Noname manuscript No.
(will be inserted by the editor)

Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization

Ningyi Liao · Dingheng Mo · Siqiang Luo · Xiang Li · Pengcheng Yin

Received: date / Accepted: date

Abstract Recent advances in data processing have stimulated the demand for learning graphs of very large scales. Graph Neural Networks (GNNs), being an emerging and powerful approach in solving graph learning tasks, are known to be difficult to scale up. Most scalable models apply node-based techniques in simplifying the expensive graph message-passing propagation procedure of GNNs. However, we find such acceleration insufficient when applied to million- or even billion-scale graphs. In this work, we propose SCARA, a scalable GNN with feature-oriented optimization for graph computation. SCARA efficiently computes graph embedding from the dimension of node features, and further selects and reuses feature computation results to reduce overhead. Theoretical analysis indicates that our model achieves sub-linear time complexity with a guaranteed precision in propagation process as well as GNN training and inference. We conduct extensive experiments on various datasets to evaluate the efficacy and efficiency of SCARA. Performance comparison with baselines shows that SCARA can reach up to 800× graph propagation acceleration than current state-of-the-art methods with fast convergence and comparable accuracy. Most notably, it is efficient to process precomputation on the largest available billion-scale GNN dataset Papers100M (111M nodes, 1.6B edges) in 13 seconds.

Keywords Graph neural networks · Personalized PageRank · Graph attribute optimization · Scalability

Siqiang Luo is the corresponding author.

Ningyi Liao
Nanyang Technological University
E-mail: liao0090@e.ntu.edu.sg

Dingheng Mo
Nanyang Technological University
E-mail: dingheng001@e.ntu.edu.sg

Siqiang Luo
Nanyang Technological University
E-mail: siqiang.luo@ntu.edu.sg

Xiang Li
East China Normal University
E-mail: xiangli@dase.ecnu.edu.cn

Pengcheng Yin
Google Research
E-mail: pcyin@google.com

1 Introduction

Recent years have witnessed the burgeoning of online services based on data represented by graphs, which leads to a rapid increase in the amount and complexity of such graph data. Graph Neural Networks (GNNs) are specialized neural models designed to represent and process graph data, and have achieved strong performance on graph understanding tasks such as node classification [18, 21, 8, 12], link prediction [34, 48, 45, 37], and community detection [35, 1, 11].

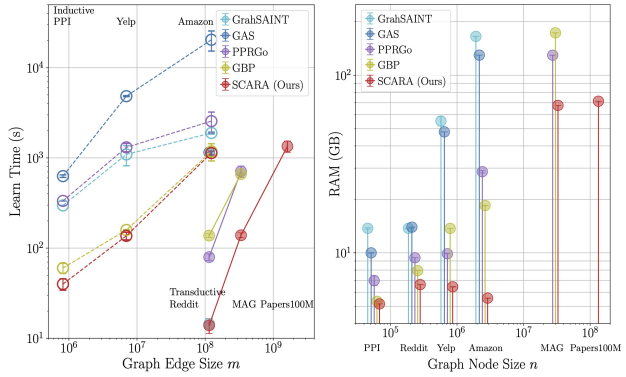
One of the most widely adopted GNN designs is the Graph Convolutional Network (GCN) [21] which learns graph representations by leveraging information of topological structure. Specifically, a GCN represents each node state by a feature vector, successively propagates the state to neighboring nodes, and updates the neighbor features using a neural network. This interleaved process of graph propagation and state update can proceed for multiple iterations.

While being able to effectively gather state information from the graph structure, GCNs are known to be resource-demanding, which implies limited scalability when deployed to large-scale graphs [43, 50]. It is also non-trivial to fit the node features of large graphs into the memory of hardware accelerators like GPUs. However, it is increasingly demanding to apply these effective models to modern real-world graph datasets. Recent studies have attempted to learn representations of large graphs such as the Microsoft Academic

Graph (MAG) with 100 million entries [31, 44]. Nonetheless, directly fitting the basic GCN model to such data would easily cause unacceptable training time or out-of-memory error. Hence, how to adopt the GCN model efficiently to these very large-scale graphs while benefiting from its performance becomes a challenging yet important problem in realistic applications.

Existing Approaches are Not Scalable Enough. Several techniques have been proposed towards more efficient learning for GNN, addressing the scalability issues. One optimization is to decouple graph propagation from feature learning and employ simple model structures to speed up computation [41, 22], which frees the GPU memory from storing entire graph data and reduces the memory footprint. Such methods typically integrate graph data management techniques such as Personalized PageRank [30] to calculate the graph representation used in the model. Another direction is easing node interdependence, which enables training on smaller batches and is achieved by neighbor sampling [18, 9], layer sampling [8, 16], and subgraph sampling [12, 46, 20]. Various sampling schemes have been applied to restrain the number of nodes contained in GNN learning pipelines and reduce computational overhead. Other algorithms are also utilized in simplifying graph propagation and learning in order to improve efficiency and efficacy, including diffusion [22, 4], self-attention [34, 33, 47], and quantization [15].

Unfortunately, such methods are nevertheless not efficient enough when applied to million-scale or even larger graphs. According to [36], the very recent state-of-the-art algorithm GBP [10] typically consumes more than 10^4 seconds solely for precomputation on the Papers100M graph (111M nodes, 1.6B edges, generated from MAG) to reach proper accuracy. In our experiments, the same model even exceeds the 192GB RAM bound on a single worker during processing. As shown in Fig. 1, the time and memory expense of current approaches increases rapidly to a prohibitive level when the graph scales up. Such cost caused by the limited scalability hinders their application in practice.



(a) Learning Time (b) RAM Usage

Fig. 1: Scalability of GNN models on different real-world datasets. **(a)** The model learning time increases with the scale of graph edges, where solid and dashed lines are transductive and inductive learning, respectively. **(b)** The model memory usage increases with the scale of graph nodes. Note that both axes are on a log scale.

Our Contributions. In this paper, we propose SCARA, a SCAlable gRaph neural network Algorithm with low time complexity and high scalability on very large datasets. On the theoretical side, the time complexity of SCARA for precomputation/training/inference matches the same sub-linear level with the state of the art, as shown in Table 1. On the practical side, to our knowledge, SCARA is the first GNN algorithm that can be applied to the billion-edge graph Papers100M with a precomputation time less than 13 seconds and complete training under a relatively strict memory limit.

Particularly, SCARA employs several feature-oriented optimizations. First, we observe that most current scalable methods repetitively compute the graph propagation information from the node-based dimension, which results in complexity at least proportional to the number of graph nodes. To address this issue, we design a Feature-Push method that realizes the information propagation from the feature vectors, which removes the linear dependency on the number of nodes in the complexity while maintaining the same precision of corresponding graph propagation values. Second, as we mainly process the feature vectors, we discover that there is significant room to reuse the computation results across different feature dimensions. Hence we propose the Feature-Reuse algorithm. Through combining calculation results, SCARA efficiently adopts several feature-based vector optimizations and prevents time-consuming repetitive propagation. By such designs, SCARA outperforms all leading competitors in our experiments in all 6 GNN learning tasks in regard to model convergence time, i.e., the sum of precomputation and training time, with highly efficient inference speed, significantly better memory overhead, and comparable or better accuracy.

In summary, we have made the following contributions[1]:

– We present the Feature-Push algorithm which propagates the graph information from the feature vectors with forward push and random walk. Our method achieves a sub-linear complexity for precomputation running time along with efficient model training and inference implemented in the mini-batch approach.

– We propose the Feature-Reuse mechanism, which utilizes the feature-oriented optimizations to further improve the efficiency of feature propagation while maintaining precision. The technique reduces the precomputation overhead of sole Feature-Push by approximately 4×.

[1] The source code and data used in the paper have been made available at: `https://github.com/gdmnl/SCARA-PPR`

– We conduct comprehensive experiments to evaluate the efficiency and effectiveness of the SCARA model on various datasets and against benchmark methods. Our model is time- and memory-efficient enough to process the billion-scale dataset Papers100M. It also achieves up to 800× faster in precomputation time than the current state of the art.

Comparison with the conference version [25]. We have made the following new contributions:
– We propose a novel Feature-Reuse algorithm, which is now formulated as an optimization problem and provided with an efficient feature-oriented solution for calculation integrating all features. Experiments in Section 4.5 demonstrate that the current approach is up to 4× faster and produces less approximation error.
– We empower the method with efficient parallel computation capabilities, mainly thanks to the Feature-Reuse algorithm conducting the majority of the resource-intensive computations in a feature-oriented manner. As shown in Section 4.4, parallelism brings over 10× speed-up compared with the former single-worker version.
– We enhance the theoretical analysis on feature PPR approximation precision for our methodology in Section 3, including detailed elaboration on the precision control strategies, and complete proofs of Lemma 2 and Lemma 3 for the updated Feature-Reuse on its precision guarantee.
– We update our experimental settings to incorporate parallelization and rerun all evaluations to align with these new settings. Furthermore, we present additional experiments and discussions on the impact of model parameters, parallel processing, and reuse schemes in Section 4.

Paper Organization. The rest of the paper is organized as follows. Section 2 introduces the preliminaries of GNN and relative scalable methods. We present a thorough analysis on the time and memory complexities of these models. Then we propose our SCARA framework in Section 3 based on the concept of feature PPR, introducing the Feature-Push and Feature-Reuse algorithms as well as the precision guarantees. Experimental results are presented in Section 4, including evaluations and discussions on efficacy, efficiency, and convergence. Finally, we summarize the paper in Section 5.

2 Preliminaries and Related Works

Notations. Consider a graph $G = \langle V, E\rangle$ with node set V and edge set E. We assume the graph is self-looped [21], that an edge is connected to the node itself for each node in V. Let $n = |V|$, $m = |E|$, and $d = m/n$. The graph connectivity is represented by the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, while the diagonal degree matrix is $\boldsymbol{D} \in \mathbb{R}^{n\times n}$. Following [10, 36], we normalize the adjacency matrix by $\boldsymbol{D}$ with convolution coefficient $r \in [0, 1]$ as $\tilde{\boldsymbol{A}}_{(r)} = \boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}$. For each node $v \in V$, denote the set of the out-neighbors by $\mathcal{N}(v) = \{u|(v, u) \in E\}$, and the out-degree of v by $d(v) = |\mathcal{N}(v)|$. Each v has an F-dimension feature vector $\boldsymbol{x}(v)$, which composes the matrix $\boldsymbol{X} \in \mathbb{R}^{n\times F}$.

A GNN recurrently computes the node representation matrix $\boldsymbol{H}^{(l)}$ as current state in the l-th layer. The model input feature matrix is $\boldsymbol{H}^{(0)} = \boldsymbol{X}$ in particular. For a conventional L-layer GCN [21], the $(l+1)$-th representation matrix $\boldsymbol{H}^{(l+1)}$ is updated as:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{1}$$

where $\boldsymbol{W}^{(l)}$ is the trainable weight matrix of the l-th layer, $\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{A}}_{(1/2)}$ is the normalized adjacency matrix, and $\sigma(\cdot)$ is the activation function such as ReLU or softmax. For analysis simplicity we keep the feature size F unchanged in all layers.

Summarized in Table 1, we present an analysis on the complexity bounds of GCN in Equation (1) to explain the restraints of its efficiency. One dominating part of the learning overhead is the training phase, where the model weights $\boldsymbol{W}^{(l)}$ are iteratively updated for I epochs and thence resource-intensive. For the L-layer GCN model training per epoch, it can be typically divided into two consecutive procedures of matrix multiplications: *Graph propagation* computes the product $\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}$, and is bounded by a complexity of $O(LmF)$ giving the adjacency matrix $\tilde{\boldsymbol{A}}$ with m entries and the propagation is conducted for L iterations. The overhead for the second procedure *feature transformation* by multiplying $\boldsymbol{W}^{(l)}$ is $O(LnF^2)$. As discovered by previous studies [12, 10], the dominating term is $O(LmF)$ when the graph is large, while the latter transformation can be accelerated by GPU computation. Hence, the full graph propagation becomes the scalability bottleneck.

In the inference phase, the model performs a similar forward inference, hence resulting in the same time complexity of $O(LmF + LnF^2)$. In terms of memory usage, the GCN typically takes $O(LnF + LF^2)$ space to store layer-wise node representations and weight matrices respectively.

Post-Propagation Model. As the graph propagation possesses the major computation overhead when the graph is scaled-up, a straightforward idea is to simplify this step and prevent it from being repetitively included in each layer. Such approaches are regarded as propagation decoupling models [49, 23, 40]. We further classify them into post- and pre-propagation variants based on the presence stage of propagation relative to feature transformation.

The post-propagation decoupling methods apply propagation only on the last model layer, enabling efficient and individual computation of the graph propagation matrix, as well as the fast and simple model training. The APPNP model [22] introduces the personalized PageRank (PPR) [30] algorithm in the propagation stage. The iterative graph propagation in the GCN updates is replaced by multiplying the PPR

Table 1: Time and memory complexities of scalable GNN models. Precomputation memory complexity indicates the usage of intermediate variables, while the training and inference memory refers to the GPU usage for storing and updating representation and weight matrices in each training iteration. The training and inference time complexities represent the forward-passing computation on the training and inference node set. Models including GCN and GraphSAINT do not have an explicit precomputation stage, which are marked as "–".

Model	Precomp. Mem.	Training Mem.	Inference Mem.	Precomp. Time	Training Time	Inference Time
GCN [21]	–	$O(LnF + LF^2)$	$O(LnF + LF^2)$	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
GraphSAINT [46]	–	$O(L^2bF + LF^2)$	$O(LnF + LF^2)$	–	$O(IL^2nF + ILnF^2)$	$O(LmF + LnF^2)$
GAS [16]	$O(LnF)$	$O(LdbF + LF^2)$	$O(LdbF + LF^2)$	$O(m + LnF)$	$O(ILmF + ILnF^2)$	$O(nF)$
APPNP [22]	$O(m)$	$O(LbF + LF^2 + db)$	$O(LbF + LF^2 + db)$	$O(m)$	$O(ITmF + ILnF^2)$	$O(TmF + LnF^2)$
PPRGo [5]	$O(n/r_{max})$	$O(LbF + LF^2 + Kb)$	$O(LbF + LF^2 + Kb)$	$O(m/r_{max})$	$O(IKnF + ILnF^2)$	$O(KnF + LnF^2)$
SGC [41]	$O(m)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(LmF)$	$O(ILnF^2)$	$O(LnF^2)$
GBP [10]	$O(nF)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(LF\sqrt{Lm\log(Ln)}/\epsilon)$	$O(ILnF^2)$	$O(LnF^2)$
SCARA (ours)	$O(nF)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(F\sqrt{m\log n}/\lambda)$	$O(ILnF^2)$	$O(LnF^2)$

matrix after the feature transformation layers:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-2, \tag{2}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\hat{\boldsymbol{\Pi}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = L-1, \tag{3}$$

where $\hat{\boldsymbol{\Pi}} = \sum_{l=0}^{L} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l$ is the PPR matrix.

In this design, the feature transformation benefit from the mini-batch scheme in both training and inference stages, hence reducing the demand for GPU memory. In Table 1, the batch size is b. Regarding computation speed, a T-round Power Iteration computation on the PPR matrix [30] leads to $O(TmF + LnF^2)$ time per epoch. The PPRGo model [5] further improves the efficiency of precomputing the PPR matrix $\boldsymbol{\Pi}$ by the Forward Push algorithm [2, 3] with an error threshold r_{max} and only records the top-K entries. However, it demands $O(n/r_{max})$ space to store the dense PPR matrix.

Pre-Propagation Model. Another line of research, namely the pre-propagation models such as SGC [41], chooses to propagate graph information in advance and encode it to the attributes matrix $\boldsymbol{X}$, forming an embedding matrix $\boldsymbol{P}$ that is utilized as the input feature to the neural network layers. In a nutshell, we summarize the model updates in the following scheme:

$$\boldsymbol{H}^{(0)} = \boldsymbol{P} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l_{(r)} \cdot \boldsymbol{X}, \tag{4}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{5}$$

where L_P denotes the depth of precomputed propagation and a_l is the layer-dependent diffusion weight.

The line of Equation (4) corresponds to the *precomputation section* and is calculated only once for each graph. The complexity of this stage is solely related to the precomputation techniques applied in the model. In SGC, the equation is given by an L-hop multiplication of $\boldsymbol{P} = \tilde{\boldsymbol{A}}^L\boldsymbol{X}$, taking $O(LmF)$ time. A recent work GBP [10] employs a PPR-based bidirectional propagation with $L_P = L$ and tunable a_l and r. Under an approximation of relative error ϵ, it improves precomputation complexity to $O(LF\sqrt{Lm\log(Ln)}/\epsilon)$ in the best case. It is notable that since GBP contains a node-based traverse scheme, it is sensitive to the scale of n in practice.

Equation (5) follows the neural network *feature transformation*, taking $\boldsymbol{P}$ as input feature. Compared to Equation (3), it completely removes the need for additional multiplication, hence both training and inference are reduced to $O(LnF^2)$. The simple GNN provides scalability in both resource-demanding training and frequently-queried inference, with the ease to employ techniques such as mini-batch training, parallel computation, and data augmentation.

Other Methods. There is a large scope of GNNs related to sampling techniques, which still possess the iterative propagation, but simplifies it by replacing the full-batch graph updates with selected nodes in mini-batches. The representatives are GraphSAGE [18] performing layer-wise sampling and GraphSAINT [46] exploiting multiple levels of information. Specifically, the efficiency-oriented variant GraphSAINT-RW incorporates L-hop random walk graph sampling, resulting in a total training complexity of $O(L^2nF + LnF^2)$. It is however not applicable in the full graph inference stage, causing the inference time and memory overheads to be identical to the vanilla GCN. GAS [16] samples layer-wise neighbors and consumes great memory for historical embedding. It has $O(LmF + LnF^2)$ training overhead, while the optimal inference complexity is benefited by the cached embedding.

In our experiments, we compare our GNN algorithm with the state-of-the-art scalable models from each of the aforementioned categories, to demonstrate the scalability and effectiveness of our algorithm.

3 SCARA Framework

We propose our SCARA framework composing Feature-Push and Feature-Reuse. The Feature-Push algorithm conducts graph propagation from the aspect of features,

while FEATURE-REUSE is a novel technique that reuses columns in the feature matrix. We also present analysis on the algorithmic complexity and precision guarantee to demonstrate the theoretical validity and effectiveness of SCARA.

3.1 Overview

To realize scalability in the network training and inference stage, and to better employ advanced Personalized PageRank (PPR) algorithms to optimize graph diffusion, we apply the backbone of propagation decoupling approach in our GNN design. Similar to previous pre-propagation models [41, 10], in precomputation stage we follow the idea of Equation (4) to compute the graph information $\boldsymbol{P}$ in advance together with the node attributes $\boldsymbol{X}$. Then, a simple yet effective feature transformation is conducted as given in Equation (5). We enhance the model structure by incorporating skip connections [22] and dense connections [10] in every intermediate layers.

Since the propagation stage is the complexity bottleneck as mentioned earlier, we focus on reducing its computation complexity. We derive Equation (4) in our propagation as:

$$\boldsymbol{P} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}_{(r)}^l \cdot \boldsymbol{X} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^l \boldsymbol{X}, \tag{6}$$

where α is the teleport probability as we set $a_l = \alpha(1-\alpha)^l$ to be associated with the form in the PPR calculation. Compared with APPNP and PPRGo, we adopt a generalized graph adjacency $\tilde{\boldsymbol{A}}_{(r)}$ with an adjustable convolution factor $r \in [0, 1]$ to fit different scales of graphs.

Our computation of Equation (6) is displayed in Algorithm 1 (FEATURE-PUSH) and explained in detail in Section 3.2. The highlight of FEATURE-PUSH is the application of propagating from features, which differs from prior works. In many real-world tasks, when a graph is scaled-up, its numbers of nodes (n) and edges (m) increase, but the node attributes dimension (F) usually remains unchanged. Thus, an algorithm with complexity mainly dependent on F enjoys better scalability than those dominated by n or m.

As the attribute matrix $\boldsymbol{X}$ is included in our computation, we then investigate how to fully utilize its implicit information to further accelerate our algorithm, which leads to the Algorithm 2 (FEATURE-REUSE). The motivation is to reduce the expensive iterative computation of $\boldsymbol{P}$ components by exploiting the previous results based on attribute vectors $\boldsymbol{x}$ on selected dimensions f. We apply a linear combination scheme with precision guarantee to lighten the constraints of Algorithm 1 while improving speed. We further describe this methodology in Section 3.3.

3.2 FEATURE-PUSH

Examining Equation (6), the embedding matrix $\boldsymbol{P}$ is the composition of graph diffusion matrix $\tilde{\boldsymbol{A}}_{(r)}$ and node attributes $\boldsymbol{X}$. Most scalable methods such as APPNP [22] and SGC [41] compute the propagation part separately from network training, resulting in a complexity at least proportional to edge size m. GBP [10] discusses a bidirectional propagation with both node-side random walk on $\boldsymbol{D}^{-1}\boldsymbol{A}$ and feature-side reverse push on $\boldsymbol{D}^{-r}\boldsymbol{X}$. Although the random walk step ensures precision guarantee, it requires long running time when not being accelerated by other methods [39, 38, 29].

We propose the FEATURE-PUSH approach that propagates graph information from the feature dimension, which is capable to utilize efficient single-source PPR algorithms through a simple but surprisingly effective transformation. Note that the graph propagation term in Equation (6) can be written as the following to rearrange the normalization order:

$$\tilde{\boldsymbol{A}}_{(r)}^l \cdot \boldsymbol{X} = \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^l \boldsymbol{X} = \boldsymbol{D}^{r-1}\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l \boldsymbol{D}^{1-r}\boldsymbol{X}. \tag{7}$$

Here, given the normalized features $\boldsymbol{D}^{1-r}\boldsymbol{X}$, single-source PPR algorithms can be alternated to efficiently propagate information with $\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l$, one feature vector each time, without doing the actual iterative matrix multiplications. In order to better derive FEATURE-PUSH, we borrow the Personalized PageRank (PPR) notations to describe our technique manipulating feature vectors. On a graph G, given a source node $s \in V$ and a target node $t \in V$, the PPR $\pi(s, t)$ represents the probability of a random walk with teleport factor $\alpha \in (0, 1)$ which starts at node s and stops at t. In general, *forward* PPR algorithms, often categorized as *single-source* PPR, start the computation from s, contrasted to *backward* or *reverse* alternatives that are developed from t [36].

When the PPR calculation is integrated with features, it shares similarities in forms but with a different interpretation. Consider the PPR problem with regard to nodes in a set $U \subseteq V$ as the source nodes. Let n_U be the size of set U. We call an n_U-dimension vector $\boldsymbol{x}$ with sum of elements $\|\boldsymbol{x}\|_1 = 1$ as a feature vector. In our context, the feature PPR $\pi(\boldsymbol{x}; t)$ represents the PPR for feature vector $\boldsymbol{x}$, and can be defined as the probability of the event that a random walk which starts at a node $s \in U$ with probability distribution $\boldsymbol{x}$ and stops at t. It can be derived from the definition that, each feature PPR $\pi(\boldsymbol{x}; t)$ can be interpreted as a generalized integration of a series of the common single-source PPR value $\pi(s, t)$ with the source node s being any arbitrary nodes in U. Hence the properties and operations of common PPR are still valid.

The notation can be extended to the matrix form when computing multiple features. Let F be the number of feature vector. The feature matrix is $\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F]$ of shape $n_U \times F$ and $\boldsymbol{x}_f$ $(1 \le f \le F)$ is the f-th column feature vector. Correspondingly, the embedding matrix is $\boldsymbol{P} =$

$[\boldsymbol{\pi}_1, \cdots, \boldsymbol{\pi}_F]$, where $\boldsymbol{\pi}_f = \boldsymbol{\pi}(\boldsymbol{x}_f)$ is the f-th column of PPR vector computed from feature $\boldsymbol{x}_f$, and is composed by $\boldsymbol{\pi}_f = \left(\pi(\boldsymbol{x}_f; t_1), \cdots, \pi(\boldsymbol{x}_f; t_{n_U})\right)^\top$ on all nodes. Calculating $\boldsymbol{P}$ from feature $\boldsymbol{X}$ is achieved by separately applying FEATURE-PUSH on each feature vector, which is exactly the implication of Equation (6). Now that the feature PPR is explained, we here look into its calculation. We define the problem of feature PPR approximation:

Definition 1 (Approximate Feature PPR) Given an absolute error bound $\lambda > 0$, a PPR threshold $0 < \delta < 1$, and a failure probability $0 < \phi < 1$, the approximate PPR query for feature vector $\boldsymbol{x}$ computes an estimation $\hat{\pi}(\boldsymbol{x}; t)$ for each $t \in U$ with $\pi(\boldsymbol{x}; t) > \delta$, such that with probability at least $1 - \phi$,

$$|\pi(\boldsymbol{x}; t) - \hat{\pi}(\boldsymbol{x}; t)| \leq \lambda. \tag{8}$$

Recognizing that GNNs require less precise propagation information to achieve proper performance [51, 32], the approximate feature PPR enables employing efficient computation based on forward PPR algorithms without loss in eventual model effectiveness [39, 42]. We employ a scalable algorithm FEATURE-PUSH to compute the embedding matrix combining Forward Push [3] and Random Walk techniques that both operate feature vectors. The algorithm makes use of both approaches, that random walk is accurate but less efficient, while forward push is fast with a loose precision guarantee. Algorithms exploiting such combination have been the state of the arts in various PPR benchmarks [39, 26]. We highlight that the differences between Algorithm 1 and [39, 26] are three-fold. First, the push starts from the feature vector, which can be seen as a generalized PPR operation taking probability distribution $\boldsymbol{x}$ into account. Unlike single source PPR that starts from only one node in the graph, the feature vector $\boldsymbol{x}$ is usually dense and hence requires specific processing. Second, the feature-based query facilitates subsequent transformation in Equation (7) and reusing in Equation (13). This design ensures that the computation result $\boldsymbol{\pi}$ satisfies the need of GNN propagation. Third, the FEATURE-PUSH design minimizes the need of additional storage and conducts most feature-wise operations in-place, which demonstrates excellent memory efficiency.

Algorithm 1 FEATURE-PUSH

Input: Graph G, node set U, feature vector $\boldsymbol{x}$, probability α, convolution factor r, push parameter β
Output: Approximate embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$

1 **for all** $u \in U$ **do**
2 $r'(\boldsymbol{x}; u) \leftarrow x(u) \cdot d(u)^{1-r}$
3 $r(\boldsymbol{x}; u) \leftarrow r'(\boldsymbol{x}; u) / \sum_{u \in U} r'(\boldsymbol{x}; u)$
4 $\hat{\pi}(\boldsymbol{x}; t) \leftarrow 0$ for all $t \in U$
5 **while** exist $u \in U$ such that $r(\boldsymbol{x}; u) > r_{max}/d(u)$ **do**
6 **for all** $v \in \mathcal{N}(u)$ **do**
7 $r(\boldsymbol{x}; v) \leftarrow r(\boldsymbol{x}; v) + (1-\alpha) \cdot r(\boldsymbol{x}; u)/d(u)$
8 $\hat{\pi}(\boldsymbol{x}; u) \leftarrow \hat{\pi}(\boldsymbol{x}; u) + \alpha \cdot r(\boldsymbol{x}; u)$
9 $r(\boldsymbol{x}; u) \leftarrow 0$
10 $r_{sum} \leftarrow \sum_{u \in U} r(\boldsymbol{x}; u), \quad N_W \leftarrow r_{sum}/\beta$
11 **for all** $u \in U$ such that $r(\boldsymbol{x}; u) \neq 0$ **do**
12 Perform $\frac{r(\boldsymbol{x};u)}{r_{sum}} \cdot N_W$ random walks from u
13 **for all** random walk stopping at t **do**
14 $\hat{\pi}(\boldsymbol{x}; t) \leftarrow \hat{\pi}(\boldsymbol{x}; t) + r_{sum}/N_W$
15 $\hat{\pi}(\boldsymbol{x}; t) \leftarrow \hat{\pi}(\boldsymbol{x}; t) \cdot d(t)^{r-1}$ for all $t \in U$
16 **return** $\hat{\boldsymbol{\pi}}(\boldsymbol{x}) \leftarrow \left(\hat{\pi}(\boldsymbol{x}; t_1), \cdots, \hat{\pi}(\boldsymbol{x}; t_{n_U})\right)^\top$

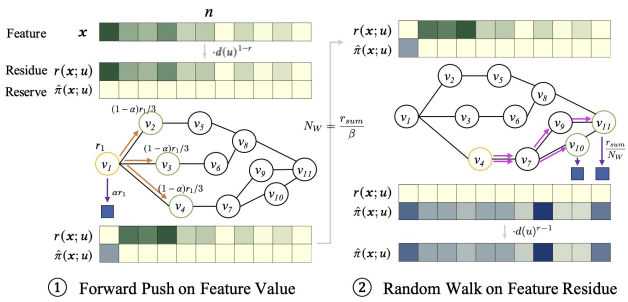


Fig. 2: Illustration of the FEATURE-PUSH process. The input feature vector is transformed into the residue vector through normalization. FEATURE-PUSH performs two consecutive steps, namely Forward Push on Feature Value and Random Walk on Feature Residue, to acquire the approximate feature PPR vector, which is output as the embedding after further normalization. The two steps are related by the push parameter β.

As shown in Algorithm 1, the FEATURE-PUSH algorithm outputs the approximation of embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$ for input feature $\boldsymbol{x}$. The node set U can either be the whole graph nodes V or a subset of V. Repeating it for F times with all features $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F$ produces all columns compositing the estimate of embedding matrix $\hat{\boldsymbol{P}}$. The algorithm first computes the approximation $\hat{\pi}(\boldsymbol{x}; t)$ for each node $t \in U$ through forward push (line 1-9 in Algorithm 1), then conducts compressed random walks to save computation (line 10-14). A running example is illustrated in Fig. 2. We analyze each step and their combination respectively.

Forward Push on Feature Value. Instead of calculating the PPR value $\pi(s, t)$, the forward push method in FEATURE-PUSH maintains a *reserve value* $\hat{\pi}(\boldsymbol{x}; t)$ directly for node $t \in U$ and feature $\boldsymbol{x}$ as the estimation of $\pi(\boldsymbol{x}; t)$. An auxiliary *residue value* $r(\boldsymbol{x}; t)$ is recorded as the intermediate result for each node-feature pair. The residue is initialized by the L_1-normalized feature vector $\boldsymbol{x}$, to convert node attributes to distributions in line with $\pi(\boldsymbol{x}; t)$ that stands for the probability with a sum of 1 for all nodes $t \in U$. The forward push algorithm subsequently updates the residue of target node t from the source node s to propagate the information. The threshold r_{max} controls the terminating condition so that the

process can stop early. Eventually, the forward push transfers α portion of node residue $r(\boldsymbol{x};t)$ into reserve value, while distributing the remaining $(1-\alpha)$ to the neighbors of s.

Random Walk on Feature Residue. Feature-Push then performs random walks with decay factor α to propagate the residue feature value. Compared with the pure random walk approach, Feature-Push only requires $\frac{r(\boldsymbol{x};t)}{r_{sum}} \cdot N_W$ number of walks per node with the same precision guarantee, benefiting from the Forward Push results. As presented in line 10, the total random walk number N_W is decided by the ratio r_{sum}/β, hence a sparser residue and lager parameter β result in less random walks required. The estimation of $\hat{\pi}(\boldsymbol{x};t)$ is eventually achieved by implementing the Monte-Carlo method [17, 38], and is updated according to the fraction of random walks terminating at t.

Combination and Normalization. The combination of forward push and random walk generates the approximate PPR matrix $\boldsymbol{\Pi}^{(l)} = \alpha(1-\alpha)^l \left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l$ for a certain l. To be aligned with the embedding matrix $\boldsymbol{P}^{(l)}$ in Equation (6), we apply the normalization by degree vector (lines 2 and 15 in Algorithm 1) to achieve the transformation in Equation (7). It is worth noting that Algorithm 1 is fully feature-oriented – it processes one feature vector at a time. Such scheme has several merits, with the first is that a series of vectorization techniques can be applied during processing each feature to accelerate computation. For space optimization, the feature vector $\boldsymbol{x}$ and result vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$ can be computed in-place and share the same memory, thus greatly reduces the overhead of storing such dense vector and in the mean time ensures memory locality.

Approximation Precision. To depict the combination between forward push and random walk processes, we define the push parameter β:

Definition 2 (Push Parameter) The push parameter β is the scale between the total left residual r_{sum} and the total number of sampled random walks N_W in Feature-Push.

The parameter β is named after its pivot role in determining the portion of forward push conducted as shown later in Lemma 4. It is the key parameter of Feature-Push, which balances absolute error guarantee and time complexity. Referencing the trade-off in [39], we set β to a specific value, namely standard push parameter $\beta_s = \frac{\lambda^2}{(2\lambda/3+2)\cdot\log(2/\phi)}$, to satisfy the guarantee of $\hat{\pi}(\boldsymbol{x};t)$ in Definition 1. In Algorithm 1, the forward push and random walk are combined in such form as line 14.

Derived from the single-source PPR analysis [3, 39], we state that our Feature-Push algorithm provides an unbiased estimation $\hat{\pi}(\boldsymbol{x};t)$ of the value $\pi(\boldsymbol{x};t)$ as the following lemma. By running Algorithm 1 feature by feature, the approximate calculation is also applicable to the PPR matrix containing multiple vectors:

Lemma 1 *Algorithm 1 produces an unbiased estimation $\hat{\pi}(\boldsymbol{x};t)$ of the value $\pi(\boldsymbol{x};t)$ satisfying Equation (8). Repeating it for F times produces an unbiased estimation $\hat{\boldsymbol{P}}$ of the embedding matrix $\boldsymbol{P}$.*

Parallel Computation. Since Algorithm 1 processes one feature vector at a time, and the execution of features is independent to each other, the acquisition on the estimation matrix $\hat{\boldsymbol{P}}$ can be safely parallelized to further celebrate efficiency. In implementation, each thread can simultaneously perform Algorithm 1 to compute the propagation from the feature vector $\boldsymbol{x}_f$ to the result PPR $\hat{\boldsymbol{\pi}}(\boldsymbol{x}_f)$, corresponding to the f-th column from matrix $\boldsymbol{X}$ to $\hat{\boldsymbol{P}}$. As stated previously, the computation is localized to a single column vector, hence performing parallel processing does not occur additional memory overhead.

3.3 Feature-Reuse

A key difference between the feature PPR and the classic single-source PPR is that, in single-source PPR, queries on nodes are orthogonal to each other, whereas in feature PPR there is similarity between different features. As a direct derivation from Equation (6), the feature PPR $\boldsymbol{P}$ is linearly related to the attribute $\boldsymbol{X}$. Hence, the feature-oriented calculation Algorithm 1 enables taking advantage of such property and utilizing computed values to estimate the PPR of another similar feature.

We propose Feature-Reuse algorithm that speeds up the feature PPR computation by leveraging and reusing the similarity between different feature vectors. We select a set of vectors as the base vectors from all features and compute their PPR values by Feature-Push. When querying the PPR value on a non-base feature vector, Feature-Reuse separates a segment of the vector that can be obtained by combining the base vectors, and estimate the PPR value of this segment directly with the PPR value of the base vectors without additional Feature-Push computation overhead.

We elaborate how to utilize the linearity of PPR values by a toy example. If we have the PPR values $\boldsymbol{\pi}(\boldsymbol{b})$ for base feature vector $\boldsymbol{b} = (0.5, 0.5)$, and need to compute the PPR vector for $\boldsymbol{x} = (0.4, 0.6)$, we can firstly decompose $\boldsymbol{x} = (0.4, 0.4) + (0, 0.2)$. We then acquire the PPR vector for $(0.4, 0.4)$ directly by $0.8\boldsymbol{\pi}(\boldsymbol{b})$, and just need to compute the PPR value of the residue $(0, 0.2)$. Intuitively, the latter PPR calculation is faster than directly processing the raw feature, thanks to the reduced dimension. We will later derive in Lemma 4 that the computation complexity is actually positively related to L_1 norm $\|\boldsymbol{x}\|_1$ of the residue vector.

To formulate the Feature-Reuse algorithm, we here derive it in the form of an optimization problem under our matrix notation. On the input side, the algorithm aims to

represent the feature matrix $\boldsymbol{X}$ by a partial of selected feature columns called base features. The number of base feature vectors is $F_B \ll F$ and they compose the base matrix $\boldsymbol{X}_B = \left[\boldsymbol{b}_1, \cdots, \boldsymbol{b}_{F_B}\right]$, $\boldsymbol{b}_f \in \boldsymbol{X}$. Then, the entire feature matrix $\boldsymbol{X}$ can be written as combinations of the bases:

$$\boldsymbol{X} = \boldsymbol{X}_B \cdot \boldsymbol{\Theta} + \boldsymbol{Z}, \tag{9}$$

where $\boldsymbol{\Theta}$ is the base coefficient matrix with shape $F_B \times F$, and $\boldsymbol{Z} = [\boldsymbol{z}_1, \cdots, \boldsymbol{z}_F]$ represents the left values in features. Equation (9) can be interpreted as a rank-F_B decomposition on raw matrix $\boldsymbol{X}$ plus a residue matrix.

To compute feature PPR, Feature-Push is applied to the column vectors of $\boldsymbol{X}_B$ and $\boldsymbol{Z}$ instead of $\boldsymbol{X}$. The feature PPR estimation on the two matrices are denoted as $\hat{\boldsymbol{P}}_B$ and $\hat{\boldsymbol{P}}_Z$, respectively. Corresponding to Equation (9), the approximate feature PPR on $\boldsymbol{X}$ can be acquired by the combinations as:

$$\check{\boldsymbol{P}} = \hat{\boldsymbol{P}}_B \cdot \boldsymbol{\Theta} + \hat{\boldsymbol{P}}_Z. \tag{10}$$

Now that to accelerate the Feature-Push calculation especially on $\boldsymbol{Z}$, we aim to sparsify the residue vector by reducing the L_1 norm of its column vectors $\|\boldsymbol{z}_f\|_1$. This is equivalent to minimizing the L_1 norm of matrix $\|\boldsymbol{Z}\|_1 = \sum_{f=1}^{F} \|\boldsymbol{z}_f\|_1$ while ensuring the low rank approximation $\boldsymbol{Y} = \boldsymbol{X}_B\boldsymbol{\Theta}$ is satisfied by selecting base features. Hence the overall optimization goal is:

$$\min\ \mathrm{rank}(\boldsymbol{Y}) + \eta \|\boldsymbol{Z}\|_1, \quad \text{s.t. } \boldsymbol{Y} + \boldsymbol{Z} = \boldsymbol{X}. \tag{11}$$

Equation (11) indicates that, Feature-Reuse actually seeks to decompose feature matrix $\boldsymbol{X}$ as the sum of a low rank component $\boldsymbol{Y}$ plus a sparse component $\boldsymbol{Z}$. Such optimization problem falls exactly the same as Robust Principal-Component Analysis (RPCA) [6, 7] when $\eta = \frac{1}{\sqrt{n}}$, which can be effectively solved by convex optimization methods such as alternating direction [27]. In general, [6] discovers that the problem can be transferred into a pair of convex problems when only one term in the derived form of Equation (11) is variable and a generic Lagrange multiplier method can be applied. Such algorithm requires only alternative matrix-wise operation and does not involve complex calculations, making it highly efficient to execute. When the iteration converges, the result matrices $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are guaranteed to be low-rank and sparse, respectively.

However, there are two major difficulties in directly exploiting the RPCA optimization for our reuse task. Examining Equation (11), its low rank matrix $\boldsymbol{Y}$ does not guarantee the decomposition of $\boldsymbol{X}_B\boldsymbol{\Theta}$ that includes base features $\boldsymbol{X}_B$ inherited from $\boldsymbol{X}$. Also, considering the scale of the feature matrix is as large as $O(nF)$, it is inefficient to employ the decomposition on the entire matrix. We hence propose several techniques to specifically address these issues and achieve our Feature-Reuse algorithm.

Algorithm 2 shows the pseudo code of Feature-Reuse that utilizes a few base features to efficiently compute the feature PPR on the entire matrix. In line 1-9, it first leverages RPCA iterations on a sampled portion of the feature matrix to finds out base features and corresponding combination coefficient. After concatenating the base feature and PPR matrices (line 10-12), it reuses these calculation results on the other features to form the approximate PPR matrix (line 13-17). We separately elaborate on these two phases.

Algorithm 2 Feature-Reuse

Input: Graph G, feature matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F]$, base size F_B, reuse parameter γ, error bound λ
Output: Approximate embedding matrix $\hat{\boldsymbol{P}}$

1 Sample feature matrix $\boldsymbol{X'}$ on node set $U' \subset U$
2 $\boldsymbol{Y}, \boldsymbol{Z}, \boldsymbol{E} \leftarrow 0,\ \mu \leftarrow 1/n_{U'}$
3 **while** $\|X' - Y - Z\|_1 > \lambda \|X'\|_1$ **do**
4 $\quad \boldsymbol{Z} \leftarrow \mathsf{Threshold}_{\mu}(\boldsymbol{X'} - \boldsymbol{Y} - \boldsymbol{E})$
5 $\quad \boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V} \leftarrow \mathsf{SVD}_{F_B}(\boldsymbol{X'} - \boldsymbol{Z} + \boldsymbol{E}),\ \boldsymbol{Y} \leftarrow \boldsymbol{USV}$
6 $\quad \boldsymbol{E} \leftarrow \boldsymbol{X'} - \boldsymbol{Y} - \boldsymbol{Z} + \mu\boldsymbol{E}$
7 $\psi_1, \cdots, \psi_{F_B} \leftarrow \arg\min_{1 \le \psi_i \le F} \sum_{f=\psi_1}^{\psi_{F_B}} \|\boldsymbol{z}_f\|_1$
8 $\boldsymbol{X}_B \leftarrow [\boldsymbol{x}_{\psi_1}, \cdots, \boldsymbol{x}_{\psi_{F_B}}],\ \boldsymbol{V}_B \leftarrow [\boldsymbol{v}_{\psi_1}, \cdots, \boldsymbol{v}_{\psi_{F_B}}]$
9 $\boldsymbol{\Theta} \leftarrow \boldsymbol{V}_B^{-1}\boldsymbol{V},\ \beta_s \leftarrow \frac{\lambda^2}{(2\lambda/3+2)\cdot\log(2n)}$
10 **for** i from 1 to F_B **do** ▷ *[in parallel]*
11 $\quad \hat{\boldsymbol{\pi}}_i \leftarrow$ Apply Alg. 1 on $\boldsymbol{b}_i$ with $\beta_B = \gamma\beta_s$
12 $\hat{\boldsymbol{P}}_B \leftarrow [\hat{\boldsymbol{\pi}}_1, \cdots, \hat{\boldsymbol{\pi}}_{F_B}]$
13 **for** f from 1 to F **do** ▷ *[in parallel]*
14 $\quad \boldsymbol{\theta}_f \leftarrow \mathrm{col}_f\,\boldsymbol{\Theta},\ \boldsymbol{z}_f \leftarrow \boldsymbol{x}_f - \boldsymbol{X}_B\boldsymbol{\theta}_f$
15 $\quad \theta_{sum} = \sum_{i=1}^{F_B} \theta_{fi}$
16 $\quad \hat{\boldsymbol{\pi}}_f \leftarrow$ Apply Alg. 1 on $\boldsymbol{z}_f$ with $\beta_Z = (1 - \gamma\theta_{sum})\,\beta_s$
17 $\quad \check{\boldsymbol{\pi}}_f \leftarrow \hat{\boldsymbol{\pi}}_f + \hat{\boldsymbol{P}}_B\boldsymbol{\theta}_f$
18 **return** $\check{\boldsymbol{P}} = [\check{\boldsymbol{\pi}}_1, \cdots, \check{\boldsymbol{\pi}}_F]$

Base Selection on Matrix Portion. Feature-Reuse first optimizes the rank-F_B and sparse components from feature matrix. Line line 3-6 in Algorithm 2 corresponds to the RPCA iterative solution [6], where $\mathsf{Threshold}_{\tau}(x) = \mathrm{sgn}(x)\max(|x| - \tau, 0)$ is shrinkage operation that zeros elements with absolute value smaller than threshold τ, and $\mathsf{SVD}_k(\cdot)$ is rank-k truncated singular value decomposition (truncated SVD). The decomposition iteration is applied to a portion of feature matrix X', containing only a subset of nodes. Studies show that the sampling size $n_{U'}$ can be as small as $O(F^2)$ while preserving the precision of RPCA decomposition [14]. Hence the complexity of such base selection scheme can be bounded by $O(n_{U'}FF_B)$, which is free from the scale of the whole graph.

When the decomposition components $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are computed from $\boldsymbol{X'}$, we utilize them to estimate the feature reuse coefficient on the entire matrix. We first select the top-F_B indices ψ from all features with minimum decomposition error of respective residue vector $\boldsymbol{z_f}$, i.e. columns of the sparse component $\boldsymbol{Z}$. Features at these indices are hence regarded as base features $\boldsymbol{b}_i = \boldsymbol{x}_{\psi_i}$. Meanwhile, the coefficient matrix Θ is computed from the low rank components corresponding to

the selected indices. This is because with neglectable approximation errors, there is $\boldsymbol{X}_B = \boldsymbol{USV}_B$ for bases and $\boldsymbol{Y} = \boldsymbol{X}_B\boldsymbol{\Theta}$ for all features. Then in line 10-12, FEATURE-PUSH is invoked to acquire the feature PPR $\hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta_B)$ with input vector $\boldsymbol{b}_i$ and push parameter β_B. The calculation results are stored to $\hat{\boldsymbol{P}}_B$ as Equation (10) for further reuse in the following phase.

Calculation Reuse on Sparse Residue. Algorithm 2 then computes the approximate values of the rest features (line 13-17). For feature f, the f-th column vector $\boldsymbol{\theta}_f$ of $\boldsymbol{\Theta}$ serves as the reuse coefficient of each bases. According to Equation (9), values in the vector $\boldsymbol{x}_f$ that can be represented by base features are removed, and the residue vector is $\boldsymbol{z}_f$, which is sparse as RPCA optimizes. We compute the feature PPR $\hat{\boldsymbol{\pi}}(\boldsymbol{z}_f, \beta_Z)$ of such sparse residue by FEATURE-PUSH. The push parameter β_Z is dependent on the particular reuse state of coefficient $\boldsymbol{\theta}_f$. Finally, the feature PPR $\check{\boldsymbol{\pi}}(\boldsymbol{x}_f)$ on raw feature $\boldsymbol{x}_f$ can be constituted as line 17, reusing the PPR computation results of base features.

Approximation Precision. In Algorithm 2, the result PPR of a base vector $\hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta_B)$ is directly computed by FEATURE-PUSH in line 11 and has its accuracy guarantee according to Lemma 1. However, the PPR of non-base features is from the combination in line 17. How to assure that such approximation still satisfies the precision guarantee in Definition 1? We demonstrate that the precision can be controlled by setting proper value to the push parameters β_B and β_Z when calling FEATURE-PUSH in line 11 and line 16.

We first write the reuse combination Equation (9) and Equation (10) in our vector notation for a feature $\boldsymbol{x}_f$. For simplicity we omit the subscript f:

$$\boldsymbol{x} = \sum_{i=1}^{F_B} \theta_i \cdot \boldsymbol{b}_i + \boldsymbol{z}, \tag{12}$$

$$\check{\boldsymbol{\pi}}(\boldsymbol{x}) = \sum_{i=1}^{F_B} \theta_i \cdot \hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta_B) + \hat{\boldsymbol{\pi}}(\boldsymbol{z}, \beta_Z). \tag{13}$$

The following lemma depicts the precision constraint of $\check{\boldsymbol{\pi}}(\boldsymbol{x})$ in Equation (13).

Lemma 2 *Given a feature vector $\boldsymbol{x}$, the ground truth of PPR vector is $\boldsymbol{\pi}(\boldsymbol{x})$, and the estimation output by Equation (13) is $\check{\boldsymbol{\pi}}(\boldsymbol{x})$. For any respective element $\pi(\boldsymbol{x}; t)$ and $\check{\pi}(\boldsymbol{x}; t)$, $|\pi(\boldsymbol{x}; t) - \check{\pi}(\boldsymbol{x}; t)| \leq \lambda$ holds with probability at least $1 - \phi$, for β_Z such that $\beta_Z > \beta_B$ and*

$$\beta_Z \leq \frac{\lambda^2/\log(2/\phi) - 2\sum_{i=1}^{F_B} \theta_i \beta_B}{2\lambda/3 + 2}. \tag{14}$$

Proof. Similar to the theory in [30], feature PPR can also be interpreted as the solution of the following linear system:

$$\boldsymbol{\pi}(\boldsymbol{x}) = \alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{AD}^{-1}\pi(\boldsymbol{x}),$$

which can be transformed to

$$(\boldsymbol{I} - (1-\alpha)\boldsymbol{AD}^{-1})\boldsymbol{\pi}(\boldsymbol{x}) = \alpha\boldsymbol{x}.$$

Denote non-singular matrix $\boldsymbol{C} = \boldsymbol{I} - (1-\alpha)\boldsymbol{AD}^{-1}$. Then

$$\boldsymbol{\pi}(\boldsymbol{x}) = \alpha\boldsymbol{C}^{-1}\boldsymbol{x}.$$

The above equation indicates that feature PPR satisfies the associative law, which means

$$\theta\boldsymbol{\pi}(\boldsymbol{x}) = \boldsymbol{\pi}(\theta\boldsymbol{x}), \quad \boldsymbol{\pi}(\boldsymbol{x}_1) + \boldsymbol{\pi}(\boldsymbol{x}_2) = \boldsymbol{\pi}(\boldsymbol{x}_1 + \boldsymbol{x}_2).$$

According to the associative law, the combination PPR $\check{\boldsymbol{\pi}}(\boldsymbol{x})$ expressed in Equation (13) satisfies

$$\begin{aligned}\mathbb{E}[\check{\boldsymbol{\pi}}(\boldsymbol{x})] &= \sum_{i=0}^{F_B} \theta_i \cdot \mathbb{E}[\hat{\pi}(\boldsymbol{b}_i, \beta_B)] + \mathbb{E}[\hat{\pi}(\boldsymbol{z}, \beta_Z)] \\ &= \sum_{i=0}^{F_B} \theta_i \boldsymbol{\pi}(\boldsymbol{b}_i) + \hat{\boldsymbol{\pi}}(\boldsymbol{z}) = \boldsymbol{\pi}(\sum_{i=0}^{F_B} \theta_i \boldsymbol{b}_i + \boldsymbol{z}) = \boldsymbol{\pi}(\boldsymbol{x}).\end{aligned}$$

Therefore $\check{\boldsymbol{\pi}}(\boldsymbol{x})$ is an unbiased estimation of $\boldsymbol{\pi}(\boldsymbol{x})$. For each base $\boldsymbol{b}_i$, we compute $\hat{\pi}(\boldsymbol{b}_i, \beta_B)$ with Algorithm 1. In each such computation of Algorithm 1, the left residue on each node v before sampling random walks at line 10 is $r(\boldsymbol{b}_i; v)$, the total left residue is $r_{sum}(\boldsymbol{b}_i)$, and $N_W(\boldsymbol{b}_i) = r_{sum}(\boldsymbol{b}_i)/\beta_B$ is the number of random walks sampled.

As each base PPR is computed independently, combining the PPR vectors by $\sum_{i=0}^{F_B} \theta_i \hat{\pi}(\boldsymbol{b}_i, \beta_B)$ is equivalent to push a vector $\theta_i \boldsymbol{b}_i$ with the same pattern of the computing process of $\hat{\pi}(\boldsymbol{b}_i, \beta_B)$, and then sample $N_W(\boldsymbol{b}_i)$ random walks on the remaining residues of $\theta_i r_{sum}(\boldsymbol{b}_i)$ in total.

For a such computing process on $\theta_i \boldsymbol{b}_i$, consider the $N_W(\boldsymbol{b}_i)$ random walks it generate from all nodes. Let the random variable $X_j(\boldsymbol{b}_i; t) = 1$ if the j-th random walk terminates at t, and otherwise be $X_j(\boldsymbol{b}_i; t) = 0$. Associating with the single-source PPR $\pi(v, t)$, we have

$$\mathbb{E}\left[\sum_{j=0}^{N_W(\boldsymbol{b}_i)} \frac{r_{sum}(\boldsymbol{b}_i)}{N_W(\boldsymbol{b}_i)} X_j(\boldsymbol{b}_i; t)\right] = \sum_{v \in V} r(\boldsymbol{b}_i; v) \cdot \pi(v, t). \tag{15}$$

Consider the summation of all base vectors,

$$\mathbb{E}\left[\sum_{i=0}^{F_B} \sum_{j=0}^{N_W(\boldsymbol{b}_i)} \frac{\theta_i r_{sum}(\boldsymbol{b}_i)}{N_W(\boldsymbol{b}_i)} X_j(\boldsymbol{b}_i; t)\right] = \sum_{i=0}^{F_B} \sum_{v \in V} \theta_i r(\boldsymbol{b}_i; v) \cdot \pi(v, t).$$

As we have the PPR estimation expressed in the form of combination of residue and random walk values:

$$\begin{aligned}\check{\pi}(\boldsymbol{x}; t) = &\sum_{i=0}^{F_B} \theta_i r(\boldsymbol{b}_i; t) + r(\boldsymbol{z}; t) \\ &+ \sum_{i=0}^{F_B} \sum_{j=0}^{N_W(\boldsymbol{b}_i)} \frac{\theta_i r_{sum}(\boldsymbol{b}_i)}{N_W(\boldsymbol{b}_i)} X_j(\boldsymbol{b}_i; t) + \frac{r_{sum}(\boldsymbol{z})}{N_W(\boldsymbol{z})} X_j(\boldsymbol{z}; t).\end{aligned}$$

By referring to Lemma 3.2 in [38], we can further acquire the precision guarantee of the PPR as:

$$\Pr[|\check{\pi}(\boldsymbol{x};t)-\pi(\boldsymbol{x};t)|>\lambda]\leq 2\cdot\exp(-\frac{\lambda^2 N_{sum}}{2\nu+2a\lambda/3}), \quad (16)$$

where the number of walks $N_{sum}=N_W(\boldsymbol{z})+\sum_{i=0}^{F_B}N_W(\boldsymbol{b}_i)$, $a=N_{sum}\cdot\max\{\frac{\theta_1 r_{sum}(b_1)}{N_W(b_1)},\cdots,\frac{\theta_{F_B} r_{sum}(b_{F_B})}{N_W(b_{F_B})},\frac{r_{sum}(\boldsymbol{z})}{N_W(\boldsymbol{z})}\}$, and

$$\begin{aligned}\nu=&\frac{1}{N_{sum}}\sum_{i=0}^{F_B}\sum_{j=0}^{N_W(\boldsymbol{b}_i)}\left(\frac{\theta_i r_{sum}(\boldsymbol{b}_i)N_{sum}}{N_W(\boldsymbol{b}_i)}\right)^2\mathbb{E}[X_j(\boldsymbol{b}_i;t)]\\&+\frac{1}{N_{sum}}\sum_{j=0}^{N_W(\boldsymbol{z})}\left(\frac{r_{sum}(\boldsymbol{z})N_{sum}}{N_W(\boldsymbol{z})}\right)^2\mathbb{E}[X_j(\boldsymbol{z};t)]. \quad (17)\end{aligned}$$

Recall that $\beta_Z<\beta_B$, therefore $\frac{r_{sum}(\boldsymbol{z})N_{sum}}{N_W(\boldsymbol{z})}>\frac{r_{sum}(\boldsymbol{b_i})N_{sum}}{N_W(\boldsymbol{b_i})}$ holds for any $\boldsymbol{b_i}$, thence $a=\frac{r_{sum}(\boldsymbol{z})N_{sum}}{N_W(\boldsymbol{z})}$.

To simplify the expression of ν, we substitute Equation (15) into Equation (17) as:

$$\begin{aligned}\nu=&\frac{1}{N_{sum}}\sum_{i=0}^{F_B}\frac{\theta_i^2 r_{sum}(\boldsymbol{b}_i)N_{sum}^2}{N_W(\boldsymbol{b}_i)}\cdot\sum_{v\in V}r(\boldsymbol{b}_i;v)\cdot\pi(v,t)\\&+\frac{1}{N_{sum}}\frac{r_{sum}(\boldsymbol{z})N_{sum}^2}{N_W(\boldsymbol{z})}\cdot\sum_{v\in V}r(\boldsymbol{z};v)\cdot\pi(v,t)\\\leq&\sum_{i=0}^{F_B}\theta_i^2\beta_B N_{sum}+\beta_Z N_{sum}.\end{aligned}$$

The last inequality is because of Definition 2, where the push coefficients are the scales as $\beta_B=r_{sum}(\boldsymbol{b}_i)/N_W(\boldsymbol{b}_i)$, $\beta_Z=r_{sum}(\boldsymbol{z})/N_W(\boldsymbol{z})$. With the expressions on a and ν, we are able to derive Equation (16) as:

$$\Pr[|\check{\pi}(\boldsymbol{x};t)-\pi(\boldsymbol{x};t)|>\lambda]\leq 2\cdot\exp(-\frac{\lambda^2}{2\sum_{i=0}^{F_B}\theta_i^2\beta_B+2\beta_Z+2\beta_Z\lambda/3}).$$

By setting the value of β_Z

$$\beta_Z\leq\frac{\lambda^2/\log(2/\phi)-2\sum_{i=0}^{F_B}\beta_B\theta_i}{2\lambda/3+2},$$

we hence prove that

$$\Pr[|\check{\pi}(\boldsymbol{x};t)-\pi(\boldsymbol{x};t)|>\lambda]\leq\phi \qquad \square$$

Lemma 2 draws to the conclusion that, when choosing a smaller push parameter β_B for base vectors, the parameter β_Z can be larger and reduce the cost of PPR computation on most feature vectors. Hence we are particular interested in the upper bound of β_Z and set the actual value close to it. As Equation (14) suggests, if β_B are the same for all base Feature-Push, then the upper bound of β_Z is dependent on the sum of reuse coefficients $\theta_{sum}=\sum_{i=1}^{F_B}\theta_i$.

Based on Lemma 2, in Feature-Reuse algorithm we propose the reuse parameter γ as the indicator of the balance between base push parameter β_B and the one on residue β_Z. The following lemma states that by setting $\beta_B=\gamma\beta_s, \beta_Z=(1-\gamma\theta_{sum})\beta_s$ as in Algorithm 2, it satisfies the precision guarantee in Definition 1:

Lemma 3 *Given a feature set $\boldsymbol{X}$, for any feature vector $\boldsymbol{x}_f\in\boldsymbol{X}$, Algorithm 2 returns an approximate PPR vector $\check{\boldsymbol{\pi}}(\boldsymbol{x}_f)$, that any of its elements $\check{\pi}(\boldsymbol{x}_f;t)$ satisfies Equation (8) with at least $1-\phi$ probability.*

Proof. In Feature-Reuse, $\theta_{sum}=\sum_{i=1}^{F_B}\theta_i$ denotes the proportion of the feature vector $\boldsymbol{x}$ computed by the base vectors, and the L_1 length of the remaining part is $1-\theta_{sum}$. Then β_Z satisfies:

$$\begin{aligned}\beta_Z=&\frac{(1-\gamma\theta_{sum})\lambda^2}{\log(2/\phi)\cdot(2\lambda/3+2)}\leq\frac{\lambda^2/\log(2/\phi)}{2\lambda/3+2}-\sum_{i=1}^{F_B}\beta_B\theta_i\\\leq&\frac{\lambda^2/\log(2/\phi)-2\sum_{i=1}^{F_B}\beta_B\theta_i}{2\lambda/3+2}.\end{aligned}$$

Therefore, parameters β_B for base vectors and β_Z for remaining vectors satisfy Equation (14). According to Lemma 2 this lemma follows. $\square$

Parallel Computation. The parallelism of Feature-Reuse is based on that of Feature-Push. Since it is fully feature-oriented, each feature can still be computed individually. For the bulk of the loops in Algorithm 2, i.e. line 10 and line 13 containing PPR calculations, the processing can be parallelized.

3.4 Complexity Analysis

We then develop theoretical analysis on the time and memory complexity of SCARA. For a single run of Algorithm 1, we have the following lemma:

Lemma 4 *When the input vector is $\boldsymbol{x}$, the time complexity of Feature-Push is bounded by $O(\sqrt{\frac{m\|\boldsymbol{x}\|_1}{\beta}})$.*

Proof. We analyze the two parts of Algorithm 1 separately. The forward push with early termination threshold r_{max} runs in $O(\|\boldsymbol{x}\|_1/r_{max})$ as it iteratively propagates the residue value in the vector [3]. For random walks on feature residue, we employ the complexity derived by [39] as $O(m\cdot r_{max}/\beta)$. Hence the overall running time of one query in Algorithm 1 is bounded by $O\left(\frac{\|\boldsymbol{x}\|_1}{r_{max}}+r_{max}\cdot\frac{m}{\beta}\right)$. By applying Lagrange multipliers, the complexity is minimized when selecting $r_{max}=\sqrt{\frac{\beta\|\boldsymbol{x}\|_1}{m}}$, and the balanced complexity is $O(\sqrt{\frac{m\|\boldsymbol{x}\|_1}{\beta}})$. $\square$

Utilizing Lemma 4, the time complexity of computing one feature PPR $\hat{\pi}(\boldsymbol{x}, \beta)$ with Algorithm 1 can be bounded by $O(\sqrt{m}\|\boldsymbol{x}\|_1/\beta)$. To get PPR value with absolute error guarantee of λ, Algorithm 1 requires a push parameter $\beta_s = \frac{\lambda^2/\log(2/\phi)}{2\lambda/3+2}$. Then without Feature-Reuse, the time complexity for computing PPR value for each normalized feature vector is bounded by $O(\sqrt{m/\beta_s})$.

When Feature-Reuse applies, let $\theta_{sum} = \sum_{i=1}^{F_B} \theta_i$ denote the proportion of a feature $\boldsymbol{x}_f$ computed by base vectors, and the L_1 length of the rest $\boldsymbol{x}'$ is $1 - \theta_{sum}$. In Algorithm 2, we compute the remaining part with push parameter of $(1 - \gamma\theta_{sum})\beta_s$, where $0 < \gamma \leq 1$. Recalling that the L_1 length of the feature vector is reduced by θ_{sum} with Feature-Reuse, we derive the time complexity of Feature-Reuse on $\boldsymbol{x}$ is $O\left(\sqrt{\frac{m(1-\theta_{sum})}{\beta_s(1-\gamma\theta_{sum})}}\right)$, which is $\sqrt{\frac{1-\theta_{sum}}{1-\gamma\theta_{sum}}}$ times smaller than those without Feature-Reuse.

For example, if we compute $\theta_{sum} = 1/2$ for a vector $\boldsymbol{x}_f$ with the base vectors, and set $\gamma = 1/4$, then the complexity of computing the PPR for $\boldsymbol{x}_f$ is $O(\sqrt{4m/7\beta_s})$, which is substantially better than the consumption without Feature-Reuse $O(\sqrt{m/\beta_s})$. The overhead of each base vector is $O(\sqrt{4m/\beta_s})$, which is only twice slower than the original complexity. As we select only a few base vectors, the additional overhead produced by computing base vectors is neglectable compared with the acceleration gained.

When Feature-Reuse applies, the complexity of computing a feature vector is not worse than the complexity without Feature-Reuse, and is equivalent to the latter only when $\theta_{sum} = 0$ (i.e. the feature vector is completely orthogonal with the base vectors). Therefore in the worst case, the complexity of SCARA on feature matrix $\boldsymbol{X}$ is equivalent to repeating F queries of Algorithm 1. By setting $\phi = 1/n$, we can derive the time overhead of SCARA precomputation as shown in Table 1. For the complexity of memory, the usage of a single-query Feature-Push can be denoted as $O(n)$. Hence the precomputation complexity of SCARA is given by the following theorem:

Theorem 1 *Time complexity of SCARA precomputation is bounded by* $O\left(F\sqrt{m\log n}/\lambda\right)$. *Memory complexity is* $O(nF)$.

4 Experimental Evaluation

We implement the SCARA model and evaluate its performance by experiments in the aspects of both efficacy and scalability. From efficacy perspective, we compare the SCARA performance with other scalable GNN competitors under similar parameter settings. To demonstrate the scalability of our model, we further investigate its time and memory overhead with these benchmarks.

4.1 Experiment Setting

Datasets. We adopt benchmark datasets of different graph properties, feature dimensions, and data splitting for large-scale node classification tasks. We present the dataset statistics in Table 2. Among the datasets, PPI, Yelp, and Amazon are for *inductive* learning, where the training and testing graphs are different and require separate graph precomputation and propagation. The given original node splittings are in Table 2. The learning tasks on the other datasets are *transductive* and are performed on the same graph structure. For a dataset with N_c target classes, we refer to convention in [21, 5] to randomly sample two sets of $20N_c$ and $200N_c$ nodes for training and validation, respectively, and the rest labeled nodes in the graph as the testing set.

Metrics. Predictions on datasets PPI, Yelp, and MAG are multi-label classification having multiple targets for each node. The other tasks are multi-class with only one target class per node. We uniformly utilize micro F1-score to assess the model prediction performance. For efficiency metrics, we record the precomputation, training, and inference time of each model. We also measure the peak RAM memory in the whole process, as the GPU memory is mainly determined by training batch size and less relevant. The evaluation is conducted on a machine with Ubuntu 20 operating system, with 192GB RAM, two 28-core Intel Xeon CPUs (2.2GHz), and an NVIDIA RTX A5000 GPU (24GB memory). The implementation is by PyTorch and C++.

Baseline Models. We select the state-of-the-art models of different scalable GNN methods analyzed in Section 2 as our baselines. GraphSAINT-RW [46] and GAS [16] are repre-

Table 2: Dataset statistics and parameters. "Split" is the percentage of nodes in training/validation/testing set. "(i)" and "(t)" stand for inductive and transductive tasks. "(m)" and "(s)" stand for multiple and single target classifications.

Dataset	Nodes n	Edges m	Features F	Classes N_c	Split	Probability α	Convolution r	Common
PPI [18]	56,944	818,716	50	121 (m)	0.79/0.11/0.10 (i)	0.3	0.0	$\lambda = 1 \times 10^{-4}$ $F_B = 0.02F$ $\gamma = 0.2$
Yelp [46]	716,847	6,977,410	300	100 (m)	0.75/0.10/0.15 (i)	0.9	0.3	
Reddit [18]	232,965	114,615,892	602	41 (s)	0.01/0.04/0.96 (t)	0.5	0.5	
Amazon [12]	2,400,608	123,718,024	100	47 (s)	0.70/0.15/0.15 (i)	0.2	0.2	
MAG [44]	27,394,820	366,143,207	200	100 (m)	0.01/0.01/0.99 (t)	0.5	0.5	
Papers100M [19]	111,059,956	1,615,685,872	128	172 (s)	0.78/0.08/0.14 (t)	0.5	0.5	

Table 3: Average results of SCARA and baselines on large-scale datasets for transductive and inductive learning. "Learn" and "Infer" columns are the learning (sum of precomputation and training) and inference time (s), respectively. "Mem." is the peak RAM memory (GB). "F1" is the micro F1-score (%) on testing sets. "–" in rows with "OOM" stands for out of memory error in the settings. "–" in the GraphSAINT "Pre." column implies that the model does not have an explicit precomputation stage. The respective models of first and second best performance in "Learn", "Infer", "Mem.", and "F1" columns are marked in **bold** and underlined fonts.

Transductive	Reddit						MAG						Papers100M					
	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1
GraphSAINT	14.4	(–	14.4)	166.2	13.7	41.6 ±4.8	–	–	–	–	OOM	–	–	–	–	–	OOM	–
GAS	1200	(49.6 +	1151)	**2.2**	14.0	38.2 ±0.3	–	–	–	–	OOM	–	–	–	–	–	OOM	–
PPRGo	79.4	(62.3 +	17.1)	29.1	9.4	41.5 ±2.3	711	(451 +	259)	85240	130	17.0 ±1.5	–	–	–	–	OOM	–
GBP	138	(124 +	13.7)	13.5	7.9	38.8 ±0.3	663	(569 +	94.4)	1452	173	34.8 ±0.1	–	–	–	–	OOM	–
SCARA (ours)	**13.9**	(0.07 +	13.8)	10.7	**5.6**	**44.1 ±0.4**	**139**	(11.6 +	127)	**1208**	**67.7**	**34.9 ±0.3**	**1346**	(12.7 +	1333)	**4.7**	**71.4**	**48.9 ±0.8**

Inductive	PPI						Yelp						Amazon					
	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1
GraphSAINT	297	(–	297)	8.0	13.7	89.1 ±0.3	1093	(–	1093)	104	55.2	**65.0 ±0.0**	1890	(–	1890)	515	165	81.9 ±0.0
GAS	629	(0.8 +	628)	5.6	10.0	**99.4 ±0.0**	4863	(18.6 +	4844)	45.6	48.0	57.2 ±0.5	20549	(96.3 +	20453)	212	130	76.3 ±0.3
PPRGo	334	(7.4 +	326)	0.8	7.0	48.3 ±0.9	1310	(6.3 +	1304)	18.3	9.9	26.3 ±0.5	2560	(95.6 +	2464)	62.0	28.6	77.2 ±1.6
GBP	60.1	(2.3 +	57.8)	0.2	5.3	99.2 ±0.1	159	(31.2 +	127)	**1.9**	13.7	61.6 ±0.1	1181	(84.9 +	1096)	**4.9**	18.5	**88.3 ±0.1**
SCARA (ours)	**39.9**	(0.05 +	39.8)	**0.2**	**5.2**	99.2 ±0.0	**137**	(0.2 +	136)	2.3	**6.4**	62.9 ±0.1	**1132**	(0.5 +	1132)	5.0	**6.6**	85.6 ±0.0

* The results are in 32-thread parallel executions and hence different from those in Table 3 of [25].

sentative of different sampling-based algorithms. For post- and pre-propagation decoupling approaches, we respectively employ the most advanced PPRGo [5] and GBP [10]. For a fair comparison, we mostly retain the implementations and settings from original papers and source codes. We uniformly apply the same 32-thread parallel executions, which is a common setting in practical application, for evaluations on all models unless specially mentioned.

Hyperparameters. For the neural network architecture, we set the layer depth $L = 4$, layer width $W = 2048$ and $W = 128$ for inductive and transductive tasks, respectively, to be aligned with optimal baseline results in [10]. In model optimization, we utilize Adam optimizer with a learning rate of 0.005. Training is employed in the mini-batch manner when applicable, with respective batch size 2048 and 64 for inductive and transductive learning. We train the model for a maximum of 1000 epochs with early stopping and acquire the best model weights based on validation. Propagation-related parameters including PPR teleport probability α, convolution coefficient r, push parameter λ, and Feature-Reuse parameters including base size F_B and reuse parameter γ are presented in Table 2 per dataset. We further analyze the settings of these parameters in Section 4.3.

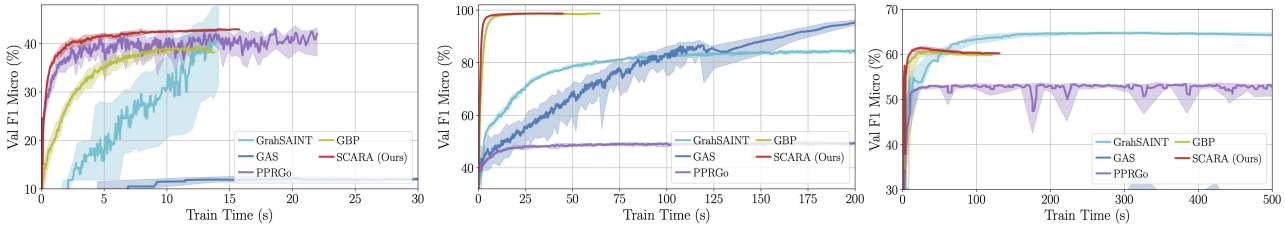
4.2 Performance Comparison

We evaluate the performance of SCARA and baselines in terms of both effectiveness and efficiency. Table 3 shows the average results of repetitive experiments on 6 large datasets, including the assessments on accuracy, memory, and the running time for different phases. Among them the key metric is learning time, which is summed up by precomputation and training times and presents the efficiency through the information retrieving process to acquire an effective model. Compared with the experiments in Table 3 of the conference version [25], settings including experiment machine and parallel processing are updated, which leads to different evaluation results. We nonetheless state that our main observation and comparison still hold.

As an overview, the experimental results demonstrate the superiority of our model achieving scalability throughout the learning phase. On all datasets, SCARA reaches $30 - 800\times$ acceleration in precomputation time than the best decoupling method, as well as comparable or better training and inference speed, and significantly better memory overhead. When the graphs are scaled-up, the time and memory footprints of SCARA increase relatively slower than other GNN baselines, which is in line with our complexity analysis. For prediction performance, SCARA converges stably in all tasks and outputs comparable or better accuracy than other scalable competitors.

The trend of scalability can be intuitively inferred from Fig. 1. The learning time is highly associated with the edge size m. Since the inductive and transductive tasks adopt different training set splits, we plot the trends separately. The memory usage of learning is more closely related with the overall number of nodes n. As the baseline models already approach our 192GB RAM limit on graphs with a smaller scale of n, it explains the reason for the OOM error of these models in Table 3.

In a more specific view from time efficiency, our SCARA model effectively speeds up the learning process in all tasks, mostly thanks to the fast and scalable precomputation for graph propagation. The simple neural model forwarding implemented in mini-batch approach also contributes to the ef-



(a) Reddit (b) PPI (c) Yelp

Fig. 3: Validation F1 convergence curves of SCARA and baseline models on **(a)** Reddit, **(b)** PPI, and **(c)** Yelp datasets. Curves only represents the process of training phase. Shaded area is the result range of multiple runs.

ficient computation of model training and inference. On the largest available dataset Papers100M, our method efficiently completes precomputation in 13 seconds, and finishes learning in an acceptable length of time, showing the scalability of processing billion-scale graphs. In comparison among several datasets, the sampling-based GraphSAINT and GAS achieve good performance, but the $O(ILmF)$ term in training complexity results in great slowdown when graphs are scaled-up. GraphSAINT is costly for its full-batch prediction stage on the whole graph, which is usually only executable on CPUs. GAS is particularly fast for transductive inference, but it comes with the price of trading off memory expense and training time to manipulate its cache. The propagation decoupling models PPRGo and GBP show better scalability, but take more time than SCARA to converge, due to the graph information yielded by precomputation algorithms. It can be seen that their node-based propagation computations become less efficient when the graph sizes grow larger, which aligns with Table 1 complexity analysis. Remarkably, SCARA achieves about 800× and 200× faster for precomputation than these two competitors on Reddit and Amazon.

Regarding memory overhead, our method also demonstrates its efficiency benefit from its scalable implementation. We discover that the major memory expense of SCARA only increases proportional to the graph attribute matrix, while PPRGo and GBP usually demand twice as large RAM, and GraphSAINT and GAS use even more for their samplers. SCARA is the only method that finishes computation on the billion-scale Papers100M graph, while all other baselines meet out of memory error on our 192GB machine.

For learning effectiveness, SCARA achieves similar or better F1-score compared with current GNN baselines. For 4 out of 5 datasets with comparable results, our model outperforms both the state-of-the-art pre-propagation approach GBP and the scalable post-propagation baseline PPRGo. Among other methods, GraphSAINT and GAS have generally good performance for certain settings, but face the price of resource-demanding learning and poor consistency across datasets.

Fig. 3 shows the validation F1-score versus training time on representative datasets and corresponding GNN models. It can be observed that when comparing the time consumption to convergence, the SCARA model is efficient in reaching the same precision faster than most methods. The performances of GAS and PPRGo in the figure are relatively suboptimal because they are relatively less stable and require more time to converge beyond the display scopes in Fig. 3. It is worth noting that some baselines fail to or only partially converge before training terminates in tasks such as PPI.

4.3 Effect of Parameters

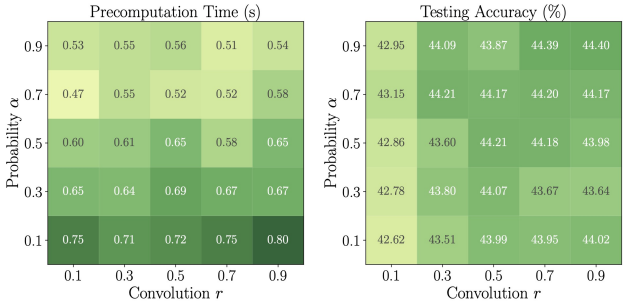
In this section we explain the selection of different parameters. For the three parameters in Feature-Push, intuitively, α is the PPR teleport probability of Feature-Push, which is dependent on graph adjacency. α is usually set to a larger value to mitigate density for graphs with higher average degrees [5]. The factor r determines the balance between left- and right-normalization as shown in Equation (6), which is usually correlated with the direction of propagation through edges. In particular, when $r = 0.5$, it degrades to the normalized adjacency matrix $\tilde{A}$ presented in APPNP [22] and PPRGo [5]. The error bound λ determines the approximation push coefficient β in Algorithm 1. Hence, λ is used to configure the trade-off between precision and speed in precomputation and tends to be larger for better efficiency.

Regarding the default selection in Table 2, we set the values of α and r for shared datasets mainly in accordance with GBP [10, 36] in order to produce comparable results. The rest Reddit and MAG are employed with the following strategy: we use $r = 0.5$ for better comparison and generality, while α is decided based on graph edge density [5]. The error bound λ can be arbitrarily large as long as it does not reduce effectiveness, we hence uniformly set it to $\lambda = 1 \times 10^{-4}$ for all datasets to provide aligned evaluations across datasets.

We conduct a grid search in Fig. 4 on the value ranges of teleport probability α and convolution coefficient r to examine their effect. In order to prevent potential influence, we use the single-thread scheme for experiments in this section. It can be inferred that a larger α has a slight improvement on precomputation efficiency, while r has no significant impact.

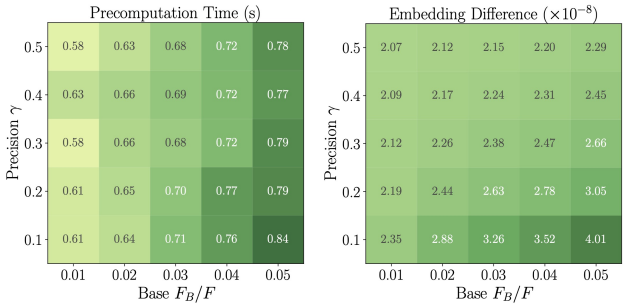
Hence, there is a trade-off in feature PPR between propagation coverage and efficiency controlled by α: a larger α indicates a higher probability of the propagation staying in the current node instead of further traveling to its neighbors, and consequently less exploration of the neighborhood. The accuracy is relatively not sensitive with variance inside the error range ($\pm 1\%$) as long as α and r values are not too extreme. It indicates that the model is robust to the changes of both parameters, based on which we are able to conclude that the parameters can be determined without requiring a sophisticated tuning.

For the base size F_B and push parameter γ in FEATURE-REUSE, we conduct additional experiments to empirically explore the algorithmic sensitivity. As above experiments show that the neural network is relatively robust and patterns are hard to infer from the testing accuracy, we thence particularly investigate the propagation stage. We use the embedding difference, which is calculated by the average absolute difference of each element in the embedding matrix $\boldsymbol{P}$ comparing with SCARA without FEATURE-REUSE, as the indicator of the feature PPR precision.



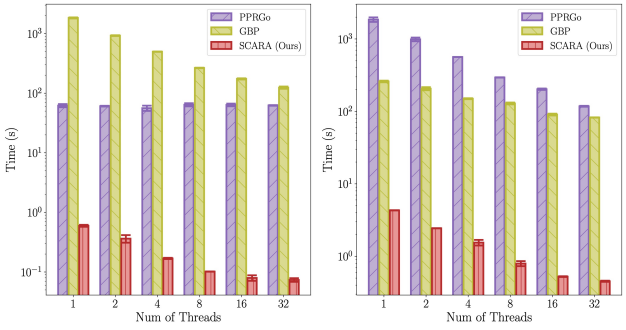
(a) Precomputation Time (b) Testing Accuracy

Fig. 4: Effect of propagation parameters teleport probability α and convolution coefficient r on SCARA **(a)** efficiency and **(b)** testing accuracy on Reddit dataset.



(a) Precomputation Time (b) Embedding Difference

Fig. 5: Effect of reuse precision parameter γ and base set size F_B on SCARA **(a)** precomputation time and **(b)** average embedding value difference on Reddit dataset. For comparison, precomputation without FEATURE-REUSE uses 2.37s.



(a) Reddit (b) Amazon

Fig. 6: Precomputation time of SCARA and decoupling baselines with different parallel schemes on **(a)** Reddit and **(b)** Amazon datasets. Note that both axes are on a log scale.

Fig. 5 presents the result on precomputation time and precision on Reddit dataset. For comparison, single-thread repetitive FEATURE-PUSH precomputation without FEATURE-REUSE uses 2.37s. It can be observed that both F_B and γ influence FEATURE-REUSE efficiency for less than ± 0.2s. Intuitively, a larger set of base features F_B requires more additional calculation time, hence hinder the overall efficiency. On the contrary, the factor γ affects less on performance as the residue vectors are still processed by subsequent calculations. The difference of embedding values is at the level of 10^{-8}, which is significantly smaller than the algorithmic error bound $\lambda = 10^{-4}$. Generally, a more aggressive reuse scheme results in relatively higher average approximation errors of the embedding values. We hence conclude that our parameter settings $F_B/F = 0.02$ and $\gamma = 0.2$ are effective for the general evaluation of SCARA.

4.4 Effect of Parallel Computation

We then employ additional experiments to study the speed-up on precomputation time brought by parallel processing. Particularly, we compare against the decoupling methods PPRGo and GBP, since sampling-based baselines GraphSAINT and GAS cannot be fit into the similar parallel scheme by design. Fig. 6 displays the efficiency results with the number of threads ranging from 1 to 32. Note that the precomputation in [25] is always single-threaded.

The experimental evaluation shows that SCARA achieves near-linear improvement when the number of parallel workers increases, demonstrating its feasibility for parallelism. Thanks to its feature-oriented design, each feature can be processed independently with efficient cache performance. Generally, adopting parallelization accelerates the precomputation by up to 10×. However, employing 32 or more threads does not significantly further improve the efficiency,

especially on smaller datasets. We argue that in this case, the main overhead becomes those non-parallelized operations.

In comparison, the two baselines PPRGo and GBP present both longer precomputation time, as well as less relative speed-up in parallelism. For PPRGo, the processing time on Reddit keeps constant even when adding more threads, implying that most of its computation expenses cannot be optimized by employing the parallel scheme.

4.5 Effect of Feature-Reuse

To examine the contribution of Feature-Reuse technique utilized in our SCARA model, we conduct ablation study to compare the performance of the reuse scheme proposed in Algorithm 2. In following notation, SCARA is the model with Feature-Reuse in precomputation following Algorithm 2. The bare iterative full-precision Feature-Push without Feature-Reuse is named as SCARA-Push, and the greedy-search based reuse scheme in [25] is regarded as SCARA-Greedy. Similarly, we test all related methods in single-thread execution to avoid noise.

We here consider the feature size as a factor of particular interest, as Feature-Reuse is a feature-oriented optimization design. We sample the node feature vectors $\boldsymbol{x}$ in the Reddit dataset to generate feature matrices $\boldsymbol{X} \in \mathbb{R}^{n\times F'}$ with different feature numbers F'. Using these features as input, we respectively evaluate the performace of graph learning. The results of average times and testing accuracies for the three variants are given in Table 4. Element-wise embedding value differences with regard to the SCARA-Push result are also presented for the two reuse schemes.

By comparing the speed-up relative to SCARA-Push without reuse, we state that Feature-Reuse substantially reduces the precomputation time for different node feature sizes. When the number of features increases, the algorithm benefits more acceleration from adopting the optimization scheme and reusing previous computations. For the full-size feature matrix with greatest improvement, SCARA achieves 3.6× speed-up compared to SCARA-Push, and 2.4× speed-up compared to SCARA-Greedy.

Table 4: Performance of SCARA variants on precomputation time (s), testing accuracy (%), and average embedding value difference ($\times 10^{-8}$) for Reddit dataset with different feature dimensions F'.

	Feature F'	100	200	400	602
Pre. Time	SCARA-Push	0.38	0.77	1.52	2.37
	SCARA-Greedy	0.30	0.56	1.08	1.54
	SCARA	0.14	0.24	0.46	0.65
Accuracy	SCARA-Push	32.7	36.6	42.0	43.9
	SCARA-Greedy	32.8	36.6	42.0	43.6
	SCARA	32.8	36.6	42.0	44.1
Embed. Diff.	SCARA-Greedy	21.0	15.0	16.5	15.6
	SCARA	0.4	0.3	0.6	2.4

Examining the reuse precision, it is inferred from Table 4 that Algorithm 2 produces less estimation error with regard to embedding values, which implies that the convergent optimization solution is not only faster but also more precise than the SCARA-Greedy search. Meanwhile, Feature-Reuse causes no significant difference on model effectiveness, as minor accuracy fluctuations are under the error bound of repetitive experiments. Interestingly, even with a feature dimension of $F' = 100$, the model achieves 32.8% testing accuracy, indicating that our feature PPR embedding matrix is capable to store adjacency and feature information that is sufficient for model learning.

5 Conclusion

In this paper, we propose SCARA, a scalable Graph Neural Network algorithm with feature-oriented optimizations. Our theoretical contribution includes showing the SCARA model has a sub-linear complexity that efficiently scales-up the graph propagation by two algorithms, namely Feature-Push and Feature-Reuse. We conduct extensive experiments on various datasets to demonstrate the time and memory scalability of SCARA in learning and inference. Our model is efficient to process billion-scale graph data and achieves up to 800× faster than the current state-of-the-art scalable GNNs in precomputation, while maintaining comparable or better accuracy.

Although our model improves the scalability of GNN propagation by employing the decoupling strategy, it is discovered that the architecture simplification may be less flexible when applied to graph variants that do not rely on neighborhood-based propagation, such as graphs under heterophily [13, 28, 24]. We believe that exploring decoupling schemes that are suitable for different variants of graphs is a promising future direction for further expanding our proposed feature-oriented framework.

Acknowledgements This research is supported by Singapore MOE funding (MOE-T2EP20122-0003, RS05/21), NTU-NAP startup grant (022029-00001), and the Joint NTU-WeBank Research Centre on FinTech. Xiang Li is supported by Shanghai Pujiang Talent Program No. 21PJ1402900, Shanghai Science and Technology Committee General Program No. 22ZR1419900 and National Natural Science Foundation of China No. 62202172.

References

1. Al-Rfou, R., Perozzi, B., Zelle, D.: Ddgk: Learning graph representations for deep divergence graph ker-

nels. In: The World Wide Web Conference, pp. 37–48 (2019)
2. Andersen, R., Borgs, C., Chayes, J., Hopcraft, J., Mirrokni, V.S., Teng, S.H.: Local Computation of PageRank Contributions. In: Algorithms and Models for the Web-Graph, vol. 4863, pp. 150–165. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
3. Andersen, R., Chung, F., Lang, K.: Local Graph Partitioning using PageRank Vectors. In: 2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06), pp. 475–486. IEEE (2006)
4. Atwood, J., Towsley, D.: Diffusion-convolutional neural networks. 29th Advances in Neural Information Processing Systems pp. 2001–2009 (2016)
5. Bojchevski, A., Klicpera, J., Perozzi, B., Kapoor, A., Blais, M., Rózemberczki, B., Lukasik, M., Günnemann, S.: Scaling graph neural networks with approximate pagerank. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining pp. 2464–2473 (2020)
6. Candès, E.J., Li, X., Ma, Y., Wright, J.: Robust principal component analysis? Journal of the ACM **58**(3), 1–37 (2011)
7. Chandrasekaran, V., Sanghavi, S., Parrilo, P.A., Willsky, A.S.: Rank-Sparsity Incoherence for Matrix Decomposition. SIAM Journal on Optimization **21**(2), 572–596 (2011)
8. Chen, J., Ma, T., Xiao, C.: Fastgcn: Fast learning with graph convolutional networks via importance sampling. In: International Conference on Learning Representations (2018)
9. Chen, J., Zhu, J., Song, L.: Stochastic training of graph convolutional networks with variance reduction. 35th International Conference on Machine Learning **3**, 1503–1532 (2018)
10. Chen, M., Wei, Z., Ding, B., Li, Y., Yuan, Y., Du, X., Wen, J.R.: Scalable graph neural networks via bidirectional propagation. 33rd Advances in Neural Information Processing Systems (2020)
11. Chen, Z., Bruna, J., Li, L.: Supervised community detection with line graph neural networks. 7th International Conference on Learning Representations (2019)
12. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 257–266 (2019)
13. Chien, E., Peng, J., Li, P., Milenkovic, O.: Adaptive universal generalized pagerank graph neural network. In: 9th International Conference on Learning Representations (2021)
14. Coudron, M., Lerman, G.: On the Sample Complexity of Robust PCA. In: 25th Advances in Neural Information Processing Systems (2012)
15. Ding, M., Kong, K., Li, J., Zhu, C., Dickerson, J.P., Huang, F., Goldstein, T.: VQ-GNN: A Universal Framework to Scale up Graph Neural Networks using Vector Quantization. 34th Advances in Neural Information Processing Systems (2021)
16. Fey, M., Lenssen, J.E., Weichert, F., Leskovec, J.: GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In: 38th International Conference on Machine Learning. PMLR 139 (2021)
17. Fogaras, D., Rácz, B., Csalogány, K., Sarlós, T.: Towards Scaling Fully Personalized PageRank: Algorithms, Lower Bounds, and Experiments. Internet Mathematics **2**(3), 333–358 (2005)
18. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 1025–1035 (2017)
19. Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., Leskovec, J., Barzilay, R., Battaglia, P., Bengio, Y., Bronstein, M., Günnemann, S., Hamilton, W., Jaakkola, T., Jegelka, S., Nickel, M., Re, C., Song, L., Tang, J., Welling, M., Zemel, R.: Open Graph Benchmark : Datasets for Machine Learning on Graphs. 33rd Advances in Neural Information Processing Systems (2020)
20. Huang, Z., Zhang, S., Xi, C., Liu, T., Zhou, M.: Scaling up graph neural networks via graph coarsening. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, vol. 1, pp. 675–684 (2021)
21. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (2017)
22. Klicpera, J., Bojchevski, A., Günnemann, S.: Predict then propagate: Graph neural networks meet personalized pagerank. 7th International Conference on Learning Representations pp. 1–15 (2019)
23. Li, X., Zhu, R., Cheng, Y., Shan, C., Luo, S., Li, D., Qian, W.: Finding Global Homophily in Graph Neural Networks When Meeting Heterophily. In: 39th International Conference on Machine Learning (2022)
24. Liao, N., Luo, S., Li, X., Shi, J.: LD2: Scalable heterophilous graph neural network with decoupled embedding. In: Advances in Neural Information Processing Systems, vol. 36 (2023)
25. Liao, N., Mo, D., Luo, S., Li, X., Yin, P.: SCARA: Scalable graph neural networks with feature-oriented optimization. Proceedings of the VLDB Endowment **15**(11), 3240–3248 (2022)

26. Lin, D., Wong, R.C.W., Xie, M., Wei, V.J.: Index-free approach with theoretical guarantee for efficient random walk with restart query. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 913–924 (2020)
27. Lin, Z., Liu, R., Su, Z.: Linearized Alternating Direction Method with Adaptive Penalty for Low-Rank Representation. In: 24th Advances in Neural Information Processing Systems (2011)
28. Liu, H., Liao, N., Luo, S.: Simga: A simple and effective heterophilous graph neural network with efficient global aggregation. arXiv e-prints (2023)
29. Mo, D., Luo, S.: Agenda: Robust personalized pageranks in evolving graphs. In: Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 1315–1324. ACM, Virtual Event Queensland Australia (2021)
30. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep. (1999)
31. Sinha, A., Shen, Z., Song, Y., Ma, H., Eide, D., Hsu, B.J.P., Wang, K.: An overview of microsoft academic service (mas) and applications. In: Proceedings of the 24th International Conference on World Wide Web, pp. 243–246 (2015)
32. Sun, L., Dou, Y., Yang, C., Wang, J., Yu, P.S., He, L., Li, B.: Adversarial attack and defense on graph data: A survey. arXiv e-prints (2018)
33. Thekumparampil, K.K., Wang, C., Oh, S., Li, L.J.: Attention-based graph neural network for semi-supervised learning. arXiv e-prints (2018)
34. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. In: 8th International Conference on Learning Representations (2017)
35. Wang, C., Pan, S., Long, G., Zhu, X., Jiang, J.: Mgae: Marginalized graph autoencoder for graph clustering. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM '17, p. 889–898. Association for Computing Machinery, New York, NY, USA (2017)
36. Wang, H., He, M., Wei, Z., Wang, S., Yuan, Y., Du, X., Wen, J.R.: Approximate Graph Propagation. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, vol. 1, pp. 1686–1696. Association for Computing Machinery (2021)
37. Wang, K., Luo, S., Lin, D.: River of no return: Graph percolation embeddings for efficient knowledge graph reasoning. arXiv e-prints (2023)
38. Wang, S., Yang, R., Wang, R., Xiao, X., Wei, Z., Lin, W., Yang, Y., Tang, N.: Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. ACM Transactions on Database Systems **44**(4), 1–37 (2019)
39. Wang, S., Yang, R., Xiao, X., Wei, Z., Yang, Y.: Fora: Simple and effective approximate single-source personalized pagerank. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining **Part F1296**, 505–514 (2017)
40. Wang, X., Zhang, M.: How powerful are spectral graph neural networks. In: 39th International Conference on Machine Learning (2022)
41. Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., Weinberger, K.: Simplifying graph convolutional networks. In: K. Chaudhuri, R. Salakhutdinov (eds.) Proceedings of the 36th International Conference on Machine Learning, vol. 97, pp. 6861–6871 (2019)
42. Wu, H., Gan, J., Wei, Z., Zhang, R.: Unifying the global and local approaches: An efficient power iteration with forward push. In: Proceedings of the 2021 International Conference on Management of Data, vol. 1, pp. 1996–2008 (2021)
43. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems **32**(1), 4–24 (2021)
44. Yang, R., Shi, J., Xiao, X., Yang, Y., Liu, J., Bhowmick, S.S.: Scaling attributed network embedding to massive graphs. Proceedings of the VLDB Endowment **14**(1), 37–49 (2021)
45. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 974–983 (2018)
46. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: Graphsaint: Graph sampling based learning method. In: International Conference on Learning Representations (2019)
47. Zhang, J., Zhang, H., Xia, C., Sun, L.: Graph-bert: Only attention is needed for learning graph representations. arXiv e-prints (2020)
48. Zhang, M., Chen, Y.: Link prediction based on graph neural networks. Advances in Neural Information Processing Systems **31**, 5165–5175 (2018)
49. Zhang, Z., Cui, P., Zhu, W.: Deep Learning on Graphs: A Survey. IEEE Transactions on Knowledge and Data Engineering **14**(8), 1–1 (2020)
50. Zhu, Z., Peng, J., Li, J., Chen, L., Yu, Q., Luo, S.: Spiking Graph Convolutional Networks. In: 31th International Joint Conference on Artificial Intelligence (2022)
51. Zügner, D., Akbarnejad, A., Günnemann, S.: Adversarial attacks on neural networks for graph data. In: Proceedings of the 24th ACM SIGKDD International Con-

ference on Knowledge Discovery & Data Mining, pp. 2847–2856 (2018)