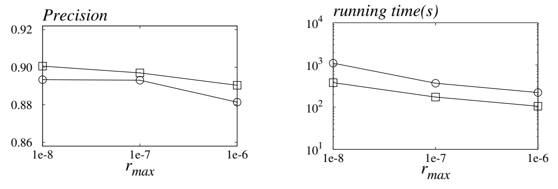
(a) Node classification on Patent.



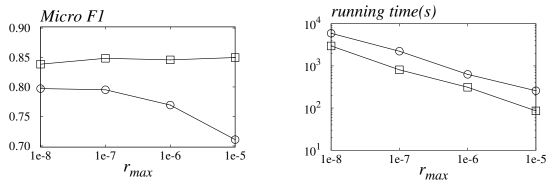
(d) Link prediction on Flickr.



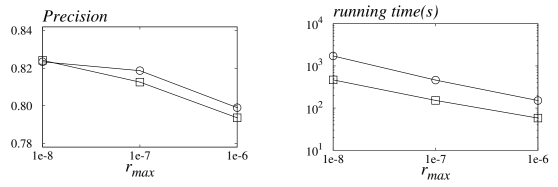
(b) Node classification on Mag-authors.



(e) Link prediction on Mag-authors.

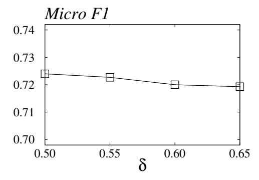


(c) Node classification on Wikipedia.

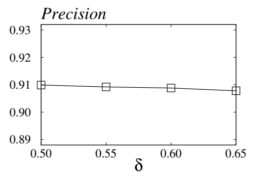


(f) Link prediction on YouTube.

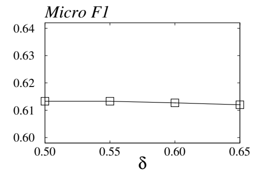
**Figure 12: Experimental results of Tree-SVD-S and Subset-STRAP with varying $r_{max}$**



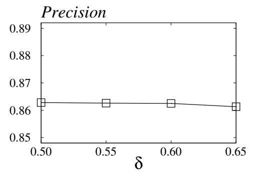
(a) Node classification on Patent.



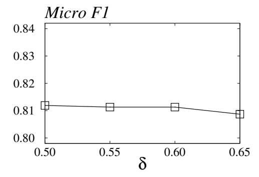
(b) Link prediction on Flickr.



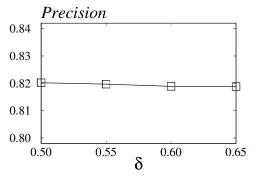
(a) Node classification on Mag-authors.



(b) Link prediction on Mag-authors.



(a) Node classification on Wikipedia.



(b) Link prediction on YouTube.

**Figure 13: Experimental results of Tree-SVD with varying $\delta$.**

Tree-SVD is far more efficient than existing static and dynamic solutions while providing identical effectiveness.

## REFERENCES

[1] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. 475–486.

[2] 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.

[3] Kenneth L. Clarkson and David P. Woodruff. 2013. Low rank approximation and regression in input sparsity time. In *STOC*. 81–90.

[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] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[11] 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.

[12] 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.

[13] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[14] Peter Lofgren. 2015. *Efficient algorithms for personalized pagerank*. Stanford University.

[15] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *SIGIR*. 719–728.

[16] Sedigheh Mahdavi, Shima Khoshraftar, and Aijun An. 2018. dynnode2vec: Scalable Dynamic Network Embedding. In *ICBD*. 3762–3765.

[17] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020).

[18] 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.

[19] 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.

[20] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. *Efficient PageRank Tracking in Evolving Networks.* 875–884.

[21] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *SIGKDD*. 1105–1114.

[22] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).

[23] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *SIGKDD*. 701–710.

[24] 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.
[25] 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.
[26] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.
[27] Anton Tsitsulin, Marina Munkhoeva, Davide Mottin, Panagiotis Karras, Ivan Oseledets, and Emmanuel Müller. 2021. FREDE: Anytime Graph Embeddings. *PVLDB* 14, 6 (2021), 1102–1110.
[28] 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.
[29] Yuan Yin and Zhewei Wei. 2019. Scalable graph embeddings via sparse transpose proximities. In *SIGKDD*. 1429–1437.
[30] 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.
[31] Hongyang Zhang, Peter Lofgren, and Ashish Goel. 2016. Approximate personalized pagerank on dynamic graphs. In *SIGKDD*. 1315–1324.
[32] Xingyi Zhang, Kun Xie, Sibo Wang, and Zengfeng Huang. 2021. Learning Based Proximity Matrix Factorization for Node Embedding. In *SIGKDD*. 2243–2253.
[33] Ziwei Zhang, Peng Cui, Haoyang Li, Xiao Wang, and Wenwu Zhu. 2018. Billion-scale network embedding with iterative random projection. In *ICDM*. 787–796.
[34] Ziwei Zhang, Peng Cui, Jian Pei, Xiao Wang, and Wenwu Zhu. 2018. TIMERS: Error-Bounded SVD Restart on Dynamic Networks. In *AAAI*. 224–231.