Scaling up Graph Neural Networks

A Confirmation Report
Submitted to the School of Computer Science and Engineering
of the Nanyang Technological University

by

Liao Ningyi

for the Confirmation for Admission
to the Degree of Doctor of Philosophy

2023

Abstract

Recent years have witnessed the burgeoning of services based on data represented by graphs, which leads to 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 is becoming increasingly popular thanks to their impressive performance on a wide range of graph learning tasks. Most of these models, however, are known to be difficult to scale up. Our examination suggests that the GNN scalability bottleneck is usually the iterative message-passing propagation procedure, which is tightly coupled with the graph structure. Hence, improving the performance of the expensive graph propagation by graph related techniques becomes the key in scaling up GNNs.

In this report, we explore the scalability issue on different graph variants and GNN designs, and propose our approaches to scale up GNN to million- or even billion-scale graphs by simplifying the graph propagation operation.

We first propose SCARA, a scalable GNN with feature-oriented optimization for graph computation, to address the propagation bottleneck by decoupling it as precomputation. 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 our model. 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.

Next, we specifically study the scalability issue of heterophilous GNN, a family of GNNs that specializes in learning graphs where connected nodes tend to have different labels. We propose a scalable model, LD^2, which simplifies the learning procedure by decoupling

graph propagation and generating expressive embeddings prior to training. We perform theoretical analysis to demonstrate that LD^2 realizes optimal time complexity in training, as well as a memory footprint that remains independent of the graph scale. The capability of our model is evaluated by extensive experiments. Being lightweight in minibatch training on large-scale heterophilous graphs, it achieves up to 15× speed improvement and efficient memory utilization, while maintaining comparable or better performance than the baselines.

Acknowledgements

I would like to extend my first and sincerest gratitude to my advisor, Prof Siqiang Luo, without whom, this work would never have been possible. The insightful perspectives and immense knowledge from him have inspired me on my research all the time. The meticulous guidance and unwavering patience from him have motivated me throughout the years.

I gratefully recognize the detailed advice, constructive feedback, and generous assistance of my Thesis Advisory Committee members Prof Cheng Long and Prof Guosheng Lin. Very special thanks to Prof Xiang Li, Prof Jiemin Shi and Dr Pengcheng Yin, who provide informative suggestion to this research.

I would like to thank my colleagues and research laboratory mates, Dingheng Mo, Haoyu Liu, and Kai Siong Yow, for the substantial and fruitful discussions among us, which boost my research towards a better level.

I would always express my gratitude to all my friends and my family, for their unparalleled understanding, encouragement, and support.

Contents

List of Figures

List of Tables

List of Algorithms

List of Publications

- **Ningyi Liao***, Dingheng Mo*, Siqiang Luo, Xiang Li, Pengcheng Yin. "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". *Proceedings of the VLDB Endowment*, Vol. 15, No. 11, pp. 3240-3248, 2022.

- Kai Siong Yow, **Ningyi Liao**, Siqiang Luo, Reynold Cheng. "Machine Learning for Subgraph Extraction: Methods, Applications and Challenges". *Proceedings of the VLDB Endowment*, Vol. 16, No. 12, 2023.

- Jun Xuan Yew, **Ningyi Liao**, Dingheng Mo, Siqiang Luo. "Example Searcher: A Spatial Query System via Example". *IEEE 39th International Conference on Data Engineering (ICDE)*, 2023.

- **Ningyi Liao**, Shufan Wang, Liyao Xiang, Nanyang Ye, Shuo Shao, Pengzhi Chu. "Achieving adversarial robustness via sparsity". *Machine Learning*, Vol. 111, pp. 685-711, 2021.

- **Ningyi Liao**, Siqiang Luo, Xiang Li, Jieming Shi. "LD^2: Scalable Heterophilous Graph Neural Network with Decoupled Embedding". Under submission of the *37th Conference on Neural Information Processing Systems*.

- **Ningyi Liao**, Dingheng Mo, Siqiang Luo, Xiang Li, Pengcheng Yin. "Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization". Under submission of *The VLDB Journal*.

- Haoyu Liu, **Ningyi Liao**, Siqiang Luo. "SIMGA: A Simple and Effective Heterophilous Graph Neural Network with Efficient Global Aggregation". Under submission of the *37th Conference on Neural Information Processing Systems*.

*Both authors contributed equally to this research.

- Kai Siong Yow, **Ningyi Liao**, Siqiang Luo, Reynold Cheng, Chenhao Ma, Xiaolin Han. "A Survey on Machine Learning Solutions for Graph Pattern Extraction". Under submission of the *The VLDB Journal*.

Chapter 1

Introduction

1.1 Motivations and Challenges

Graphs are ubiquitous in modeling entities and their relationships. The marriage of graphs and deep learning leads to Graph Neural Networks (GNNs), which describe a set of neural networks that process data represented by graph structures. Recent advances in Graph Neural Networks (GNNs) has shown the power in a wide range of applications, such as computer vision [1, 2], natural language processing [3, 4], spatial-temporal prediction [5, 6], and natural science [7, 8].

Message-passing GNN designs. Compared to deep learning algorithms on Euclidean data, GNNs utilize neural architectures to perform tasks on graph data of nodes and edges, modeling both instances and their relationships. One of the most widely adopted GNN designs is the Graph Convolutional Network (GCN) [3] which learns graph representations by leveraging information of topological structure through an operation called message passing, or interchangeably, graph convolution. 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. In this means, the information of one node is passed to another based on edge connections.

Scalability issues of current GNNs. While being able to effectively gather state information from the graph structure, the family of networks represented by GCN are known to be resource-demanding, which implies limited scalability when deployed to large-scale graphs [9, 10]. 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, which typically have million-scale nodes with various kinds of attributes. 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.

1.2 Major Contributions

This report highlights our studies towards scaling up GNNs in two aspects. We firstly enhance the precomputation phase in decoupled GNNs by integrating graph management techniques and feature-oriented optimizations. Then, we look into introducing the decoupled architecture into a specific field of heterophilous graphs.

In the first part, we propose SCARA, a scalable GNN with high scalability on very large datasets. The model hierarchy combines two algorithms proposed by us, namely Feature-Push and Feature-Reuse. The Feature-Push algorithm propagates graph information from the feature vectors with forward push and random walk. The Feature-Reuse mechanism further utilizes feature-oriented optimizations to improve the efficiency of feature propagation while maintaining precision. Powered by these two approaches, SCARA realizes a sub-linear complexity for precomputation running time along with efficient model training and inference implemented in the mini-batch approach. It also demonstrates efficient memory usage in processing billion-scale graphs.

Our second work examines the scalability issues for graphs under heterophily and proposes LD^2, a scalable GNN for heterophilous graphs with Low-Dimension embeddings and Long-Distance aggregation. The model removes the reliance on iterative train-time full-graph computations by utilizing the decoupled network architecture that learns on precomputed embeddings. We design a set of feature and topology embeddings by applying multi-hop discriminative propagation, encoding expressive node representation within a compact size. An end-to-end precomputation algorithm is proposed for efficient embedding calculation.

LD^2 achieves theoretically optimized training, highlighting time complexity that is only linear to the number of nodes $O(n)$ and memory overhead independent of the graph scale. To the best of our knowledge, LD^2 is the first model to achieve such optimization in the context of heterophilous GNNs.

1.3 Outline of the Report

This report is organized as follows:

- Chapter 1 introduces the background and advances of common GNN designs under the message-passing framework. It points out the current issues in scalability of such GNN designs, deriving the research question of scaling up GNNs to large graphs while maintaining or even improving the performance. We also summarize our contributions under this topic.
- Chapter 2 reviews the architectures of related GNN variants, including existing approaches in efficient and scalable designs. We develop a comprehensive analysis on the scalability bottleneck of these models on time and memory complexity.
- Chapter 3 proposes the **SCARA** model as the scalable decoupled GNN with feature-oriented optimization. We subsequently derive our methodology in improving the precomputation with enhanced algorithms. Extensive experiments are conducted to demonstrate the model efficiency on billion-scale graphs.
- Chapter 4 presents the **LD^2** model as the scalable decoupled heterophilous GNN with Low-Dimension embeddings and Long-Distance aggregation. We tackle the specific scenario of graphs under heterophily and propose the multi-channel scheme to utilize the decoupling technique. Experimental evaluations show that the model achieves improved minibatch training performance especially on large graphs.
- Chapter 5 concludes our current progress. We also conduct further discussions and expectations of future research.

Chapter 2

Literature Review

2.1 Notations and Vanilla GNNs

2.1.1 Graph Notations

In a graph $G = (V, E)$ with node set V and edge set E, the number of nodes, the number of edges, and the average degree are denoted by $n = |V|$, $m = |E|$, and $d = m/n$, respectively. The neighborhood of an ego node $u \in V$ is the set $\mathcal{N}(u) = \{v|(u, v) \in E\}$, and its degree $d(u) = |\mathcal{N}(u)|$. The diagonal degree matrix is $\boldsymbol{D} = \text{diag}(d(1), d(2), \cdots, d(n))$. The graph connectivity is represented by the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$. We adopt the general normalization scheme [11, 12] with coefficients $a, b \in [0, 1]$ and $\bar{\boldsymbol{A}} = \boldsymbol{D}^{-a}\boldsymbol{A}\boldsymbol{D}^{-b}$. The normalized version with self-loop edges is also frequently used, which is denoted as $\tilde{\boldsymbol{A}} = (\boldsymbol{I} + \boldsymbol{D})^{-a}(\boldsymbol{I} + \boldsymbol{A})(\boldsymbol{I} + \boldsymbol{D})^{-b}$, and the corresponding Laplacian matrix is $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$. Each node $u \in V$ is represented by an F-dimensional attribute vector $\boldsymbol{x}(u)$, which composes the attribute matrix $\boldsymbol{X} \in \mathbb{R}^{n\times F}$.

Following notations are frequently used in graph spectral theory. Consider an undirected graph whose adjacency matrix $\tilde{\boldsymbol{A}}$ is symmetric, the eigendecomposition of the normalized graph adjacency and Laplacian matrices respectively as $\tilde{\boldsymbol{A}} = \boldsymbol{U}\boldsymbol{M}\boldsymbol{U}^\top$ and $\tilde{\boldsymbol{L}} = \boldsymbol{V}\boldsymbol{N}\boldsymbol{V}^\top$, where $\boldsymbol{M} = \text{diag}(\mu_1, \cdots, \mu_n)$, $|\mu_1| \geq |\mu_2| \geq \cdots \geq |\mu_n|$, $\boldsymbol{N} = \text{diag}(\nu_1, \cdots, \nu_n)$, $0 = \nu_1 < \nu_2 \leq \cdots \leq \nu_n$, and $\boldsymbol{U}, \boldsymbol{V}$ are the matrices of corresponding eigenvectors.

Intuitively, since $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$, the leading eigenvalues $\mu_1, \mu_2, \cdots$ of $\tilde{\boldsymbol{A}}$ correspond to the smallest of those $\nu_1, \nu_2, \cdots$ of $\tilde{\boldsymbol{L}}$. These eigenvalues are known as the *low-frequency spectrum* of the graph that correlates to graph connectivity. Specially, $\nu_2 > 0$ if and only if the graph is connected, which is our case. Similarly, small values of μ_f and large values of ν_i represent the high frequency part of the graph. Graph spectrum is a graph invariant despite the status of node labels.

2.1.2 GNN Taxonomy

Motivated by the increasing neural network structures of deep learning approaches, various operations have been introduced to GNN designs to build a broad range of neural networks on a graph. Early studies propose the notion of GNN as the neural network application in graph domains [13]. These *recurrent GNNs* compute and propagate node information through the graph iteratively to learn a convergent state. Gate-based methods of Gate Recurrent Units and Long Short-Term Memory are employed as propagation optimizations [14, 15]. The convolution operation is generalized from grid data to graph and composed the core of *convolutional GNNs* [3, 16]. The basic idea is to represent a node by aggregating the feature of itself as well as its neighbors.

A previous line of convolutional GNNs is based on *spectral* convolution, which is the operator defined in the domain after graph Fourier transform, and is introduced to build GNNs including Spectral CNN [17] and ChebNet [18]. In comparison, *spacial*-based GNNs apply graph convolution directly according to graph topology and have a wide range of varieties such as DCNN [16] and GraphSage [19]. The proposal of GCN [3] bridges the gap between the above two domains of operators. Thanks to the similarity of convolutional layers, techniques from grid data of attention mechanism and skip connection have been generalized to further boost the convolutional GNNs [20, 21].

2.1.3 Vanilla GCN Propagation

A GNN recurrently computes the node representation matrix $\boldsymbol{H}^{(l)}$ as current state in the l-th layer. For the vanilla L-layer GCN [3], the model input feature matrix is $\boldsymbol{H}^{(0)} = \boldsymbol{X}$

TABLE 2.1: Precomputation, training, and inference time complexity of common GNN models.

Model	Precomp. Time	Training Time	Inference Time
GCN [3]	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
Cluster-GCN [22]	$O(m)$	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
GraphSAINT [23]	–	$O(IL_PLnF^2)$	$O(LmF + LnF^2)$
GAS [24]	$O(m + LnF)$	$O(ILmF + ILnF^2)$	$O(nF)$
APPNP [25]	$O(m)$	$O(IL_PmF + ILnF^2)$	$O(L_PmF + LnF^2)$
PPRGo [26]	$O(m/\delta)$	$O(IKnF + ILnF^2)$	$O(KnF + LnF^2)$
SGC [27]	$O(L_PmF)$	$O(ILnF^2)$	$O(LnF^2)$
GBP [11]	$O(L_PF\sqrt{L_Pm\log(L_Pn)}/\epsilon)$	$O(ILnF^2)$	$O(LnF^2)$

TABLE 2.2: Precomputation, training, and inference memory complexity of common GNN models.

Model	Precomp. Mem.	Training Mem.	Inference Mem.
GCN [3]	–	$O(LnF + LF^2)$	$O(LnF + LF^2)$
Cluster-GCN [22]	$O(n)$	$O(Ln_bF + LF^2)$	$O(LnF + LF^2)$
GraphSAINT [23]	–	$O(L_PLn_bF + LF^2)$	$O(LnF + LF^2)$
GAS [24]	$O(LnF)$	$O(Ldn_bF + LF^2)$	$O(Ldn_bF + LF^2)$
APPNP [25]	$O(m)$	$O(Ln_bF + LF^2 + nn_b)$	$O(Ln_bF + LF^2 + nn_b)$
PPRGo [26]	$O(n/\delta)$	$O(Ln_bF + LF^2 + Kn_b)$	$O(Ln_bF + LF^2 + Kn_b)$
SGC [27]	$O(m)$	$O(Ln_bF + LF^2)$	$O(Ln_bF + LF^2)$
GBP [11]	$O(nF)$	$O(Ln_bF + LF^2)$	$O(Ln_bF + LF^2)$

in particular, and the $(l+1)$-th representation matrix $\boldsymbol{H}^{(l+1)}$ is updated as:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{2.1}$$

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 simplicity we assume the feature size F to be constant in all layers.

Summarized in Tables 2.1 and 2.2, we present an analysis on the complexity bounds of GCN in Eq. (2.1) to explain the restraints of its efficiency in computational time and memory, respectively. In the tables, training and inference memory indicate the GPU usage for storing and updating representation and weight matrices, while precomputation is usually conducted on RAM. The memory complexity indicates the usage of intermediate variables, and fixed storage such as the graph adjacency matrices are omitted. Training and inference time complexity represent the forward-passing computational operations on respective node sets.

One dominating part of the GNN learning overhead is the training phase, where the model weights $\boldsymbol{W}^{(l)}$ are iteratively updated for I epochs and is 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 $\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}$, which can be regarded as repetitive sparse-dense matrix multiplications, 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 $O(LnF^2)$. In the inference phase, the model performs a similar forward prediction, hence results in the same time complexity of $O(LmF + LnF^2)$.

As discovered by previous studies [11, 22], 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 with respect of GNN learning time.

In terms of memory usage, GCN typically requires $O(LnF+LF^2)$ space to store layer-wise node representations and weight matrices, respectively. For large-scale cases where $n \gg F$, the overhead of dense node representations $O(LnF)$ becomes the primary term [28].

2.2 Sampling-based GNNs

2.2.1 Layer-wise Sampling Models

The above analysis indicates that the scalability bottleneck of GCN lies in the time complexity of graph propagation as well as the memory overhead of full-graph representation. There is a large scope of GNNs attempting to address the issue by sampling techniques, which simplify the propagation by replacing the entire graph with subgraphs in minibatches [19, 22, 23]. The message-passing scheme conducted in Eq. (2.1) is hence not on the whole graph, but a sampled subgraph with corresponding adjacency and embedding matrices.

Some studies utilize *layer-wise* sampling, where node samples are generated differently in the propagation of each layer. GraphSAGE [19] typically perform random sampling

to generate a smaller neighbor set of each node with equal size, while FastGCN [29] and LADIES [30] randomly samples nodes in the entire graph.

GAS [24] 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.

2.2.2 Graph-wise Sampling Models

Another popular direction is *graph-wise* sampling with hierarchical consideration of the whole graph structure. Cluster-GCN [22] divides a graph into subgraphs based on the result of classical graph clustering algorithms. It requires $O(m)$ precomputation time of finding clustersm, while the training time is bounded by $O(LmF + LnF^2)$.

GraphSAINT [23] proposes various schemes to utilize different levels of information. The GraphSAINT model with L-step random walk considers $O(Ln_b)$ nodes in each training iteration, hence produces a total complexity of $O(L^2nF + L^2nF^2)$. Unfortunately, the sampling approach is not applicable in the full graph inference stage, resulting in its inference time and memory overhead being the same with GCN.

2.3 Decoupled GNNs

2.3.1 Post-Propagation Decoupled Models

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 [31, 32]. 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 iterative graph propagation in the GCN updates is replaced by multiplying the PPR matrix after the feature transformation layers:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-2, \tag{2.2}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{\Pi}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = L-1, \tag{2.3}$$

$\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ is a matrix representing graph propagation, usually in the form $\boldsymbol{\Pi} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l$, where L_P denotes the precomputed propagation hops and a_l is the hop-dependent diffusion weight.

The *feature transformation* in this case is Eq. (2.2), which only contains L layers of consecutive weight multiplication, much similar to a simple Multi-Layer Perceptron (MLP) and is sometimes simply noted as $\boldsymbol{H}^{(L)} = \mathrm{MLP}(\boldsymbol{X})$. The design is benefit from the mini-batch scheme in both training and inference stages, hence reducing the demand for GPU memory.

Eq. (2.3) corresponds to the *graph propagation* stage in common GNNs. Since it is after the feature transformation, it is regarded as post-propagation. Compared to Eq. (2.1), it is decoupled from the iterative calculation of multiple layers, and only conducted once per training epoch. However, it is worth noting that such post-propagation is only decoupled from the iterative layer propagation, but still requires computation in the training iteration.

The APPNP model [25] introduces the personalized PageRank (PPR) [33] algorithm in the propagation stage, where $\hat{\boldsymbol{\Pi}} = \sum_{l=0}^{L_P} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l$ is the PPR matrix of L_P hops. APPNP performs the post-propagation on the PPR matrix by an L_P-round Power Iteration [33], which leads to a computation speed of $O(L_P mF + LnF^2)$ per epoch. By adapting minibatch training, it does not need to load the full feature matrix into GPU memory, but only the corresponding rows in matrix $\boldsymbol{H}$ and $\boldsymbol{\Pi}$, hence is with a reduced complexity of $O(Ln_bF + LF^2 + nn_b)$, where n_b is the batch size.

The PPRGo model [26] further improves the efficiency of precomputing the PPR matrix $\boldsymbol{\Pi}$ by the Forward Push algorithm [34, 35] with an error threshold δ and only records the top-K entries. However, it demands $O(n/\delta)$ space to store the dense PPR matrix.

2.3.2 Pre-Propagation Decoupled Models

Another line of research, namely the pre-propagation models, 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:

$$\boldsymbol{H}^{(0)} = \boldsymbol{P} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l \cdot \boldsymbol{X}, \tag{2.4}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)} \boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{2.5}$$

where L_P denotes the precomputed propagation hops and a_l is the hop-dependent diffusion weight.

The line of Eq. (2.4) corresponds to the *precomputation* section performing the graph propagation. As the embedding matrix is calculated only once for each graph, it saves the propagation time in following L-layer model updates of feature transformation of multiple epochs compared to GCN in Eq. (2.1). The complexity of this stage is completely free from the training iteration and is solely related to the precomputation techniques applied in the model.

Eq. (2.5) follows the neural network *feature transformation*, taking $\boldsymbol{P}$ as input feature. Compared to Eq. (2.3), 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.

In SGC [27], the embedding matrix is given by a fix-hop multiplication of $\boldsymbol{P} = \tilde{\boldsymbol{A}}^{L_P} \cdot \boldsymbol{X}$. S^2GC [36] performs constant summation on the powers of graph adjacency to obtain a low-frequency embedding $\boldsymbol{P} = \frac{1}{L_P} \sum_{l=0}^{L_P} \tilde{\boldsymbol{A}}^l \cdot \boldsymbol{X}$. GDC [37] formulates the generalized form $\boldsymbol{P} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l \cdot \boldsymbol{X}$. It however mostly focuses on the Heat Kernel form where $a_l = e^{-t} \cdot \frac{t^l}{l!}$. Adapting different embedding schemes, these models calculates the propagation by vanilla matrix calculation, hence all require $O(LmF)$ precomputation time.

The model GBP [11] employs a PPR-based bidirectional propagation with $L_P = L$ and tunable a_l and r to optimize the precomputation calculation. 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. The embedding matrix in GBP is a dense matrix that requires $O(nF)$ memory.

AGP [12] proposes further generalization in two aspects. First, it extends graph normalization $\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ to arbitrary $\boldsymbol{D}^{-a}\boldsymbol{A}\boldsymbol{D}^{-b}$ with $a, b \in [0, 1]$. Second, it efficiently computes propagation with general coefficients a_l. In the paper, it explores the SGC, APPNP (PPR), and GDC (Heat Kernel) schemes.

2.4 Heterophilous Graphs and GNNs

2.4.1 Graphs Under Heterophily

For multiclass classification task on graph $G = (V, E)$, a node $u \in V$ is labeled by $y(u) \in \{0, 1, \cdots, N_c - 1\}$, where N_c is the number of classes. The term *homophily* indicates that connected nodes tend to be similar to each other in terms of classes, while in *heterophily* scenarios the majority are different. Note that heterophily is not the same of heterogeneous, where in the latter case nodes are dissimilar with respect to types but not labels.

We measure the graph heterophily by node homophily score [38], which is the average proportion of the neighbors with the same class of each node:

$$\mathcal{H}_n = \frac{1}{|V|}\sum_{u\in V}\frac{|\{v \in \mathcal{N}(u) : y(v) = y(u)\}|}{|\mathcal{N}(u)|}. \tag{2.6}$$

Generally, $\mathcal{H}_n \in [0, 1]$. A homophily score closer to 0 indicates higher heterophily, and vice versa.

TABLE 2.3: Precomputation, training, and inference time complexity of heterophilous GNN models.

Model	Precomp. Time	Training Time	Inference Time
GPRGNN [39]	$O(m)$	$O(IL_PmF + ILnF^2)$	$O(L_PmF + LnF^2)$
GCNJK [40]	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
MixHop [41]	–	$O(IL_PLmF + ILnF^2)$	$O(L_PLmF + LnF^2)$
LINKX [42]	–	$O(ImF + ILnF^2)$	$O(mF + LnF^2)$

TABLE 2.4: Precomputation, training, and inference memory complexity of heterophilous GNN models.

Model	Precomp. Mem.	Training Mem.	Inference Mem.
GPRGNN [39]	$O(m)$	$O(LnF + LF^2 + m)$	$O(LnF + LF^2 + m)$
GCNJK [40]	–	$O(L_CnF + L_CF^2)$	$O(L_CnF + L_CF^2)$
MixHop [41]	–	$O(CLnF + CLF^2)$	$O(CLnF + CLF^2)$
LINKX [42]	–	$O(L_Cn_bF + L_CF^2 + nF)$	$O(L_Cn_bF + L_CF^2 + nF)$

2.4.2 Iterative Heterophilous GNNs

In the context of GNNs under heterophily, most current models belong to the iterative design, that they alter the vanilla propagation Eq. (2.1) for better performance but do not change its iterative nature. Similarly, we compare the complexity of representative heterophilous models in Tables 2.3 and 2.4.

Usually, heterophilous GNNs consider full-graph information, relying upon the complete graph adjacency matrix to compute inter-node relationships. These high-order calculations are shown to be effective in retrieving information beyond immediate neighbors, but come at the price of more complex propagation operations.

H_2GCN [43] examines the homophily-dominant property for 2-hop neighbors, and simultaneously performs propagation on both 1-hop and 2-hop adjacency matrices $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{A}}_2$. Specifically, the 2-hop matrix $\tilde{\boldsymbol{A}}_2$ is the adjacency matrix of the induced subgraph consisting of only strict 2-hop neighbors $\bar{\mathcal{N}}_2(u) = \{v | t \in \mathcal{N}(u), v \in \mathcal{N}(t), v \notin \mathcal{N}(u)\}$. The two representations are usually aggregated by a jumping knowledge layer.

MixHop [41] concatenates identity, 1-hop, and 2-hop propagations in each of its layer $\boldsymbol{H}^{(l+1)} = \sigma(\boldsymbol{H}^{(l)}\boldsymbol{W}_0^{(l)} \| \tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{W}_1^{(l)} \| \tilde{\boldsymbol{A}}^2\boldsymbol{H}^{(l)}\boldsymbol{W}_2^{(l)})$, where $(\cdot\|\cdot)$ denotes matrix concatenation. Such aggregation results in expanding width of representations over multiple layers.

GeomGCN [38] incorporates geometric measures besides node connections to build the overview of the entire graph, while GloGNN [32] considers global information during message-passing, which is equivalent to a propagation of layer representations of nodes from different hops.

Another common practice for non-homophilous design is altering transformation to learn from multiple features, i.e. channels. Denote the number of channels as C, employing multi-channel learning per layer respectively increases the memory budget for node representations and weight matrices by C. In Table 2.4 we denote $L_C = L + C$ for simplicity.

GCNJK [40] records individual layer representations as channels, which is widely use by following works such as H_2GCN. MixHop also include a multi-channel design that aggregating embeddings from different hops.

FAGCN [44] introduces the high-frequency filter $\epsilon \boldsymbol{I} - \bar{\boldsymbol{A}}$. In each layer, it respectively applies low- and high-frequency filters to the layer representation and aggregates by attention mechanism.

GGCN [45] proposes the process of assigning signs to edges based on inter- and intra-node similarity. In practice, they utilize cosine similarity between node feature vectors. Its aggregation is performed on the representations corresponding to the positive edges, the negative edges, and the raw representation of previous layer, with weights of each channel controlled by a learnable scalar factor.

ACM [46] explores the channel mixing mechanism, similarly applying multiple channels to learn the layer representation. For each layer, low-frequency, high-frequency, and identity channels are respectively applied to the current representation before a learnable node-wise aggregation: $\tilde{\boldsymbol{A}}\boldsymbol{H}\boldsymbol{W}_l$, $(\boldsymbol{I} - \tilde{\boldsymbol{A}})\boldsymbol{H}\boldsymbol{W}_h$, $\boldsymbol{I}\boldsymbol{H}\boldsymbol{W}_i$

In spite of their advantageous capabilities, recent studies discover that heterophilous GNNs are naturally unsuitable for sampling-based minibatching, since their distant or full-graph information is heavily overlooked in batches built on locality [47]. Evaluations show that simply fitting these models to learn from induced subgraph samples causes performance degradation [42].

2.4.3 Decoupled Heterophilous GNNs

Applying the decoupling technique to heterophilous GNNs is non-trivial due to the full-graph relationships. Very few models fall into this classification at the current stage. To our knowledge, GPRGNN [39] and LINKX [42] are the only models conceptually similar to this scheme, but both remain sensitive to the graph scale.

GPRGNN performs learnable propagation on the output of feature transformation $\boldsymbol{H}^{(L)} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l \cdot \mathrm{MLP}(\boldsymbol{X})$. The adjacency matrix is inevitable for deriving weight parameters a_l. Therefore, GPRGNN is still regarded as a full-batch model.

LINKX alternatively exploits a simple architecture $\boldsymbol{H}^{(L)} = \mathrm{MLP}(\boldsymbol{X}\boldsymbol{W}_X \oplus \boldsymbol{A}\boldsymbol{W}_A)$, where $\oplus$ denotes representation fusion. Although it supports minibatching, the matrix $\boldsymbol{A}$ is involved as an input feature in learning. Hence it still suffers from $O(nF)$ model size and $O(mF)$ forward prediction time.

Chapter 3

Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization

3.1 Introduction

The modern years have witnessed the burgeoning of online services based on data represented by graphs, which leads to rapid increase in the amount and complexity of such graph data. Graph Neural Networks (GNNs) describe a set of neural networks that process data represented by graphs, and have achieved strong performance on graph understanding tasks such as node classification [3, 19, 22, 29], link prediction [20, 48–50], and community detection [51–53]. Recent studies have attempted to learn representations of large graphs such as the Microsoft Academic Graph (MAG) with 100 million entries [54, 55]. Nonetheless, directly fitting the basic models like GCN to such data would easily cause unacceptable training time or out-of-memory error.

This work is published as: Ningyi Liao*, Dingheng Mo*, Siqiang Luo, Xiang Li, Pengcheng Yin. "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". *Proceedings of the VLDB Endowment*, Vol. 15, No. 11, pp. 3240-3248, 2022.

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 [25, 27], which frees the GPU memory from storing entire graph data and reduces memory footprint. Such methods typically integrate graph data management techniques such as Personalized PageRank [33] 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 [19, 56], layer sampling [24, 29], and subgraph sampling [22, 23, 57]. 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 [16, 25], self-attention [20, 58, 59], and quantization [60].

Unfortunately, such methods are nevertheless not efficient enough when applied to million-scale or even larger graphs. According to [12], the very recent state-of-the-art algorithm GBP [11] 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, implying that the cost of such an approach is still prohibitively high to be applied in practice.

In this research, 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. On the practical side, to our knowledge, SCARA is the first GNN algorithm that can be applied to billion-scale 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 compositing the 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.

3.2 Method

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.

3.2.1 SCARA Model 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 models [11, 27], in precomputation stage we follow the idea of pre-propagation decoupling 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 Eq. (2.5). We enhance the model structure by incorporating skip connections [25] and dense connections [11] in every intermediate layers. Thence, it enjoys a $O(Ln_bF + LF^2)$ memory footprint and $O(LnF^2)$ time complexity during training and inference.

Since the propagation stage is the complexity bottleneck as mentioned earlier, we focus on reducing its computation complexity. We derive Eq. (2.4) in our propagation as:

$$\boldsymbol{P} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}_{(r)}^l \cdot \boldsymbol{X} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^l \boldsymbol{X}, \tag{3.1}$$

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, and utilize a symmetric normalization factor $a = 1-r, b = r, r \in [0, 1]$.

Our computation of Eq. (3.1) is displayed in Algorithm 3.1 (Feature-Push) and explained in detail in Section 3.2.2. 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 3.2 (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 3.1 while improving speed. We further describe this methodology in Section 3.2.3.

3.2.2 Feature-Push

Examining Eq. (3.1), 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 [25] and SGC [27] compute the propagation part separately from network training, resulting in a complexity at least proportional to edge size m. GBP [11] 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 [61, 62].

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 Eq. (3.1) can be written as the following to rearrange the normalization order:

$$\tilde{\boldsymbol{A}}_{(r)}^{l} \cdot \boldsymbol{X} = \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^{l} \boldsymbol{X} = \boldsymbol{D}^{r-1}\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^{l} \boldsymbol{D}^{1-r}\boldsymbol{X}. \tag{3.2}$$

Here, given the normalized features $\boldsymbol{D}^{1-r}\boldsymbol{X}$, single-source PPR algorithms can be alternated to efficiently propagate information with $(\boldsymbol{A}\boldsymbol{D}^{-1})^{l}$, 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 [12].

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 $\boldsymbol{\pi}_f = (\pi(\boldsymbol{x}_f;t_1), \cdots, \pi(\boldsymbol{x}_f;t_{n_U}))^{\top}$ 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 Eq. (3.1). Now that the feature PPR is explained, we here look into its calculation. We define the problem of feature PPR approximation:

Definition 3.1 (**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 $1 - \phi$,

$$|\pi(\boldsymbol{x}; t) - \hat{\pi}(\boldsymbol{x}; t)| \leq \lambda. \tag{3.3}$$

Recognizing that GNNs require less precise propagation information to achieve proper performance [63, 64], the approximate feature PPR enables employing efficient computation based on forward PPR algorithms without loss in eventual model effectiveness [61, 65]. We employ a scalable algorithm Feature-Push to compute the embedding matrix combining Forward Push [35] 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

Algorithm 3.1 Feature-Push

Input: Graph G, node set U, feature vector $\boldsymbol{x}$, probability α, convolution factor r, push parameter β

Output: Approximate embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$

for all $u \in U$ **do**
  $r'(\boldsymbol{x}; u) \leftarrow x(u) \cdot d(u)^{1-r}$
  $r(\boldsymbol{x}; u) \leftarrow r'(\boldsymbol{x}; u) / \sum_{u \in U} r'(\boldsymbol{x}; u)$
$\hat{\pi}(\boldsymbol{x}; t) \leftarrow 0$ for all $t \in U$
while exist $u \in U$ such that $r(\boldsymbol{x}; u) > r_{max}/d(u)$ **do**
  for all $v \in \mathcal{N}(u)$ **do**
    $r(\boldsymbol{x}; v) \leftarrow r(\boldsymbol{x}; v) + (1 - \alpha) \cdot r(\boldsymbol{x}; u)/d(u)$
  $\hat{\pi}(\boldsymbol{x}; u) \leftarrow \hat{\pi}(\boldsymbol{x}; u) + \alpha \cdot r(\boldsymbol{x}; u)$
  $r(\boldsymbol{x}; u) \leftarrow 0$
$r_{sum} \leftarrow \sum_{u \in U} r(\boldsymbol{x}; u), \quad N_W \leftarrow r_{sum}/\beta$
for all $u \in U$ such that $r(\boldsymbol{x}; u) \neq 0$ **do**
  Perform $\frac{r(\boldsymbol{x}; u)}{r_{sum}} \cdot N_W$ random walks from u
  for all random walk stopping at t **do**
    $\hat{\pi}(\boldsymbol{x}; t) \leftarrow \hat{\pi}(\boldsymbol{x}; t) + r_{sum}/N_W$
$\hat{\pi}(\boldsymbol{x}; t) \leftarrow \hat{\pi}(\boldsymbol{x}; t) \cdot d(t)^{r-1}$ for all $t \in U$
return $\hat{\boldsymbol{\pi}}(\boldsymbol{x}) \leftarrow (\hat{\pi}(\boldsymbol{x}; t_1), \cdots, \hat{\pi}(\boldsymbol{x}; t_{n_U}))^\top$

combination have been the state of the arts in various PPR benchmarks [61, 66]. We highlight that the differences between Algorithm 3.1 and [61, 66] 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 Eq. (3.2) and reusing in Eq. (3.8). 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.

As shown in Algorithm 3.1, the FEATURE-PUSH algorithm outputs the approximation of embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$ for input feature $\boldsymbol{x}$. 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 3.1), then conducts compressed random walks to save computation (line 10-14). We analyze each method 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 [62, 67], 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 Eq. (3.1), we apply the normalization by degree vector (lines 2 and 15 in Algorithm 3.1) to achieve the transformation in Eq. (3.2). It is worth noting that Algorithm 3.1 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 3.2 (**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 Theorem 3.4. It is the key parameter of FEATURE-PUSH, which balances absolute error guarantee and time complexity. Referencing the trade-off in [61], 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 3.1. In Algorithm 3.1, the forward push and random walk are combined in such form as line 14.

Derived from the single-source PPR analysis [35, 61], 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 3.1 feature by feature, the approximate calculation is also applicable to the PPR matrix containing multiple vectors:

Lemma 3.1. *Algorithm 3.1 produces an unbiased estimation $\hat{\pi}(\boldsymbol{x};t)$ of the value $\pi(\boldsymbol{x};t)$ satisfying Eq. (3.3). Repeating it for F times produces an unbiased estimation $\hat{\boldsymbol{P}}$ of the embedding matrix $\boldsymbol{P}$.*

Parallel Computation. Since Algorithm 3.1 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 3.1 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.

3.2.3 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, while in feature PPR there is similarity between different features. The feature-oriented calculation as Algorithm 3.1 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.

As a toy example, if we have the PPR $\boldsymbol{\pi}(\boldsymbol{b})$ for base feature vector $\boldsymbol{b} = (0.5, 0.5)$, and need to compute the PPR 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 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 elaborate in Theorem 3.4 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 $\boldsymbol{X}_B = [\boldsymbol{b}_1, \cdots, \boldsymbol{b}_{F_B}]$, $\boldsymbol{b}_f \in \boldsymbol{X}$. Then, the entire feature matrix $\boldsymbol{X}$ can be written as combinations of the bases:

$$\boldsymbol{X} = \boldsymbol{X}_B \cdot \boldsymbol{\Theta} + \boldsymbol{Z}, \tag{3.4}$$

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. Eq. (3.4) 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 Eq. (3.4), the approximate feature PPR on $\boldsymbol{X}$ can be acquired by the combinations as:

$$\check{\boldsymbol{P}} = \hat{\boldsymbol{P}}_B \cdot \boldsymbol{\Theta} + \hat{\boldsymbol{P}}_Z. \tag{3.5}$$

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:

$$\min\ \operatorname{rank}(\boldsymbol{Y}) + \eta \|\boldsymbol{Z}\|_1, \quad \text{s.t. } \boldsymbol{Y} + \boldsymbol{Z} = \boldsymbol{X}. \tag{3.6}$$

Eq. (3.6) 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) [68, 69] when $\eta = \frac{1}{\sqrt{n}}$, which can be effectively solved by convex optimization methods such as alternating direction [70]. In general, [68] discovers that the problem can be transferred into a pair of convex problems when only one term in the derived form of Eq. (3.6) 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 Eq. (3.6), 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 3.2 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 3.2 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 $\boldsymbol{Y}, \boldsymbol{Z}, \boldsymbol{E} \leftarrow \boldsymbol{0}$, $\mu \leftarrow 1/n_{U'}$

3 **while** $\|\boldsymbol{X}' - \boldsymbol{Y} - \boldsymbol{Z}\|_1 > \lambda\|\boldsymbol{X}'\|_1$ **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. 3.1 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. 3.1 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 3.2 corresponds to the RPCA iterative solution [68], 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 [71]. 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 Eq. (3.5) for further reuse in the following phase.

Calculation Reuse on Sparse Residue. Algorithm 3.2 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 Eq. (3.4), 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 $\hat{\boldsymbol{\pi}}(\boldsymbol{x}_f)$ on raw feature $\boldsymbol{x}_f$ can be constituted as line 17, reusing the PPR computation results of base features.

Approximation Precision. In Algorithm 3.2, 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 Theorem 3.1. 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 3.1? 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 Eq. (3.4) and Eq. (3.5) in our vector notation for a feature $\boldsymbol{x}_f$. For simplicity we omit the subscript f:

$$\boldsymbol{x} = \sum_{i=1}^{F_B} \theta_i \cdot \boldsymbol{b}_i + \boldsymbol{z}, \tag{3.7}$$

$$\check{\boldsymbol{\pi}}(\boldsymbol{x}) = \sum_{i=1}^{F_B} \theta_i \cdot \hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta_B) + \hat{\boldsymbol{\pi}}(\boldsymbol{z}, \beta_Z). \tag{3.8}$$

The following lemma depicts the precision constraint of $\check{\boldsymbol{\pi}}(\boldsymbol{x})$ in Eq. (3.8).

Lemma 3.2. *Given a feature vector $\boldsymbol{x}$, the ground truth of PPR vector is $\boldsymbol{\pi}(\boldsymbol{x})$, and the estimation output by Eq. (3.8) 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*

$$\beta_Z \leq \frac{\lambda^2 / \log(2/\phi) - 2\sum_{i=1}^{F_B} \theta_i \beta_B}{2\lambda/3 + 2}. \tag{3.9}$$

Proof. Similar to the theory in [33], feature PPR can also be interpreted as the solution of the following linear system:

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

which can be transformed to

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

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

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

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 Eq. (3.8) 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 3.1. In each such computation of Algorithm 3.1, 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

$$\mathbb{E}\left[\sum_{j=0}^{N_W(\boldsymbol{b}_i)} \frac{r_{sum}(\boldsymbol{b}_i)}{N_W(\boldsymbol{b}_i)} X_j(\boldsymbol{b}_i; t)\right] = \sum_{v \in V} r(\boldsymbol{b}_i; v) \cdot \pi(v, t). \tag{3.10}$$

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 [62], we can further acquire the precision guarantee of the PPR as:

$$\Pr[|\check{\pi}(\boldsymbol{x};t)-\pi(\boldsymbol{x};t)|>\lambda]\leq 2\cdot\exp(-\frac{\lambda^2 N_{sum}}{2\nu+2a\lambda/3}), \tag{3.11}$$

where the number of walks $N_{sum}=N_W(\boldsymbol{z})+\sum_{i=0}^{F_B}N_W(\boldsymbol{b}_i)$,

$$a=N_{sum}\cdot\max\{\tfrac{\theta_1 r_{sum}(b_1)}{N_W(b_1)},\cdots,\tfrac{\theta_{F_B} r_{sum}(b_{F_B})}{N_W(b_{F_B})},\tfrac{r_{sum}(\boldsymbol{z})}{N_W(\boldsymbol{z})}\},$$

and

$$\begin{aligned}\nu=&\frac{1}{N_{sum}}\sum_{i=0}^{F_B}\sum_{j=0}^{N_W(\boldsymbol{b}_i)}\left(\frac{\theta_i r_{sum}(\boldsymbol{b}_i)N_{sum}}{N_W(\boldsymbol{b}_i)}\right)^2\mathbb{E}[X_j(\boldsymbol{b}_i;t)]\\&+\frac{1}{N_{sum}}\sum_{j=0}^{N_W(\boldsymbol{z})}\left(\frac{r_{sum}(\boldsymbol{z})N_{sum}}{N_W(\boldsymbol{z})}\right)^2\mathbb{E}[X_j(\boldsymbol{z};t)].\end{aligned} \tag{3.12}$$

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 Eq. (3.10) into Eq. (3.12) 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 3.2, 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 Eq. (3.11) 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

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

□

Theorem 3.2 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 Eq. (3.9) 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 Theorem 3.2, 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 3.2, it satisfies the precision guarantee in Definition 3.1:

Lemma 3.3. *Given a feature set $\boldsymbol{X}$, for any feature vector $\boldsymbol{x}_f \in \boldsymbol{X}$, Algorithm 3.2 returns an approximate PPR vector $\check{\boldsymbol{\pi}}(\boldsymbol{x}_f)$, that any of its elements $\check{\pi}(\boldsymbol{x}_f;t)$ satisfies Eq. (3.3) 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 Eq. (3.9). According to Theorem 3.2 this lemma follows. □

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 3.2, i.e. line 10 and line 13 containing PPR calculations, the processing can be parallelized.

3.2.4 Complexity Analysis

We then develop theoretical analysis on the time and memory complexity of SCARA. For a single run of Algorithm 3.1, we have the following lemma:

Lemma 3.4. *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 3.1 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 [35]. For random walks on feature residue, we employ the complexity derived by [61] as $O(m \cdot r_{max}/\beta)$. Hence the overall running time of one query in Algorithm 3.1 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 $O(\sqrt{\frac{m\|\boldsymbol{x}\|_1}{\beta}})$. □

Utilizing Theorem 3.4, the time complexity of computing one feature PPR $\hat{\pi}(\boldsymbol{x}, \beta)$ with Algorithm 3.1 can be bounded by $O(\sqrt{m\|\boldsymbol{x}\|_1/\beta})$. To get PPR value with absolute error guarantee of λ, Algorithm 3.1 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 3.2, 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 3.1. By setting $\phi = 1/n$, we can derive the time overhead of SCARA precomputation. 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 3.5. *Time complexity of* SCARA *precomputation is bounded by* $O\left(F\sqrt{m \log n}/\lambda\right)$. *Memory complexity is* $O\left(nF\right)$.

3.3 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.

3.3.1 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 3.1. 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 3.1. 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 [3, 26] 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 as our baselines. GraphSAINT-RW [23] and GAS [24] are representative of different sampling-based algorithms. For post- and pre-propagation decoupling approaches, we respectively employ the most advanced PPRGo [26] and GBP [11]. 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 neural network architecture, we set layer depth $L = 4$, layer width $W = 2048$ and $W = 128$ for inductive and transductive tasks, respectively, to be

TABLE 3.1: 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	Feat. F	Classes N_c	Split	Prob. α	Conv. r	Common
PPI [19]	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 [23]	716,847	6,977,410	300	100 (m)	0.75/0.10/0.15 (i)	0.9	0.3	
Reddit [19]	232,965	114,615,892	602	41 (s)	0.01/0.04/0.96 (t)	0.5	0.5	
Amazon [22]	2,400,608	123,718,024	100	47 (s)	0.70/0.15/0.15 (i)	0.2	0.2	
MAG [55]	27,394,820	366,143,207	200	100 (m)	0.01/0.01/0.99 (t)	0.5	0.5	
Papers100M [72]	111,059,956	1,615,685,872	128	172 (s)	0.78/0.08/0.14 (t)	0.5	0.5	

aligned with optimal baseline results in [11]. 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 3.1 per dataset. We further analyze the settings of these parameters in Section 3.3.3.

3.3.2 Performance Comparison

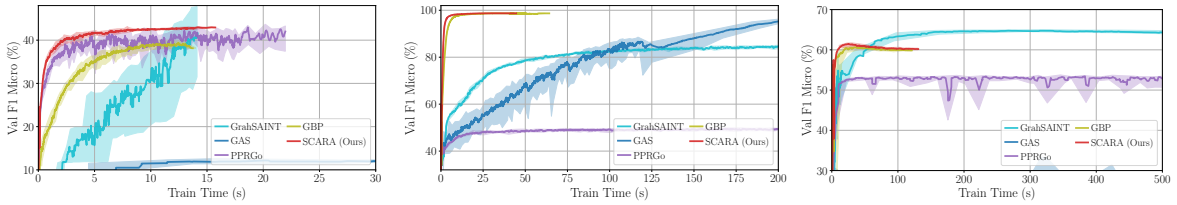
We evaluate the performance of SCARA and baselines in terms of both effectiveness and efficiency. Table 3.2 shows the average results of repetitive experiments on 6 large datasets, including the assessments on accuracy, memory, and the running time for different phases. Among them the key metric is learning time, which is summed up by precomputation and training times and presents the efficiency through the information retrieving process to acquire an effective model. The training curves are given in Figure 3.1.

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 stablely in all tasks and outputs comparable or better accuracy than other scalable competitors.

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 efficient 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

TABLE 3.2: 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. "OOM" stands for out of memory error. 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	–	(– –)	–	–	–	–	(– –)	–	–	–
GAS	1151	(– 1151)	**2.2**	14.0	38.2 ±0.3	–	(– –)	–	–	–	–	(– –)	–	–	–
PPRGo	79.4	(62.3 + 17.1)	29.1	9.4	41.5 ±2.3	711	(451 + 259)	85240	130	17.0 ±1.5	–	(– –)	–	–	–
GBP	138	(124 + 13.7)	13.5	7.9	38.8 ±0.3	663	(569 + 94.4)	1452	173	34.8 ±0.1	–	(– –)	–	–	–
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**	**35.7 ±0.9**
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	628	(– 628)	5.6	10.0	**99.4 ±0.0**	4844	(– 4844)	45.6	48.0	57.2 ±0.5	20453	(– 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



(A) Reddit (B) PPI (C) Yelp

FIGURE 3.1: 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.

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 our 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.

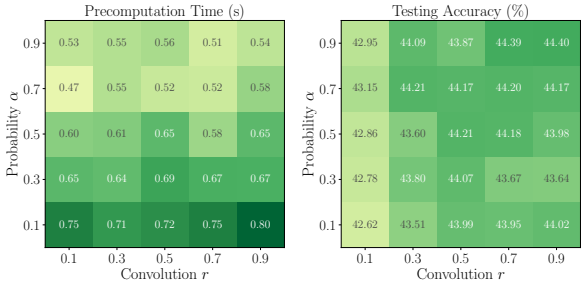
Figure 3.1 shows the validation F1-score versus training time on representative datasets and corresponding GNN models. It can be observed that when comparing the time consumption to convergence, the SCARA model is efficient in reaching the same precision faster than most methods. The performances of 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 Figure 3.1. It is worth noting that some baselines fail to or only partially converge before training terminates in tasks such as PPI.

3.3.3 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. The factor r controls the normalization strength of the degree matrix $\boldsymbol{D}$ as shown in Eq. (3.1). Especially, when $r = 0.5$, it degrades to the normalized adjacency matrix $\tilde{\boldsymbol{A}}$ presented in APPNP [25] and PPRGo [26]. The error bound λ determines the approximation push coefficient β in Algorithm 3.1. Hence, λ is used to configure the trade-off between precision and speed in precomputation and tends to be larger for better efficiency.

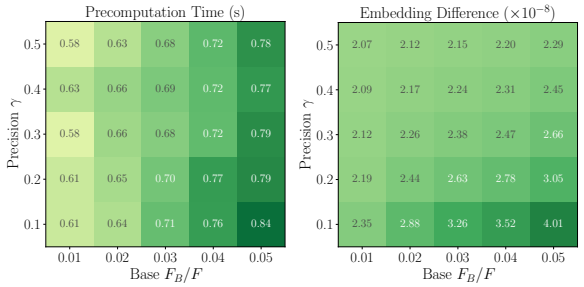
Regarding the default selection in Table 3.1, we set the values of α and r for shared datasets mainly in accordance to GBP [11, 12] in order to produce comparable results. The rest Reddit and MAG are employed with the following strategy: we use $r = 0.5$ for better comparison and generality, while α is decided based on graph edge density [26]. 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 Figure 3.2 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. This is because in feature PPR, a larger α indicates a higher probability of the propagation staying in the current node instead of further traveling to its neighbors. The accuracy is relatively not sensitive with variance inside the error range ($\pm 1\%$) as long as



(A) Precomp. Time (B) Testing Accuracy

FIGURE 3.2: Effect of propagation parameters teleport probability α and convolution coefficient r on SCARA **(a)** efficiency and **(b)** testing accuracy on Reddit dataset.



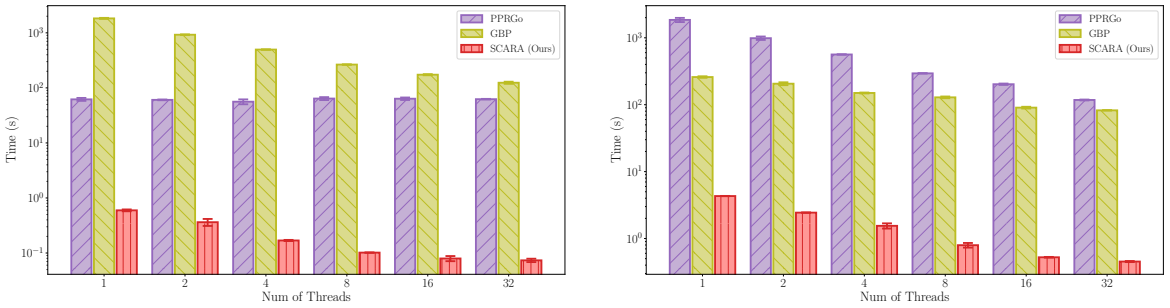
(A) Precomp. Time (B) Embedding Diff.

FIGURE 3.3: 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.

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.

Figure 3.3 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.



(A) Reddit (B) Amazon

FIGURE 3.4: 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.

3.3.4 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. Figure 3.4 displays the efficiency results with the number of threads ranging from 1 to 32.

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.

3.3.5 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 3.2. In following notation, SCARA is the model with Feature-Reuse in precomputation following Algorithm 3.2. We compare it with the bare iterative full-precision Feature-Push without Feature-Reuse. 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 3.3. Element-wise embedding value differences with regard to the Feature-Push result are also presented for the two reuse schemes.

By comparing the speed-up relative to Feature-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 Feature-Push.

Examining the reuse precision, it is inferred from Table 3.3 that Feature-Reuse causes no significant difference on model effectiveness, as minor accuracy fluctuations are under

Table 3.3: 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 Size	$\boldsymbol{F' = 100}$	$\boldsymbol{F' = 200}$	$\boldsymbol{F' = 400}$	$\boldsymbol{F' = 602}$
Precomp. Time	Feature-Push	0.38	0.77	1.52	2.37
	SCARA	0.14	0.24	0.46	0.65
Accuracy	Feature-Push	32.7	36.6	42.0	43.9
	SCARA	32.8	36.6	42.0	44.1
Embedding Difference	SCARA	0.4	0.3	0.6	2.4

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.

3.4 Summary and Discussion

In this work, 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.

We also note two potential extensions of our current model. Firstly, the SCARA model is designed for common homophilous graphs, where similar nodes tend to be connected with each other, and the Feature-Push propagation based on node neighbors is beneficial to the performance. However, not all graph datasets follow such assumption, and these graphs under heterophily would require specific approaches. Secondly, SCARA focuses on the precomputation improvement of pre-propagation decoupled model. Although demonstrating strong performance, the decoupling scheme is fixed with regard to architectural design, resulting in limited flexibility when adapting to different downstream tasks. It would be of potential interest to explore generalizing the SCARA optimization to other types of GNN models. In the next chapter, we propose a model attempting to address the first issue on heterophilous graphs.

Chapter 4

Scalable Heterophilous Graph Neural Network with Decoupled Embedding

4.1 Introduction

Graph Neural Networks (GNNs) combine graph processing techniques and neural networks to learn from graph-structured data, and have shown remarkable performance in recent advances of graph learning. Common GNN models rely on the principle of *homophily*, which assumes that connected nodes tend to be similar to each other in terms of classes [73]. This inductive bias introduces additional information from the graph structure and improves model performance in applicable tasks [74].

However, this assumption does not always hold in practice. A broad range of real-world graphs are *heterophilous*, where class labels of neighboring nodes usually differ from the ego node [75]. In such cases, the aggregation mechanism employed by conventional GNNs, which only passes messages from a node to its neighbors, may mix the information from non-homophilous nodes and cause them to be less discriminative. Consequently, the locality-based design is considered less advantageous or even potentially harmful in

This work is based on the work: Ningyi Liao, Siqiang Luo, Xiang Li, Jieming Shi. "LD^2: Scalable Heterophilous Graph Neural Network with Decoupled Embedding". Under submission of the *37th Conference on Neural Information Processing Systems*.

these applications [47, 76]. Various models have been proposed to address the heterophily problem, giving rise to a class of specialized GNNs known as heterophilous GNNs. Common strategies to address heterophily include discovering non-local or global graph relations [38, 39, 41, 43, 77, 78], and retrieving expressive node information through enhanced network architectures [32, 40, 44–46, 79].

Scalability has become a prominent concern in GNN studies. The ever-increasing sizes of graph data nowadays can easily exceed the memory limit of devices such as GPUs, rendering these solutions impractical for large-scale tasks [28, 80]. We observe that this issue is particularly critical in the context of heterophilous GNNs, due to an inherent conflict that most current models have not taken into account: heterophily-oriented designs usually rely on non-local information calculated by certain types of whole-graph operations. As the graph structure is involved, the time and memory overhead escalates substantially with the graph size. A recent investigation [42] reveals that all the evaluated full-graph GNNs run out of 24GB GPU memory when applied to the million-scale graph wiki (1.77M nodes, 244M edges). It is thus crucial to develop GNNs scalable to large graphs while retaining the capability for heterophily.

In this work, we examine the scalability problem and propose LD², a scalable GNN model for heterophilous graphs with Low-Dimension embeddings and Long-Distance aggregation. The model highlights simplicity by decoupling graph dependency from iterative computations and solely learning from multiple precomputed embeddings. Derived from node attributes and graph topology, these novel embeddings are able to aggregate node relations of varying objectives and distances in the graph into low-dimensional features. To facilitate the decoupled scheme, we specifically propose an algorithm to efficiently estimate all embeddings before training, which enjoys time complexity only linear to the graph scale and a guaranteed precision bound. After the precomputation, a simple but powerful multi-channel neural network is subsequently employed to learn from the extracted node features. Theoretical and empirical results showcase that the combination of embeddings effectively retrieves representations among heterophilous nodes. On the efficiency aspect, LD² benefits from its scalable design, including a straightforward minibatch scheme, optimal training and inference time, and superior memory utilization.

4.2 Method

In this section, we first present an overview of the LD² model in Section 4.2.1, then respectively motivate the selection of the adjacency embedding and feature embedding in Sections 4.2.2 and 4.2.3. Lastly, an end-to-end scalable algorithm, namely A²Prop, is proposed in Section 4.2.4 to efficiently and concurrently compute all the embeddings.

4.2.1 LD²: A Decoupled Heterophilous GNN

In order to achieve superior time and memory scalability for heterophilous GNNs, we employ the concept of decoupling, which removes the dependency of graph adjacency propagation in training iterations. The main idea of our model is first generating *embeddings* from raw *features* including node attributes and adjacency in a precomputation stage. Then, these embeddings are taken as inputs to learn *representations* by a simple neural network model. We embrace the multi-channel architecture [46, 81] to enhance flexibility, where the input data is a list consisting of embedding matrices $[\boldsymbol{P}_1, \boldsymbol{P}_2, \cdots, \boldsymbol{P}_C]$. Each embedding is separately processed and then merged in the network.

LD² utilizes diverse embeddings based on pure graph adjacency and node attributes, denoted as $\boldsymbol{P}_A(\boldsymbol{A})$ and $\boldsymbol{P}_X(\boldsymbol{X}, \boldsymbol{A})$, respectively. Both types of embeddings can be produced by our precomputation A²Prop following Algorithm 4.1. The initial layer of the LD²

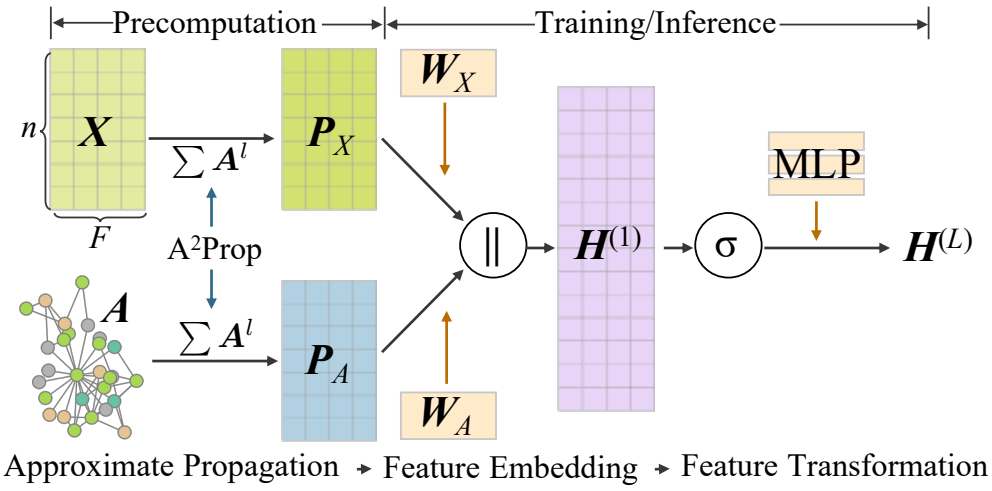


FIGURE 4.1: LD² framework: decoupled precomputation and training.

network applies a separate linear transformation to each embedding input, and the results are concatenated to form the representation matrix. Lastly, an L-layer MLP is leveraged for the classification task. The high-level framework of LD^2 is depicted in Figure 4.1 and can be expressed as follow: The overall framework can be expressed as follow:

$$\text{Precompute} : \boldsymbol{P}_A, \boldsymbol{P}_X = \text{A}^2\text{Prop}(\boldsymbol{A}, \boldsymbol{X}), \tag{4.1}$$

$$\text{Transform} : \boldsymbol{H}^{(L)} = \text{MLP}(\boldsymbol{P}_A\boldsymbol{W}_A \| \boldsymbol{P}_X\boldsymbol{W}_X). \tag{4.2}$$

Training/Inference Complexity. Our decoupled model design enables a simple on-demand minibatch scheme in training and inference, that only n_b rows corresponding to the batch nodes in the embedding matrices are loaded into GPU and processed by the network transformation. For LD^2 with C channels, the GPU memory footprint is therefore bounded by $O(L_C n_b F + L_C F^2)$. It is worth noting that such complexity does not depend on the graph scale n or m. Consequently, the training is freely configurable with an arbitrary GPU memory budget. Regarding computation operations, the time complexity of forward inference through the graph is $O(LnF^2)$, being just linear to n. As the memory and time complexity only contain essential operations of MLP transformation with no additional expense, this is the optimal scale with respect to the iterative training of GNN architectures.

4.2.2 Low-dimension Adjacency Embedding

Several studies reveal that, despite the feature information of nodes, the pure graph structure is equally or even more important in the context of heterophilous GNNs [38, 42, 75]. Particularly, the most informative aspects are often associated with 2-hop neighbors, i.e., "neighbors of neighbors" of ego nodes. [43] proves that even under heterophily, the 2-hop neighborhood is expected to be homophily-dominant. We thence intend to explicitly model such topological similarity.

The 2-hop relation can be described by the 2-hop adjacency matrix $\boldsymbol{A}^2$. Note that as the sparse matrix $\boldsymbol{A}$ has m entries, the number of entries in $\boldsymbol{A}^2$ is at the scale of $O(md)$, which indicates that directly applying 2-hop graph propagation in the training stage will demand even more expensive time and memory overhead to be scaled up. We instead propose an approximate scheme that seeks to prevent the 2-hop adjacency from repetitive

processing, and retrieves a low-dimensional but expressive embedding prior to training in the precomputation stage. In other words, we utilize the embedding to resemble 2-hop information which can be directly learned by the neural network transformation. Denote the F-dimensional embedding as $\boldsymbol{P}_A \in \mathbb{R}^{n\times F}$. We aim to minimize its approximation error in Frobenius norm ($\|\cdot\|_F$):

$$\boldsymbol{P}_A = \underset{\boldsymbol{P}\in\mathbb{R}^{n\times F}}{\arg\min} \ \|\boldsymbol{A}^2 - \boldsymbol{P}\boldsymbol{P}^T\|_F^2. \tag{4.3}$$

The solution to Eq. (4.3) can be derived from the eigendecomposition of the symmetric matrix $\boldsymbol{A}^2$, that $\boldsymbol{P}_A^* = \boldsymbol{U}|\boldsymbol{\Lambda}|^{1/2}$, where $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \cdots, \lambda_F)$ is the diagonal matrix with top-F eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_F$, and $\boldsymbol{U} \in \mathbb{R}^{n\times F}$ is the matrix consisting of corresponding orthogonal eigenvectors. The eigenvalues are also called *frequencies* of the graph, and large eigenvalues of the adjacency matrix refer to low-frequency signals in the graph spectrum.

Spectral Analysis. Let $A^2(u,v)$ be the entry (u,v) of matrix $\boldsymbol{A}^2$. Its diagonal degree matrix is $\boldsymbol{D}_2 = \mathrm{diag}(d_2(1), d_2(2), \cdots, d_2(n))$, $d_2(u) = \sum_{v\in V} A^2(u,v)$. Denote $\boldsymbol{P}_A(u)$ as the F-dimensional embedding vector of node u. We show that the embedding $\boldsymbol{P}_A^*$ defined by Eq. (4.3) is also the solution to the following optimization problem:

$$\boldsymbol{P}_A = \underset{\boldsymbol{P}\in\mathbb{R}^{n\times F}, \boldsymbol{P}^\top \boldsymbol{D}_2 \boldsymbol{P} = \boldsymbol{\Lambda}}{\arg\min} \sum_{u,v\in V} A^2(u,v)\|\boldsymbol{P}(u) - \boldsymbol{P}(v)\|^2. \tag{4.4}$$

This is because:

$$\begin{aligned}\sum_{u,v} A^2(u,v)\|\boldsymbol{P}(u) - \boldsymbol{P}(v)\|^2 &= 2\sum_u d_2(u)\|\boldsymbol{P}(u)\|^2 - 2\sum_{u,v} A^2(u,v)\boldsymbol{P}(u)\boldsymbol{P}(v) \\ &= 2\,\mathrm{tr}(\boldsymbol{P}^\top \boldsymbol{D}_2 \boldsymbol{P} - \boldsymbol{P}^\top \boldsymbol{A}^2 \boldsymbol{P}).\end{aligned}$$

As $\boldsymbol{P}^\top \boldsymbol{D}_2 \boldsymbol{P}$ is fixed, finding the minimum of Eq. (4.4) is equivalent to optimizing $\max_{\boldsymbol{P}} \boldsymbol{P}^\top \boldsymbol{A}^2 \boldsymbol{P}$, of which the solution is exactly $\boldsymbol{P}_A^*$ according to the property of eigenvectors. Equation (4.4) implies that, 2-hop neighbors $(u,v), t \in \mathcal{N}(u), v \in \mathcal{N}(t)$ in the graph will share similar embeddings $\boldsymbol{P}_A(u)$ and $\boldsymbol{P}_A(v)$.

In fact, the low-dimensional embedding $\boldsymbol{P}_A^*$ can be interpreted as the adjacency spectral embedding of the 2-hop graph $\boldsymbol{A}^2$. Graph spectral embedding is a technique concerning the low-frequency spectrum of a graph, and is employed in tasks such as graph clustering [82]. As $\boldsymbol{P}_A$ corresponds to the dominant eigenvalues of $\boldsymbol{A}^2$, the embedding provides an approximate representation of the 2-hop neighborhoods based on the overall graph topology.

Alternatively, if we regard the adjacency information solely as features input into the network like LINKX, $\boldsymbol{P}_A$ correlates to the uncentered principal components of matrix $\boldsymbol{A}$. Thus learning a linear transformation $\boldsymbol{P}_A\boldsymbol{W}_A$ with weight matrix $\boldsymbol{W}_A \in \mathbb{R}^{F\times F}$ is the low-rank approximation of $\boldsymbol{A}\boldsymbol{W}_{A0}$ where $\boldsymbol{W}_{A0} \in \mathbb{R}^{n\times F}$, but with less computational cost. Compared to other works attempting to generate graph embeddings based on graph geometric or similarity measures [33, 38, 45, 78, 83], our approach offers the advantages of lower dimensionality and efficient calculation as demonstrated in Section 4.2.4.

4.2.3 Long-distance Feature Embedding

Decoupling the node features through approximate propagation has been extensively studied in regular GNNs with various schemes [12, 25–27, 36, 37]. Nonetheless, these approaches are based on the homophily assumption and focus on local neighborhoods. In order to apply decoupled propagation to heterophilous graphs and exploit the multi-channel ability of our model, we formulate the general form of approximate propagation as the weighted sum of powers of a propagation matrix applied to the input feature:

$$\boldsymbol{P}_X = \sum_{l=1}^{L_P} \theta_l \boldsymbol{T}^l \boldsymbol{X}, \tag{4.5}$$

where examples of matrix $\boldsymbol{T}$ include $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{L}}$, which respectively correspond to aggregative and discriminative operations.

LD² jointly utilizes the following channels:

(1) *Inverse* summation of 1-hop Laplacian propagations where $\theta_l = 1$, $\boldsymbol{T} = \tilde{\boldsymbol{L}}$:

$$\boldsymbol{P}_{X,H} = \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} \tilde{\boldsymbol{L}}^l \boldsymbol{X};$$

(2) *Constant* summation of 2-hop adjacency propagations where $\theta_l = 1$, $\boldsymbol{T} = \bar{\boldsymbol{A}}^2$:

$$\boldsymbol{P}_{X,L2} = \frac{1}{L_{P,L2}} \sum_{l=1}^{L_{P,L2}} \bar{\boldsymbol{A}}^{2l} \boldsymbol{X};$$

(3) raw node attributes where $\theta_1 = 1, \theta_l = 0 (l > 1)$, $\boldsymbol{T} = \boldsymbol{I}$:

$$\boldsymbol{P}_{X,0} = \boldsymbol{X}.$$

Intuitively, the first two channels perform distinct propagations on node feature $\boldsymbol{X}$ as illustrated in Figure 4.2, and employ inverse or constant summation to aggregate multi-hop information, in contrast to the local *decaying* summation ($l \to \infty, \theta_l \to 0$) commonly adopted in homophilous GNNs. Hence, such summations are suitable for retrieving long-range information under heterophily. The raw matrix $\boldsymbol{X}$ is also directly used as one input channel to depict node identity, which is a ubiquitous practice known as skip connection, identity mapping, or all-pass filter in heterophilous GNNs [32, 41, 46, 84].

The inverse embedding $\boldsymbol{P}_{X,H}$ is based on the intuition that, as neighbors tend to be different from the ego node, their features are also dissimilar. Hence in propagation, the embedding of the ego node should contain the previous embedding of itself, as well as the inverse of adjacent embeddings, which is exactly the interpretation of propagating node features by graph Laplacian matrix $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$. The second embedding $\boldsymbol{P}_{X,L2}$ performs a 2-hop propagation through the graph and aggregates the results of multi-scale neighbors. It echoes the earlier statement on the importance of 2-hop adjacency $\bar{\boldsymbol{A}}^2$ from the feature aspect. Note that for high-order propagation here, adjacency matrix $\bar{\boldsymbol{A}}$ escaping self-loops is shown to be advantageous in capturing non-local homophily [43, 79].

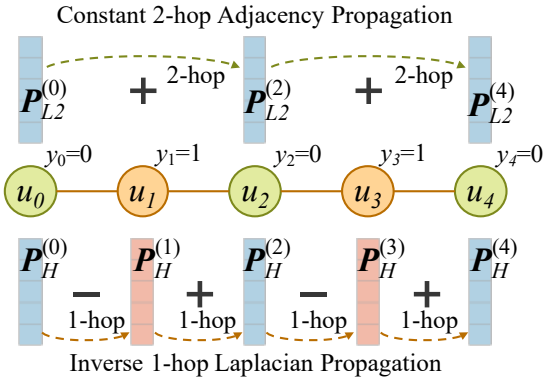


FIGURE 4.2: Two types of LD² propagations under heterophily.

Spectral Analysis. Assume that $\|\boldsymbol{X}(u)\| = \|\boldsymbol{P}(u)\| = 1$. We first examine the following regularization problem optimizing $\boldsymbol{P}$ based on input $\boldsymbol{X}$ for homophilous graphs [36]:

$$\boldsymbol{P}_{X,L} = \underset{\|\boldsymbol{P}(u)\|=1,\forall u\in V}{\arg\min} \sum_{u,v\in V} \tilde{A}(u,v)\|\boldsymbol{P}(u) - \boldsymbol{P}(v)\|^2 + \|\boldsymbol{P} - \boldsymbol{X}\|_F^2. \tag{4.6}$$

Differentiating the objective function with respect to $\boldsymbol{P}$ leads to:

$$(\boldsymbol{I} - \tilde{\boldsymbol{A}})\boldsymbol{P} - \boldsymbol{X} = 0.$$

Therefore the solution is:

$$\boldsymbol{P}^*_{X,L} = (\boldsymbol{I} - \tilde{\boldsymbol{A}})^{-1}\boldsymbol{X} = \sum_{l=0}^{\infty} \tilde{\boldsymbol{A}}^l \boldsymbol{X}.$$

In the implementation, a limited $L_{P,L}$-hop summation is used instead due to the over-smoothing issue that the infinite form converges to identical embeddings across nodes. This low-pass filter $\boldsymbol{P}_{X,L} = \frac{1}{L_{P,L}} \sum_{l=0}^{L_{P,L}} \tilde{\boldsymbol{A}}^l \boldsymbol{X}$ is investigated in S²GC [36] as an approach for balancing locality and multi-hop propagation. Its interpretation can be observed from its objective function Eq. (4.6), that it simultaneously minimizes the embedding difference of neighboring nodes as well as the approximation closeness to the input feature $\boldsymbol{X}$.

To obtain our first channel, we preferably introduce the low-frequency regularization to 2-hop adjacency, as 1-hop neighbors exhibit heterophily. Therefore, replacing $\tilde{A}(u,v)$ in Eq. (4.6) with $\bar{A}^2(u,v)$ yields our constant 2-hop embedding $\boldsymbol{P}_{X,L2}$. It shares similar spectral properties with S²GC for acting as a low-pass filter in 2-hop neighborhoods, while maintaining certain long-distance knowledge thanks to the multi-scale aggregation.

The other channel utilized in LD², i.e. the inverse Laplacian propagation, can be derived as:

$$\begin{aligned}
\boldsymbol{P}_{X,H} &= \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} \tilde{\boldsymbol{L}}^l \boldsymbol{X} \\
&= \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} (\boldsymbol{I}\boldsymbol{X} - \tilde{\boldsymbol{A}}^l \boldsymbol{X}) \\
&= \boldsymbol{X} - \boldsymbol{P}_{X,L}.
\end{aligned}$$

From the above analysis, $\boldsymbol{X}$ is the feature from all-pass filters for each hop, while $\boldsymbol{P}_{X,L}$ being the low-frequency information. The embedding $\boldsymbol{P}_{X,H}$ thus acts as a high-pass filter applied to the input feature, focusing on discriminative structures. In terms of spatial domain interpretation, high-frequency information corresponds to the embedding differences between the ego node and 1-hop neighbors [44]. It is noticeable that these three channels $\boldsymbol{P}_{X,L2}, \boldsymbol{P}_{X,H}, \boldsymbol{P}_{X,0}$ respectively represent low-pass, high-pass, and all-pass propagations through the graph while addressing heterophily. Combining them as inputs to the neural network benefits model performance by expressive information at various distances including identity, local, and global perspectives.

4.2.4 Approximate Propagation Precomputation

Conventionally, calculating the graph propagation $\tilde{\boldsymbol{A}} \cdot \boldsymbol{P}$ for an arbitrary feature matrix $\boldsymbol{P}$ is conducted by sparse-dense matrix multiplication. However, such an approach does not recognize the property of the adjacency matrix $\tilde{\boldsymbol{A}}$, that it can be represented by the adjacency list of nodes, and non-zero values in its data are solely determined by node degrees. Furthermore, since the propagation result is subsequently processed by the neural network, it is not necessary to be precise as the model is robust to handle noisy data [80, 85]. We first define the precision bound for approximate embedding:

Definition 4.1 (**Approximate Vector Embedding**). Given a relative error bound $0 < \epsilon < 1$, a norm threshold $\delta > 0$, and a failure probability $0 < \phi < 1$, the estimation $\hat{\boldsymbol{P}}(u)$ for an arbitrary embedding vector $\boldsymbol{P}(u)$ should satisfy that, for each $u \in V$ with $\|\boldsymbol{P}(u)\| > \delta$, such that with probability at least $1 - \phi$,

$$\|\boldsymbol{P}(u) - \hat{\boldsymbol{P}}(u)\| \leq \epsilon \cdot \|\boldsymbol{P}(u)\|. \tag{4.7}$$

Graph power iteration algorithm is the variant of power iteration particularly applied for calculating powers of adjacency matrix $\boldsymbol{A}$. In essence, the algorithm can be derived by maintaining a *residue* $\boldsymbol{R}^{(l)}(u)$ that holds the current l-hop propagation results for each node, and iteratively updating the next-hop residues of neighboring nodes $\boldsymbol{R}^{(l+1)}(v), v \in \mathcal{N}(u)$ for all nodes u. For each iteration, the *reserve* $\hat{\boldsymbol{P}}^{(l)}$ is also added up and converges to an underestimation of $\boldsymbol{P}$.

We propose Algorithm 4.1 for our specific scenario, namely Approximate Adjacency Propagation (A²Prop). Based on power iteration, our algorithm is greatly generalized to accommodate normalized adjacency, feature vectors for nodes, and a limited number of hops. We show that the algorithmic output can be bounded by Definition 4.1. For L_P iterations, denote the acceptable error per entry for push as δ_P, the matrix-wise absolute error is:

$$\|\boldsymbol{P} - \hat{\boldsymbol{P}}\|_{1,1} \leq \sum_{f=1}^{L_P} \sum_{f=1}^{F} \sum_{u \in V} d(u)\delta_P = L_P m F \delta_P.$$

By setting $\delta_P = \epsilon\delta / L_P m$, the estimation $\hat{\boldsymbol{P}}$ satisfies Definition 4.1.

Approximate Feature Embedding. The feature embedding in the form $\boldsymbol{P}_X = \sum_{l=0}^{L_P} \boldsymbol{T}^l \boldsymbol{X}$ can be computed by iteratively applying graph power iterations to the raw feature as initial residue $\boldsymbol{R}^{(0)} = \boldsymbol{X}$. The implicit propagation behavior is described by the matrix $\boldsymbol{T}$. For example, for Laplacian propagation $\boldsymbol{T} = \tilde{\boldsymbol{L}}$, the ego node u residue is updated by $\boldsymbol{R}^{(l+1)}(u) = \boldsymbol{R}^{(l+1)}(u) + \boldsymbol{R}^{(l)}(u)$, while its neighbors $v \in \mathcal{N}(u)$ are affected as $\boldsymbol{R}^{(l+1)}(v) = \boldsymbol{R}^{(l+1)}(v) - \boldsymbol{R}^{(l)}(u)/d^a(v)d^b(u)$. Hence we introduce a propagation coefficient

Algorithm 4.1 A²Prop: Approximate Adjacency Propagation

Input: graph G, feature matrix $\boldsymbol{X}$, max hop L_P, normalization factor a, b, propagation factor α_T, summation factor θ_l, push threshold δ_P

Output: adjacency embedding $\boldsymbol{P}_A$, feature embedding $\boldsymbol{P}_X$

$\boldsymbol{R}_A^{(0)} \leftarrow N(0,1)$, $\boldsymbol{R}_X^{(0)} \leftarrow \boldsymbol{X}$
for l from 0 to $L_P - 1$ **do**
  for all $u \in V$ such that $\|\boldsymbol{R}^{(l)}(u)\| > \delta_P$ **do**
    for all $v \in \mathcal{N}(u) \cap \{u\}$ **do**
      $\boldsymbol{R}_A^{(l+1)}(v) \leftarrow \boldsymbol{R}_A^{(l+1)}(v) + \alpha_A(u,v) \cdot \boldsymbol{R}_A^{(l)}(u)$
      $\boldsymbol{R}_X^{(l+1)}(v) \leftarrow \boldsymbol{R}_X^{(l+1)}(v) + \frac{\alpha_T(u,v)}{d^a(v)d^b(u)} \cdot \boldsymbol{R}_X^{(l)}(u)$
    $\boldsymbol{P}_X(u) \leftarrow \boldsymbol{P}_X(u) + \theta_l \cdot \boldsymbol{R}_X^{(l)}(u)$
  if $l \bmod 2 = 1$ and $l < L_P - 1$ **then**
    $\boldsymbol{P}_A \leftarrow \texttt{orthonormalize}(\boldsymbol{R}_A^{(l)})$
  empty $\boldsymbol{R}_A^{(l)}, \boldsymbol{R}_X^{(l)}$
$\boldsymbol{P}_A \leftarrow \boldsymbol{P}_A \cdot |(\boldsymbol{R}_A^{(L_P)})^\top \cdot \boldsymbol{P}_A|^{1/2}$
$\boldsymbol{P}_X \leftarrow \boldsymbol{P}_X + \theta_{L_P} \cdot \boldsymbol{R}_X^{(L_P)}$
return $\boldsymbol{P}_A, \boldsymbol{P}_X$

$\alpha_T(u,v)$, that $\alpha_L(u,u) = d^{a+b}(u), \alpha_L(u,v) = -1, v \in \mathcal{N}(u)$. For propagation $\tilde{\boldsymbol{A}}$ and $\bar{\boldsymbol{A}}$, the coefficient is just $\alpha_A(u,v) = 1$ with $\alpha_A(u,u) = 1, 0$, respectively.

In each iteration l, the reserve is updated after propagation according to the coefficient θ_l to sum up corresponding embeddings. Intuitively, one multiplication of $\bar{\boldsymbol{A}}^2$ is equivalent to two iterations of $\bar{\boldsymbol{A}}$ propagation. Hence for $\boldsymbol{P}_{X,L2}$ there is $\theta_l = l \bmod 2 = 0, 1, 0, 1, \cdots$ under the summation scheme in Algorithm 4.1. Since all embeddings we consider are constant, that is, $\theta_l \in \{0, 1\}$, the reserve can be simply increased without the rescaling terms in more general cases such as [12].

Approximate Adjacency Embedding. The adjacency embedding is represented by leading eigenvectors $\boldsymbol{P}_A = \boldsymbol{U}|\boldsymbol{\Lambda}|^{1/2}$. This eigendecomposition of $\boldsymbol{A}^2$ can be solved by the truncated power iteration: Initialize the $n \times F$ residue by i.i.d. Gaussian noise $\boldsymbol{R}^{(0)} = N(0,1)$. For each iteration l, firstly multiply the residue by $\boldsymbol{A}^2$ as $\boldsymbol{R}^{(l+1)} = \boldsymbol{A}^2\boldsymbol{R}^{(l)}$; then, perform column-wise normalization to the residue $\texttt{orthonormalize}(\boldsymbol{R}^{(l+1)})$ so that its columns are orthogonal to each other and of L2 norm 1. After convergence, the matrix satisfies $\boldsymbol{A}^2\boldsymbol{R}^{(L_P)} = \boldsymbol{R}^{(L_P)}\boldsymbol{\Lambda}$ within the error bound, which leads to the estimated output $\hat{\boldsymbol{U}} = \boldsymbol{R}^{(L_P)}, \hat{\boldsymbol{P}}_A = \hat{\boldsymbol{U}}|\hat{\boldsymbol{\Lambda}}|^{1/2}$.

Similarly, the 2-hop power iteration of $\boldsymbol{P}_A$ can be merged with those for $\boldsymbol{P}_X$ with a shared maximal iteration L_P, and orthonormalization is conducted every two $\boldsymbol{A}$ iterations. When the algorithm converges with error bound δ, the number of iteration follows $L_P = O(\log(F/\delta)/(1 - |\lambda_{F+1}/\lambda_F|))$. By selecting proper values for F and δ, the algorithm produces satisfying results within L_P iterations.

Precomputation Complexity. Since A²Prop serves as a general approximation for various adjacency-based propagations, the computation of all feature channels can be performed simultaneously in a single run. The memory overhead of the algorithm is mainly the residue and reserve matrices for C embedding channels, which is $O(CnF)$ in total. Note that A²Prop precomputation is performed in the main memory, and benefits from a less-constrained budget compared to GPU memory.

For each iteration, neighboring connections are accessed for at most m times. The time complexity of Algorithm 4.1 can thus be bounded by $O(L_P m F)$. Its loops over nodes and features can be parallelized and vectorized to reduce execution time. Moreover, the power iteration design is also amendable for further enhancements, such as reduction to

sub-linear complexity, better cache performance, and precision-efficiency trade-offs. We leave these potential improvements on A²Prop for future work.

4.3 Experimental Evaluation

We implement the LD² model and evaluate its performance from the perspectives of both efficacy and scalability. We mainly highlight key empirical results compared to minibatch GNNs on large-scale heterophilous graphs.

4.3.1 Experiment Setting

Datasets. We mainly perform experiments on large-scale heterophilous datasets [38, 42] for the transductive node classification task, with the largest available graph wiki ($m =$ 244M) included. We leverage settings as per [42] such as the random train/test splits and the induced subgraph testing for GSAINT-sampling models, while addressing several issues revealed by [86] before assessments. Statistics of these datasets are listed in Table 4.1, including the 1-hop and 2-hop node homophily scores for non-multilabel datasets. The empirical results support our analysis that regardless of heterophily, 2-hop neighbors in the graph tend to exhibit higher homophily.

TABLE 4.1: Dataset statistics and homophily scores. $\mathcal{H}_{n,1}$ and $\mathcal{H}_{n,2}$ are 1-hop and 2-hop homophilous scores, respectively.

Dataset	**Nodes** n	**Edges** m	d	F	N_c	**Notes**	$\mathcal{H}_{n,1}$	$\mathcal{H}_{n,2}$
squirrel [38]	5,201	401,907	77.275	2,089	5	–	0.217	0.214
penn94 [42]	41,536	1,403,756	33.796	4.814	2	–	0.504	0.478
arxiv-year [42]	169,343	1,327,142	7.837	128	5	directed	0.289	0.337
genius [42]	421,858	1,344,722	3.188	12	2	–	0.368	0.823
twitch-gamers [42]	168,114	6,965,671	41.434	7	2	–	0.562	0.531
pokec [42]	1,632,803	23,934,767	14.659	65	2	–	0.454	0.605
snap-patents [42]	2,738,035	16,705,984	6.101	269	5	directed	0.220	0.298
wiki [42]	1,770,981	244,278,050	137.934	600	5	–	0.306	–

Baselines. We focus on GNN models applicable to *minibatch* training in our evaluation regarding scalability, and hence most *full-batch* networks are excluded in the main experiments. Conventional baselines include MLP which only processes node attributes without considering graph topology, as well as PPRGo [26] and SGC [27] representing decoupled schemes for homophilous graphs. GCNJK [40] and MixHop [41] are GNNs under non-homophily. GSAINT random walk sampling [23] is utilized to empower them for minibatching. LINKX is the decoupled heterophilous GNN proposed by [42]. Simple i.i.d. node batching is adopted for decoupled networks.

Model and Training Hyperparameters. We particularly explore model hyperparameters including the number of layers L, i.e. model depth, and the number of hidden size, i.e. layer width, since these settings are mostly correlated with the efficacy and efficiency performance of models. For minibatch training, we comprehensively tune the hyperparameters of batch size and learning rate among baselines to produce comparable performance. We exploit the validation set to select the training epoch with best validation accuracy, and use early stopping if the model training converges.

We select above hyperparameters based on the following principle: We first refer to their original papers and implementations and explore model depth and width, in order to achieve relatively optimal reproduced performance. Then we select the largest batch size applicable to the GPU while preventing out of memory error for efficiency consideration. Other hyperparameters including weight decays and learning rates are tuned accordingly. For other architectural and training settings, we mostly follow the implementation in [42] when applicable, in order to produce similar evaluation to the benchmark.

Evaluation Metrics. We uniformly use classification accuracy on the test set to measure network effectiveness. Note that since the datasets are updated and the minibatch scheme is employed, results may be different from their original works. In order to evaluate scalability performance, we conduct repeated experiments and record the network training/inference time and peak memory footprint as efficiency metrics. For precomputed methods, we consider the learning process combining both precomputation and training. Evaluations are conducted on a machine with 192GB RAM, two 28-core Intel Xeon CPUs (2.2GHz), and an NVIDIA RTX A5000 GPU (24GB memory).

TABLE 4.2: Average test accuracy (%) of minibatch LD² and baselines on heterophilous datasets. "> 12h" means the model requires more than 12h clock time to produce proper results. Respective results of the first and second best performances on each dataset are marked in **bold** and underlined fonts.

Dataset	squirrel	genius	penn94	arxiv-year	twitch	pokec	snap-patents	wiki
MLP	33.16 ±0.59	82.47 ±0.06	74.41 ±0.48	37.23 ±0.31	61.26 ±0.19	61.81 ±0.07	23.03 ±1.48	35.64 ±0.10
PPRGo	33.95 ±0.49	79.81 ±0.00	58.75 ±0.31	39.35 ±0.12	47.19 ±2.26	50.61 ±0.04	(> 12h)	(> 12h)
SGC	59.39 ±0.62	79.85 ±0.01	68.31 ±0.27	43.40 ±0.16	57.05 ±0.21	56.58 ±0.06	37.70 ±0.06	28.12 ±0.08
GCNJK-GS	27.63 ±4.72	80.65 ±0.07	65.91 ±0.16	48.26 ±0.64	59.91 ±0.42	59.38 ±0.21	33.64 ±0.05	42.95 ±0.39
MixHop-GS	33.24 ±2.44	80.63 ±0.04	75.00 ±0.37	49.26 ±0.16	61.80 ±0.00	64.02 ±0.02	34.73 ±0.15	45.52 ±0.11
LINKX	60.14 ±0.92	82.51 ±0.10	**78.63** ±0.25	**50.44** ±0.30	64.15 ±0.18	68.64 ±0.65	52.69 ±0.05	50.59 ±0.12
LD² (ours)	**66.87** ±0.02	**85.31** ±0.06	75.52 ±0.10	50.29 ±0.11	**64.33** ±0.19	**74.93** ±0.10	**58.58** ±0.34	**52.91** ±0.16

TABLE 4.3: Time and memory overhead of LD² and baselines on large-scale datasets. "Learn", "Infer", and "Mem." respectively refer to minibatch learning and inference time (s) and peak GPU memory (GB). Precomputation time is appended when applicable. "> 12h" means the model requires more than 12h clock time to produce proper results. Respective results of the first and second best performances among heterophilous models per metric are marked in **bold** and underlined fonts.

Dataset	twitch-gamers			pokec			snap-patents			wiki		
	Learn	Infer	Mem.	Learn	Infer	Mem.	Learn	Infer	Mem.	Learn	Infer	Mem.
MLP	6.36	0.02	0.61	47.86	0.11	13.77	27.39	0.28	9.33	133.55	0.62	18.15
PPRGo	10.46+15.88	0.41	9.64	121.95+56.11	2.69	3.82	(> 12h)			(> 12h)		
SGC	0.09+0.74	0.01	0.28	1.05+8.08	0.01	0.28	4.94+23.54	0.01	0.42	12.66+7.98	0.01	0.52
GCNJK-GS	71.48	0.02*	7.33	27.33	0.09*	9.03	19.02	0.23*	9.21	95.52	0.69*	16.36
MixHop-GS	52.12	0.01*	1.49	71.35	0.03*	12.91	45.24	0.16*	19.58	84.22	0.23*	16.28
LINKX	10.99	0.19	2.35	28.77	0.33	9.03	39.80	0.22	21.53	180.71	1.14	14.53
LD² (ours)	0.85+**1.96**	**0.01**	**1.44**	17.95+**6.18**	**0.01**	**3.82**	31.32+**6.96**	**0.02**	**3.96**	28.12+**6.50**	**0.01**	**4.47**

* Inference time of GSAINT sampling is not precise since they are conducted on induced subgraphs smaller than the raw graph.

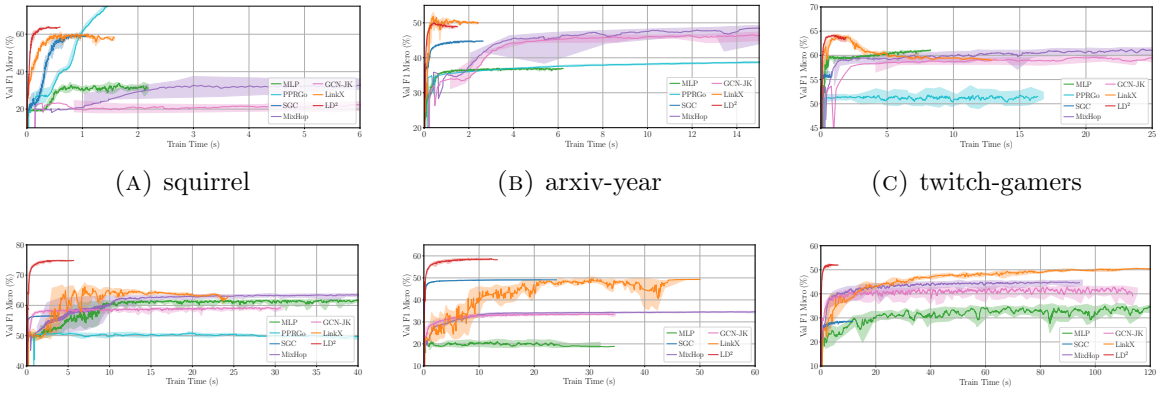
4.3.2 Performance Comparison

The main evaluations of LD² and baselines on 8 large heterophilous datasets are presented in Tables 4.2 and 4.3 for effectiveness and efficiency metrics, respectively. As an overview, our model demonstrates its scalability in completing training and inference with fast running time and efficient memory utilization, especially on large graphs. At the same time, it achieves comparable or superior prediction accuracy against the state-of-the-art minibatch heterophilous GNNs in most datasets.

Time Efficiency. More specifically, compared to heterophilous benchmarks on the four largest graphs with million-scale data, LD² speeds up the minibatch training process by 3–15 times, with an acceptable precomputation cost. Its inference time is also consistently below 0.1 seconds. The outstanding efficiency of LD² is mainly attributed to the simple model architecture that removes graph-scale operations while ensuring rapid convergence. In contrast, the execution speeds of MixHop and LINKX are highly susceptible to node and edge sizes, given their design dependency on the entire input graph. The extensive parameter space also causes them to converge slower, necessitating relatively longer training times. PPRGo shows limited scalability due to the costly post-transformation propagation. The superiority of LD² efficiency even holds when compared to simple methods such as MLP and SGC, indicating that the model is favorable for incorporating additional heterophilous information with no significant training overhead. The empirical results affirm that LD² exhibits optimized training and inference complexity at the same level as simple models.

Memory Footprint. LD² remarkably reduces run-time GPU memory consumption. As the primary overhead only comprises the model parameters and batch representations, it enables flexible configuration of the model size and batch size to facilitate powerful training. Even for the largest graph wiki with $n = 1.77$M and $F = 600$, the footprint remains below 5GB under our hyperparameter settings. Other heterophilous GNNs, though adopting the minibatch scheme, experience high memory requirements and even occasionally encounter out-of-memory errors during experiments, as their space-intensive graph propagations are executed on the GPU. Consequently, when the graph scales up, they can only be applied with highly constrained model capacities to conserve space, potentially resulting in compromised performance.

Test Accuracy. With regard to efficacy, LD² achieves top testing accuracy on 6 out of 8 heterophilous graphs and comparable performance on the remaining ones. It also consistently outperforms the sampling-based GCNJK and MixHop, as well as conventional GNNs. Particularly, by extracting embeddings from not only node features but pure graph topology as well, LD² obtains significant improvements over feature-based networks on datasets such as squirrel, genius, and wiki, demonstrating the importance of pure graph information in heterophilous learning. We deduce that the relatively suboptimal accuracy on penn94 may be correlated with the difficulty of fitting one-hot encoding



(A) squirrel (B) arxiv-year (C) twitch-gamers

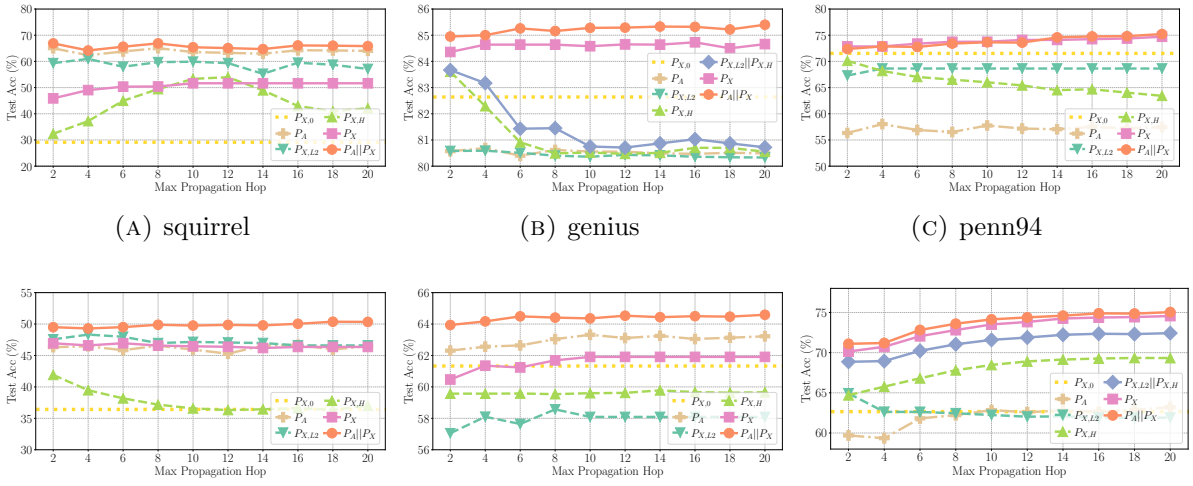
(D) pokec (E) snap-patents (F) wiki

FIGURE 4.3: Validation accuracy convergence curves of minibatch LD² and baseline models on 6 heterophilous datasets. Curves only represents the process of the training phase. Shaded area is the result range of multiple runs.

features into informative embeddings. Consistent with the previous studies [42], regular GNN baselines suffer from performance loss on most heterophilous graphs, while MLP achieves comparably high accuracy when node attributes are discriminative enough. For non-homophilous models GCNJK and MixHop, the minibatch scheme hinders them from reaching higher results because of the neglect of their full-graph relationships.

Convergence Curve. To examine the effect of model and training settings, in Figure 4.3, we display the model convergence curve, i.e. validation accuracy versus training time on heterophilous datasets and minibatch models corresponding to Table 4.2. It can be obviously observed that LD² outperforms other baseline methods on most datasets, demonstrating more stable curve, faster convergence, and significantly shorter overall training time. It is worth noting that the convergence of some baselines is beyond the display scopes in Figure 4.3.

Among other baselines, on small graphs, LINKX is relatively fast compared to GCNJK and MixHop which generally take more time per epoch. However, its large parameter space results in unstable performance, and hence requires more epochs to converge. When the graph scales larger, the efficiency of LINKX degrades due to its full-graph dependency. For simple and non-heterophilous models, though the decoupling design benefits them for less epoch time, their accuracies are suboptimal, and hence experience more training epochs than LD². Particularly, the PPRGo model is so large that it overfits on validation sets of small graphs such as squirrel.



(A) squirrel (B) genius (C) penn94

(D) arxiv-year (E) twitch-gamers (F) pokec

FIGURE 4.4: Effect of A²Prop propagation hops on the effectiveness of different adjacency and feature embedding channels and their combinations on 6 heterophilous datasets.

4.3.3 Effect of Propagation Channels and Parameters

To gain deeper insights into the multi-channel embeddings of LD², we specifically present the results of learning on separate inputs in Figure 4.4, where we explore the range of L_P in $[2, 4, 6, \cdots, 20]$ for each each embeddings per dataset. To study the long-distance information retrieval ability of our model, we also evaluate the effect on different embedding combinations, such as $P_{X,L2}\|P_{X,H}$, $P_X = P_{X,0}\|P_{X,L2}\|P_{X,H}$, and $P_X\|P_A$. These partial combinations are similarly input into the MLP for training following Eq. (4.2). Comparison among different channel combinations is useful for studying the effect of each embedding channel.

It can be observed that different graphs imply varying patterns when embedding channels and propagation hops are changed. For the genius dataset where raw node attributes already achieve an accuracy above 82%, applying the other two feature embeddings further improves the result. While the adjacency embedding alone shows secondary performance, integrating it with other channels proves beneficial. In comparison, on pokec, it is the inverse embedding $\boldsymbol{P}_{X,H}$ that becomes the key contributor, and larger hops produce better results. Generally, as the graph scale increases, employing more propagation hops becomes advantageous in capturing distant information. The effect of the inverse embedding $\boldsymbol{P}_{X,H}$ decreases when adding multiple hops in graphs such as genius, pokec, penn94, and arxiv-year. However, on the small heterophilous graph squirrel, it reaches maximum when

$L_P = 10$, indicating that non-local inter-node relationships are beneficial in this case. The observation supports our design that by adopting multi-channel and long-distance embeddings, LD² is powerful in capturing implicit information of various frequencies and scales that is important in the presence of heterophily.

4.3.4 Case Study

In order to intuitively illustrate the effect of approximate propagation used in LD², here we consider a toy example. Figure 4.5 depicts a graph with 9 nodes and 10 edges. Its nodes belong to 3 classes, and the connections are mostly heterophilous.

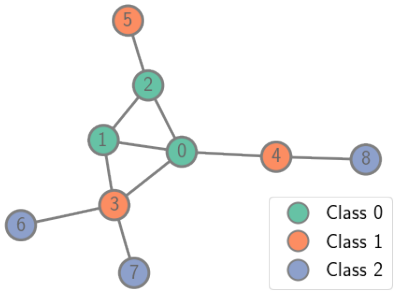


FIGURE 4.5: An example heterophilous graph where $\mathcal{H}_{n,1} = 0.204$.

We specifically focus on the inverse Laplacian propagation $\boldsymbol{P}_{X,H} = \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} \tilde{\boldsymbol{L}}^l \boldsymbol{X}$, as the 2-hop propagation is not suitable for such a small graph.

We first consider the $F = 3$ feature distribution with values in $[-1, 0, 1]$ as shown in the left side of Figure 4.6. Nodes inside the same class are of the same value in each feature

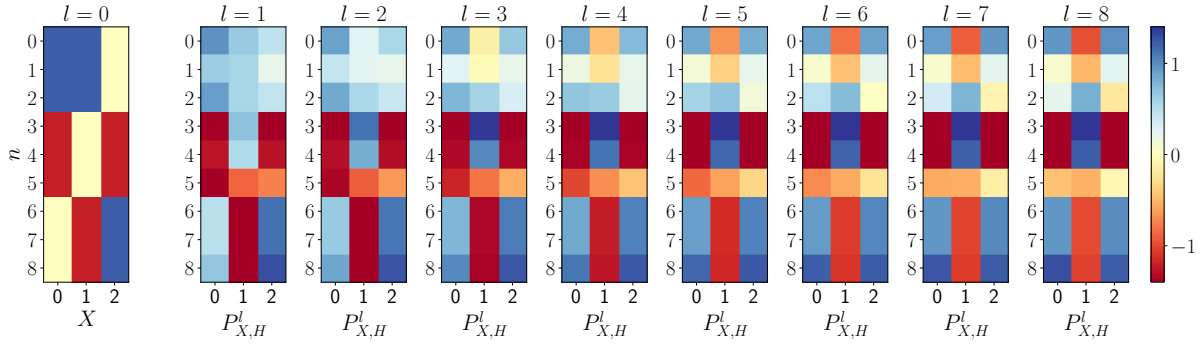


FIGURE 4.6: Progression of inverse Laplacian embedding with increasing propagation hops on given positive-negative distribution raw feature.

dimension. We then perform $l = 1$ to 8 times of propagation and illustrate the embedding in the figure. It can be interpreted that the propagation is useful in assigning inverse values to neighboring nodes based on the current ego node embedding. When the number of hops increases, the embedding gradually converges in each feature dimension. It is intuitive in the figure that, with proper steps of propagation, such as $l = 3$ or 4, it is easy to distinguish nodes in different classes, that their embeddings show different patterns. The Laplacian propagation procedure is hence useful for classifying heterophilous nodes in this case.

In another example, we investigate the one-hot style node feature, where nodes in the same class are assigned with 1 for one feature, and 0 for others. No negative value exists in the raw feature. In this case, the embedding produced by Laplacian propagation quickly converges. When setting $l = 3$ or 4, it is difficult to distinguish class 0 and 2, since all their nodes exhibit a similar pattern of having negative values in feature dimension $F = 0, 2$ and positive values in $F = 1$.

The example illustrates the propagation procedure of the heterophilous filter. We intend to use the case study to explain the difficulty of LD² adapting to certain patterns of input features, such as one-hot encoding. We believe this is partially the reason that LD² with only feature embeddings achieves suboptimal accuracy on graphs with such one-hot features including squirrel and penn94.

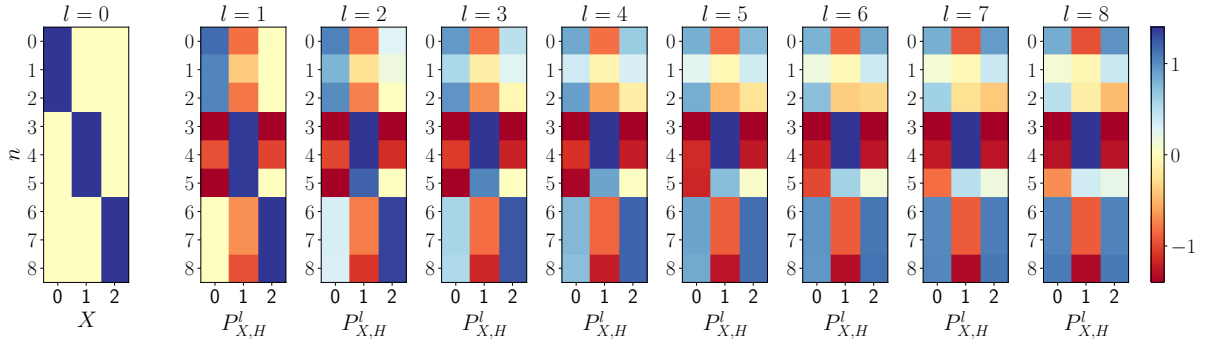


FIGURE 4.7: Progression of inverse Laplacian embedding with increasing propagation hops on given one-hot distribution raw feature.

4.4 Summary and Discussion

In this work, we propose LD², a scalable GNN for heterophilous graphs, which leverages long-distance propagation to capture non-local relationships among nodes, and incorporates low-dimensional yet expressive embeddings for effective learning. The model decouples full-graph dependency from iterative training, and adopts an efficient precomputation algorithm for approximating multi-channel embeddings. Theoretical and empirical evidence demonstrates its optimized training characteristics, including time efficiency with complexity linear to $O(n)$, and GPU memory independence from the graph size n and m. As a noteworthy result, LD² successfully applies to million-scale datasets under heterophily, with learning times as short as 1 minute and GPU memory expense below 5GB.

In the experiments, we observe that LD² exhibits varying performance on graphs with different types of features.In Figure 4.4 we display the effects of these feature embeddings when changing propagation steps, and in Section 4.3.4 we examine a toy model attempting to explain the reason behind the varying propagations. We think that the propagation of LD² may be less effective for generating expressive embeddings from certain types of features. As mentioned in Section 4.2.4, the complexity of our precomputation algorithm A²Prop is $O(L_P mF)$. The evidence indicates that the efficiency bottleneck of the precomputation lies in the linear dependency on the graph and feature size in the algorithm.

Given these current limitations, we believe that efforts towards more robust precomputation schemes and better adaptability to diverse features could further enhance the non-homophilous model in the future. Although the A²Prop is efficient in implementation, we do recognize that there are graph centrality algorithms and decoupled GNN precomputations reaching sub-linear complexity. A²Prop is potentially configurable for these enhancements. Secondly, various data augmentation approaches are able to transfer the one-hot features to other feature distributions, for instance, by using an embedding model or a simple MLP. Applying LD² to propagate the augmented embeddings is a promising way to address its limitation on feature distribution.

Chapter 5

Conclusion and Future Works

5.1 Conclusion

Recent advances in data processing have stimulated the demand for learning graphs of very large scales. GNNs, being an emerging and powerful approach in solving graph learning tasks, are known to be difficult to scale up. In this work, we present the current progress and result of our efforts towards scaling up GNNs to large graph.

A literature review is firstly conducted in Chapter 2, where we start from the basic design of vanilla GNN architectures, and perform a thorough analysis on the complexity and computation bottleneck of such design. Then, we subsequently review various existing approaches on improving the GNN efficiency and scalability, introducing the sampling and decoupling techniques. Lastly, we investigate the scalability issue on heterophilous graphs.

In Chapter 3, we notice that 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. Our solution to the bottleneck is SCARA, a scalable GNN with decoupling propagation and feature-oriented optimization. By integrating advanced graph management techniques, we propose the precomputation algorithms Feature-Push and Feature-Reuse, which is decoupled with the scale of graph node size and employs efficient computation from the feature perspective. We perform theoretical analysis to derive the precision guarantee as well as the sub-linear complexity of SCARA. Comprehensive empirical evaluations

demonstrate the significant improvement of SCARA learning time compared to state-of-the-art models, while it is also superior with fast inference speed, small memory overhead, and comparable or improved accuracy. We also discuss the effect of parameters and different algorithmic settings in detail.

In Chapter 4, we especially investigate the heterophilous graphs. We discover that most existing heterophilous models incorporate iterative full-graph computations to capture node relationships. These approaches have limited application to large-scale graphs due to their high computational costs and challenges in adopting minibatch schemes. To address the problem, we propose LD^2, a scalable GNN model for heterophilous graphs with Low-Dimension embeddings and Long-Distance aggregation. We introduce the decoupling scheme to non-homophilous settings with augmentations such as multi-channel embeddings and multi-hop propagations. Based on the complexity analysis, we subsequently design embeddings for effectively representing pure topology and node attribute data under heterophily. The A^2Prop precomputation algorithm is derived with vector-wise precision guarantee as well as adaptability to the multiple heterophilous embeddings. To evaluate the model effect, we conduct experiments compared to other minibatch baselines on 8 large heterophilous datasets, and show that the model is capable for realize impressive speed and memory optimization while achieving good accuracy. Parameter exploration and case study are performed to further validate the effectiveness of our design.

5.2 Future Works

5.2.1 Scaling Up a Broader Range of Models

As demonstrated in Chapters 3 and 4, by integrating scalable techniques and simple GNN models, we significantly improve the efficiency performance of decoupled models on common, i.e. homophilous, graphs as well as heterophilous variants. Our analysis also indicates that the ability of pre-propogation models may be constrained by certain inherent limitations, such as difficulties in generality and flexibility. Nonetheless, we do note that our current approaches may still be applicable when introduced to a more general range of GNN architectures. We here list two potential candidates.

The iterative- and post-propagation GNNs under the message passing framework are still widely used today [20, 25]. As these models execute more propagation operations during training and inference, their performance are more sensitive to the propagation efficiency. Currently, most of these models follow the naive matrix multiplication as graph propagation. However, as we have shown in Chapter 3, such calculation can be effectively approximated without affecting eventual accuracy. In addition, a precision guarantee can be provided by exploiting the random walk scheme. Hence, introducing approximate propagation to iterative- and post-propagation architectures is a promising direction to widen the application scenario of our techniques while benefiting the scalability of more mainstream GNN models.

Another popular type of scalable solutions rely on the sampling scheme, since they can be easily fit to a wide range of GNN models and applications without heavy modification. We notice that few advanced graph algorithms with favorable scalability properties have been applied in the GNN operations [57]. We are thence motivated in introducing graph partitioning algorithms such as [87] to further optimize the sampling process with higher efficiency and better usage of graph information. For example, by incorporating methods efficiently finding subgraphs based on node connectivity and neighboring information, computational expensive models such as GAT [20, 58] and Graph Transformer [88, 89] may be alleviated and achieve better scalability.

5.2.2 Benchmarking GNN Scalability and Efficiency

Several existing papers survey the GNN performance by evaluating on corresponding benchmarks [72, 90–92]. However, most of these works focus on the efficacy of GNN models, and the efficiency aspect is often overlooked. As recent research stresses more on the importance of GNN scalability, it becomes challenging to balance the efficiency while maintaining model performance. Reviews suggest that the gains of scalability may come with the price of corrupting graph completeness and losing training information [9, 93]. Hence it is a meaningful task of evaluating GNNs based on both effectiveness and efficiency at the same time.

One possible research direction lies in the robustness of GNN learning. Researchers reveal that neural networks have certain robustness against random noise [94]. In the case of

GNN, the performance of trained models is marginally affected by graph perturbation such as nodes or edges changes [63, 64]. In other words, there is a possibility that some graph operations can be less precise, while the potential of neural networks can be exploited without greatly sacrificing GNN accuracy, as the approximate propagation we shown in Chapters 3 and 4. Performing quantitative evaluation on such property enlightens a possible path to GNN design that is both effective and efficient, which will be of great interest to the community.

5.2.3 Exploring Scalable GNN on Graph Variants

In Chapter 4 we address the scalability issue by introducing the decoupled model to heterophilous data, which is a category of graphs attracting attentions in recent years. There are, however, a much wider range of graphs classified as different variants, such as heterogeneous graphs and dynamic graphs. Studies towards scaling up these graphs are also welcomed by applications based on these variants.

There are few studies focusing on GNN training on dynamic graphs. A related category is Spatial-Temporal GNNs (STGNNs) that learn node attributes and dependencies with sequential inputs changing dynamically over time [95]. The general thought of STGNNs is to aggregate spatial information with basic GNNs layers, and extract temporal sequences with RNN structures simultaneously. However, these works often cannot achieve desired performance in an online fashion, as they are based on classic GNNs that are known to have a high training cost. A very recent paper [96] introduces the decoupled model into dynamic graphs, paving new possibility of enhancing scalability of GNNs on these types of graphs.

References

[1] Danfei Xu, Yuke Zhu, Christopher B Choy, and Li Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5410–5419, 2017.

[2] Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. Graph r-cnn for scene graph generation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 670–685, 2018.

[3] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[4] Diego Marcheggiani and Ivan Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1506–1515, 2017.

[5] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In *International Conference on Learning Representations*, 2018.

[6] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, pages 3634–3640, 2018.

[7] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pages 4502–4510, 2016.

[8] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[9] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 1 2021.

[10] Zulun Zhu, Jiaying Peng, Jintang Li, Liang Chen, Qi Yu, and Siqiang Luo. Spiking Graph Convolutional Networks. In *31th International Joint Conference on Artificial Intelligence*, may 2022.

[11] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji Rong Wen. Scalable graph neural networks via bidirectional propagation. *33rd Advances in Neural Information Processing Systems*, 2020.

[12] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. Approximate graph propagation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1686–1696, Virtual Event Singapore, August 2021. ACM. doi: 10.1145/3447548.3467243.

[13] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. ISSN 10459227.

[14] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations*, 2016.

[15] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 7th International Joint Conference on Natural Language Processing*, pages 1556–1566, 2015.

[16] James Atwood and Don Towsley. Diffusion-convolutional neural networks. *29th Advances in Neural Information Processing Systems*, pages 2001–2009, 2016. ISSN 10495258.

[17] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *2nd International Conference on Learning Representations*, 2014.

[18] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29:3844–3852, 2016.

[19] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

[20] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *8th International Conference on Learning Representations*, 2017.

[21] Afshin Rahimi, Trevor Cohn, and Timothy Baldwin. Semi-supervised user geolocation via graph convolutional networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2009–2019, Melbourne, Australia, 7 2018. Association for Computational Linguistics.

[22] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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*, pages 257–266, 2019.

[23] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based learning method. In *International Conference on Learning Representations*, 2019.

[24] Matthias Fey, Jan E. Lenssen, Frank Weichert, and Jure Leskovec. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *38th International Conference on Machine Learning*. PMLR 139, 2021.

[25] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. *7th International Conference on Learning Representations*, pages 1–15, 2019.

[26] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2464–2473, Virtual Event CA USA, August 2020. ACM. doi: 10.1145/3394486.3403296.

[27] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97, pages 6861–6871, 6 2019.

[28] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: Staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15(9): 1937–1950, 2022. ISSN 21508097. doi: 10.14778/3538598.3538614.

[29] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018.

[30] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *33rd Advances in Neural Information Processing Systems*, 2019.

[31] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 14(8):1–1, 2020. ISSN 1041-4347.

[32] Xiang Li, Renyu Zhu, Yao Cheng, Caihua Shan, Siqiang Luo, Dongsheng Li, and Weining Qian. Finding global homophily in graph neural networks when meeting heterophily. In *39th International Conference on Machine Learning*, 2022.

[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web january. Technical report, 1999.

[34] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S. Mirrokni, and Shang-Hua Teng. Local computation of pagerank contributions. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Anthony Bonato, and Fan R. K. Chung, editors, *Algorithms and Models for the Web-Graph*, volume 4863, pages 150–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi: 10.1007/978-3-540-77004-6_12.

[35] Reid Andersen, Fan Chung, and Kevin Lang. Local Graph Partitioning using PageRank Vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.

[36] Hao Zhu and Piotr Koniusz. Simple spectral graph convolution. In *9th International Conference on Learning Representations*, 2021.

[37] Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In *32nd Advances in Neural Information Processing Systems*, 2019.

[38] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. Geom-gcn: Geometric graph convolutional networks. *International Conference on Learning Representations*, 2020.

[39] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. Adaptive universal generalized pagerank graph neural network. In *9th International Conference on Learning Representations*, October 2021.

[40] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *35th International Conference on Machine Learning*, volume 80, Stockholm, Sweden, 2018. PMLR.

[41] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *36th International Conference on Machine Learning*, volume 97, Long Beach, California, 2019. PMLR.

[42] Derek Lim, Felix Hohne, Xiuyu Li, Sijia Linda Huang, Vaishnavi Gupta, Omkar Bhalerao, and Ser-Nam Lim. Large scale learning on non-homophilous graphs: New benchmarks and strong simple methods. In *34th Advances in Neural Information Processing Systems*, 2021.

[43] Jiong Zhu, Mark Heimann, Yujun Yan, Lingxiao Zhao, Leman Akoglu, and Danai Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. In *33rd Advances in Neural Information Processing Systems*, page 12, 2020.

[44] Deyu Bo, Xiao Wang, Chuan Shi, and Huawei Shen. Beyond low-frequency information in graph convolutional networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3950–3957, May 2021. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v35i5.16514.

[45] Yujun Yan, Milad Hashemi, Kevin Swersky, Yaoqing Yang, and Danai Koutra. Two sides of the same coin: Heterophily and oversmoothing in graph convolutional neural networks. In *22nd IEEE International Conference on Data Mining*, November 2022.

[46] Sitao Luan, Chenqing Hua, Qincheng Lu, Jiaqi Zhu, Mingde Zhao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. Revisiting heterophily for graph neural networks. In *36th Advances in Neural Information Processing Systems*, 2022.

[47] Xin Zheng, Yixin Liu, Shirui Pan, Miao Zhang, Di Jin, and Philip S. Yu. Graph neural networks for graphs with heterophily: A survey, February 2022.

[48] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems*, 31:5165–5175, 2018.

[49] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[50] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[51] Chun Wang, Shirui Pan, Guodong Long, Xingquan Zhu, and Jing Jiang. Mgae: Marginalized graph autoencoder for graph clustering. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 889–898, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Rami Al-Rfou, Bryan Perozzi, and Dustin Zelle. Ddgk: Learning graph representations for deep divergence graph kernels. In *The World Wide Web Conference*, pages 37–48, 2019.

[53] Zhengdao Chen, Joan Bruna, and Lisha Li. Supervised community detection with line graph neural networks. *7th International Conference on Learning Representations*, 2019.

[54] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, pages 243–246, 2015.

[55] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S Bhowmick. Scaling attributed network embedding to massive graphs. *Proceedings of the VLDB Endowment*, 14(1):37–49, 2021.

[56] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *35th International Conference on Machine Learning*, 3:1503–1532, 2018.

[57] Zengfeng Huang, Shengzhong Zhang, Chong Xi, Tang Liu, and Min Zhou. Scaling up graph neural networks via graph coarsening. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, volume 1, pages 675–684, 8 2021.

[58] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. *arXiv e-prints*, 2018.

[59] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-bert: Only attention is needed for learning graph representations. *arXiv e-prints*, 2020.

[60] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John P Dickerson, Furong Huang, and Tom Goldstein. VQ-GNN: A Universal Framework to Scale up Graph Neural Networks using Vector Quantization. *34th Advances in Neural Information Processing Systems*, 2021.

[61] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 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.

[62] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. *ACM Transactions on Database Systems*, 44(4):1–37, dec 2019. ISSN 0362-5915.

[63] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2847–2856, 2018.

[64] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. Adversarial attack and defense on graph data: A survey. *arXiv e-prints*, 2018.

[65] Hao Wu, Junhao Gan, Zhewei Wei, and Rui Zhang. Unifying the global and local approaches: An efficient power iteration with forward push. In *Proceedings of the 2021 International Conference on Management of Data*, volume 1, pages 1996–2008, 6 2021.

[66] Dandan Lin, Raymond Chi-Wing Wong, Min Xie, and Victor Junqiu Wei. Index-free approach with theoretical guarantee for efficient random walk with restart query. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 913–924, 2020.

[67] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards Scaling Fully Personalized PageRank: Algorithms, Lower Bounds, and Experiments. *Internet Mathematics*, 2(3):333–358, jan 2005. ISSN 1542-7951.

[68] Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM*, 58(3):1–37, 2011.

[69] Venkat Chandrasekaran, Sujay Sanghavi, Pablo A. Parrilo, and Alan S. Willsky. Rank-Sparsity Incoherence for Matrix Decomposition. *SIAM Journal on Optimization*, 21 (2):572–596, 2011.

[70] Zhouchen Lin, Risheng Liu, and Zhixun Su. Linearized Alternating Direction Method with Adaptive Penalty for Low-Rank Representation. In *24th Advances in Neural Information Processing Systems*, 2011.

[71] Matthew Coudron and Gilad Lerman. On the Sample Complexity of Robust PCA. In *25th Advances in Neural Information Processing Systems*, 2012.

[72] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, Jure Leskovec, Regina Barzilay, Peter Battaglia, Yoshua Bengio, Michael Bronstein, Stephan Günnemann, Will Hamilton, Tommi Jaakkola, Stefanie Jegelka, Maximilian Nickel, Chris Re, Le Song, Jian Tang, Max Welling, and Rich Zemel. Open Graph Benchmark : Datasets for Machine Learning on Graphs. *33rd Advances in Neural Information Processing Systems*, 2020.

[73] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27(1):415–444, August 2001.

[74] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. arXiv:1806.01261, October 2018.

[75] Kristen M. Altenburger and Johan Ugander. Monophily in social networks introduces similarity among friends-of-friends. *Nature Human Behaviour*, 2(4):284–290, March 2018. ISSN 2397-3374. doi: 10.1038/s41562-018-0321-8.

[76] Adam Breuer, Roee Eilat, and Udi Weinsberg. Friend or faux: Graph-based early detection of fake accounts on social networks. In *Proceedings of The Web Conference 2020*, WWW '20, pages 1287–1297, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3366423.3380204.

[77] Jiong Zhu, Ryan A. Rossi, Anup Rao, Tung Mai, Nedim Lipka, Nesreen K. Ahmed, and Danai Koutra. Graph neural networks with heterophily. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):11168–11176, May 2021.

[78] Susheel Suresh, Vinith Budde, Jennifer Neville, Pan Li, and Jianzhu Ma. Breaking the limit of graph neural networks by improving the assortativity of graphs with local mixing patterns. *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021.

[79] Sunil Kumar Maurya, Xin Liu, and Tsuyoshi Murata. Simplifying approach to node classification in graph neural networks. *Journal of Computational Science*, 62:101695, July 2022. ISSN 18777503. doi: 10.1016/j.jocs.2022.101695.

[80] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. Scara: Scalable graph neural networks with feature-oriented optimization. *Proceedings of the VLDB Endowment*, 15(11):3240–3248, 2022. doi: 10.14778/3551793.3551866.

[81] Xiao Wang, Meiqi Zhu, Deyu Bo, Peng Cui, Chuan Shi, and Jian Pei. Am-gcn: Adaptive multi-channel graph convolutional networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 1243–1253, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403177.

[82] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003. doi: 10.1162/089976603321780317.

[83] Renjie Liao, Zhizhen Zhao, Raquel Urtasun, and Richard S Zemel. Lanczosnet: Multi-scale deep graph convolutional networks. In *International Conference on Learning Representations*, 2019.

[84] Chen Ming, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. Simple and deep graph convolutional networks. In *37th International Conference on Machine Learning*, volume 119, pages 1703–1713. PMLR, 2020.

[85] Yao Ma, Xiaorui Liu, Tong Zhao, Yozen Liu, Jiliang Tang, and Neil Shah. A unified view on graph neural networks as graph signal denoising. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 1202–1211, Virtual Event Queensland Australia, October 2021. ACM. doi: 10.1145/3459637.3482225.

[86] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. A critical look at evaluation of gnns under heterophily: Are we really making progress? In *11th International Conference on Learning Representations*, 2023.

[87] Peter A. Lofgren, Siddhartha Banerjee, Ashish Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1436–1445, 2014.

[88] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. *Advances in neural information processing systems*, 32, 2019.

[89] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. In *AAAI Workshop on Deep Learning on Graphs: Methods and Applications*, 2020.

[90] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A. Mojumder, Kihoon Jung, Jose L. Abellan, Yash Ukidave, Ajay Joshi, John Kim, and David Kaeli. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–23. IEEE, March 2021. doi: 10.1109/ISPASS51385.2021.00013.

[91] Chaoyang He, Keshav Balasubramanian, Emir Ceyani, Carl Yang, Han Xie, Lichao Sun, Lifang He, Liangwei Yang, Philip S. Yu, Yu Rong, Peilin Zhao, Junzhou Huang, Murali Annavaram, and Salman Avestimehr. Fedgraphnn: A federated learning system and benchmark for graph neural networks. In *9th International Conference on Learning Representations Workshop on Distributed and Private Machine Learning (DPML)*, 2021.

[92] Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 23:1–48, December 2022.

[93] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.

[94] Alhussein Fawzi, Seyed Mohsen Moosavi-Dezfooli, and Pascal Frossard. Robustness of classifiers: From adversarial to random noise. In *29th Advances in Neural Information Processing Systems*, pages 1632–1640, 2016.

[95] Yueyang Wang, Ziheng Duan, Yida Huang, Haoyan Xu, Jie Feng, and Anni Ren. Mthetgnn: A heterogeneous graph embedding framework for multivariate time series forecasting. *arXiv e-prints*, 2020.

[96] Yanping Zheng, Zhewei Wei, and Jiajun Liu. Decoupled graph neural networks for large dynamic graphs, May 2023.