Scaling up Graph Neural Networks

Ningyi Liao

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2025

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

19/06/2025

. . . . . .

Date

Ningyi Liao

. . . . . .

Ningyi Liao

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

04/07/2025

Date

Luo Siqiang

Prof. Siqiang Luo

Authorship Attribution Statement

This thesis contains material from 7 papers, all published in the following peer-reviewed journals or from papers accepted at conferences. In all 7 papers, I am listed as the first author or co-first author (marked by *).

Chapter 3 is published as a conference paper and a journal extension:

- **Ningyi Liao***, Dingheng Mo*, Siqiang Luo, Xiang Li, Pengcheng Yin. "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 15, no. 11, pp. 3240–3248, 2022. DOI: 10.14778/3551793.3551866.
- **Ningyi Liao**, Dingheng Mo, Siqiang Luo, Xiang Li, Pengcheng Yin. "Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization". *The VLDB Journal*, vol. 33, no. 3, pp. 667–683, 2023. DOI: 10.1007/s00778-023-00829-6.

The contributions of the co-authors are as follows:

- I performed the literature review, implemented majority of the code, designed and conducted the experiments, and prepared majority of the manuscript drafts.
- Dingheng Mo designed the first two algorithms, assisted in implementing the code, and prepared corresponding proofs in the manuscript.
- Prof Siqiang Luo provided the initial project direction, supervised the project, and co-edited the manuscript drafts.
- Prof Xiang Li provided the experimental platform and provided comments on the manuscript.
- Dr Pengcheng Yin offered insights regarding the machine learning and application perspective, and revised the manuscript.

Chapter 4 is published as **Ningyi Liao**, Siqiang Luo, Xiang Li, Jieming Shi. "LD^2: Scalable Heterophilous Graph Neural Network with Decoupled Embeddings". In *36th Conference on Neural Information Processing Systems (**NeurIPS**)*, pp. 10197–10209, 2023.

The contributions of the co-authors are as follows:

- I co-developed the idea of the project, performed the literature review, implemented the code, designed and conducted the experiments, and prepared the manuscript drafts.
- Prof Siqiang Luo co-developed the idea of the project, supervised the project, and revised the manuscript.

- Prof Xiang Li offered insights on the project topic and the heterophily perspective, and provided comments on the manuscript.
- Prof Jieming Shi offered insights regarding the graph computation perspective and provided comments on the manuscript.

Chapter 5 is published as **Ningyi Liao***, Zihao Yu*, Siqiang Luo, Gao Cong. "HubGT: Fast Graph Transformer with Decoupled Hierarchy Labeling". In *39th Conference on Neural Information Processing Systems (**NeurIPS**)*, 2025.

The contributions of the co-authors are as follows:

- I co-developed the idea of the project, implemented majority of the code, conducted experiments on the proposed method, and prepared the manuscript drafts.
- Zihao Yu co-developed the idea of the project, performed the literature review, implemented prototype version of the code, conducted the experiments on baseline methods, and assisted in preparing the manuscript drafts.
- Prof Siqiang Luo supervised the project, provided advice on designing the algorithm, and offered guidance on the manuscript.
- Prof Gao Cong offered guidance on the manuscript.

Chapter 6 is published as Zihao Yu*, **Ningyi Liao***, Siqiang Luo. "GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 17, no. 9, pp. 2269–2278, 2024. DOI: 10.14778/3665844.3665856.

The contributions of the co-authors are as follows:

- Zihao Yu co-developed the idea of the project, performed the literature review, implemented the code, conducted the experiments, and prepared the manuscript drafts.
- I provided ideas on the graph learning perspective, assisted and reviewed code implementation and experiments, and significantly edited the manuscript drafts.
- Prof Siqiang Luo provided the initial project direction, supervised the project, and offered guidance on the manuscript.

Chapter 7 is published as **Ningyi Liao**, Zihao Yu, Ruixiao Zeng, Siqiang Luo. "UNIFEWS: You Need Fewer Operations for Efficient Graph Neural Networks". In *42nd International Conference on Machine Learning (**ICML**)*, vol. 267, pp. 37587–37609, 2025.

The contributions of the co-authors are as follows:

- I developed the idea of the project, performed the literature review, developed majority of the theoretical framework, implemented the code, conducted experiments on the proposed method, and prepared the manuscript drafts.
- Zihao Yu conducted the experiments on baseline methods and assisted in reviewing the manuscript drafts.
- Ruixiao Zeng developed the proof under realistic distribution and assisted in reviewing part of the experiments.

- Prof Siqiang Luo supervised the project and offered guidance on the manuscript.

Chapter 8 is published as **Ningyi Liao**, Haoyu Liu, Zulun Zhu, Siqiang Luo, Laks V.S. Lakshmanan. "A Comprehensive Benchmark on Spectral GNNs: The Impact on Efficiency, Memory, and Effectiveness". In *ACM SIGMOD International Conference on Management of Data (**SIGMOD**)*, vol. 3, no. 4, 2026. DOI: 10.1145/3749156.

The contributions of the co-authors are as follows:

- I co-developed the idea of the project, coordinated project management, performed majority of the literature review, developed the theoretical framework and taxonomy, designed and implemented majority of the code framework, designed and conducted the main experiments, and prepared the majority of the manuscript drafts.
- Haoyu Liu assisted in performing the literature review and designing the code and experiments, implemented the code for 5 variable filters, conducted experiments on signal regression, and prepared corresponding part of the manuscript drafts.
- Zhulun Zhu performed the initial literature review and theoretical analysis, conducted experiments on graph clustering, and prepared initial parts on the taxonomy and framework of the manuscript drafts.
- Prof Siqiang Luo provided the initial project direction, supervised the project, and revised the manuscript.
- Prof Laks V.S. Lakshmanan provided insights on the project direction, shaped the key observations, and revised the manuscript.

19/06/2025

Ningyi Liao

. . . . . .

Date

. . . . . .

Ningyi Liao

Acknowledgements

Four years prior, I could scarcely have envisioned how indelibly this chapter would imprint itself upon this period of my life, nor how its echoes would reverberate intermittently through the seasons yet to unfold. Now inscribed as my particulars, the entirety of its wakefulness and weariness, its endeavors and futilities, its elations and melancholies, has crystallized into gleaming lived realities, exuberantly transcending the very word of 'doctorate'. In the quiet retracing, along the path traversed, the names of those who illumined my steps manifest, each with unique, earnest senses. Unfeignedly and overwhelmingly, it is the arrival that presents the invaluable juncture for my acknowledgements to them, without whom, the journey would have remained impossible.

First and foremost, I wish to express my most profound and heartfelt gratitude to my Ph.D. supervisor, Prof. Siqiang Luo. From the tentative research explorations to the confidence you instilled with steadfast encouragement, from the visionary perspectives and counsel to the exacting exchanges and practices, your guidance has not only established the fundamental bedrock of my studies in this thesis but has also anchored my resolve and affirmation. The tenets you have cultivated, a commitment to become a researcher marked by independence, excellence, and impact, coupled with your role as their epitome, shall endure as the guiding beacon throughout my academic career.

My earnest gratefulness is dedicated to the members of my thesis advisory committee, Prof. Cheng Long and Prof. Guosheng Lin, for their patient engagement, insightful advice, and unwavering support throughout my doctoral studies, especially during the pivotal moments of the qualification examination and the thesis defense. The generosity with which you imparted your extensive expertise and incisive feedback has served to broaden the scope and deepen the substance of my research.

My sincere appreciation also extends to my collaborators and coauthors, Prof. Xiang Li, Dr. Pengcheng Yin, Prof. Reynold Cheng, Prof. Jieming Shi, Prof. Laks V.S. Lakshmanan, Prof. Gao Cong, Prof. Xiaokui Xiao, and amongst others. Working with these talented minds has been a long-cherished opportunity. The intellectual vision,

collective erudition, and scholarly rigor have provided invaluable experiences that have elevated my academic development, leaving a lasting impact on the way I seek knowledge.

To my colleagues, Dingheng Mo, Weiping Yu, Haoyu Liu, Zulun Zhu, Junfeng Liu, Zihao Yu, Dr. Kai Wang, Dr. Kai Siong Yow, Yuxin Qi, who worked together with me in conducting research, as well as other members of our Graph Analysis and Data System Lab and lab-mates in Data Management and Analytics Lab, I offer my deepest thanks. From stimulating discussions that sparked wonderful ideas, to dedicated teamwork that lead to remarkable outcomes, I have always been appreciating for working alongside such outstanding teammates. Together, we have not only advanced our shared work, but also cultivated a vibrant community grounded in genuine companionship. I have also been fortunate to mentor several enthusiastic graduate and undergraduate students: Cheng Han Lee, Yuyuan Song, Jun Xuan Yew, Ruixiao Zeng, Junxiang Xu, Gerald Wei Yong Yip, Yiming Yang, Ivan Khai Ze Lim, with whom I explored a variety of intriguing topics. In addition, I am grateful to Nanyang Technological University and the Joint NTU-WeBank Research Centre on FinTech for supporting my Ph.D. fellowship.

Lastly, and most importantly, I am always indebted to my family for their understanding and encouragement, which go beyond what words can fully convey. I dedicate this thesis to them.

"Per aspera ad astra."

To my dear family

Summary

Graph Neural Networks (GNNs) integrate machine learning models to understand graph-structured data that characterize relationships, and have achieved exceptional utility in various domains. The primary aim of this thesis is to study the prominent scalability issue: as the graph size increases, common GNNs exhibit prohibitive computational overhead and unsatisfactory performance. Therefore, scaling up GNNs is essential for processing massive real-world data and broadening their practical applications.

The scope of this thesis focuses on large-scale GNN designs and improvements. State-of-the-art methods are proposed for incorporating advanced graph data management techniques and handling diverse graph patterns efficiently. Novel theoretical frameworks and comprehensive benchmarks are also presented for understanding and tackling the scalability issue.

Overall, this thesis delivers multiple new GNN models capable of processing up to billion-scale graphs. It also contributes to fundamental analyses and practical insights advancing GNN developments and deployments at large scale.

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 have emerged remarkably for their impressive performance on a wide range of graph learning tasks. Most of these models, however, are known to be difficult to apply to large-scale data. When the graph size increases, regular GNNs become impractical due to the prohibitive computational overhead and unsatisfying performance. Hence, understanding and enhancing the scalability of GNN models is a compelling and prominent task for their broader application in handling real-world graph data.

In this thesis, we explore the scalability issue on different graph variants and GNN designs, uncover the theoretical groundings from the perspective of graph processing, and propose methods to scale up GNN to up to billion-scale graphs with top-tier performance.

In the first part, we propose novel GNN model designs by enhancing graph data management techniques. Our examination reveals that the scalability bottleneck of GNN designs typically lies in repetitive graph-related computation and leads to a coupled overhead with data scale. To apply GNNs to large graphs, it is crucial to specifically managing the graph computation. Our first model, **SCARA**, is developed for efficiently computing graph embedding from the dimension of node features. The other model, **LD**^{2}, is specialized for graphs under heterophily exhibiting distinct data distribution. Theoretical analyses are offered for both models to assess their approximation precision, graph expressiveness, and computational complexity. Experiment results demonstrate that both designs achieve state-of-the-art scalability performance in terms of computation speed and memory footprint without compromising accuracy.

In the second part, we extend the idea of boosting graph computations to broader graph learning approaches. In **HubGT**, we examine the promising Transformer-based GNN model, analyze its scalability issues, and propose dedicated graph processing algorithms for fast and effective data retrieval. In **GENTI**, we introduce the strategy of decoupling and accelerating graph element computation for the subgraph learning problem, which

has particular use in handling dynamic graph links. These applications highlight the utility of our intuition for scaling up graph learning through advanced graph management techniques, especially decoupling on the computational device layer, graph-oriented storage on the data structure layer, and approximation on the algorithmic layer. Both works showcase practical efficiency in processing larger graphs at a faster speed, significantly advancing the capabilities of these diverse graph learning approaches.

In the third part, we dive deep into understanding the GNN scalability issue concerning graph data management. From the theoretical side, our **UNIFEWS** framework innovatively integrates the graph signal processing framework with the graph sparsification mechanism to provide an interpretation of the approximate graph propagation schemes, which incorporate and generalize the scalable computations in SCARA and LD^2. Our derivation establishes a new framework for characterizing graph sparsification as an approximation for the GNN learning process with a bounded error, offering solid theoretical support for a range of scalable GNN designs. From the empirical side, we conduct a benchmark study focusing on **Spectral GNN** models, which are unique for their scalable learning schemes. Based on our characterization and taxonomy, we formulate and analyze over 30 GNNs as spectral filters, including the SCARA and LD^2 paradigms, with a unified and scalable implementation applicable to million-scale graphs. Within this framework, extensive evaluations are conducted with comprehensive metrics on time efficiency, memory footprint, and effectiveness, where novel observations and practical guidelines are exclusively available due to the benchmark scale and coverage. Challenging the prevailing belief, our benchmark reveals an intricate landscape regarding the effectiveness and efficiency of spectral graph filters, demonstrating the potential to achieve desirable performance through tailored spectral manipulation of graph data.

In conclusion, this thesis aims to advance the design and implementation of scalable GNNs, especially by identifying key performance bottlenecks and underpinning graph management theories. Our contributions not only equip various graph learning systems with the capability to operate on real-world, billion-scale graphs, but also provide pioneering theoretical foundations and practical guidelines that inspire future research regarding GNN performance at large scale.

Contents

List of Figures

List of Tables

List of Algorithms

List of Author's Awards, Patents, and Publications

Awards

- **PREMIA Best Student Paper Award 2025, Certificate of Commendation**, "A Comprehensive Benchmark on Spectral GNNs: The Impact on Efficiency, Memory, and Effectiveness". In *ACM SIGMOD International Conference on Management of Data (**SIGMOD**)*, 2025.
- **PREMIA Best Student Paper Award 2024, Certificate of Merit**, "GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs". In *Proceedings of the VLDB Endowment (**VLDB**)*, 2024.

Patents

- **Ningyi Liao**, Dingheng Mo, Siqiang Luo, "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". *Singapore Provisional* (2022–2023) No. 10202203849U.

Journal Articles

- **Ningyi Liao**, Dingheng Mo, Siqiang Luo, Xiang Li, Pengcheng Yin. "Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization". *The VLDB Journal*, vol. 33, no. 3, pp. 667–683, 2023. DOI: 10.1007/s00778-023-00829-6.

Conference Proceedings

- **Ningyi Liao**, Haoyu Liu, Zulun Zhu, Siqiang Luo, Laks V.S. Lakshmanan. "A Comprehensive Benchmark on Spectral GNNs: The Impact on Efficiency, Memory, and Effectiveness". In *ACM SIGMOD International Conference on Management of Data (**SIGMOD**)*, vol. 3, no. 4, 2026. DOI: 10.1145/3749156.

- **Ningyi Liao**[*], Zihao Yu[*], Siqiang Luo, Gao Cong. "HubGT: Fast Graph Transformer with Decoupled Hierarchy Labeling". In *39th Conference on Neural Information Processing Systems (**NeurIPS**)*, 2025.

- **Ningyi Liao**, Zihao Yu, Ruixiao Zeng, Siqiang Luo. "UNIFEWS: You Need Fewer Operations for Efficient Graph Neural Networks". In *42nd International Conference on Machine Learning (**ICML**)*, vol. 267, pp. 37587–37609, 2025.

- Zihao Yu[*], **Ningyi Liao**[*], Siqiang Luo. "GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 17, no. 9, pp. 2269–2278, 2024. DOI: 10.14778/3665844.3665856.

- **Ningyi Liao**, Siqiang Luo, Xiang Li, Jieming Shi. "LD^2: Scalable Heterophilous Graph Neural Network with Decoupled Embeddings". In *36th Conference on Neural Information Processing Systems (**NeurIPS**)*, pp. 10197–10209, 2023.

- **Ningyi Liao**[*], Dingheng Mo[*], Siqiang Luo, Xiang Li, Pengcheng Yin. "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 15, no. 11, pp. 3240–3248, 2022. DOI: 10.14778/3551793.3551866.

- **Ningyi Liao**, Siqiang Luo, Xiaokui Xiao, Reynold Cheng. "Advances in Designing Scalable Graph Neural Networks: The Perspective of Graph Data Management". In *ACM SIGMOD International Conference on Management of Data (**SIGMOD** Tutorial)*, pp. 844–850, 2025. DOI: 10.1145/3722212.3725634.

- Haoyu Liu, **Ningyi Liao**, Siqiang Luo. "SIGMA: An Efficient Heterophilous Graph Neural Network with Fast Global Aggregation". In *IEEE 41th International Conference on Data Engineering (**ICDE**)*, pp. 1924–1937, 2025. DOI: 10.1109/ICDE65448.2025.00147.

[*]Both authors contributed equally to this research.

- Kai Siong Yow, **Ningyi Liao**, Siqiang Luo, Reynold Cheng. "Machine Learning for Subgraph Extraction: Methods, Applications and Challenges". In *Proceedings of the VLDB Endowment (**VLDB** Tutorial)*, vol. 16, no. 12, pp. 3864–3867, 2023. DOI: 10.14778/3611540.3611571.
- Jun Xuan Yew, **Ningyi Liao**, Dingheng Mo, Siqiang Luo. "Example Searcher: A Spatial Query System via Example". In *IEEE 39th International Conference on Data Engineering (**ICDE 2023** Demo)*, pp. 3635–3638, 2023. DOI: 10.1109/ICDE55515.2023.00286.

Tutorials and Invited Talks

- "Scaling up Graph Neural Networks." In APSIPA 3-Minute Thesis (3MT) Competition, *17th Asia Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC 2025)*, Singapore, October 2025.
- "Advances in Designing Scalable Graph Neural Networks: The Perspective of Graph Data Management – Specific Algorithms" (1.5h/3h). In *ACM International Conference on Management of Data (SIGMOD)*, Berlin, German, June 2025.
- "Scalable Heterophilous Graph Neural Network with Decoupled Embeddings". In *Singapore ACM SIGKDD Symposium*, SMU, Singapore, July 2024.
- "Machine Learning for Subgraph Extraction: Methods, Applications and Challenges – Community Detection & Maximum Common Subgraph" (0.5h/1.5h). In *49th International Conference on Very Large Databases (VLDB)*, Vancouver, Canada, September 2023.

Manuscripts under Submission

- Weiping Yu*, **Ningyi Liao***, Siqiang Luo, Junfeng Liu. "RAGDoll: Efficient Offloading-based Online RAG System on a Single GPU". Under submission of *ACM International Conference on Management of Data (**SIGMOD**)*, 2025. arXiv: 2504.15302.

- 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 *Data Mining and Knowledge Discovery*, 2023. arXiv: 2204.01057.

- Yuxin Qi, **Ningyi Liao**, Siqiang Luo, Xi Lin, Xinpeng Jiang, Kai Han, Jianhua Li. "Fine-Grained Differentially Private Graph Neural Network". Preparing for submission, 2024.

Chapter 1

Introduction

Graph is a ubiquitous data structure for modeling entities and their relationships as nodes and edges, which is particularly adept at revealing and characterizing complex interactions. It functions as a universal tool for abstracting real-world objects and finds applications in an extensive range of fields. Unlike ordinary structured data such as images and text, graphs are highly irregular, which implies that the interconnected data is inherently complicated and demands specialized management beyond common geometry-based mathematical tools [14, 15, 16].

Deep learning methods in the form of neural networks have driven breakthroughs in handling various types of data [17, 18, 19, 20, 21, 22, 23]. The marriage of deep learning and graphs leads to Graph Neural Networks (GNNs), which describe a set of neural networks dedicated to learning tasks on data represented in graph structures [24, 25, 26, 27, 28]. GNNs have been proven powerful for encapsulating entities as well as interactions, and have been successfully applied to diverse scenarios and disciplines, including social science [29, 30, 31], material science [32, 33, 34], neuroscience [35, 36, 37], and biomedical engineering [38, 39, 40].

In the era of big data, where unprecedented volumes of information are being generated, scalability has become a prominent issue in graph data management. Typical industry-level graphs can be composed of millions or billions of nodes and edges [41, 42, 43, 44]. While GNNs are effective in understanding graph information, they are known to be resource-intensive, rendering them prohibitive for data on such a large scale [45, 46, 47]. When the graph size increases, GNN computational overhead escalates significantly due to the tight coupling between its architecture and the graph data structure, which leads to

a even more compelling scalability issue. Empirical studies also observe that the data management steps in GNNs are notably more time- and memory-consuming than those in common deep neural networks [48, 49, 50]. Hence, scaling up GNNs to large graphs becomes a challenging yet important problem when designing and utilizing these models.

In this thesis, we attempt to study the scalability issues in applying GNN models to large graphs by designing and enhancing dedicated data management techniques. The following part of this chapter first introduces relevant concepts in the context of data management and graph learning, and subsequently identifies the scalability bottlenecks of current GNN models. Then, specific research objectives are developed for addressing the existing gaps in scaling up GNNs, and the contributions of the proposed methods are summarized. Lastly, the outline of the thesis is presented.

1.1 Background and Motivation

Graph-structured Data. A graph, also known as a network, typically consists of a set of nodes and a set of edges, as illustrated in Figure 1.1, and can be further categorized into different variants by examining the properties of its components. For example, graphs can be directed or undirected, homogeneous or heterogeneous, homophilous or heterophilous, according to the divisions of edge directions, edge types, and neighborhood similarity, respectively [51, 52]. Graph-structured data in particular variants usually carry additional prior knowledge for better understanding the data, but also require careful and sophisticated technique designs [53, 54].

(a) An example graph. **(b)** Adjacency matrix. **(c)** Feature matrix.

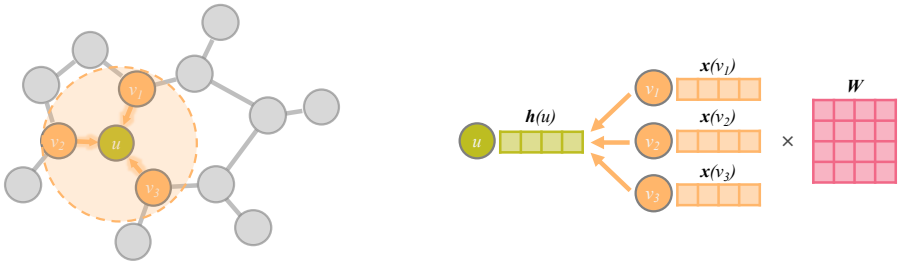
FIGURE 1.1: Example of a graph and its representations. **(a)** An example graph with node features and labels. **(b)** The adjacency matrix of the graph. **(c)** The feature matrix of the graph.

Classic graph data management algorithms tend to focus on problems within the graph structure itself, e.g., ranking nodes, finding paths, and identifying specific substructures. In comparison, the discipline of *graph learning* attempts to build data-driven models for extracting graph knowledge, representing the underlying information, and solving particular graph-related problems [55, 56, 57, 58]. Depending on the granularity, graph learning can be categorized into node-, edge-, and graph-level tasks; depending on the output requirement, distinct models exist for analytical and generative tasks. We characterize graph neural networks as analytical models with a distinctive neural network architecture [16, 47, 59, 60].

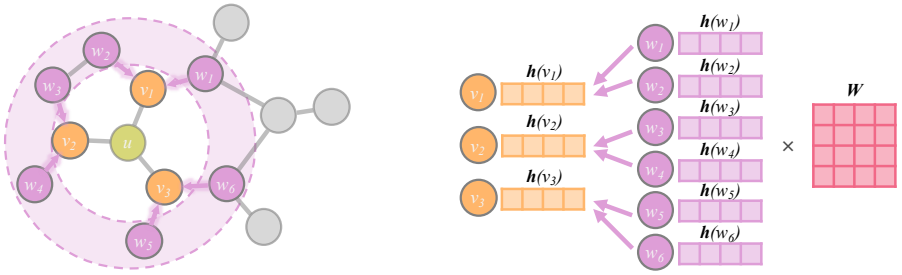
Convolutional GNNs. Inspired by the emerging deep learning approaches, various neural operations have been introduced to the field of graph learning to build GNN models. Compared to previous neural networks, GNNs specifically feature the ability of representing non-Euclidean data through dedicated architectures that integrate graph topology, capturing not only instance identities but also their relationships [15, 61, 62]. The *convolution* operation is generalized from grid data to graphs and composes the core of convolutional GNNs [25, 63, 27], while advantageous techniques such as the attention mechanism and skip connection have also been incorporated to the convolutional network architectures [28, 64, 65, 66, 67, 68].

The high-level idea of graph convolution is to represent a node by aggregating its own features as well as those of its neighbors. Specifically, the fundamental GCN model [25] represents each node's state by a feature vector, successively propagates the state message to its neighboring nodes, and updates the neighbors' features using a neural network. This interleaved process containing graph propagation and representation update can conducted for multiple iterations, as depicted in Figure 1.2. By this means, the information of one node is passed to another based on edge connections, and the process is named *message passing*. By integrating graph operations such as sparsification and downsampling, propagation edges can be reduced from the basic message-passing scheme [69, 70]. Conversely, graph rewiring and upsampling methods are able to augment the convolution by appending global nodes [71, 72]. A unique array of models known as spectral GNNs [24, 73, 26] invokes graph spectral theory [74, 75] for describing the convolution operator in the frequency domain and inclines to the spectral filter designs.

A notorious scalability issue rooted in convolutional GNN architectures is the *neighbor explosion* phenomenon, that the number of nodes included in convolution computation potentially grows exponentially when performing the iterative message-passing



(a) 1-hop convolution.



(b) 2-hop convolution.

FIGURE 1.2: Message-passing for a graph node u. **(a)** Graph convolution and representation update for 1-hop neighbors. **(b)** Graph convolution and representation update for 2-hop neighbors.

operation [76, 77, 78, 79]. As a consequence, conventional solutions for ordinary data structures, such as mini-batching, fail when deploying GNNs to large-scale data, and the model scalability is critically limited [80, 81]. For advanced GNN models, the scalability issue becomes increasingly prominent due to the complicated computations, especially those correlated with the graph topology [82, 83, 84].

Sequential GNNs. Compared to convolutional models, sequential GNNs typically process graphs as sequential data without explicitly passing messages through the graph topology. These include the early works of recurrent GNNs, which compute and iterate node information through the graph to learn a convergent state. Gate-based methods, including Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM), are employed as recursion optimizations [85, 86]. Recently, Graph Transformers (GTs) have emerged as a class of promising sequential architectures, combining graph embedding techniques and the Transformer architecture [87] to pursue substantial expressive power [88, 89, 90]. These models rely less on the iterative graph convolution, but choose to extract graph knowledge through alternative operations, such as representing node-pair relationship by Shortest Path Distance (SPD).

Subgraph GNNs. Subgraph GNNs, or alternatively known as Subgraph-based Graph Representation Learning (SGRL), learn graph representation from a specified area around nodes of interest, namely subgraphs [91, 92, 93]. Their design intuition is that, applying neural network models to the substructure instead of the whole graph favorably captures the underlying structural information. Compared to the canonical full-graph GNNs, subgraph models preserve expressiveness of the relative intra-node knowledge and offers better scalability for graph learning tasks emphasizing graph structure, such as link prediction [94, 95, 96].

1.2 Research Objectives

This thesis aims to mitigate the existing gap in graph data processing and neural network research by addressing the issues in scaling up GNN models and providing a better understanding in the lens of graph data management. The high-level objectives include designing graph data management techniques to provide scalable solutions for various GNN models, as well as investigating and characterizing the graph-relevant techniques within these methods, and can be summarized by the following research goals:

- **Analyze and design scalable GNN schemes**: Versatile graph data processing algorithms will be integrated with the GNN architecture to mitigate the graph computational overhead of different model designs. Theories on approximate graph computation will be established to characterize the scalable learning scheme.
- **Develop scalable approaches for different scenarios**: Diverse graph variants and GNN architectures will be investigated to identify specific scalability bottlenecks. Novel models utilizing dedicated data management techniques will be developed to address the scalability issue.
- **Harmonize effectiveness and efficiency performance**: Model accuracy and scalability will be evaluated to understand the comprehensive performance when scaling up GNNs. Practical guidelines and solutions will be offered for improving model scalability while ensuring efficacy.

In particular, we are motivated to tackle the following research questions:

- What is the bottleneck of convolutional GNN architectures when applied to large-scale data, and how to address the bottleneck by utilizing graph data management techniques?
- What is the specific scalability issue for applying convolutional GNNs to graph data under heterophily, and how to accommodate this challenge by augmenting the data processing pipeline?
- How different are the efficiency limitations for Transformer-based GNNs, and how to design dedicated graph data management techniques suitable for these models?
- What are the scalability issues in subgraph learning approaches, and how to boost these methods by the idea of managing graph data efficiently?
- What is the underlying mechanism and theoretical guarantee of using data management techniques for approximating GNN computation?
- What is the relationship between the effectiveness and efficiency in the context of various scalable GNN designs and training schemes?

The scope of this study on scalable GNNs mainly focuses on theoretical contributions to designing and improving GNN models, especially regarding their scalability and efficiency performance. Issues raised during scaling up these models, e.g., the impact of scalable solutions on model expressiveness and efficacy, will be explored as well. There is another line of studies addressing the scalability issue by system-level approaches [97, 48, 98, 99, 100, 101, 102], especially by distributed computation [103, 104, 105, 106, 107, 108, 109, 110, 50], which is highly orthogonal to our study from the algorithmic perspective.

1.3 Major Contributions

The straight-forward solution to scale up graph neural networks is to provide effective graph data processing techniques for different types of models. From this perspective, we elaborate our contributions in proposing two scalable model architectures.

SCARA: Decoupled GNN with Feature-Oriented Optimization. We first examine the decoupled architecture, which is a strategy for separating and specifically enhancing the computation-intensive graph propagation without interfering representation learning in graph convolution, and propose two algorithms exploiting the properties of node features, namely FEATURE-PUSH and FEATURE-REUSE. The FEATURE-PUSH algorithm rethinks the message-passing scheme and sparingly propagates graph information from the feature dimension by evaluating the current status, successfully saving the need for iterative operations. The FEATURE-REUSE mechanism further utilizes feature-wise similarity by decomposing the base vectors to arrange feature propagations and reduce overlapped computations while maintaining precision. Powered by these two approaches, SCARA achieves a sub-linear time complexity for graph propagation, along with efficient network training and inference implemented in a mini-batch manner. It also demonstrates scalable memory usage when processing graphs up to the billion scale.

LD^2: Heterophilous GNN with Decoupled Embeddings. Convolutional GNNs for heterophilous graphs are unique for their non-local propagation architectures considering the shifted graph inductive bias. However, this design also aggravates the GNN scalability issue for including relatively coupled graph operations. We are among the pioneer works to explore scalable solutions under heterophily and propose LD^2, which is symbolized by its low-dimension embeddings and long-distance aggregation. By utilizing the decoupled architecture and performing learning on precomputed embeddings, the model removes the reliance on iterative train-time full-graph computations in typical heterophilous GNNs. 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 is proposed for efficient embedding calculation. LD^2 achieves optimized training performance, highlighting time complexity that is only linear to the number of nodes and memory overhead independent of the graph scale.

Then, we extend the solution of boosting graph-related computation to more diverse graph learning scenarios with distinctive model architectures and graph understanding tasks, which leads to our contributions in broadening two scalable graph learning approaches.

HubGT: Hierarchy Labeling for Graph Transformer. While Graph Transformer has emerged as a promising GNN architecture, it remains challenging in effectively representing graph information while ensuring learning efficiency. We specifically investigate the scalability issues in current GT designs, and propose HubGT as an efficient Graph

Transformer with novel hierarchical sampling based on the hub labels, effectively embedding local and global graph topology. We also enable the powerful positional encoding based on relative distance on large-scale graphs. For boosting graph computation, we design a hierarchical index dedicated to HubGT graph processing, featuring construction under linear complexity and $O(1)$ query overhead. The decoupled precomputation and simple training contribute to fast and scalable mini-batch GT training. HubGT is scalable on up to million-scale graphs, achieving top-tier accuracy and demonstrating the fastest inference speed.

GENTI: Walk-based Representation for Dynamic Subgraph GNN. Subgraph GNNs have demonstrated scalability and expressiveness for large-scale tasks. The core challenge of applying subgraph GNNs to dynamic graphs lies in accommodating the extraction of subgraphs to evolving data with efficient computation. To address the efficiency bottleneck, we propose GENTI as a scalable subgraph GNN on dynamic graphs with GPU-oriented designs. On the algorithmic level, we decouple the resource-intensive subgraph extraction stage to be separately conducted on CPUs and GPUs, which enables full GPU utilization during subgraph processing and offers improved efficiency. On system level, we design the data structure for maintaining subgraphs on GPU with improved memory complexity and fast batch processing ability. We also enhance the graph storage to render it applicable for scalable and dynamic updating and sampling. GENTI is capable of applying to billion-scale dynamic graphs and achieves significant speedups in empirical running time.

On top of novel model designs, properly scaling up GNNs also calls for deeper understanding regarding the underlying mechanism, especially the model performance at large scale. To this end, we offer research in characterizing a range of models from two data-centered perspectives.

Understanding GNN Approximation as Graph Sparsification. Graph sparsification is a popular technique for efficiently performing graph operation with a lower amount of components, therefore saving computational overhead and benefiting scalability. We propose UNIFEWS as a general sparsification particularly for convolutional GNNs, which is practical for many common models to reduce graph-scale computation and alleviate computational cost. UNIFEWS employs a unified scheme involving both graph connection reduction and model weight compression. In fact, the graph propagations in both SCARA and LD^2 can be considered as special cases of the UNIFEWS scheme. We develop a novel

theoretical analysis of the graph sparsification technique and bound the approximation procedure through the comprehensive lens of graph spectral signal processing. The graph sparsification approach characterized by UNIFEWS is effective in reaching high sparsity without compromising accuracy, paving a promising solution towards efficient GNN operations unifying graph edges and model parameters.

Benchmarking Performance of Spectral GNNs. In a broader sense, convolutional models with the decoupled architecture including SCARA and LD^2 can be regarded as spectral GNNs, where different graph filters in the frequency domain are utilized to process graph signals, and subsequently represented by learnable network architectures. We thereby benchmark GNNs under the umbrella of spectral models with dedicated designs and analysis, as well as a unified framework for practical implementation. We offer an open-source, plug-and-play collection for spectral models and filters. The framework realizes integrated and efficient implementations covering a variety of spectral GNN designs. Designed in a spectral-oriented manner, our implementation ensures that most filters are easily adaptable to a wide range of GNN transformation, architecture, and learning schemes. Based on the framework, we provide extensive study on the effectiveness and efficiency of spectral filters across various common configurations. We analyze the spectral properties of the filters and offer practical guidelines for using spectral GNNs, especially highlighting the scalability characteristics of these models. Experimental evaluations show that the model achieves improved minibatch training performance especially on large graphs.

A broad range of realistic applications can benefit from GNNs with better scalability, enabling effective learning on industry-level big data with millions or even billions of graph components. Examining and enhancing GNN scalability also helps researchers better understand the underlying mechanisms for integrating machine learning and graph data, advancing research and applications in areas of data mining, data management, and machine learning. Additionally, compared to mainstream models that demand professional servers, some proposed GNNs are scalable enough to run on a single machine with commercial-level hardware, which brings forward the possibility of deploying scalable GNNs on customer-side machines with better data convenience, privacy, and democratization.

1.4 Organization of the Thesis

As outlined in Figure 1.3, the thesis is organized in the following structure.

Chapter 1 introduces the background and advances of graph neural networks for graph representation learning, while specifying the current issues in scalability of common GNN designs. It then outlines the research objectives, questions, and contributions for scaling up GNNs to large graphs in this thesis.

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.

The remaining chapters are divided into three parts:

Part I: *Designing Scalable GNN Models.* This part focuses on developing novel message-passing GNN model designs for different scenarios utilizing dedicated graph data management techniques.

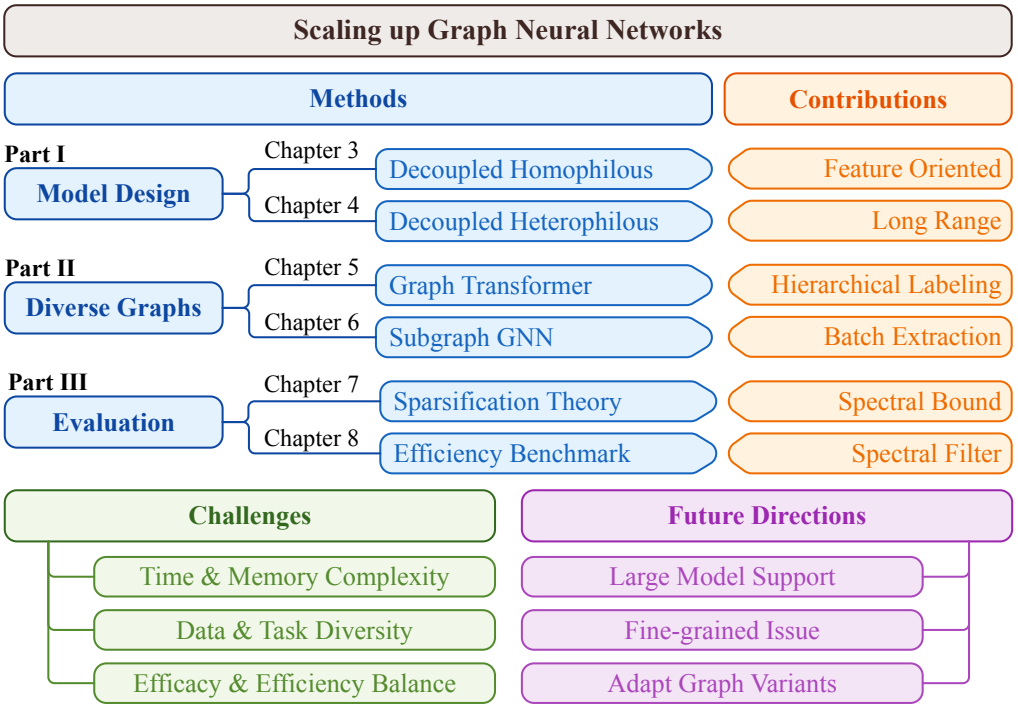


FIGURE 1.3: Overall framework of the thesis, including respective methods under each parts and corresponding solutions and contributions. We further list the high-level challenges as stated in Chapter 1 and future directions as in Chapter 9.

Chapter 3 proposes the **SCARA** model as the scalable decoupled GNN with feature-oriented optimization. We subsequently derive our methodology in performing graph propagation as precomputation with algorithms manipulating node features, and provide theoretical analysis on approximation precision and complexity.

Chapter 4 develops 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 fully utilize the embedding information.

Part II: *Boosting Diverse Graph Learning Approaches*. This part broadens the scalability solutions based on graph computation acceleration to more diverse graph learning approaches.

Chapter 5 designs the **HubGT** model as the fast Graph Transformer boosted by decoupled graph computation and hierarchical graph representations. We uncover limitations in current GT designs, then address them by the end-to-end hub labeling algorithm for fast neighborhood generation and positional encoding.

Chapter 6 offers the **GENTI** model as the scalable subgraph GNN with GPU-powered walk-based representation learning on dynamic graphs. We especially identify the system and algorithmic bottlenecks in subgraph extraction and design new pipelines and data structures facilitating more efficient graph processing.

Part III: *Evaluating GNN Scalability Performance*. This part presents a theoretical and empirical analysis of the performance of scalable GNN models from the perspective of graph data processing and representation learning.

Chapter 7 relates approximate GNN propagation to spectral graph sparsification. It highlights the theoretical explanation by formulating graph learning as an optimization problem, as well as the empirical benefit of the unified model compression.

Chapter 8 presents a comprehensive benchmark for evaluating the effectiveness and efficiency of spectral GNNs, covering theoretical formulation of model architectures, taxonomy of filters, and a practical implementation and evaluation framework.

Chapter 9 concludes the thesis by reviewing the key results and contributions. We also conduct further discussions and expectations regarding promising future research topics under the topic of GNN scalability and graph data management.

Chapter 2

Literature Review

2.1 Graph Data and Notations

2.1.1 Simple Graph

In an undirected graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$, the number of nodes, the number of edges, and the average degree are denoted by $n = |\mathcal{V}|$, $m = |\mathcal{E}|$, and $d = m/n$, respectively. The neighborhood of an ego node $u \in \mathcal{V}$ is the set $\mathcal{N}(u) = \{v \mid (u, v) \in \mathcal{E}\}$, and its degree $d(u) = |\mathcal{N}(u)|$. The diagonal degree matrix is $\boldsymbol{D} = \mathrm{diag}(d(1), d(2), \cdots, d(n))$, and the self-looped variant is $\bar{\boldsymbol{D}} = \boldsymbol{D} + \boldsymbol{I}$.

The graph connectivity is represented by the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, and its variant with self-loop edges is $\bar{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$. To balance the scale of values and facilitate further calculations, it is more common to use the adjacency matrix normalized by node degree. Here we adopt the graph normalization scheme [111, 112] with coefficient $\rho \in [0, 1]$ denoting distinct considerations for in- and out-degrees, and the normalized adjacency is $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{D}}^{\rho-1} \bar{\boldsymbol{A}} \bar{\boldsymbol{D}}^{-\rho}$.

2.1.2 Attributed Graph and Heterophily

Each node in the graph $u \in \mathcal{V}$ is represented by an F_0-dimensional attribute vector $\boldsymbol{X}[u]$, which composes the attribute matrix $\boldsymbol{X} \in \mathbb{R}^{n \times F_0}$. In other words, $\boldsymbol{X}[u]$ is a row of $\boldsymbol{X}$.

From the other perspective, we denote the feature-wise length-n attribute vector, i.e., a column of $\boldsymbol{X}$, as $\boldsymbol{x}$.

For multiclass classification task on graph $\mathcal{G}$, a node $u \in \mathcal{V}$ 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 [113]. We measure the graph heterophily by node homophily score [114], which is the average proportion of the neighbors with the same class of each node:

$$\mathcal{H}_{\text{node}} = \frac{1}{n} \sum_{u \in V} \frac{|\{v \in \mathcal{N}(u) : y(v) = y(u)\}|}{d(u)}. \tag{2.1}$$

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

2.1.3 Spectral Graph Theory

Spectral graph theory [115] relates the graph with signal processing techniques, which utilizes the graph Laplacian matrix $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$ or its normalized counterpart $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$. Consider an undirected graph whose adjacency matrix $\tilde{\boldsymbol{A}}$ is symmetric, the eigen-decomposition of the normalized graph adjacency and Laplacian matrices respectively as $\tilde{\boldsymbol{A}} = \boldsymbol{V}\boldsymbol{M}\boldsymbol{V}^\top$ and $\tilde{\boldsymbol{L}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$, where the graph *spectrum* $\boldsymbol{M} = \text{diag}(\mu_1, \cdots, \mu_n)$ is the diagonal matrix of eigenvalues $|\mu_1| \geq |\mu_2| \geq \cdots \geq |\mu_n|$, or equivalently $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \cdots, \lambda_n)$, $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n \leq 2$. $\boldsymbol{V}, \boldsymbol{U}$ are the matrices of corresponding eigenvectors.

The eigenvalues are also known as the signal *frequencies*. 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 $\lambda_1, \lambda_2, \cdots$ of $\tilde{\boldsymbol{L}}$. These eigenvalues are known as the low-frequency spectrum of the graph that correlates to graph connectivity. Specially, $\lambda_2 > 0$ if and only if the graph is connected, which is our case. Similarly, small values of μ_j and large values of λ_i represent the high frequency part of the graph. Graph spectrum is a graph invariant despite the status of node labels.

2.1.4 Dynamic Graph

A dynamic graph $\mathcal{G}_T = \langle \mathcal{V}, \mathcal{E}, T \rangle$ consists of a series of graph instances over a time span T. At each specific timestamp, a graph instance is similar to a static graph with respective nodes and edges $\mathcal{G}_t = \langle \mathcal{V}_t, \mathcal{E}_t \rangle$. Depending on their representation with regard to time intervals, dynamic graphs can be classified into two categories, namely *Continuous-Time Dynamic Graphs* (CTDGs) and *Discrete-Time Dynamic Graphs* (DTDGs) [116]. In particular, CTDGs possess continuous timestamps $t \in [0, T]$, while DTDGs include a set of discrete ones $t \in \{t_0, t_1, \cdots, t_T\}$. CTDGs are generally more expressive in depicting temporal information, since a DTDG can be equivalently considered as a CTDG in special case [116, 117].

For dynamic graphs, the graph components of nodes and edges are changing overtime, which are knows as *events*. A CTDG can be constructed by a series of edge additions and deletions events. The graph $\mathcal{G}$ can thus be represented by an event sequence $\mathcal{S} = \{(u, v, t, op)\}$. Each event implies that the edge from node u to v experiences the operation denoted by op at timestamp t. The operation op can either be insertion or deletion of the edge (u, v).

2.2 Convolutional GNN

2.2.1 Vanilla Spatial GNN and Operation

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 [25], the model input feature matrix is $\boldsymbol{H}^{(0)} = \boldsymbol{X}$ in particular, and the $(l+1)$-th representation matrix $\boldsymbol{H}^{(l+1)}$ is updated as:

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

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{F \times F}$ is the trainable weight matrix of the l-th layer, $\tilde{\boldsymbol{A}}$ is the normalized adjacency matrix, and $\sigma(\cdot)$ is the activation function such as ReLU or softmax. The representation can be initialized by node attributes $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. For simplicity in complexity analysis, we assume the feature size F to be constant in all layers and $F_0 = F$.

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

Model	Precomp. Time	Training Time	Inference Time
GCN [25]	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
Cluster-GCN [120]	$O(m)$	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
GraphSAINT [121]	–	$O(IL_PLnF^2)$	$O(LmF + LnF^2)$
GAS [122]	$O(m + LnF)$	$O(ILmF + ILnF^2)$	$O(nF)$
APPNP [123]	$O(m)$	$O(IL_PmF + ILnF^2)$	$O(L_PmF + LnF^2)$
PPRGo [124]	$O(m/\delta)$	$O(IKnF + ILnF^2)$	$O(KnF + LnF^2)$
SGC [125]	$O(L_PmF)$	$O(ILnF^2)$	$O(LnF^2)$
GBP [111]	$O(L_PF\sqrt{L_Pm\log(L_Pn)}/\epsilon)$	$O(ILnF^2)$	$O(LnF^2)$

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

Model	Precomp. Mem.	Training Mem.	Inference Mem.
GCN [25]	–	$O(LnF + LF^2)$	$O(LnF + LF^2)$
Cluster-GCN [120]	$O(n)$	$O(Ln_bF + LF^2)$	$O(LnF + LF^2)$
GraphSAINT [121]	–	$O(L_PLn_bF + LF^2)$	$O(LnF + LF^2)$
GAS [122]	$O(LnF)$	$O(Ldn_bF + LF^2)$	$O(Ldn_bF + LF^2)$
APPNP [123]	$O(m)$	$O(Ln_bF + LF^2 + nn_b)$	$O(Ln_bF + LF^2 + nn_b)$
PPRGo [124]	$O(n/\delta)$	$O(Ln_bF + LF^2 + Kn_b)$	$O(Ln_bF + LF^2 + Kn_b)$
SGC [125]	$O(m)$	$O(Ln_bF + LF^2)$	$O(Ln_bF + LF^2)$
GBP [111]	$O(nF)$	$O(Ln_bF + LF^2)$	$O(Ln_bF + LF^2)$

In order to generalize the expression of GCN in Eq. (2.2) to a broader range of message-passing GNNs, we abstract the graph-related operation in the l-th layer as a propagation function applying on the graph matrix $f^{(l)} : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$, then the recursive spatial GNN update is:

$$\boldsymbol{H}^{(l+1)} = \varphi\left(f^{(l)}(\tilde{\boldsymbol{A}}) \cdot \boldsymbol{H}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{2.3}$$

where φ denotes the transformation consisting of learnable weights and non-linear activations. More complex schemes such as jumping knowledge [65] and concatenation [118, 119] have also been proposed, but the computational schemes remain similar.

Complexity Analysis. Summarized in Tables 2.1 and 2.2, we present an analysis on the complexity bounds of GCN in Eq. (2.2) 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 $\boldsymbol{P} = f(\tilde{\boldsymbol{A}}) \cdot \boldsymbol{H}$, which can be regarded as repetitive sparse-dense matrix multiplications. For each propagation *hop* multiplying a graph matrix $\boldsymbol{A}$, or similarly, a matrix $f(\tilde{\boldsymbol{A}})$ derived from $\boldsymbol{A}$ with the same level of sparsity with $O(m)$ entries, upon representation $\boldsymbol{H}$, the time complexity is $O(mF)$. Performing L-hop propagation leads to a time complexity of $O(LmF)$.

The overhead for the second procedure **feature transformation** by multiplying $\boldsymbol{W}$ to $\boldsymbol{H}$ is $O(nF^2)$, and increased to $O(LnF^2)$ by stacking L *layers*. 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 [120, 111], 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 [49]. The memory space for a graph propagation matrix is $O(m)$. If the matrix remains static during propagation, we usually omit it in complexity analysis for simplicity.

Learning Schemes. The learning scheme describes how the model and data are deployed to CPU and GPU for training and inference. **Full-batch** (FB) training loads all input data onto the GPU, which suffices the need on small-scale graphs and is the de facto scheme for most GNNs. However, for larger graphs, the critical scalability issue emerges, as the full graph topology exceeds the GPU memory capacity.

To mitigate the overhead, a common model-agnostic solution is to employ **graph partition** (GP) algorithms to divide and train the graph data in batches. However, this process impacts the graph topology and undermines GNN expressiveness [78, 79].

In contrast, the **mini-batch** (MB) scheme depicts separating processing graph and model learning, usually with decoupled architectures. It chooses to perform graph-related operations to generate one or more node-wise representation matrices in a precomputation stage on CPU. Then, only the intermediate representations with batch size n_b are loaded onto the GPU in batches during training, which prevents the memory footprint from being coupled with the graph size and enjoys better scalability. Since the graph operation is invariant across different model architectures and training schemes with varied efficiency and scalability scenarios, it is possible to share the same formulation and implementation in these scenarios, which can be further enhanced by dedicated graph data management techniques.

2.2.2 Sampling-based GNN

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 mini-batches [27, 121, 120]. The message-passing scheme conducted in Eq. (2.2) 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 [27] typically perform random sampling to generate a smaller neighbor set of each node with equal size, while FastGCN [126] and LADIES [127] randomly samples nodes in the entire graph.

GAS [122] 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.

Another popular direction is **graph-wise** sampling with hierarchical consideration of the whole graph structure. Cluster-GCN [120] divides a graph into subgraphs based on the result of classical graph clustering algorithms. It requires $O(m)$ precomputation time of finding clusters, while the training time is bounded by $O(LmF + LnF^2)$.

GraphSAINT [121] 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.

SGNN-LS [128] investigates the utilization of spectral sparsification for sampling GNNs in the polynomial form, showcasing that graph sparsification process is useful in offering bounded approximation for message-passing GNNs.

2.2.3 Decoupled GNN

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 [16, 72]. 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:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-2, \tag{2.4}$$

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

$\boldsymbol{\Pi} \in \mathbb{R}^{n\times n}$ 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. (2.4), 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. (2.5) 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. (2.2), 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 [123] introduces the personalized PageRank (PPR) [129] algorithm in the propagation stage, where $\hat{\mathbf{\Pi}} = \sum_{l=0}^{L_P} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l$ is the PPR matrix of L_P hops. APPNP performs the post-propagation on the PPR matrix by an L_P-round Power Iteration [129], 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 $\mathbf{\Pi}$, hence is with a reduced complexity of $O(Ln_bF + LF^2 + nn_b)$, where n_b is the batch size.

The PPRGo model [124] further improves the efficiency of precomputing the PPR matrix $\mathbf{\Pi}$ by the Forward Push algorithm [130, 131] 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.

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:

$$\boldsymbol{H}^{(0)} = \boldsymbol{P} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l \cdot \boldsymbol{X}, \tag{2.6}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{2.7}$$

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

The line of Eq. (2.6) 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. (2.2). 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. (2.7) follows the neural network *feature transformation*, taking $\boldsymbol{P}$ as input feature. Compared to Eq. (2.5), 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 [125], the embedding matrix is given by a fix-hop multiplication of $\boldsymbol{P} = \tilde{\boldsymbol{A}}^{L_P} \cdot \boldsymbol{X}$. S^2GC [132] 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 [133] 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 [111] employs a PPR-based bidirectional propagation with hops $L_P = L$, and tunable layer-wise coefficient a_l and graph normalization ρ 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 [112] 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.

2.2.4 Heterophilous GNN

In the context of GNNs under heterophily, most current models belong to the **iterative** design, that they alter the vanilla propagation Eq. (2.2) 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 [71] 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

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

Model	Precomp. Time	Training Time	Inference Time
GPRGNN [134]	$O(m)$	$O(IL_Pmf + ILnF^2)$	$O(L_Pmf + LnF^2)$
GCNJK [65]	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
MixHop [135]	–	$O(IL_PLmF + ILnF^2)$	$O(L_PLmF + LnF^2)$
LINKX [136]	–	$O(ImF + ILnF^2)$	$O(mF + LnF^2)$

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

Model	Precomp. Mem.	Training Mem.	Inference Mem.
GPRGNN [134]	$O(m)$	$O(LnF + LF^2 + m)$	$O(LnF + LF^2 + m)$
GCNJK [65]	–	$O(L_Qnf + L_QF^2)$	$O(L_Qnf + L_QF^2)$
MixHop [135]	–	$O(QLnF + QLF^2)$	$O(CLnF + CLF^2)$
LINKX [136]	–	$O(L_Qn_bF + L_QF^2 + nF)$	$O(L_Qn_bF + L_QF^2 + nF)$

consisting of only strict 2-hop neighbors $\bar{\mathcal{N}}_2(u) = \{v \mid t \in \mathcal{N}(u), v \in \mathcal{N}(t), v \notin \mathcal{N}(u)\}$. The two representations are usually aggregated by a jumping knowledge layer.

MixHop [135] 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 [114] incorporates geometric measures besides node connections to build the overview of the entire graph, while GloGNN [72] 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 constituted by different graph propagation functions f, which is regarded as *channels*. Denote the number of channels as Q, employing multi-channel learning per layer respectively increases the memory budget for node representations and weight matrices by Q. In Table 2.4 we denote $L_Q = L + Q$ for simplicity.

GCNJK [65] 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 [137] 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 [138] 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 [139] 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 [140]. Evaluations show that simply fitting these models to learn from induced subgraph samples causes performance degradation [136].

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 [134] and LINKX [136] are the only models conceptually similar to this scheme, but both remain sensitive to the graph scale.

GPRGNN [134] 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 [136] 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.

2.3 Graph and GNN Simplification

2.3.1 Iterative Graph Sparsification

The application of graph simplification techniques on GNN can be traced back to the early stage of its development [69]. These studies modify the graph components, i.e., nodes and edges, in order to simplify the GNN learning process and bring performance enhancement. Simplification techniques mainly aim to reduce GNN message-passing operations while maintaining its scheme. Consequently, these methods usually enjoy better expressiveness and generality to different model architectures. Among them, a large portion of works sparsify edge connections for accuracy improvement, typically through deleting edges.

DropEdge [141] randomly samples edges for GNN training to facilitate better efficiency and alleviated over-smoothing. NeuralSparse [142] employs supervised learning to sample k-neighbor for each node and achieves accuracy improvement.

LSP [143] removes edges based on local topology, while AdaptiveGCN [144] and SGCN [145] choose to implement sparsity by learnable edge features and graph adjacency.

In comparison, FastGAT [146] and DSpar [147] relate graph modification to *spectral sparsification* mainly as an attempt to boost efficiency. FastGAT eliminates the GAT [28] model connections by calculating the Effective Resistance [148] metric, which nonetheless requires an additional $O(m \log n)$ computation time. DSpar employs a degree-based approximate calculation and successfully applies to larger datasets.

2.3.2 Decoupled Propagation Personalization

A particularized branch emerges for simplifying decoupled GNNs. Since the graph operation is isolated, it is more flexible for fine-grained modification, which replaces the graph-level message-passing with separate maneuver on the edge connection for each propagation hop. To make use of the model propagation and perform more adaptive maneuvers, some studies design fine-grained optimizations on each diffusion step in a node-dependent fashion to resolve GNN defects such as inefficiency [120, 111, 9] and over-smoothing [149].

NDLS [150] and NIGCN [151] personalize the hop number per node in the decoupled GNN design as an approach to acknowledge the local structure and mitigate oversmoothing. Shadow-GNN [152] explicitly selects the appropriate node-wise aggregation neighborhood.

Another set of works [153, 154, 155, 156] achieves customized diffusion by gradually learning the hop-level importance of nodes at the price of introducing more overhead, which are less relevant to the efficient GNN computation.

2.3.3 Architecture Compression

The concept of GNN pruning takes the neural network architecture into account and exploits sparsification for efficiency without hindering effectiveness [157]. [158] consider structural channel pruning by reducing the dimension of weight matrices, while CGP [159] appends a prune-and-regrow scheme on the graph structure.

A line of research refers to the *lottery ticket hypothesis* [160, 161] in the context of graph learning in search of a smaller yet effective substructure in the network. Among these approaches, GEBT [162] finds the initial subnetwork with simple magnitude-based pruning on both adjacency and weight matrices, whilst ICPG [163] learns the importance of edges as an external task. GLT [164] and DGLT [165] employ an alternative optimization strategy that progressively processes the graph and network components.

Other compression techniques such as *quantization* [166, 167, 168] are also investigated as approaches to reduce the GNN model size.

2.3.4 Graph Coarsening

Different from simplification, modification techniques change the message-passing scheme in GNNs, leading to new models with distinct expressiveness after modification. Graph coarsening describes the modification approach that reduces the graph size by contracting nodes into subsets.

GraphZoom [169], GOREN [170], and [171] explore various coarsening schemes and enhance training scalability, while GCond [172] alternatively adopts an analogous condensation strategy.

2.4 Sequential GNN

2.4.1 Vanilla Graph Transformer and Positional Encoding

A Transformer layer [87] first projects three representations given an input matrix $\boldsymbol{H} \in \mathbb{R}^{n \times F}$:

$$\boldsymbol{Q} = \boldsymbol{H}\boldsymbol{W}_Q, \quad \boldsymbol{K} = \boldsymbol{H}\boldsymbol{W}_K, \quad \boldsymbol{V} = \boldsymbol{H}\boldsymbol{W}_V, \tag{2.8}$$

where $\boldsymbol{W}_Q \in \mathbb{R}^{F \times F_K}$, $\boldsymbol{W}_K \in \mathbb{R}^{F \times F_K}$, $\boldsymbol{W}_V \in \mathbb{R}^{F \times F_V}$ are learnable weights.

For a multi-head self-attention module with N_H heads, each attention head possesses its own representations $\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i, i = 1, \cdots, N_H$, and then the output $\tilde{\boldsymbol{H}}$ across all heads is calculated as:

$$\tilde{\boldsymbol{H}}_i = \sigma\left(\frac{\boldsymbol{Q}_i \boldsymbol{K}_i^\top}{\sqrt{F_K}} + \boldsymbol{P}\right)\boldsymbol{V}_i, \quad \tilde{\boldsymbol{H}} = (\tilde{\boldsymbol{H}}_1 \| \cdots \| \tilde{\boldsymbol{H}}_{N_H})\,\boldsymbol{W}_O, \tag{2.9}$$

where $\cdot\|\cdot$ denotes the matrix concatenation operation, and σ is the softmax function. The projection dimension is usually set as $F_K = F_V = F/N_H$.

Beside the representation $\boldsymbol{H}$, positional encoding (PE) $\boldsymbol{P}$ in Eq. (2.9) can also incorporate graph topology into the GT attention module by encoding pair-wise information. Typical encoding approaches include graph proximity [89, 173, 174], Laplacian eigenvectors [88, 175, 176], and shortest path distance [177, 178, 179]. The complexity of such design depends on the specific acquisition of the pair-wise encoding.

2.4.2 Kernal-based Sequential GNN

The majority of vanilla GTs [88, 89, 90] are typically proposed for graph-level learning tasks on small graphs with full-batch training (FB). This is relevant to their quadratic complexity as revealed by Table 2.6, that the critical scalability bottleneck lies in Eq. (2.9) by representing n nodes with $O(n^2F)$ time and memory overhead.

To mitigate the quadratic complexity and foster scalable learning, scalable GTs employ mini-batch training, which replaces the full representation $\boldsymbol{H}$ with batches containing n_b nodes and reduces memory complexity to $O(n_b^2 F)$. Kernal-based GTs [181, 180, 182, 183] generate batches by neighborhood sampling (NS) and utilize graph kernels, i.e., functions

TABLE 2.5: Precomputation and training time complexity of sequential GNN models.

Taxonomy	Model	Precompute Time	Train Time
Vanilla (FB)	Graphormer [89]	$O(n^3)$	$O(Ln^2F)$
	GRPE [177]	$O(n^2)$	$O(Ln^2F)$
Kernel-based (NS)	GraphGPS [180]	$O(n^3)$	$O(LnF^2 + LmF)$
	NodeFormer [181]	–	$O(LnF^2 + LmF)$
	DIFFormer [182]	–	$O(LnF^2 + LmF)$
	PolyNormer [183]	–	$O(LnF^2 + LmF)$
Hierar-chical (RS)	NAGphormer [184]	$O(LmF_0)$	$O(LnF^2)$
	PolyFormer [185]	$O(LmF_0)$	$O(LnF^2)$
	ANS-GT [173]	$O(ns^2 + md^L)$	$O(LnF^2 + Lns^2F + Lm)$
	GOAT [186]	$O(nF)$	$O(LnF^2 + LmF)$
	HSGT [187]	$O(n + md^L)$	$O(LnF^2 + LmF)$

TABLE 2.6: Precomputation and training memory complexity of sequential GNN models.

Taxonomy	Model	RAM Mem.	GPU Mem.
Vanilla (FB)	Graphormer [89]	$O(n^2)$	$O(Ln^2F)$
	GRPE [177]	$O(n^2)$	$O(Ln^2F)$
Kernel-based (NS)	GraphGPS [180]	$O(nF + n^2)$	$O(Ln_bF + m)$
	NodeFormer [181]	$O(nF + m)$	$O(Ln_bF + m)$
	DIFFormer [182]	$O(nF + m)$	$O(Ln_bF + m)$
	PolyNormer [183]	$O(nF + m)$	$O(Ln_bF + m)$
Hierar-chical (RS)	NAGphormer [184]	$O(LnF)$	$O(Ln_bF^2)$
	PolyFormer [185]	$O(LnF)$	$O(Ln_bF^2)$
	ANS-GT [173]	$O(nF + ns^2 + md^L)$	$O(Ln_bF + n_bs^2)$
	GOAT [186]	$O(nF + m)$	$O(Ln_b^2 + Ln_bF + m)$
	HSGT [187]	$O(nF + md^L)$	$O(Ln_b^2 + Ln_bF + Lm)$

modeling node-pair relations, for attention computation to exploit edge connections. Typically, they necessitate iterative processing of graph data with $O(LmF)$ complexity throughout learning. When the graph scale is large, this term becomes dominant since the edge size m is significantly larger than the node size n. Hence, we argue that such a design is not sufficiently scalable.

GraphGPS [180] combines rich positional and structural encodings with interleaved local message-passing and linear global attention to achieve both expressivity and linear complexity in node and edge counts. However, its preprocessing stage (e.g., computing structural encodings) incurs an $O(n^3)$ time complexity.

NodeFormer [181] introduces a kernelized Gumbel-Softmax operator to enable linear-time all-pair message passing for scalable graph structure learning.

DIFFormer [182] proposes an energy-constrained diffusion framework that derives closed-form optimal diffusivity to build a diffusion-based Transformer encoder.

PolyNormer [183] presents a polynomial-expressive graph Transformer that learns high-degree equivariant polynomials via a linear local-to-global attention scheme.

2.4.3 Hierarchical Sequential GNN

Alternatively, the category of hierarchical GTs aims to embed rich topological information through the input data $\boldsymbol{X}$ so that the power of GT can be leveraged to learn the node identity and relations simultaneously. In a nutshell, this is usually achieved by encoding multiple levels of graph structures, e.g., node attributes, clusters, and global representations, as hierarchical input features. Its core design involves crafting an effective embedding scheme to comply with GT expressivity.

Since the graph topology is embedded in an permutation-invariant manner, mini-batching can be performed through random sampling (RS). The model can enjoy better scalability if the graph processing is fully independent of GT attention. Ideally, hierarchical GTs can process graph topology in $O(m)$ complexity in precomputation and employ RS during training for better scalability. Nonetheless, we note that existing models, except for NAGphormer and PolyFormer, still involve graph-level operations during training as in Table 2.5, which hinders GPU utilization and causes additional overhead.

NAGphormer [184] aggregates multi-hop neighborhood features into a fixed-length token sequence per node (via Hop2Token), enabling true mini-batch Transformer training on large graphs.

PolyFormer [185] builds a hierarchical Transformer by progressively coarsening the graph and exchanging information between levels to capture both local and global structure.

ANS-GT [173] uses an adversarial bandit to adaptively sample informative nodes and a two-stage local/global attention scheme to capture long-range dependencies. its adaptive sampling requires $O(ns^2 + Lm)$ preprocessing, making it costly on large graphs.

GOAT [186] implements approximate global self-attention via low-dimensional projection (K-Means) plus a local sampling module to support both homophilous and heterophilous graphs. however, the projection and codebook updates add extra overhead and approximation error.

HSGT [187] leverages multi-level graph coarsening and dedicated horizontal/vertical Transformer blocks to fuse information across scales. yet it demands hierarchical sampling and historical embedding maintenance, increasing implementation complexity.

2.5 Subgraph GNN

2.5.1 Vanilla Subgraph GNN

Subgraph GNN methods, i.e., subgraph-based graph representation learning (SGRL), have been extensively applied to a variety of tasks on graph-structured data [104, 188, 189, 190, 191]. The intuition of SGRL is to extract and learn from a subgraph around queried nodes to emphasize local structural information and enhance model expressiveness. Common SGRL methods adopt a pipeline of extracting subgraphs for each queried node, generating features from subgraph structure, and lastly learning predictions for the given task.

The representative SEAL [91, 92] displays the superiority of predicting links by representing the subgraph. SUREL [96] and SUREL+ [93] propose a scalable co-design framework enabling the online extraction and learning of the data. GDGNN [192] alternatively chooses to decouple the pipeline and applies separated computations.

Being the core of SGRL, subgraph extraction has been accentuated through the design of samplers, which can be further divided into two categories. **Walk-based samplers** [91, 92, 193, 194] generate a certain number of random walks starting from each seed node and encode such structural information as features. In comparison, **metric-based samplers** [195, 196, 192, 93] rely on graph metrics such as personalized PageRank (PPR) [197] for assessing the proximity between given nodes.

2.5.2 Dynamic Subgraph GNN

In order to learn from graphs with dynamic updates, some GNNs are equipped with techniques for obtaining and handling features for both temporal and topology information [116].

Early studies commonly exploit **sequential GNNs** based on Recurrent Neural Networks (RNNs) structures to deal with the sequential data. For example, ROLAND [198] proposes a learning strategy integrating RNN with the dynamic GNN, while D-DGNN [199] incorporates structural metrics and a simplified GNN into LSTM [200]. TGL [201] improves the graph storage for dynamic updates and network training.

Subgraph-based designs [202, 203] introduce specific metrics to sample representative subgraphs and embed temporal information. In particular, SGRLs featuring walk-based samplers represented by CAW [204] and NeurTWs [205] have the merit to extend random walks into the temporal dimension and efficiently retrieve graph dynamics. Intuitively, most recent updates of the graph have greater impact on the prediction result. Hence, they tend to extract and model only the relevant nodes closest to the current timestamp, greatly reducing the overhead in both structural and temporal domains. These dynamic solutions entail an extra round of graph update computation for the whole time range, which significantly impedes their deployment with streaming updates.

Particularly, since the temporal information is usually excessive for model learning, one of the key components in these methods is the temporal sampling strategy. Conventional models such as TGN [206], TGL [201], and GraphMixer [207] employ straightforward schemes such as most recent truncation or uniform sampling. More effective sampling based on temporal weights has also been explored in research such as FreeDyG [208].

2.6 Scalable GNN Design

2.6.1 Typical Scalable Methods

The *scalable GNN* is regarded as a family of model designs highlighting capability in processing large graphs, typically by reducing time and memory overhead during training and inference. Compared to GNNs focusing on efficacy, these designs prioritize

enhancements in efficiency and scalability, usually orienting the management of graph data [80, 8]. A general goal of scalable GNNs is to reduce or shift the computational overhead of graph operations so that the critical GPU memory bottleneck can be addressed by performing mini-batch training. From the perspective of graph operations, the majority of scalable GNN designs can be categorized into the following approaches.

Graph Partition: Due to constraint of GPU memory when loading large-scale graph data, a common model-agnostic solution is employing graph partition algorithms to divide the graph into smaller subgraphs. The approach is especially suitable for distributed learning, where subgraphs are allocated to multiple devices for training. Both classic partition techniques based on graph topology and those tailored for GNNs are utilized, and algorithmic goals include optimizing computational and communication overhead.

Graph Sampling: As introduced in Section 2.2.2, data sampling is one of the classical approaches for addressing the scalability issue in machine learning. Graph sampling implies randomly selecting specific graph nodes and edges according to certain metrics, and forming them as batches to learn during training iterations. Based on the scope of sample selection, strategies can be categorized into node-, layer-, and subgraph-level [70]. While the sampling strategy reduces computational overhead through learning, the iterative process ensures statistically similar learning outcomes.

Decoupled Graph Computation: As detailed in Section 2.2.3, the idea of decoupling is to separate graph propagation and feature transformation apart in message-passing GNNs. The implication of decoupling strategy is that, messages generated through graph propagation can be disentangled from layer-by-layer updates and instead learned in an aggregated fashion. By this means, graph operations can be conducted with dedicated algorithmic and device optimizations, which addresses the scalability bottleneck while retaining model capability.

System-level Optimization: Another prominent direction is to achieve scalability through dedicated system optimizations for the GNN training and inference pipeline, usually focusing on the canonical models [81]. These works look into the bottleneck operations such as input data loading, intermediate data transfer, model training blocking, and implement techniques including caching, streamlining, and batch processing to enhance the computation and realize better capability in learning large-scale graph data. The system-level optimization is largely orthogonal to algorithmic designs towards scalable GNNs, while the integration between them is a long-lasting promising topic.

2.6.2 Scalability Challenges and Objectives

Corresponding to the design goals and empirical observations, the scalability issue of GNNs can be elaborated in different perspectives. We specifically identify the following challenges under the topic of scalable GNNs.

Neighborhood Explosion: The intensive scale of neighborhoods in multi-layer model learning is a persistent issue hindering GNN time complexity and empirical performance. Various graph processing techniques have been proposed on the topic of how to conduct the graph-scale computation efficiently without losing graph information.

Limited Memory: The realistic graphs also poses practical challenges in storing and maintaining the large amount of data, especially in the highly-constrained GPU memory. Therefore, an array of scalable GNNs explores efficient management of the data by reducing or shifting the memory overhead.

Multi-scale: The graph property of heterophily is revealed to be dominant in tasks such as anomaly detection. In this case, nodes are connected with dissimilar neighbors, while conventional GNNs face difficulties due to their concentration on graph locality. Multi-scale GNNs mitigate the issue by supplementing non-local graph information. However, it comes into conflict with common scalable designs that tend to diminish global dependency. How to provide multi-scale ability to scalable GNN models is thus a challenging topic.

Fine-grained: While scalable GNNs are designed to retrieve information from a large amount of data, it is uncovered that their prediction accuracy decreases on certain graph nodes. Fine-grained operations are useful for elevating attention on specific nodes or personalizing graph propagations. As the manipulation easily incurs additional time and memory overhead, it also entails novel scalable solutions to adapt these particular managements.

Part I

Designing Scalable Graph Neural Network Models

Chapter 3

Decoupled 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 [27, 25, 126, 120], link prediction [28, 91, 209, 45], and community detection [210, 211, 212]. Recent studies have attempted to learn representations of large graphs such as the Microsoft Academic Graph (MAG) with 100 million entries [41, 46]. Nonetheless, directly fitting the basic models like GCN to such data would easily cause unacceptable training time or out-of-memory error.

Several techniques have been proposed towards more efficient learning for GNN, addressing the scalability issues. One optimization is to decouple graph propagation from feature

This work is published as a conference paper and a journal extension:

- [1] **Ningyi Liao***, Dingheng Mo*, Siqiang Luo, Xiang Li, Pengcheng Yin. "SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 15, no. 11, pp. 3240–3248, 2022. DOI: 10.14778/3551793.3551866. (*Both authors contributed equally to this paper.)
- [2] **Ningyi Liao**, Dingheng Mo, Siqiang Luo, Xiang Li, Pengcheng Yin. "Scalable Decoupling Graph Neural Network with Feature-Oriented Optimization". *The VLDB Journal*, vol. 33, no. 3, pp. 667–683, 2023. DOI: 10.1007/s00778-023-00829-6.

learning and employ simple model structures to speed up computation [125, 123], 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 [129] 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 [27, 213], layer sampling [126, 122], and subgraph sampling [120, 121, 171]. 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 [123, 63], self-attention [28, 214, 215], and quantization [166].

Unfortunately, such methods are nevertheless not efficient enough when applied to million-scale or even larger graphs. According to [112], the very recent state-of-the-art algorithm GBP [111] 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 as illustrated in Figure 3.1. Similar to previous models [125, 111], in precomputation stage we follow

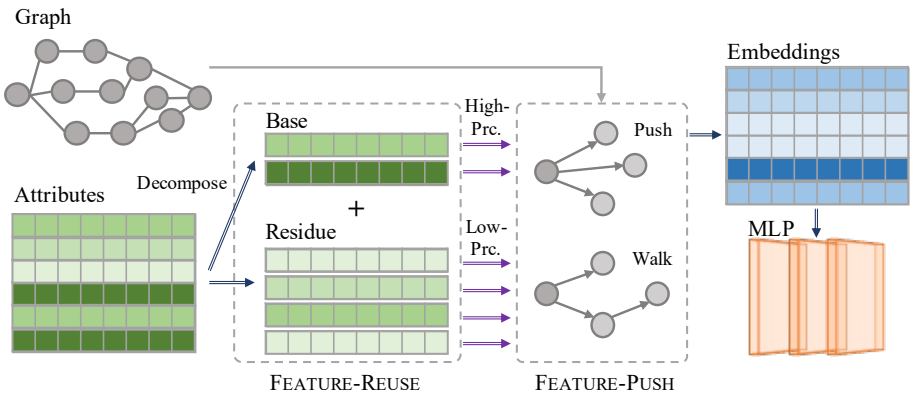


FIGURE 3.1: SCARA framework: Two feature-oriented algorithms FEATURE-PUSH ad FEATURE-REUSE for processing graph data.

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. (2.7). We enhance the model structure by incorporating skip connections [123] and dense connections [111] 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 continue from Eq. (2.6) where graph normalization is applied with factor ρ to distinguish in- and out-edges, and derive our propagation as:

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

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 graph normalization with factor $\rho \in [0, 1]$.

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 3.3 (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 $\tilde{\boldsymbol{A}}_{(\rho)}$ and node attributes $\boldsymbol{X}$. Most scalable methods such as APPNP [123] and

SGC [125] compute the propagation part separately from network training, resulting in a complexity at least proportional to edge size m. GBP [111] discusses a bidirectional propagation with both node-side random walk on $\boldsymbol{D}^{-1}\boldsymbol{A}$ and feature-side reverse push on $\boldsymbol{D}^{-\rho}\boldsymbol{X}$. Although the random walk step ensures precision guarantee, it requires long running time when not being accelerated by other methods [216, 217].

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:

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

Here, given the normalized features $\boldsymbol{D}^{1-\rho}\boldsymbol{X}$, single-source PPR algorithms can be alternated to efficiently propagate information with $\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^{l}$, one feature vector each time, without doing the actual iterative matrix multiplications. In order to better derive FEATURE-PUSH, we borrow the Personalized PageRank (PPR) notations to describe our technique manipulating feature vectors. On a graph G, given a source node $s \in \mathcal{V}$ and a target node $t \in \mathcal{V}$, 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 [112].

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 $\mathcal{U} \subseteq \mathcal{V}$ as the source nodes. Let n_U be the size of set $\mathcal{U}$. 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 $s \in \mathcal{U}$ 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 $\mathcal{U}$. 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 $\boldsymbol{x}_f$ $(1 \leq f \leq F)$ is the f-th column feature vector. Correspondingly, the

embedding matrix is $\boldsymbol{P} = [\boldsymbol{\pi}_1, \cdots, \boldsymbol{\pi}_F]$, where $\boldsymbol{\pi}_f = \boldsymbol{\pi}(\boldsymbol{x}_f)$ is the f-th column of PPR vector computed from feature $\boldsymbol{x}_f$, and is composed by $\boldsymbol{\pi}_f = \left(\pi(\boldsymbol{x}_f; t_1), \cdots, \pi(\boldsymbol{x}_f; t_{n_U})\right)^\top$ on all nodes. Calculating $\boldsymbol{P}$ from feature $\boldsymbol{X}$ is achieved by separately applying FEATURE-PUSH on each feature vector, which is exactly the implication of 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 $t \in \mathcal{U}$ 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 [218, 219], the approximate feature PPR enables employing efficient computation based on forward PPR algorithms without loss in eventual model effectiveness [216, 220]. We employ a scalable algorithm FEATURE-PUSH to compute the embedding matrix combining Forward Push [131] and Random Walk techniques that both operate

ALGORITHM 3.1: FEATURE-PUSH

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

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

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

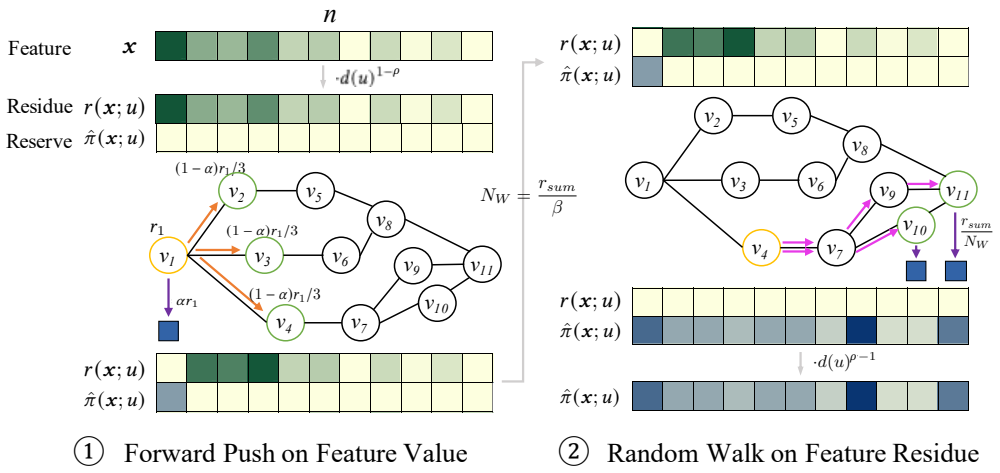


FIGURE 3.2: Illustration of the FEATURE-PUSH process. The input feature is transformed into the residue vector through the initial normalization $d(u)^{1-\rho}$. FEATURE-PUSH performs two consecutive steps, namely Forward Push on Feature Value and Random Walk on Feature Residue, to acquire the approximate feature PPR vector, which is output as the embedding after the remaining normalization $d(u)^{\rho-1}$. The two steps are related by the push parameter β.

feature vectors. The algorithm makes use of both approaches, that random walk is accurate but less efficient, while forward push is fast with a loose precision guarantee. Algorithms exploiting such combination have been the state of the arts in various PPR benchmarks [216, 221]. We highlight that the differences between Algorithm 3.1 and [216, 221] 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 $t \in \mathcal{U}$ through forward push (line 1-9 in Algorithm 3.1), then conducts compressed random walks to save computation (line 10-14). A running example is illustrated in Figure 3.2. 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 $t \in \mathcal{U}$ 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 $t \in \mathcal{U}$. 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 [222, 217], 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 [216], 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 [131, 216], 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.

We firstly present Algorithm 3.2 as a naive algorithm for reusing the features in a greedy manner searching for selecting base vectors $\boldsymbol{b}_i$ and approximating the feature set. Particularly, for each feature $\boldsymbol{x}_f$, it iteratively projects the vector to the closest base

ALGORITHM 3.2: FEATURE-GREEDY

Input: Graph $\mathcal{G}$, feature set $\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F]$, base set size F_B, decomposition threshold δ_0, reuse parameter γ, error bound λ

Output: Approximate embedding matrix $\hat{\boldsymbol{P}}$

$\beta_s \leftarrow \frac{\lambda^2}{(2\lambda/3+2)\cdot\log(2n)}$
$M(\boldsymbol{x}_f) \leftarrow 0$ for all $\boldsymbol{x}_f \in \boldsymbol{X}$, $\boldsymbol{X}_B = \emptyset$
for all $\boldsymbol{x}_f \in \boldsymbol{X}$ **do**
    $\boldsymbol{x}_{f^*} \leftarrow \arg\min_{\boldsymbol{x}_{f^*} \in \boldsymbol{X}} \|\boldsymbol{x}_{f^*} - \boldsymbol{x}_f\|_1$
    $M(\boldsymbol{x}_{f^*}) \leftarrow M(\boldsymbol{x}_{f^*}) + 1$
for i from 1 to F_B **do**
    $\boldsymbol{b}_i \leftarrow \arg\max_{\boldsymbol{b}_i \in \boldsymbol{X}} M(\boldsymbol{b}_i)$
    $\boldsymbol{X}_B \leftarrow \boldsymbol{X}_B \cup \boldsymbol{b}_i, \quad \boldsymbol{X} \leftarrow \boldsymbol{X} - \boldsymbol{b}_i$
for i from 1 to F_B **do**
    $\hat{\boldsymbol{\pi}}_i \leftarrow$ Apply Alg. 3.1 on $\boldsymbol{b}_i$ with $\beta^* = \gamma\beta_s$
for all $\boldsymbol{x}_f \in \boldsymbol{X}$ **do**
    $\theta_i \leftarrow 0$ for i from 1 to F_B
    $\boldsymbol{x}' \leftarrow \boldsymbol{x}_f, \quad \delta \leftarrow 1, \quad \vartheta \leftarrow 1$
    while $\vartheta \cdot \delta > \delta_0$ **do**
        $\boldsymbol{b}_i \leftarrow \arg\min_{\boldsymbol{b}_i \in \boldsymbol{X}_B} \|\boldsymbol{x}' - \boldsymbol{b}_i\|_1$
        $\vartheta \leftarrow \arg\min_{\vartheta} \|\boldsymbol{x}' - \vartheta\boldsymbol{b}_i\|_1, \delta \leftarrow \delta/2$
        $\boldsymbol{x}' \leftarrow \boldsymbol{x}' - \vartheta\boldsymbol{b}_i, \quad \theta_i \leftarrow \theta_i + \vartheta$
    $\boldsymbol{\pi}^*_f \leftarrow$ Apply Alg. 3.1 on $\boldsymbol{x}'$ with $\beta' = \left(1 - \gamma \sum_{i=1}^{F_B} \theta_i\right) \beta_s$
    for i from 1 to F_B **do**
        $\boldsymbol{\pi}^*_f \leftarrow \boldsymbol{\pi}^*_f + \theta_i \cdot \hat{\boldsymbol{\pi}}_i$
return $\hat{\boldsymbol{P}} = [\boldsymbol{\pi}^*_1, \cdots, \boldsymbol{\pi}^*_F]$

by evaluating distance in vector norms, until reaching an approximation threshold δ_0. Intuitively, it is less efficient due to the iterative computation of norm calculation and vector reduction.

To better formulate the FEATURE-REUSE algorithm, we here derive it in the form of an optimization problem under our matrix notation. On the input side, the algorithm aims to represent the feature matrix $\boldsymbol{X}$ by a partial of selected feature columns called base features. The number of base feature vectors is $F_B \ll F$ and they compose the base matrix $\boldsymbol{X}_B = \left[\boldsymbol{b}_1, \cdots, \boldsymbol{b}_{F_B}\right]$, $\boldsymbol{b}_f \in \boldsymbol{X}$. Then, the entire feature matrix $\boldsymbol{X}$ can be written as combinations of the bases:

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

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) [223, 224] when $\eta = \frac{1}{\sqrt{n}}$, which can be effectively solved by convex optimization methods such as alternating direction [225]. In general, [223] 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 Y and 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 Y does not guarantee the decomposition of $X_B\Theta$ that includes base features X_B inherited from 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 3.3 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

ALGORITHM 3.3: 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 $\quad \boldsymbol{Z} \leftarrow \texttt{Threshold}_{\mu}(\boldsymbol{X}' - \boldsymbol{Y} - \boldsymbol{E})$
5 $\quad \boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V} \leftarrow \texttt{SVD}_{F_B}(\boldsymbol{X}' - \boldsymbol{Z} + \boldsymbol{E})$, $\boldsymbol{Y} \leftarrow \boldsymbol{USV}$
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 $\quad \boldsymbol{\theta}_f \leftarrow \text{col}_f \boldsymbol{\Theta}$, $\boldsymbol{z}_f \leftarrow \boldsymbol{x}_f - \boldsymbol{X}_B\boldsymbol{\theta}_f$
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** $\check{\boldsymbol{P}} = [\check{\boldsymbol{\pi}}_1, \cdots, \check{\boldsymbol{\pi}}_F]$

to form the approximate PPR matrix (line 13-17). We separately elaborate on these two phases.

Base Selection on Matrix Portion. FEATURE-REUSE first optimizes the rank-F_B and sparse components from feature matrix. Line 3-6 in Algorithm 3.3 corresponds to the RPCA iterative solution [223], where $\texttt{Threshold}_\tau(x) = \text{sgn}(x)\max(|x| - \tau, 0)$ is shrinkage operation that zeros elements with absolute value smaller than threshold τ, and $\texttt{SVD}_k(\cdot)$ 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 [226]. 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 Y and Z are computed from 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 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 3.3 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 $\check{\boldsymbol{\pi}}(\boldsymbol{x}_f)$ on raw feature $\boldsymbol{x}_f$ can be constituted as line 17, reusing the PPR computation results of base features.

Approximation Precision. In Algorithm 3.3, 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*

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

The proof of Lemma 3.2 can be found in Appendix A.

Lemma 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 Lemma 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 3.3, 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 3.3 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:

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

Therefore, parameters β_B for base vectors and β_Z for remaining vectors satisfy Eq. (3.9). According to Lemma 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 3.3, 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 [131]. For random walks on feature residue, we employ the complexity derived by [216] 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 3.3, 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* $O(nF)$.

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 [25, 124] 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 [121] and GAS [122] are representative of different sampling-based algorithms. For post- and pre-propagation decoupling approaches, we respectively employ the most advanced PPRGo [124] and GBP [111]. 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

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	Class N_c	Split	Prob. α	Conv. ρ	Common
PPI [27]	56,944	818,716	50	121 (m)	79/11/10 (i)	0.3	0.0	
YELP [121]	716,847	6,977,410	300	100 (m)	75/10/15 (i)	0.9	0.3	
REDDIT [27]	232,965	114,615,892	602	41 (s)	01/04/96 (t)	0.5	0.5	$\lambda = 1 \times 10^{-4}$
AMAZON [120]	2,400,608	123,718,024	100	47 (s)	70/15/15 (i)	0.2	0.2	$F_B = 0.02F$
MAG [46]	27,394,820	366,143,207	200	100 (m)	01/01/99 (t)	0.5	0.5	$\gamma = 0.2$
PAPERS100M [43]	111,059,956	1,615,685,872	128	172 (s)	78/08/14 (t)	0.5	0.5	

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
GSAINT	14.4 (– 14.4)	166.2	13.7	41.6 ±4.8	– – –	–	OOM	–	– – –	–	OOM	–
GAS	1151 (– 1151)	**2.2**	14.0	38.2 ±0.3	– – –	–	OOM	–	– – –	–	OOM	–
PPRGo	79.4 (62.3+17.1)	29.1	9.4	41.5 ±2.3	711 (451+259)	85240	130	17.0 ±1.5	– – –	–	OOM	–
GBP	138 (124+13.7)	13.5	7.9	38.8 ±0.3	663 (569+94.4)	1452	173	34.8 ±0.1	– – –	–	OOM	–
SCARA (ours)	**13.9** (0.07+13.8)	10.7	**5.6**	**44.1 ±0.4**	**139** (11.6+127)	**1208**	**67.7**	**34.9 ±0.3**	**1346** (12.7+1333)	**4.7**	**71.4**	**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
GSAINT	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

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 aligned with optimal baseline results in [111]. 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 ρ, 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

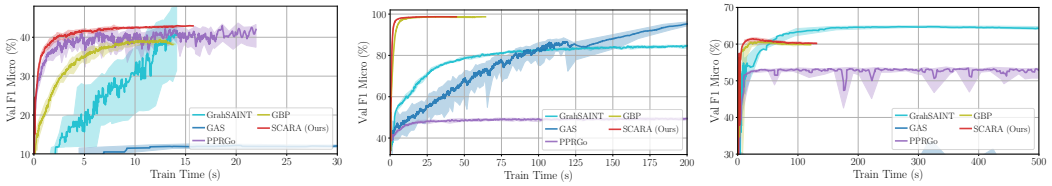
We evaluate the performance of SCARA and baselines in terms of both effectiveness and efficiency. Table 3.2 shows the average results of repetitive experiments on 6 large datasets, including the assessments on accuracy, memory, and the running time for different phases. Among them the key metric is learning 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 3.3.

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× 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 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.



(a) REDDIT **(b)** PPI **(c)** YELP

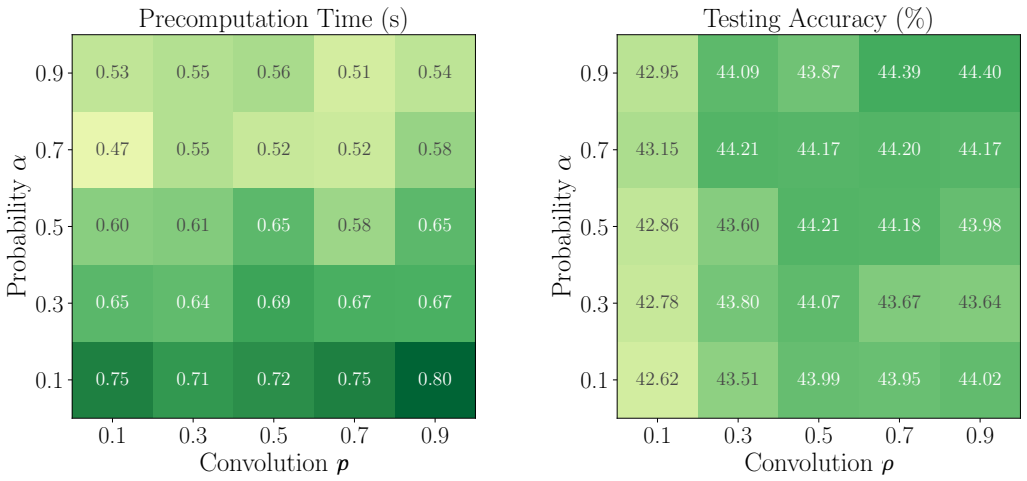
FIGURE 3.3: Validation F1 convergence curves of SCARA and baseline models on **(a)** REDDIT, **(b)** PPI, and **(c)** YELP datasets. Curves only represents the process of training phase. Shaded area is the result range of multiple runs.

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 3.3 shows the validation F1-score versus training time on representative datasets and corresponding GNN models. It can be observed that when comparing the time consumption to convergence, the SCARA model is efficient in reaching the same precision faster than most methods. The performances of GAS and PPRGo in the figure are relatively suboptimal because they are relatively less stable and require more time to converge beyond the display scopes in Figure 3.3. It is worth noting that some baselines fail to or only partially converge before training terminates in tasks such as PPI.

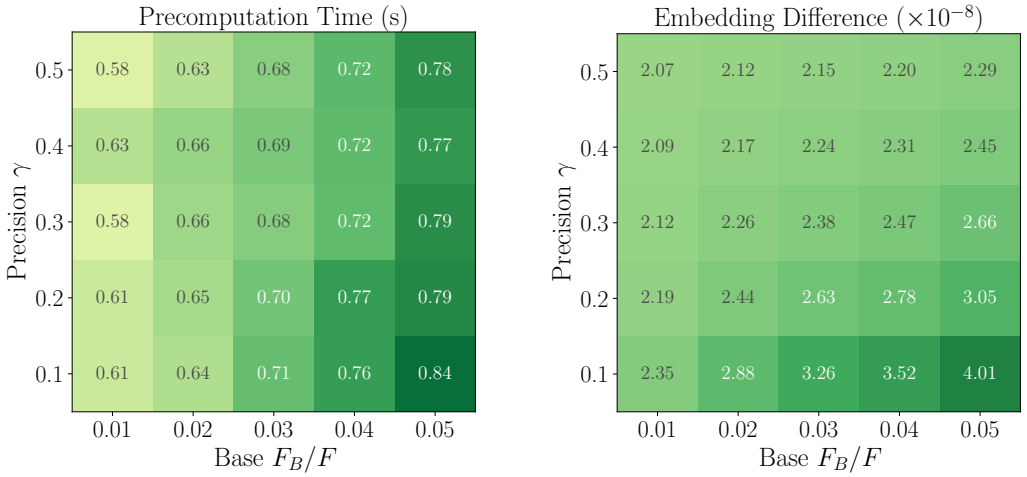
3.3.3 Effect of Parameters

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 ρ controls the normalization strength of the degree matrix $\boldsymbol{D}$ as shown in Eq. (3.1). Especially, when $\rho = 0.5$, it degrades to the normalized adjacency matrix $\tilde{\boldsymbol{A}}$ presented in APPNP [123] and PPRGo [124]. 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.



(a) Precomp. Time **(b)** Testing Accuracy

FIGURE 3.4: Effect of propagation parameters teleport probability α and convolution coefficient ρ on SCARA **(a)** efficiency and **(b)** testing accuracy on REDDIT dataset.



(a) Precomp. Time **(b)** Embedding Diff.

FIGURE 3.5: Effect of reuse precision parameter γ and base set size F_B on SCARA **(a)** precomputation time and **(b)** average embedding value difference on REDDIT dataset.

Regarding the default selection in Table 3.1, we set the values of α and ρ for shared datasets mainly in accordance to GBP [111, 112] in order to produce comparable results. The rest REDDIT and MAG are employed with the following strategy: we use $\rho = 0.5$ for better comparison and generality, while α is decided based on graph edge density [124]. 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 3.4 on the value ranges of teleport probability α and

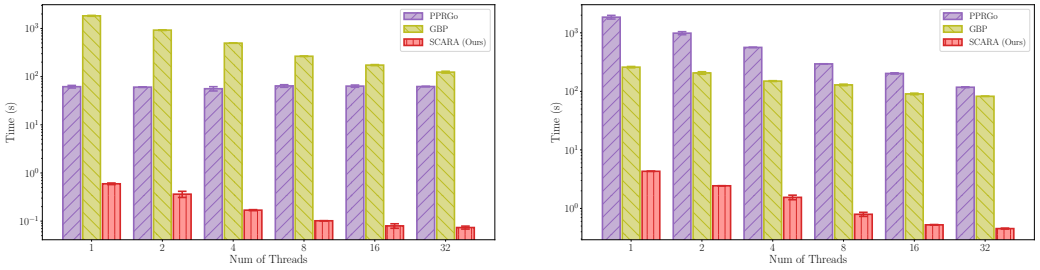
convolution coefficient ρ to examine their effect. In order to prevent potential influence, we use the single-thread scheme for experiments in this section. It can be inferred that a larger α has a slight improvement on precomputation efficiency, while ρ has no significant impact. This is because in 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 α and ρ 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 3.5 presents the result on precomputation time and precision on REDDIT dataset. For comparison, single-thread repetitive FEATURE-PUSH precomputation without FEATURE-REUSE uses 2.37s. It can be observed that both F_B and γ influence FEATURE-REUSE efficiency for less than ± 0.2s. Intuitively, a larger set of base features F_B requires more additional calculation time, hence hinder the overall efficiency. On the contrary, the factor γ affects less on performance as the residue vectors are still processed by subsequent calculations. The difference of embedding values is at the level of 10^{-8}, which is significantly smaller than the algorithmic error bound $\lambda = 10^{-4}$. Generally, a more aggressive reuse scheme results in relatively higher average approximation errors of the embedding values. We hence conclude that our parameter settings $F_B/F = 0.02$ and $\gamma = 0.2$ are effective for the general evaluation of SCARA.

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



(a) REDDIT **(b)** AMAZON

FIGURE 3.6: Precomputation time of SCARA and decoupling baselines with different parallel schemes on **(a)** REDDIT and **(b)** AMAZON datasets. Note that both axes are on a log scale.

into the similar parallel scheme by design. Figure 3.6 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 3.3. In following notation, FEATURE-REUSE is SCARA with FEATURE-REUSE in precomputation following Algorithm 3.3. We compare it with the bare iterative full-precision FEATURE-PUSH without FEATURE-REUSE, and the naive greedy-search based reuse scheme FEATURE-GREEDY. Similarly, we test all related methods in single-thread execution to avoid noise.

We here consider the feature size as a factor of particular interest, as FEATURE-REUSE is a feature-oriented optimization design. We sample the node feature vectors $\boldsymbol{x}$ in the REDDIT dataset to generate feature matrices $\boldsymbol{X} \in \mathbb{R}^{n \times F'}$ with different feature numbers F'. Using these features as input, we respectively evaluate the performace of graph learning. The results of average times and testing accuracies for the three variants are given in Table 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, FEATURE-REUSE achieves 3.6× speed-up compared to FEATURE-PUSH, and 2.4× speed-up compared to FEATURE-GREEDY.

Examining the reuse precision, it is inferred from Table 3.3 that FEATURE-REUSE produces less estimation error with regard to embedding values, which implies that the convergent optimization solution is not only faster but also more precise than the FEATURE-GREEDY search. Meanwhile, FEATURE-REUSE causes no significant difference on model effectiveness, as minor accuracy fluctuations are under the error bound of repetitive experiments. Interestingly, even with a feature dimension of $F' = 100$, the model achieves 32.8% testing accuracy, indicating that our feature PPR embedding matrix is capable to store adjacency and feature information that is sufficient for model learning.

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

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.

Another promising future direction is applying the idea of decoupled computation and efficient graph propagation to dynamic data. The intuition is that compact graph computations, such as PPR, can be efficiently updated under graph changes. Hence, it is possible to partially update the graph representation and perform incremental model training. [227] covers some of the ideas for dynamic graphs, while the effective and efficient implementation of dynamic and scalable GNNs remains largely under-explored. An alternative approach is Spatial-Temporal GNNs (STGNNs), which learn node attributes and dependencies with sequential inputs changing dynamically over time [228]. The general idea of STGNNs is to aggregate spatial information with basic GNN layers and extract temporal sequences with RNN structures simultaneously. However, these works often cannot achieve the desired performance in an online fashion, as they are based on classic GNNs known to have a high training cost.

Chapter 4

Heterophilous Graph Neural Network with Decoupled Embeddings

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 [53]. This inductive bias introduces additional information from the graph structure and improves model performance in applicable tasks [54].

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 [113]. 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 these applications [229, 140]. 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

This work is published as: [3] **Ningyi Liao**, Siqiang Luo, Xiang Li, Jieming Shi. "LD^2: Scalable Heterophilous Graph Neural Network with Decoupled Embeddings". In *36th Conference on Neural Information Processing Systems (**NeurIPS**)*, pp. 10197–10209, 2023.

graph relations [135, 71, 114, 71, 230, 134], and retrieving expressive node information through enhanced network architectures [137, 138, 139, 65, 72, 118].

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 [49]. 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 [136] 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 LD^2, 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, LD^2 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 LD^2 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. Since the decoupled design relies on fixed embeddings, we embrace the multi-channel architecture [231, 139] 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 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 LD² is depicted in Figure 4.1 and

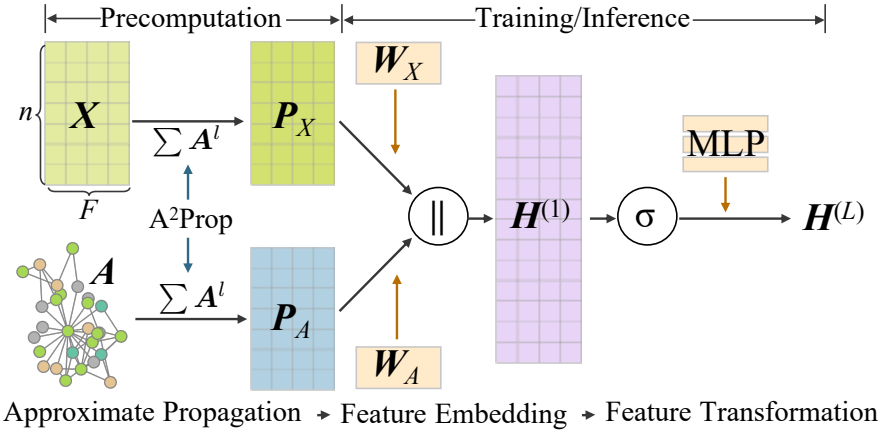


FIGURE 4.1: LD² framework: decoupled precomputation and training.

can be expressed as follow: The overall framework can be expressed as follow:

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

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

The overall learning pipeline is as follows: Given an input graph, Algorithm 4.1 is applied to compute the embeddings $\boldsymbol{P}_A$ and $\boldsymbol{P}_X$ following Eq. (4.1) in the precomputation stage. The computation is performed only once, and the static embeddings are saved for future use. Second, the model parameters are trained, and the inference for node representation follows Eq. (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

In this study, we aim to explore node embeddings that can effectively retrieve graph information under heterophily, while also being easily achieved through decoupled graph computation. As Section 2.2.4 suggests, homophily-oriented graph propagation is hindered by heterophilous graph connections. Therefore, we specifically seek to incorporate information beyond local homophily from both graph structure and node attributes into our embeddings under heterophily.

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 [113, 114, 136]. Particularly, the most informative aspects are often associated with 2-hop

neighbors, i.e., "neighbors of neighbors" of ego nodes. [71] 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:

$$\sum_{u,v} A^2(u,v)\|\boldsymbol{P}(u) - \boldsymbol{P}(v)\|^2 = 2\sum_u d_2(u)\|\boldsymbol{P}(u)\|^2 - 2\sum_{u,v} A^2(u,v)\boldsymbol{P}(u)\boldsymbol{P}(v) \tag{4.5}$$

$$= 2\,\mathrm{tr}(\boldsymbol{P}^\top\boldsymbol{D}_2\boldsymbol{P} - \boldsymbol{P}^\top\boldsymbol{A}^2\boldsymbol{P}). \tag{4.6}$$

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 [232]. 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 [136] introduced in Section 2.2.4, $\boldsymbol{P}_A$ 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 [129, 233, 114, 230, 138], 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 [123, 125, 133, 124, 132, 112]. 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:

$$\boldsymbol{P}_X = \sum_{l=1}^{L_P} \theta_l \boldsymbol{T}^l \boldsymbol{X}, \tag{4.7}$$

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

LD^2 jointly utilizes the following feature channels in Eqs. (4.8) to (4.10):

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

$$\boldsymbol{P}_{X,H} = \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} \tilde{\boldsymbol{L}}^l \boldsymbol{X}; \tag{4.8}$$

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

$$\boldsymbol{P}_{X,L2} = \frac{1}{L_{P,L2}} \sum_{l=1}^{L_{P,L2}} \bar{\boldsymbol{A}}^{2l} \boldsymbol{X}; \tag{4.9}$$

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

$$\boldsymbol{P}_{X,0} = \boldsymbol{X}. \tag{4.10}$$

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 [135, 67, 139, 72].

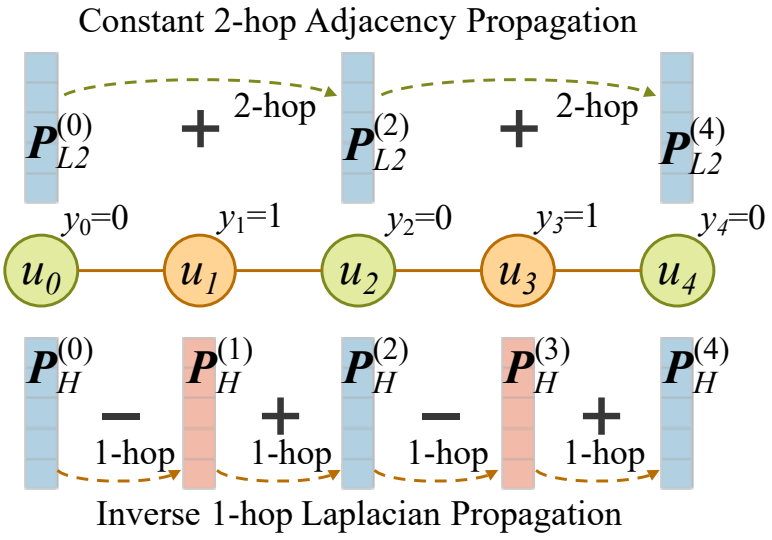


FIGURE 4.2: Two types of LD^2 propagations under heterophily.

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 [71, 118].

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 [132]:

$$\boldsymbol{P}_{X,L} = \underset{\|\boldsymbol{P}(u)\|=1,\forall u\in V}{\arg\min} \sum_{u,v\in V} \tilde{A}(u,v)\|\boldsymbol{P}(u) - \boldsymbol{P}(v)\|^2 + \|\boldsymbol{P} - \boldsymbol{X}\|_F^2. \tag{4.11}$$

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

$$(\boldsymbol{I} - \tilde{\boldsymbol{A}})\boldsymbol{P} - \boldsymbol{X} = 0. \tag{4.12}$$

Therefore the solution is:

$$\boldsymbol{P}^*_{X,L} = (\boldsymbol{I} - \tilde{\boldsymbol{A}})^{-1}\boldsymbol{X} = \sum_{l=0}^{\infty} \tilde{\boldsymbol{A}}^l \boldsymbol{X}. \tag{4.13}$$

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 S^2GC [132] as an approach for balancing locality and multi-hop propagation. Its interpretation can be observed from its objective function Eq. (4.11), 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. (4.11) with $\bar{A}^2(u,v)$ yields our constant 2-hop embedding $\boldsymbol{P}_{X,L2}$. It shares similar spectral properties with S^2GC 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:

$$\boldsymbol{P}_{X,H} = \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} \tilde{\boldsymbol{L}}^l \boldsymbol{X} = \frac{1}{L_{P,H}} \sum_{l=1}^{L_{P,H}} (\boldsymbol{I}\boldsymbol{X} - \tilde{\boldsymbol{A}}^l \boldsymbol{X}) = \boldsymbol{X} - \boldsymbol{P}_{X,L}. \tag{4.14}$$

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 [137]. 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 [234]. 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 $1 - \phi$,

$$\|\boldsymbol{P}(u) - \hat{\boldsymbol{P}}(u)\| \leq \epsilon \cdot \|\boldsymbol{P}(u)\|. \tag{4.15}$$

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 (A^{2}Prop). 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 $\alpha_T(u,v)$, 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.

ALGORITHM 4.1: A^{2}Prop: 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 mod 2 = 1 and $l < L_P - 1$ **then**
    $\boldsymbol{P}_A \leftarrow$ `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 $\boldsymbol{P}_A, \boldsymbol{P}_X$

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 [112].

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 $O(L_P mF)$. 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 LD^2 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 [114, 136] for the transductive node classification task, with the largest available graph WIKI (m = 244M) included. We leverage settings as per [136] such as the random train/test splits and the induced subgraph testing for GSAINT-sampling models, while addressing several issues revealed by [235] 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.

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 [124] and SGC [125] representing decoupled schemes for homophilous graphs. GCNJK [65] and MixHop [135] are GNNs under non-homophily. GSAINT random walk sampling [121] is utilized to empower them for minibatching. LINKX is the decoupled heterophilous GNN proposed by [136]. Simple i.i.d. node batching is adopted for decoupled networks.

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 [114]	5,201	401,907	77.275	2,089	5	–	0.217	0.214
PENN94 [136]	41,536	1,403,756	33.796	4.814	2	–	0.504	0.478
ARXIV-YEAR [136]	169,343	1,327,142	7.837	128	5	directed	0.289	0.337
GENIUS [136]	421,858	1,344,722	3.188	12	2	–	0.368	0.823
TWITCH-GAMERS [136]	168,114	6,965,671	41.434	7	2	–	0.562	0.531
POKEC [136]	1,632,803	23,934,767	14.659	65	2	–	0.454	0.605
SNAP-PATENTS [136]	2,738,035	16,705,984	6.101	269	5	directed	0.220	0.298
WIKI [136]	1,770,981	244,278,050	137.934	600	5	–	0.306	–

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 [136] 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).

4.3.2 Performance Comparison

The main evaluations of LD^2 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, LD^2 speeds up the minibatch training process by

TABLE 4.2: Average test accuracy (%) of minibatch LD^2 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.

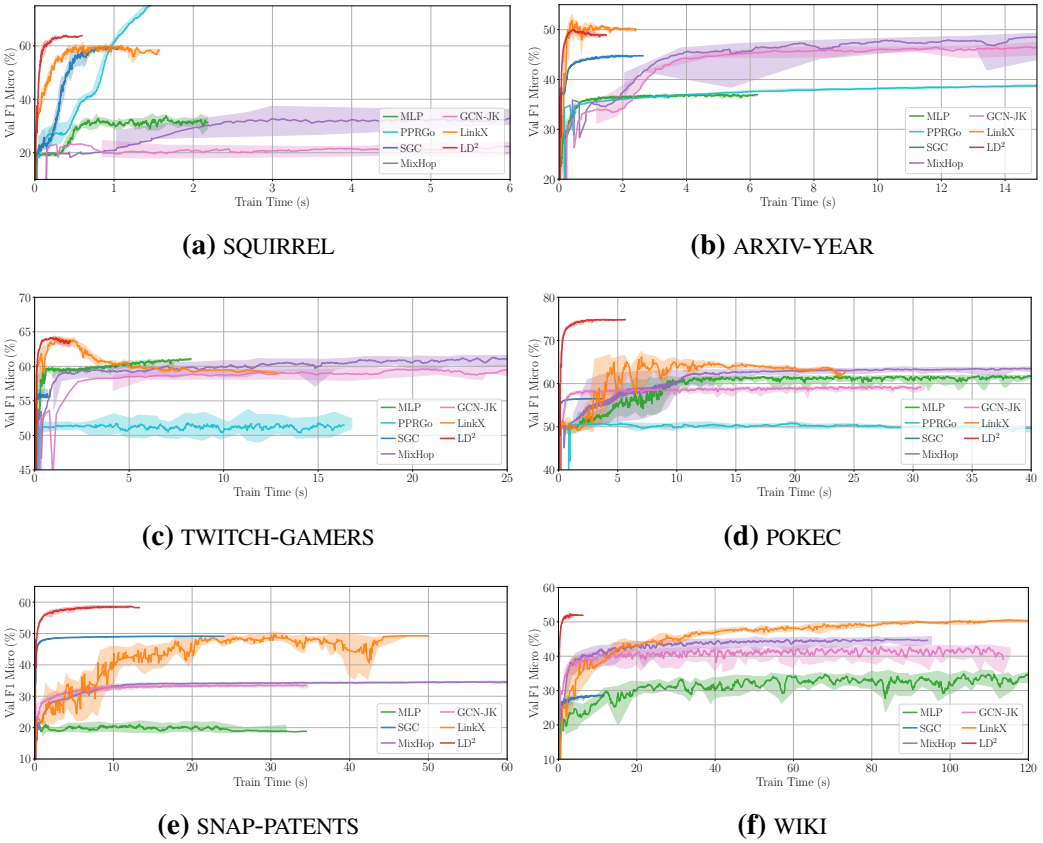
Dataset	SQUIRREL	GENIUS	PENN94	ARXIV-YEAR	TWITCH	POKEC	SNAP-PATENTS	WIKI
MLP	33.16 ±0.59	82.47 ±0.06	74.41 ±0.48	37.23 ±0.31	61.26 ±0.19	61.81 ±0.07	23.03 ±1.48	35.64 ±0.10
PPRGo	33.95 ±0.49	79.81 ±0.00	58.75 ±0.31	39.35 ±0.12	47.19 ±2.26	50.61 ±0.04	(> 12h)	(> 12h)
SGC	59.39 ±0.62	79.85 ±0.01	68.31 ±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
LD^2 (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 LD^2 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
LD^2 (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.

3–15 times, with an acceptable precomputation cost. Its inference time is also consistently below 0.1 seconds. The outstanding efficiency of LD^2 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 LD^2 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 LD^2 exhibits optimized training and inference complexity at the same level as simple models.



(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.

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, LD^2 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 features into informative embeddings. Consistent with the previous studies [136], 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 LD^2 outperforms other baseline methods on most datasets, demonstrating more stable curve, faster convergence, and significantly shorter overall training time. Due to the simple neural network structure, the model quickly leads to overfitting on datasets such as TWITCH-GAMERS and ARXIV-YEAR. 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 LD^2. Particularly, the PPRGo model is so large that it overfits on validation sets of small graphs such as SQUIRREL.

4.3.3 Effect of Embedding Schemes

Embedding Schemes. We conduct additional exploration on validating the effectiveness of utilizing $\boldsymbol{A}^2$ adjacency spectral embedding (ASE) as the LD^2 embedding scheme. Experiments of other schemes are shown in Table 4.4, including the shortest path distance used as spatial encodings in Graphormer SPD [89] and node2vec embedding [236]. There are also rank-F approximations in Eq. (4.3) by replacing $\boldsymbol{A}^2$ with $\boldsymbol{A}$ and $\boldsymbol{A}^3$, which are respectively denoted as ASE($\boldsymbol{A}$) and ASE($\boldsymbol{A}^3$).

We compare the advantages of our proposed ASE($\boldsymbol{A}^2$) with other structural embeddings such as SPD from three aspects. Regarding **effectiveness**, as we analyzed in Section 4.2.2, the ASE embedding is able to capture structural information especially on the homophilous components of the 2-hop graph, while SPD is more specific to encode distance information of directly connected nodes and node2vec represents local neighborhood, which are less suitable for heterophilous graphs.

As for **memory efficiency**, ASE is a low-dimensional embedding of shape $n \times F$, where the feature dimension F is generally much smaller than the graph scale. This implies better scalability compared to SPD embedding which is a dense $n \times n$ matrix. Node2vec requires additional $O(nd^2)$ space for storing the random walk results. As the average degree $d = m/n$ is at the scale of $O(\log n)$ to $O(n)$, node2vec is less scalable especially to large and dense graphs, which explains the OOM error on genius.

ASE also benefits from better **time efficiency** in our model as described in Section 4.2.4, which can be computed along with feature embeddings with a complexity linear to edge size m, while node2vec exhibits the same $O(nd^2)$ complexity. The node2vec design is based on random walks, which is known to be less efficient for time efficiency and cache locality, which further degrades its scalability. Additionally, our ASE and other embeddings can be efficiently computed by an end-to-end algorithm as described in Section 4.2.4.

Robustness under Noise. We also conduct experiments to evaluate the model performance under different levels of noise and incompleteness. The results are shown in Table 4.5. We mainly consider three types of noises, which are analyzed respectively as following:

Push threshold: We vary the threshold δ_P in Algorithm 4.1 to control the precision of propagation. A larger δ_P implies less precise propagation ignoring small feature values,

TABLE 4.4: Performance of LD^2 with alternative adjacency embeddings on selected datasets. Particularly, $\boldsymbol{P}_A$ = ASE($\boldsymbol{A}^2$) indicates the proposed LD^2 model with adjacency spectral embedding denoted in Eq. (4.3). "Pre." and "RAM" respectively refer to precomputation time (s) and peak GPU memory (GB).

$\boldsymbol{P}_A$	ASE($\boldsymbol{A}^2$)			ASE($\boldsymbol{A}$)			ASE($\boldsymbol{A}^3$)			**node2vec**			**SPD**		
Dataset	Acc	Pre.	RAM	Acc	Pre.	RAM	Acc	Pre.	RAM	Acc	Pre.	RAM	Acc	Pre.	RAM
SQUIRREL	66.87	3.87	0.64	60.95	2.47	0.47	62.26	3594.90	0.61	50.19	1620.58	4.58	54.50	323.30	0.99
PENN94	75.52	27.19	0.53	74.09	1.83	0.40	74.36	306.85	0.72	73.15	24654.00	22.82	(> 12h)		
GENIUS	85.31	0.79	0.60	84.68	0.63	0.58	84.56	293.28	0.62	(OOM)			(> 12h)		

while $\delta_P = 10^{-5}$ is the original setting. It can be seen that by improving the precision, the final learning accuracy does not change significantly. It indicates that our setting of $\delta_P = 10^{-5}$ is sufficient for propagation and does not affect the learning performance.

Edge removal: We randomly remove a percentage of edges to generate an incomplete variant of the graph. The LD^2 model is then applied to learn on the incomplete graph. The removal causes a negative impact on the accuracy. However, as the node attributes $\boldsymbol{X}$ are kept unchanged under the noise, the model is still able to achieve a reasonable performance.

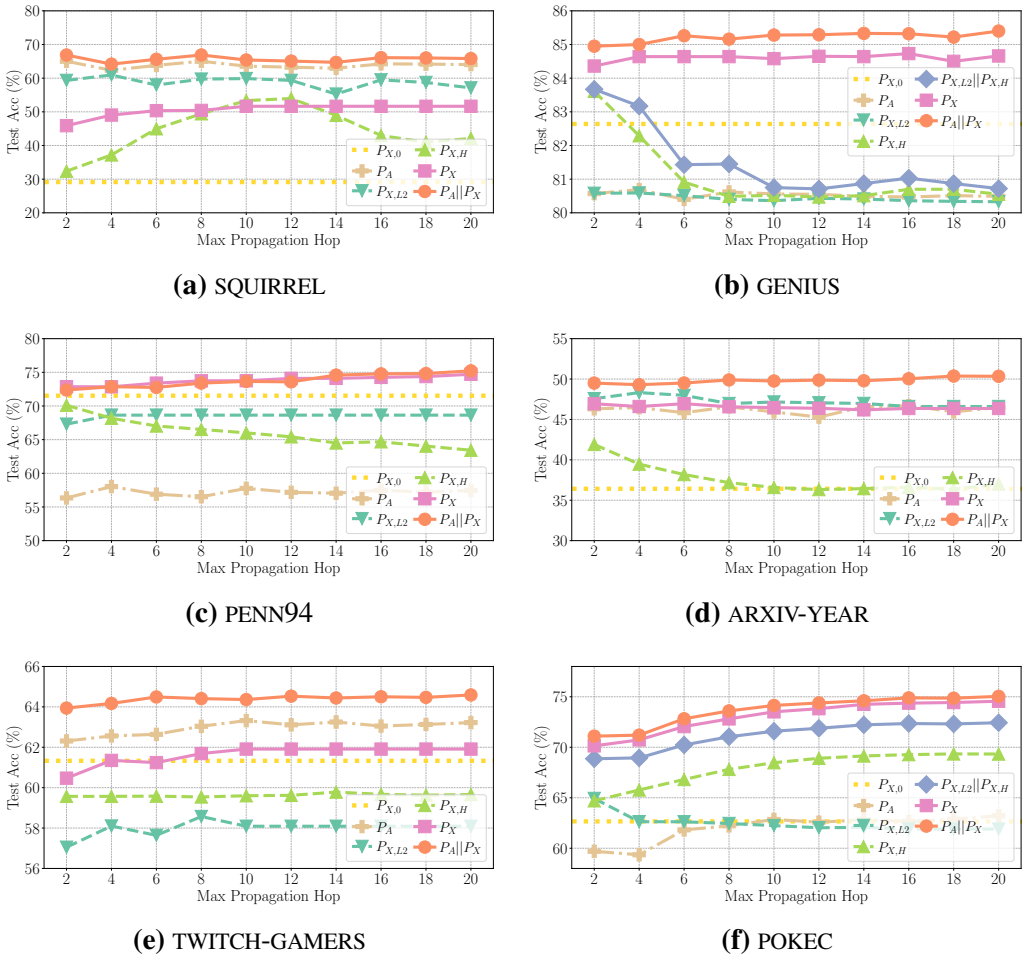
Attribute noise: We apply Gaussian noise with standard deviations proportional to the deviation of each feature dimension to the raw node attribute matrix $\boldsymbol{X}$ before pre-computation. This is more aggressive as the noise level is much larger than the scale of propagation precision.Consequently, the model suffers a more significant accuracy reduction. However, as the noise level increases, the model's performance converges towards the performance achieved by only learning the adjacency embedding $\boldsymbol{P}_A$. This is because the adjacency information is unaffected under such kind of noise.

4.3.4 Effect of Parameters

Propagation Channels and Hops. To gain deeper insights into the multi-channel embeddings of LD^2, 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.

TABLE 4.5: Performance of LD^2 with different noisy data on selected datasets. Particularly, $\delta_P = 10^{-5}$ is the original LD^2 model result presented in main experiments.

Noise	**Push Threshold δ_P**			**Edge Removal**			**Attribute Noise**		
Dataset	10^{-5}	10^{-6}	10^{-7}	10%	20%	40%	0.5σ	1σ	2σ
PENN94	75.52	75.56	75.57	72.63	72.11	71.53	69.39	67.41	62.57
GENIUS	85.31	85.29	85.16	84.98	84.77	84.30	81.29	81.22	81.18



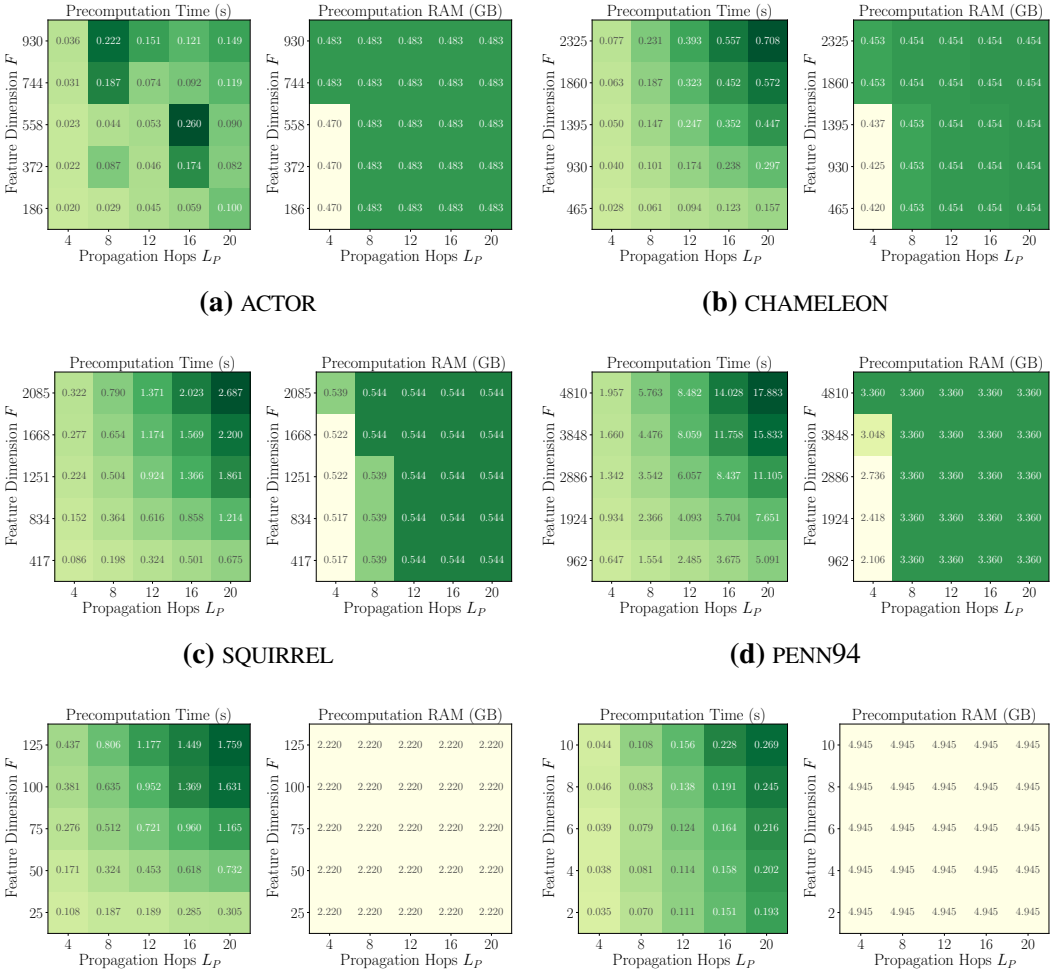
(a) SQUIRREL **(b)** GENIUS

(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.

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



(a) ACTOR **(b)** CHAMELEON

(c) SQUIRREL **(d)** PENN94

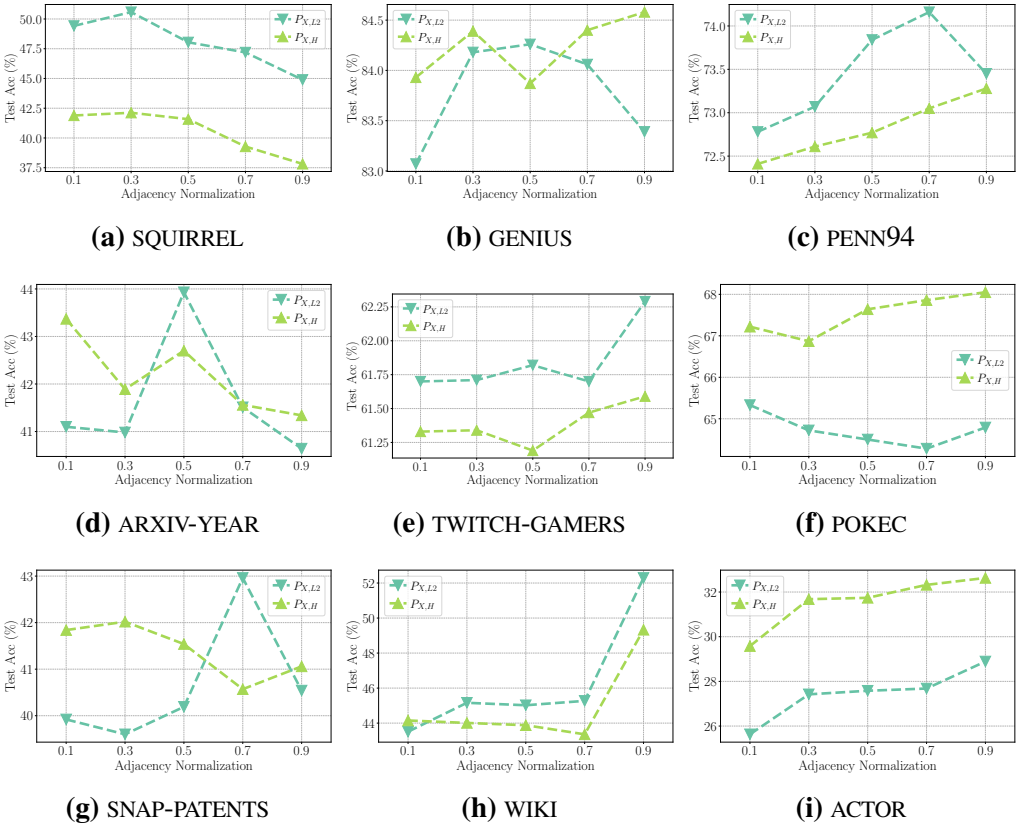
(e) ARXIV-YEAR **(f)** GENIUS

FIGURE 4.5: Effect of propagation hops and feature dimensions on A^{2}Prop precomputation time and memory overhead on 6 heterophilous datasets.

implicit information of various frequencies and scales that is important in the presence of heterophily.

Propagation Efficiency. The maximal propagation hop L_P and feature dimension F also affects the A^{2}Prop precomputation algorithmic efficiency. To study the effect, we change L_P in $[4, 8, 12, 16, 20]$ and the target dimension F in $[F/5, 2F/5, 3F/5, 4F/5, F]$ to perform A^{2}Prop computation. For precision related hyperparameters in A^{2}Prop, we commonly use relative error bound $\epsilon = 0.1$, norm threshold $\delta = 1 \times 10^{-5}$, and failure probability $\phi = 0.01$.

Results on 6 datasets with respect to precomputation time and RAM memory are displayed in Figure 4.5. As the graph and feature size scale up, the time and memory overhead of A^{2}Prop also increase, mostly linear to the scale of n and F, which is in consistent to



(a) SQUIRREL **(b)** GENIUS **(c)** PENN94

(d) ARXIV-YEAR **(e)** TWITCH-GAMERS **(f)** POKEC

(g) SNAP-PATENTS **(h)** WIKI **(i)** ACTOR

FIGURE 4.6: Effect of A²Prop adjacency normalization on the effectiveness of different feature embedding channels on 9 heterophilous datasets.

our complexity analysis. Figure 4.5 also implies that, when the graph scale is large, the dominant factor affecting the precomputation time is the number of propagation hops L_P, while the influence from F is relatively minor. When varying L_P and F, the RAM memory overhead of A²Prop merely changes, indicating its efficient usage.

Graph Normalization. For graph adjacency normalization coefficients $a, b \in [0, 1]$ applied to features, we uniformly use $b = 1 - a$ and only tune the coefficient a for the normalized feature embeddings $P_{X,L2}$ and $P_{X,H}$. We explore the choices respectively for each dataset.

Results of changing normalization are shown in Figure 4.6 on 9 heterophilous graphs. Different graphs have various favors of adapting the normalization, considering the varying implicit meanings of their features. In general, the accuracy gap is not significant for most varying parameter values. Note that the entire LD² model comprehensively extracts information from different inputs, we hence state that it is relatively not sensitive to the normalization parameter value.

4.3.5 Case Study

In order to intuitively illustrate the effect of approximate propagation used in LD^2, here we consider a toy example. Figure 4.7 depicts a graph with 9 nodes and 10 edges. Its nodes belong to 3 classes, and the connections are mostly heterophilous.

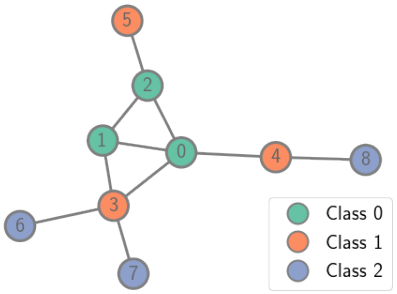


FIGURE 4.7: An example heterophilous graph where $\mathcal{H}_{n,1} = 0.204$.

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 4.8. Nodes inside the same class are of the same value in each 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.

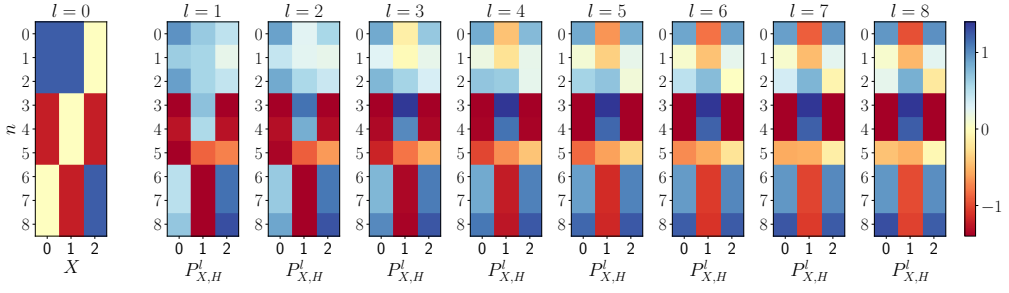


FIGURE 4.8: Progression of inverse Laplacian embedding with increasing propagation hops on given positive-negative distribution raw feature.

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 LD^2 adapting to certain patterns of input features, such as one-hot encoding. We believe this is partially the reason that LD^2 with only feature embeddings achieves suboptimal accuracy on graphs with such one-hot features including SQUIRREL and PENN94.

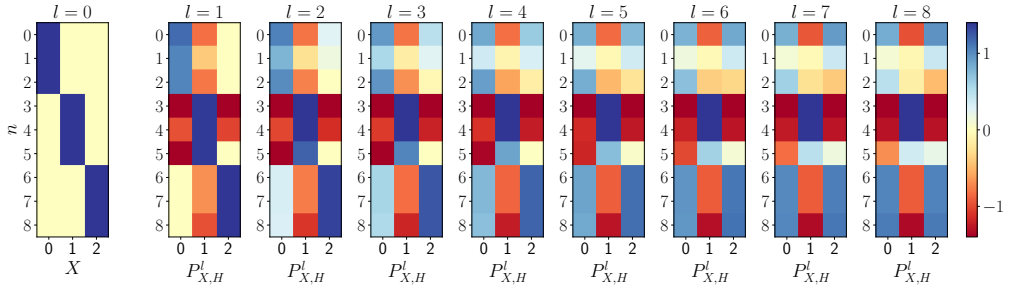


FIGURE 4.9: Progression of inverse Laplacian embedding with increasing propagation hops on given one-hot distribution raw feature.

4.4 Summary and Discussion

In this work, we propose LD^2, 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, LD^2 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 LD^2 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 4.3.5 we examine a toy model attempting to explain the reason behind the varying propagations. We think that the propagation of LD^2 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 A^{2}Prop 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 A^{2}Prop is efficient in implementation, we do recognize that there are graph centrality algorithms and decoupled GNN precomputations reaching sub-linear complexity. A^{2}Prop 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 LD^2 to propagate the augmented embeddings is a promising way to address its limitation on feature distribution.

Part II

Boosting Diverse Graph Learning Approaches

Chapter 5

Hierarchy Labeling for Graph Transformer

5.1 Introduction

Graph Transformers (GTs) characterize a family of neural networks that introduce the advantageous Transformer architecture [87] to the realm of graph data learning. These models have garnered increasing research interest for tasks such as knowledge graph retrieval, molecule analysis, and Large Language Model alignment [89, 90, 176, 237]. Despite their achievements, vanilla GTs are highly limited to specific tasks because of the full-graph attention mechanism, which has computational complexity at least quadratic to the graph size, rendering it impractical for a single graph with more than thousands of nodes. Enhancing the scalability of GTs is thus a prominent task for enabling these powerful models to handle a wider range of graph data at large scales.

To scale up Graph Transformers, existing studies explore various strategies to divide and represent the graph structure into smaller batches and employ mini-batch training. One approach is to simplify the Transformer *architecture* with a specialized attention module based on graph topology [181, 180, 182], which learns on existing edge connections instead of all-pair interactions. Alternatively, sophisticated *representations* are developed

This work is based on: [4] **Ningyi Liao***, Zihao Yu*, Siqiang Luo, Gao Cong. "HubGT: Fast Graph Transformer with Decoupled Hierarchy Labeling". In *39th Conference on Neural Information Processing Systems (**NeurIPS**)*, 2025.

for inputting graph information to GT models as node embeddings and positional encodings (PEs). These works feature graph processing techniques such as adjacency-based spatial propagation [184, 186], polynomial spectral transformation [185, 183], and hierarchical graph coarsening [173, 187]. However, we identify two major drawbacks of existing scalable GTs: in terms of efficacy, most models primarily concentrate on local adjacency, which undermines GT expressivity in capturing full-graph information; in terms of efficiency, graph-scale operations still persist throughout their training iterations, and the overhead substantially increases as the graphs become larger.

In this work, we propose HubGT, a scalable Graph Transformer exploiting the hub labeling technique to produce rich hierarchical graph information with efficient computation. HubGT is inspired by the well-studied concept of graph labeling, which identifies important hubs in the graph structure and imparts fast computation of the shortest path distance (SPD) for node pairs [238, 239, 240]. We innovatively introduce the graph label hierarchy to enhance GT capability, which is superior to the conventional adjacency-based scheme in establishing global connections to influential graph hubs. To further represent node-pair interactions, HubGT is the pioneering work to employ SPD as positional encoding for large-scale GTs, which is only practicable under its efficient calculation. The expressive representations empower HubGT to capture graph knowledge, especially the local node-pair relationships and global connectivity beyond direct edges and excel in complex graph data patterns ranging from homophily to heterophily.

To efficiently construct graph labels and calculate SPD for GT learning, we design a three-level hierarchical index orienting the specific query requirement in HubGT, which leverages both local connections and global hubs to facilitate distance computation. Graph indexing can be fully decoupled as precomputation and constructed with $O(m)$ overhead. Then, querying SPD on the index can be performed in $O(1)$ time, rendering HubGT learning as simple as training normal Transformers under $O(n)$ complexity without interweaving graph topology, where m and n are the numbers of graph edges and nodes, respectively. Both precomputation and training of HubGT achieve theoretical complexities on par with the respective state-of-the-art GTs and are significantly faster in practice thanks to the simplicity of hub labeling.

5.2 Motivating Study

5.2.1 Graph Hierarchy beyond Adjacency

The expressiveness of GTs mainly stems from the full-graph attention formulated in Eq. (2.9), which captures critical node pairs in the graph topology to learn node representations [89, 181]. However, since the mini-batch scheme replaces it with in-batch attention, its capability is potentially hindered. To compensate the information loss, scalable GTs usually invoke more powerful embedding and encoding techniques, enhancing the *global* graph view for more candidate nodes by expanding the receptive field. In canonical mini-batch GT models [89, 181, 180, 184], the graph information is typically derived from the graph *adjacency* $\boldsymbol{A}$, which symbolizes neighborhood information $\mathcal{E}$. For both kernel-based and hierarchical GT variants, their expressiveness in distinguishing different graph structures is characterized by the substructure used in attention tokens [179].

However, recent advances in message-passing GNNs reveal that, the adjacency alone is insufficient for retrieving topological information in graph learning [241]. In complicated scenarios such as heterophilous graphs, the local graph topology may be ambiguous or even misleading [242, 243, 244]. To illustrate, Figure 5.1 shows the distribution of node neighbors considering their classification labels depicted by the homophily score [114]. A lower homophily score implies high heterophily, where less neighbors share the same label with the ego node. On graphs such as CHAMELEON and SQUIRREL, a large portion of nodes exhibits zero homophily, indicating that substructures depending on graph edges $\mathcal{E}$ barely include neighbors with the same label for GT to attend.

We are thence motivated to improve GT by augmenting the existing graph connections $\mathcal{E}$ to an extended edge set $\hat{\mathcal{E}}$ containing additional node interactions beyond the neighborhood. As shown in Figure 5.1, establishing more edges enhances nodes of lower scores with more homophilous connections, effectively addressing the zero-homophily issue. Therefore, encompassing both local connections and global knowledge is capable of forming a hierarchical structure beyond the original graph adjacency, which benefits the model in retrieving a wide spectrum of information [173, 186, 187].

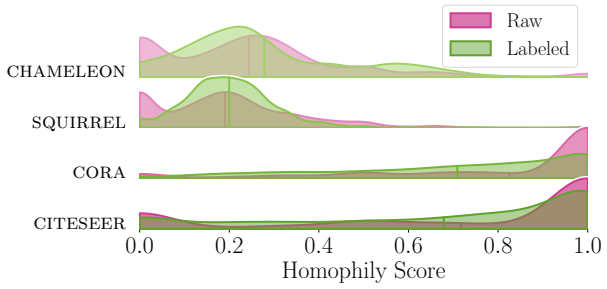


FIGURE 5.1: Relative distribution of homophily scores between original and extended graphs. A higher score implies more similar nodes in the neighborhood.

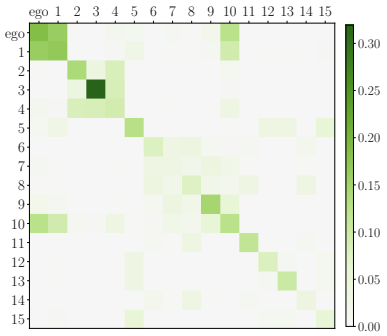
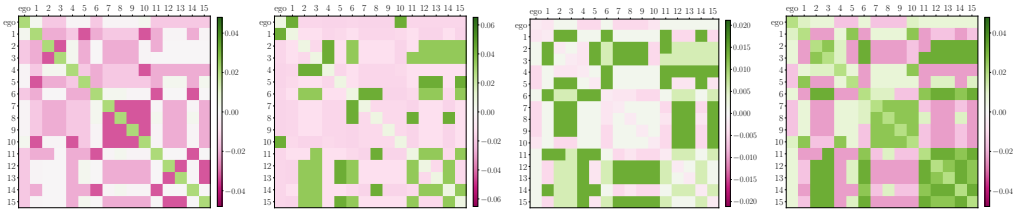


FIGURE 5.2: L-hop adjacency as positional encoding.



(a) $L = 0$ **(b)** $L = 1$ **(c)** $L = 2$ **(d)** $L = 3$

FIGURE 5.3: SPD PE as attention bias in different GT layers on CITESEER.

5.2.2 Dense Positional Encoding for Large Graphs

While an array of Positional Encoding (PE) schemes are utilized for explicitly integrating graph knowledge into GT learning, they are shown to possess various information aspects and expressiveness [89, 180, 237]. In particular, shortest path distance (SPD) as a form of *pairwise* PE [89] is revealed to be at least expressive as the Weisfeiler-Lehman (WL) graph isomorphism test [237] and empirically superior in graph classification tasks [245, 180, 246]. However, the conventional Floyd-Warshall algorithm for calculating whole-graph SPD entails $O(n^3)$ time and $O(n^2)$ space, which is prohibitive to large graphs.

On the other hand, scalable GTs commonly employ *sparse* PEs such as L-hop Adjacency $(\boldsymbol{D}^{-1/2}(\boldsymbol{A}+\boldsymbol{I})\boldsymbol{D}^{-1/2})^L$, which is straightforward to acquire by sparse matrix multiplication in $O(md^L)$ complexity [173, 187]. However, since nodes in large graphs are loosely connected, the in-batch PE under mini-batch training can be too sparse to represent pairwise relationships. As observed in Figure 5.2, PE values are only significant between the ego node and two direct neighbors, while diminishing to zero for more distant nodes

in the same batch. The values for non-ego node pairs are also small, rendering the bias ineffective in attending to node-pair interactions.

To this end, we attempt to apply the more effective pairwise SPD for encoding arbitrary node pairs on large graphs. Figure 5.3 suggests that the scheme is superior in providing rich information covering both positive and negative relationships, even for nodes that are not directly connected. Therefore, it is also more suitable for the extended hierarchical graph structure $\hat{\mathcal{E}}$ involving global connections to distant nodes.

5.3 Efficient Hierarchical Labeling

In this section, we present how HubGT achieves the hierarchy including both local and global graph structures beyond edge connections. In particular, we introduce the three levels for graph indexing and querying with respective algorithms. Utilization of the hierarchy for graph embedding and encoding during GT learning will be elaborated in the next section.

5.3.1 Label Graph and Its Properties

Proposed for efficient calculation of node-pair SPD queries, representative hub labeling approaches [247, 248] build a label index $\mathcal{L}(v)$ for node v in the graph, which is a set of hub nodes u and corresponding shortest distances $b(u, v)$ between the node pairs. The label of graph nodes is said to form a *2-hop cover* if for an arbitrary node pair $u, v \in \mathcal{V}$, there exists a node $w \in \mathcal{L}(u) \cap \mathcal{L}(v)$, and the SPD can be calculated by:

$$b(u, v) = b(u, w) + b(w, v), \tag{5.1}$$

where $b(u, w)$ and $b(w, v)$ are stored in labels $\mathcal{L}(u)$ and $\mathcal{L}(v)$, respectively.

To build the labels for all nodes in the graph with efficient search space and minimal index size, the Pruned Landmark Labeling (PLL) algorithm [249, 250] searches labels by a pruned Breadth First Search (BFS) for each node. The algorithm replies on a particular order of all nodes in $\mathcal{V}$. Denoting each node by a unique index $1, \cdots, n$, $u < v$ indicates that node u precedes node v in the sequence. During construction, PLL tends to regard low-index nodes as *hubs* and connecting more edges to them, while eliminating the

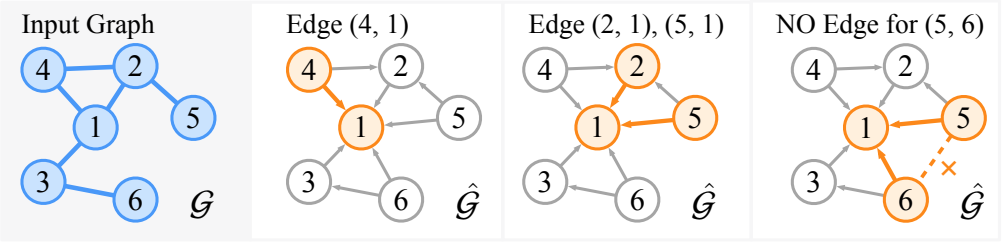
labeling from high-index nodes. Hence, a hierarchy of nodes are intrinsically formed by labeling. Notably, the algorithm is agnostic to the search order. In this work, we follow [238] to adopt the descending order of node degrees as it can be efficiently acquired and offers decent performance. While other orders such as betweenness centrality are adopted for labeling [251, 252], they nonetheless incur additional computational overhead.

In this part, we formally formulate the hierarchy in graph labels. Firstly, we leverage the node labels to build an expander graph by considering the existence of labels as new edge connections. The wholistic label set $\mathcal{L}$ can be regarded as a derived graph $\hat{\mathcal{G}} = \langle \mathcal{V}, \hat{\mathcal{E}} \rangle$ with directed and weighted edges, namely the *label graph*. Its edge set depicts the elements in node labels computed by graph labeling, that an edge $(u, v) \in \hat{\mathcal{E}}$ if and only if $(v, \delta_r) \in \mathcal{L}(u)$, and the edge weight is exactly the distance in graph labels $\delta_r = b(u, v)$. The in- and out-neighborhoods based on edge directions are $\mathcal{N}_{in}(v) = \{u \mid (u, v) \in \hat{\mathcal{E}}\}$ and $\mathcal{N}_{out}(v) = \{u \mid (v, u) \in \hat{\mathcal{E}}\}$, respectively. For simplicity, we assume that the original graph $\mathcal{G}$ is undirected, while properties for a directed $\mathcal{G}$ can be acquired by separately considering two label sets $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$ for in- and out-edges in $\mathcal{E}$.

We summarize the following three properties of the label hierarchy produced by PLL:

Property 5.1. *For an edge $(u, v) \in \mathcal{E}$, there is $(v, u) \in \hat{\mathcal{E}}$ when $u < v$, and $(u, v) \in \hat{\mathcal{E}}$ when $u > v$.*

Referring to Algorithm 5.1, when the current node is v and $v < u$, $\delta_r = 1$ holds since u is the direct neighbor of v. Hence, $(v, 1)$ is added to label $\mathcal{L}(u)$ at this round, which is equivalent to adding edge (u, v) to $\hat{\mathcal{E}}$. Similarly, $(v, u) \in \hat{\mathcal{E}}$ holds when $v > u$. For example, the edge $(1, 4)$ in Figure 5.4(a) is represented by the directed edge $(4, 1)$ in Figure 5.4(b). Property 5.1 implies that $\mathcal{N}(v) \subset \mathcal{N}_{in}(v) \cup \mathcal{N}_{out}(v)$, i.e., the neighborhood



(a) Raw Graph **(b)** Property 5.1 **(c)** Property 5.2 **(d)** Property 5.3

FIGURE 5.4: Examples of properties of the label graph $\hat{\mathcal{G}}$ corresponding to the original graph $\mathcal{G}$. Number inside each node denotes its index in descending order of node degrees.

of the original graph is also included in the label graph, and is further separated into two sets according to the relative order of neighboring nodes.

Property 5.2. *For a shortest path $\mathcal{P}(u,v)$ in $\mathcal{G}$, there is $(w,v) \in \hat{\mathcal{E}}$ for each $w \in \mathcal{P}(u,v)$ satisfying $w > v$.*

[249] proves that there is $v \in \mathcal{L}(w)$ for $w \in \mathcal{P}(u,v)$ and $w > v$. Therefore, considering shortest paths starting with node v of a small index, i.e., v being a "landmark" node, then succeeding nodes $w > v$ in the path are connected to v in $\hat{\mathcal{G}}$. In Figure 5.4(a), the shortest path between $(1,5)$ passing node 2 results in edges $(2,1)$ and $(5,1)$ in Figure 5.4(c), since nodes 2 and 5 are in the path and their indices are larger than node 1. When the order is determined by node degree, high-degree nodes appear in shortest paths more frequently, and consequently link to a majority of nodes, including those long-tailed low-degree nodes in $\hat{\mathcal{G}}$.

Property 5.3. *For a shortest path $\mathcal{P}(u,v)$ in $\mathcal{G}$, if there is $w \in \mathcal{P}(u,v)$ and $w < v$, then $(u,v) \notin \hat{\mathcal{E}}$.*

According to the property of shortest path, there is $b(u,v) = b(u,w) + b(w,v)$. Hence, the condition of line 11 in Algorithm 5.1 is not met at the v-th round when visiting w. In other words, the traversal from v is pruned at the preceding node w. By this means, the in-neighborhood $\mathcal{N}_{in}(v)$ is limited in the local subgraph with shortest paths ending at landmarks. As shown in Figure 5.4(d), the shortest path between $(5,6)$ passes node 1, indicating that $(5,6)$ are not directly connected since their distance can be acquired by edges $(5,1)$ and $(6,1)$. As a consequence, the neighborhood of node 5 in $\hat{\mathcal{G}}$ is constrained by nodes 1 and 2, preventing connections to more distant nodes such as 3 or 6.

In brief, Theorem 5.1 ensures that neighboring nodes in the raw graph $\mathcal{G}$ are still connected in $\hat{\mathcal{G}}$. Theorems 5.2 and 5.3 jointly imply that a small number of hub nodes, or landmarks, naturally emerge when more shortest paths pass through these nodes, and reside in a large number of node labels during the labeling process. On the contrary, for insignificant nodes with higher indices, the pruned traversal constrains the visit to the local neighborhood and limit their label size. Thus, we reckon that the PLL process builds a hierarchy embedded in the node labels, distinguishing global hubs while preserving original adjacency.

5.3.2 Problem Statement

In light of the limitations of existing scalable GTs on large graphs, our HubGT aims to address the two challenges in Section 5.2 simultaneously while ensuring computational efficiency: (1) The dedicated data structure provides expressive graph information for generating node *embeddings* that represent the graph hierarchy beyond mere adjacency. (2) Node-pair SPD can be efficiently calculated based on the hierarchy, which is then used as positional *encoding*.

Unlike existing hierarchical GTs relying on the original graph, we are thence motivated to extend the graph topology and incorporate hub labeling to represent graph information. Given a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, the graph labeling process [253] forms artificial edges $\hat{\mathcal{E}}$ to record the distance between nodes and stored as node labels. We consider this label graph $\hat{\mathcal{G}} = \langle \mathcal{V}, \hat{\mathcal{E}} \rangle$ containing additional directed and weighted edges for GT learning.

Once graph labels are built, we adopt them in generating batches for HubGT learning, which can be formulated as the following problem for querying the index structure:

Problem Definition. Given a node r, we aim to exploit the label graph $\hat{\mathcal{G}}$ to: (1) sample a set of nodes $\mathcal{S}(r)$ such that $(u, r) \in \hat{\mathcal{E}}$ or $(r, u) \in \hat{\mathcal{E}}$ for $u \in \mathcal{S}(r)$ solely based on the built index; (2) at the same time, acquire SPDs $b(u, v)$ for all node pairs $u, v \in \mathcal{S}(r)$.

Compared with traditional neighborhood-based batching, generating from the constructed labels preserves local neighbors while adding global hubs according to Theorem 5.3. This is preferable for GTs as it extends the receptive field beyond local neighbors described by graph adjacency and highlights those distant but important hubs in the whole graph for learning node interactions. Meanwhile, it maintains the form of 1-hop sampling, rather than multi-hop operations with potentially higher overhead.

5.3.3 HubGT Indexing

Design Goals. The stated problem characterizes the query requirements for designing our specific HubGT algorithms. In particular, we embrace the decoupling principle in consideration of scalability, separating the GT workflow into *precomputation* and *training* phases, as shown in Figure 5.5. Since graph labels are deterministic, they can be computed in an individual indexing stage beforehand. The training stage, on the other

hand, focuses on variable operations, including querying neighboring nodes and SPD, to learn model weights through repetitive epochs.

As analyzed in Table 2.5, GT training usually dominates the learning overhead and determines the deployment budget in practice. From the aspect of graph computation, a training epoch sends queries from every nodes in the graph. In comparison, the index is computed only once and used throughout training. Therefore, our primary design objective is to minimize the repetitive overhead of querying nodes and distances during training. The secondary goal is to achieve a reasonable time and space overhead for indexing. To achieve this, we propose our novel HubGT indexing and corresponding querying algorithms encompassing three levels of hierarchies, which are respectively introduced as follows.

Hierarchy-1: Local hub labeling. Examining the problem definition in Section 5.3.2, we observe that, unlike the conventional SPD query on an arbitrary node pair in the graph, the query for $b(u, v)$ in HubGT always orients an intermediary node r. In other words, u, v are 2-hop neighbors in the label graph $\hat{\mathcal{G}}$ disregarding edge directions. To leverage this relevance, we broaden the derivation of 2-hop labeling in [249], that the distance candidate orienting $b_r(u, v)$ can be acquired by utilizing the connectivity with r:

$$b_r^{(1)}(u, v) = b(u, r) + b(r, v) - \delta_r(u, v), \tag{5.2}$$

where $\delta_r(u, v) = 2, 1$, or 0 depending on the relative position between (u, r) and (v, r) in $\mathcal{G}$. Denote $\mathcal{M}_r^i(v) = \{w \in \mathcal{N}(r) \mid b(r, v) - b(w, v) = i\}$, $i = -1, 0$, or 1, where $\mathcal{N}(r) = \{w \mid (w, r) \in \mathcal{E}\}$ is the neighborhood orienting r in $\mathcal{G}$. The node sets $\mathcal{M}_r^i(v)$ of different integers i effectively characterize the relative position of the node of interest v with respect to the intermediary node r. When $\mathcal{M}_r^1(u) \cap \mathcal{M}_r^1(v) \neq \emptyset$, there is $\delta_r(u, v) = 2$; when $\mathcal{M}_r^0(u) \cap \mathcal{M}_r^1(v) \neq \emptyset$ or $\mathcal{M}_r^1(u) \cap \mathcal{M}_r^0(v) \neq \emptyset$, there is $\delta_r(u, v) = 1$; otherwise $\delta_r(u, v) = 0$.

We develop Algorithm 5.1 as the first level of hierarchy (H-1) based on PLL and Eq. (5.2). During the pruned BFS, we record the neighbor sets $\mathcal{M}_r^1(v)$ and $\mathcal{M}_r^0(v)$ for nodes added to labels $\mathcal{L}(u)$ along with their indices and distances. Additionally, we also maintain an inverse label set $\mathcal{L}'(v)$ such that $u \in \mathcal{L}'(v)$ if and only if $v \in \mathcal{L}(u)$. Therefore, each element in labels $\mathcal{L}(v)$ or $\mathcal{L}'(v)$ is a quadruple $(u, b(u, v), \mathcal{M}_v^1(u), \mathcal{M}_v^0(u))$.

Regarding computational overhead, Eq. (5.2) can be boosted by the bit-parallel computation similar to [249]: noting that all $\mathcal{M}_r^i(v)$ can be stored bit-wise in a 32- or 64-length

word for each node v, the distance calculation and set intersection in Eq. (5.2) can be performed as bit-parallel operations, efficiently handling all neighbors in one word. Hence, constructing the H-1 index shares the same overhead with PLL, while offering the extended distance information based on local computation orienting hubs.

In specific, we also explicitly impose a maximum capacity s for each node label $\mathcal{L}(v)$ in Algorithm 5.1. This is because at most s neighbors are required to form the batch in our problem. Notably, the algorithm is agnostic to the search order. In this work, we follow [238] to adopt the descending order of node degrees as it can be efficiently acquired and offers decent performance. While other orders such as betweenness centrality are adopted for labeling [251, 252], they nonetheless incur additional computational overhead.

ALGORITHM 5.1: H-1 Indexing

Input: Graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, Max neighbor size s
Output: H-1 Labels $\mathcal{L}, \mathcal{L}'$

Sort $\mathcal{V}$ based on degree $d(v)$
$\mathcal{L}(v) \leftarrow \emptyset$, $\mathcal{L}'(v) \leftarrow \emptyset$ for all $v \in \mathcal{V}$
for $r = 1$ to n **do**
    $(q(v), \mathcal{M}_r^1(v), \mathcal{M}_r^0(v)) \leftarrow (\infty, \emptyset, \emptyset)$ for all $v \in \mathcal{V}$
    $(q(v), \mathcal{M}_r^1(v), \mathcal{M}_r^0(v)) \leftarrow (1, \{v\}, \emptyset)$ for all $v \in \mathcal{N}(r)$
    Queue $Q \leftarrow \{(r, 0)\}$, $(q(r), \mathcal{M}_r^1(r), \mathcal{M}_r^0(r)) \leftarrow (0, \emptyset, \emptyset)$
    while $Q \neq \emptyset$ **do**
        $Q^1 \leftarrow \emptyset$, $Q^0 \leftarrow \emptyset$
        Pop the first element $(v, b(v, r))$ from Q
        Get $b^{(0)}(v, r)$ from Eq. (5.3) with existing labels
        if $b(v, r) < b^{(0)}(v, r)$ and $|\mathcal{L}(v)| < s$ **then**
            $\mathcal{L}(v) \leftarrow \mathcal{L}(v) \cup (r, b(v, r))$
            $\mathcal{L}'(r) \leftarrow \mathcal{L}'(r) \cup (v, b(v, r))$
            for all $u \in \mathcal{N}(v)$ such that $u > r$ **do**
                if $q(u) > q(v)$ **then**
                    Push $(u, b(v, r) + 1)$ to the end of Q
                    $Q^1 \leftarrow Q^1 \cup \{(u, v)\}$, $q(u) \leftarrow q(v) + 1$
                else if $q(u) = q(v)$ **then**
                    $Q^0 \leftarrow Q^0 \cup \{(u, v)\}$
        for all $(u, v) \in Q^0$ **do**
            $\mathcal{M}_r^0(u) \leftarrow \mathcal{M}_r^0(u) \cup \mathcal{M}_r^1(v)$
        for all $(u, v) \in Q^1$ **do**
            $\mathcal{M}_r^1(u) \leftarrow \mathcal{M}_r^1(u) \cup \mathcal{M}_r^1(v)$
            $\mathcal{M}_r^0(u) \leftarrow \mathcal{M}_r^0(u) \cup \mathcal{M}_r^0(v)$
return $\mathcal{L}(v), \mathcal{L}'(v)$ for all $v \in \mathcal{V}$

Hierarchy-2: Global hub labeling. H-1 index presents the idea of utilizing the relative position orienting a given hub r to compute distances of 2-hop pairs present in the query. It can be further extended to some *global* hubs by indexing their labels as another level of hierarchy (H-2), which corresponds to the bit-parallel BFS in [249]. More specifically, we select a small set of nodes with low indices and perform BFS without pruning following Algorithm 5.1. For each global node t, the label $(b(v,t), \mathcal{M}_t^1(v), \mathcal{M}_t^0(v))$ is computed and stored for all $v \in \mathcal{V}$. Then, the distance of an arbitrary node pair $b_t^{(2)}(u,v)$ can be similarly computed by Eq. (5.2) orienting t.

In our implementation, the H-2 construction is performed prior to H-1 and H-2, offering an additional condition for BFS pruning and cache saving. As demonstrated in [249], the global index design is advantageous in reducing the overall label size and facilitating a faster indexing time. It also benefits from bit-parallel processing and fast SPD computation similar to H-1.

Hierarchy-0: 2-hop pair caching. Although the H-1 query Eq. (5.2) produces the shortest distance $b(u,v)$ when $b_r^{(1)}(u,v) \leq b(u,v) + \delta_r(u,v)$, it is possible that the actual shortest path $\mathcal{P}(u,v)$ does not pass through r or its neighbors. In this case, the query falls back to the classic node labeling scheme:

$$b^{(0)}(u,v) = \min_{w \in \mathcal{L}(u) \cap \mathcal{L}(v)} \{b(u,w) + b(w,v)\}, \tag{5.3}$$

where labels $\mathcal{L}$ are computed by Algorithm 5.1. Note that $b(v,v) = 0$.

We regard the 2-hop labeling scheme as the fundamental hierarchy (H-0) as it guarantees answer to any queries. However, calculation by Eq. (5.3) results in a complexity of $O(|\mathcal{L}(u)| + |\mathcal{L}(v)|) = O(s)$, which is still not satisfying under the repetitive querying scenario in GT training. To further improve query speed, we choose to cache the frequently queried 2-hop shortest distances $b(u,v)$ and index them by the end node as $I_v[u]$ for $u < v$. By employing a suitable data structure such as a hash map for each v, checking the existence and acquiring $I_v[u]$ for a given node pair can be completed in $O(1)$.

The precomputation for building the H-0 index is in Algorithm 5.2. Examining Theorems 5.2 and 5.3, it can be inferred that the intermediary node presents on the shortest path $r \in \mathcal{P}(u,v)$ if and only if $r \in \mathcal{L}(u) \cap \mathcal{L}(v)$. We hence equivalently rearrange the traversal order of 2-hop pairs $u \in \mathcal{L}'(r),\ r \in \mathcal{L}(v)$ for all possible presence of shortest

paths, where $\mathcal{L}, \mathcal{L}'$ are produced by Algorithm 5.1. The search and $\mathcal{I}_v$ construction for each node v can be performed in parallel since they are mutually independent.

In Algorithm 5.2, we utilize two structures $\mathcal{I}_v^{(0)}$ and $\mathcal{I}_v^{(1)}$ to respectively record the distances acquired by Eq. (5.3) and Eq. (5.2), and save the distance to index $\mathcal{I}_v$ only if there exists a node r that cannot calculate the shortest distance using the H-1 index. In this way, we effectively constrain the H-0 index size, and ensure that it only caches the cases where H-1 may not produce the shortest distance.

5.3.4 HubGT Querying

After building the index H-2, H-1, and H-0 successively, querying label graph neighbors $\mathcal{S}(r)$ and their distances as in Section 5.3.2 can be achieved solely on the $\mathcal{L}, \mathcal{L}', \mathcal{I}$ without resorting to the graph structure. In Algorithm 5.3, we showcase the sampling procedure given an ego node r and respective sample sizes s_{out} and s_{in} for out- and in-connections stored in $\mathcal{L}(r)$ and $\mathcal{L}'(r)$, respectively. The node-pair distance inside $\mathcal{S}(r)$ is presented as a symmetric matrix $\boldsymbol{B}_r$, and its entry value $\boldsymbol{B}_r[u, v] = \boldsymbol{B}_r[v, u] = b(u, v)$.

The distance query on (u, v) consecutively accesses H-0, H-1, and H-2 indices in Algorithm 5.3. If the H-0 distance $\mathcal{I}_v[u]$ exists, it indicates the shortest distance according to Algorithm 5.2. Otherwise, the distance is either achieved by the local or global hub labeling following Eq. (5.2) orienting a particular intermediary node r or t. Notably,

ALGORITHM 5.2: H-0 Indexing

Input: Graph labels $\mathcal{L}, \mathcal{L}'$
Output: H-0 Index $\mathcal{I}$

for $v = 1$ to n **do** ▷ *[in parallel]*
  $\mathcal{I}_v^{(0)}[u] \leftarrow \infty, \mathcal{I}_v^{(1)}[u] \leftarrow -\infty$ for all $u \in \mathcal{V}$
  for all $r \in \mathcal{L}(v)$ **do**
    for all $u \in \mathcal{L}'(r)$ such that $u < v$ **do**
      Get $b_r^{(1)}(v, u)$ from Eq. (5.2)
      if $b_r^{(1)}(v, u) > \mathcal{I}_v^{(1)}[u]$ **then** $\mathcal{I}_v^{(1)}[u] \leftarrow b_r^{(1)}(v, u)$
      $b^{(0)}(v, u) \leftarrow b(v, r) + b(r, u)$
      if $b^{(0)}(v, u) < \mathcal{I}_v^{(0)}[u]$ **then** $\mathcal{I}_v^{(0)}[u] \leftarrow b^{(0)}(v, u)$
  for all $u \in \mathcal{V}$ such that $\mathcal{I}_v^{(0)}[u] \neq \infty$ **do**
    if $\mathcal{I}_v^{(0)}[u] < \mathcal{I}_v^{(1)}[u]$ and $|\mathcal{I}_v| < s^2$ **then**
      $\mathcal{I}_v[u] \leftarrow \mathcal{I}_v^{(0)}[u]$
return $\mathcal{I}_v$ for all $v \in \mathcal{V}$

queries regarding the ego node $\mathcal{S}(r)$ can be performed by only accessing the index of r without referring to others, which offers better memory locality and runtime efficiency.

Complexity Analysis. We respectively investigate the time and memory complexity of each indexing and querying stages. For H-1 and H-2 indexing, the algorithm completes the traversal of all nodes in $O(ns + ms)$ time, considering the label size $|\mathcal{L}(v)| \leq s$. The total index size is therefore bounded by $O(ns)$. For H-0 labeling in Algorithm 5.2, the computational overhead is $O(nss')$, where s' is the average size of $\mathcal{L}'$. Empirically, we enforce the index size to be less than s^2 for every $\mathcal{I}_v$. In summary, the time and memory complexities for the three-level indexing are $O(ns^2 + ms)$ and $O(ns^2)$, respectively. We particularly highlight that the empirical label sizes observed in experiments for both H-0 and H-1 are substantially smaller than the theoretical bound, thanks to the hierarchy that effectively reduces redundant information.

Querying one node pair distance by Algorithm 5.3 can be achieved in $O(1)$ time when bit parallel and accessing hash map are both $O(1)$ operations. By selecting sample sizes such that $s = s_{in} + s_{out} + 1$, the subgraph size for each query node is $|\mathcal{S}(r)| = s$. In consequence, querying every node as $r \in \mathcal{V}$ in each training iteration, with each query node possessing at most s^2 pair-wise distance computations, entails $O(ns^2)$ overhead. Compared to the conventional full-graph SPD query under $O(n^3)$ complexity, this is a significant improvement benefiting model scalability.

ALGORITHM 5.3: Query for node r

Input: Index $\mathcal{L}, \mathcal{L}', \mathcal{I}$, Sample sizes s_{in}, s_{out}
Output: Sampled subgraph nodes $\mathcal{S}(r)$ and SPD matrix $\boldsymbol{B}_r$

$\mathcal{S}(r) \leftarrow \{r\}$
Sample s_{out} nodes from $\mathcal{L}(r)$ into $\mathcal{S}(r)$
Sample s_{in} nodes from $\mathcal{L}'(r)$ into $\mathcal{S}(r)$
for all (u, v) such that $u \in \mathcal{S}(r), v \in \mathcal{S}(r), u < v$ **do**
    if $\mathcal{I}_v[u]$ exists **then**
        $b(u, v) \leftarrow \mathcal{I}_v[u]$
    else
        Get $b_r^{(1)}(u, v)$ from Eq. (5.2)
        Get $b_t^{(2)}(u, v)$ from Eq. (5.2) for all global t
        $b(u, v) \leftarrow \min\{b_r^{(1)}(u, v), b_t^{(2)}(u, v)\}$
    $\boldsymbol{B}_r[u, v] \leftarrow b(u, v)$, $\boldsymbol{B}_r[v, u] \leftarrow b(u, v)$
return $\mathcal{S}(r)$ and $\boldsymbol{B}_r$

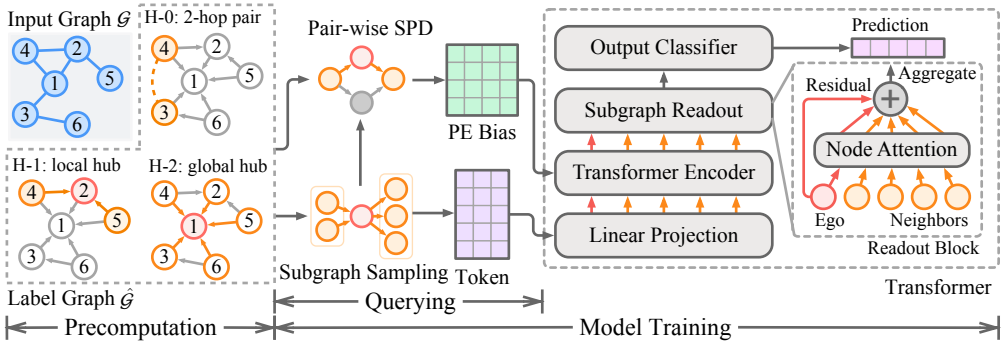


FIGURE 5.5: HubGT framework with precomputation and training stages. The **precomputation** stage processes the input graph and constructs the label index. During **training**, subgraph nodes and SPDs are queried from the index and applied as different Transformer inputs. Querying on CPU and training on GPU are pipelined and conducted simultaneously.

Comparison with Traditional Hub Labeling. We highlight that HubGT and canonical hub labeling address distinct query scenarios and design goals, leading to different performance of index construction and SPD querying. In HubGT, we prioritize querying with $O(1)$ overhead and local label access, in order to prevent the excessive queries from blocking GT training. To this end, we adopt the H-0 cache with additional precomputation and index size for fast distance access, while leveraging the neighboring property on label graph to construct H-1 index. Contrarily, classic hierarchical labeling approaches [249, 252] are designed for arbitrary node-pair queries, which is not applicable to the intermediary node technique in our H-1 local labeling, and entails $O(s)$ query time.

5.4 HubGT Model Design

To efficiently exploit the label graph in HubGT, in this section, we elaborate our end-to-end design representing graph hierarchy as both graph embeddings and positional encoding in GT learning. Figure 5.5 illustrates the overview of the HubGT pipeline.

5.4.1 Subgraph for Node Embeddings

In mini-batch GT training, both graph embeddings and positional encoding are essential to represent graph information for model learning. Each embedding is also known as a *token*, which is the minimum unit for model representation. Alternatively, the encoding

$\boldsymbol{P}$ is used in attention layers in Eq. (2.9) for depicting pair-wise relationship within the subgraph. Leveraging the label graph hierarchy in HubGT benefits both of these two inputs: (1) The labels provide expressive graph information for generating *embeddings*, encompassing both local neighbors and global hubs. (2) Node-pair SPD can be efficiently calculated, which is then used as positional *encoding* for evaluating relevance between nodes.

To generate embeddings of a node v in HubGT, we leverage the subgraph hierarchy in the label graph, i.e., neighboring nodes in labels $\mathcal{L}(v)$ and $\mathcal{L}'(v)$. Considering that the neighborhood size is variable, we convert it into a fixed-length token $\mathcal{S}(v)$ by sampling to align the GT architecture. When $\mathcal{S}(v)$ contains neighbors from $\mathcal{L}(v)$ and $\mathcal{L}'(v)$, as well as the ego node v itself, its length is $s = s_{in} + s_{out} + 1$, which justifies the problem definition in Section 5.3.2. The node embedding is generated by concatenating the raw attributes $\boldsymbol{X}[u]$ of subgraph nodes $u \in \mathcal{S}(v)$, denoted as $\boldsymbol{X}[\mathcal{S}(v)]$.

The relative values of hyperparameters s_{in} and s_{out} can be used to balance the ratio between in-neighbors and out-neighbors in $\hat{\mathcal{G}}$, which correspond to local long-tailed nodes and distant landmark nodes in $\mathcal{G}$, respectively. Compared to canonical GT tokens that represent the graph node in the context of the full graph, HubGT only relies on a small but informative subgraph of fixed size s. When the graph scales up, HubGT enjoys better scalability as its token size does not increase with the graph size.

Global Hubs. Section 5.3.3 signifies a set of global nodes used for H-2 labeling. For generating node embeddings, we further leverage these nodes as global hubs, that we consider these hubs connected to every nodes $v \in \mathcal{V}$ and can be similarly sampled during token generation. In HubGT training, we set their attributes to be learnable along with model weights. This scheme actually generalizes the virtual node utilized in [89] to a set of hubs. The learnable embeddings are able to aggregate information from connected nodes throughout learning, which eventually offers a representation of the graph in a higher level of hierarchy, and benefits GT for retrieving graph-level information.

5.4.2 Distance for Positional Encoding

For positional encoding, we specifically choose to use SPD, which depicts relative node position in the graph by pair-wise distances. In the context of mini-batch GT training, SPD is advantageous in providing dense representation for every node pairs regardless

of their position in the graph, and can be efficiently acquired without auxiliary memory overhead. In comparison, graph proximity and Laplacian encoding commonly used for mini-batch GTs only possess meaningful values for local nodes within a limited hops, and are usually too sparse in an RS batch with most entries as zeros, which limits its expressivity in depicting global relationships.

Despite the strength of SPD encoding, it has not been employed for large-scale GTs due to the impractical $O(n^3)$ overhead in naive calculation. Thanks to the efficient graph labeling of HubGT, we are able to efficiently acquire SPD inside subgraphs by performing queries on the index. For each node r, Algorithm 5.3 calculates the distances $\boldsymbol{B}_r$ of all node pairs inside the token $\mathcal{S}(r)$ under $O(s^2)$ complexity. A learnable embedding scheme $f_B : \mathbb{N} \to \mathbb{R}$ achieved by a linear layer is then employed to map the distance of each entry in $\boldsymbol{B}_r$ to the attention bias $\boldsymbol{P}$ used in Eq. (2.9), that $\boldsymbol{P}[u, v] = f_B(\boldsymbol{B}_r[u, v])$.

In implementation, querying a batch of ego nodes by Algorithm 5.3 can be conducted in parallel. The SPD query on the CPU can be pipelined with GT training on the GPU, so that training with the current batch of embeddings and encoding can be performed simultaneously with fetching and loading the data of the next batch. Thanks to the fast query design, the HubGT workflow ensures that accessing the index does not block the computation-intensive GT training, allowing seamless training that can potentially be enhanced by further acceleration techniques developed for the Transformer architecture.

5.4.3 Overall Architecture

HubGT adapts the scalable GT architecture [89, 173] incorporating subgraph precomputation and mini-batch training. The token features of each node v are learned from the subgraph embeddings as $\boldsymbol{H}[v] = \mathrm{MLP}_X(\boldsymbol{X}[\mathcal{S}(v)])$, where $\mathrm{MLP}_X : \mathbb{R}^{s \times F_0} \to \mathbb{R}^{s \times F}$ with hidden dimension F. Then, L Transformer layers characterized by Eqs. (2.8) and (2.9) are applied to predict the node representation $\boldsymbol{H}[v]$ with positional encoding $\boldsymbol{P}$. Lastly, we introduce a readout block to calculate attention within the token $\mathcal{S}(v)$ to aggregate the output of each ego node:

$$\begin{aligned} \boldsymbol{Z}[v] &= \mathrm{MLP}_Z\left(\boldsymbol{H}^{(L)}[v] + \sum_{u \in \mathcal{S}(v)} \alpha_u \boldsymbol{H}^{(L)}[u]\right), \\ \text{where} \quad \alpha_u &= \frac{\exp\left((\boldsymbol{H}^{(L)}[v] \| \boldsymbol{H}^{(L)}[u])\boldsymbol{W}_E\right)}{\sum_{u \in \mathcal{S}(v)} \exp\left((\boldsymbol{H}^{(L)}[v] \| \boldsymbol{H}^{(L)}[u])\boldsymbol{W}_E\right)}, \end{aligned} \tag{5.4}$$

and MLP_Z is the output classifier.

Data Batching and Transferring. Remarkably, HubGT inputs of embedding and encoding can be manipulated in-place on $\boldsymbol{X}$ and $\boldsymbol{B}_r$, and no graph-scale computation is required during learning iterations. Therefore, mini-batch training can be easily implemented by randomly sampling batches of ego nodes, and only strides of $\boldsymbol{X}$ and batches of $\boldsymbol{B}_r$ are loaded onto GPUs.

Complexity Analysis. During model training, one epoch of L-layer feature transformation on all nodes entails $O(LnF)$ complexity, while positional encoding is performed under $O(ns^2)$. The RAM footprint is $O(ns^2)$ and $O(nsF)$ for sampled tokens and features, respectively. For mini-batch training with batch size n_b, the VRAM overhead on GPU for a batch of node representations and bias matrices is $O(Ln_bF)$ and n_bs^2, respectively. It can be observed that the GPU memory footprint is determined only by batch size and is independent of the graph scale, ensuring favorable scalability.

5.5 Experimental Evaluation

We comprehensively evaluate the performance of HubGT with a wide range of datasets and baselines. In Section 5.5.2, we highlight the model efficiency throughout learning phases as well as its effectiveness under both homophily and heterophily. Section 5.5.4 provides in-depth insights into the HubGT design in exploiting graph hierarchy.

5.5.1 Experimental Settings

Tasks and Datasets. We focus on the node classification task on 14 benchmark datasets covering both homophily [254, 255, 43] and heterophily [136, 235]. Compared to conventional graph learning tasks used in GT studies, this task requires learning on large single graphs, which is suitable for assessing model scalability. Evaluation is conducted on a server with 32 Intel Xeon CPUs (2.4GHz), an Nvidia A30 GPU (24GB memory), and 512GB RAM.

Table 5.1 displays the scales and heterophily status of graph datasets utilized in our work. Undirected edges twice in the table. We follow common data processing and evaluation

protocols. CHAMELEON and SQUIRREL are the filtered version from [235], while OGBN-MAG is the homogeneous variant. We employ 60/20/20 random data splitting percentages for training, validation, and testing sets, respectively, except for REDDIT and OGBN-MAG, where the original split is used.

Baselines. Since the scope of this work lies in the efficacy and efficiency enhancement of the GT architecture, we primarily compare against leading scalable Graph Transformer models with attention-based layers and mini-batch capability. Methods including DIFFormer [182] and PolyNormer [183] are considered as kernel-based approaches. NAGphormer [184], GOAT [186], HSGT [187], and ANS-GT [173] stand for hierarchical GTs. An state-of-the-art message-passing GNN SGFormer [256] is also included for comparison.

TABLE 5.1: Statistics of graph datasets. f and N_c are the numbers of input attributes and label classes, respectively. "Train" is the portion of training set w.r.t. labeled nodes.

Heterophily	**Dataset**	**Nodes** n	**Edges** m	F	N_c	**Train**
Homophily	CHAMELEON	890	17,708	2325	5	60%
	SQUIRREL	2,223	93,996	2089	5	60%
	TOLOKERS	11,758	1,038,000	10	2	60%
	PENN94	41,554	2,724,458	4814	2	60%
	GENIUS	421,961	1,845,736	12	2	60%
	TWITCH-GAMER	168,114	13,595,114	7	2	60%
	POKEC	1,632,803	30,622,564	65	2	60%
Heterophily	CORA	2,708	10,556	1433	7	60%
	CITESEER	3,279	9,104	3703	6	60%
	PUBMED	19,717	88,648	500	3	60%
	PHYSICS	34,493	495,924	8415	5	60%
	OGBN-ARXIV	169,343	2,315,598	128	40	54%
	REDDIT	232,965	114,615,892	602	41	60%
	OGBN-MAG	736,389	10,792,672	128	349	85%

Hyperparameters. Parameters regarding the precomputation stage for graph structures are discussed in Section 5.5.4. For subgraph sampling, we perform parameter search for relative ratio of in/out neighbors represented by s_{out} in rage $[0, 48]$.

For network architectural hyperparameters, we use $L = 4$ Transformer layers with $N_H = 8$ heads and $F = 128$ hidden dimension for our HubGT model across all experiments. The dropout rates for inputs (features and bias) and intermediate representation are 0.1 and 0.5, respectively. The AdamW optimizer is used with a learning rate of 10^{-4}. The model

is trained with 300 epochs with early stopping. Since baseline GTs employ different batching strategies, it is difficult to unify the batch size across all models. We set the batch size to the largest value in the available range without incurring out of memory exception on our 24GB GPU, intending for a fair efficiency evaluation considering both learning speed and space.

Evaluation Metrics. We use Receiver Operating Characteristic Area Under the Curve (ROC AUC) as the efficacy metric on TOLOKERS and classification accuracy on the other datasets, following [235]. For a binary classification, AUC can be computed by $\text{AUC} = \frac{1}{|\mathcal{V}_+||\mathcal{V}_-|} \sum_{v^+ \in \mathcal{V}_+} \sum_{v^- \in \mathcal{V}_-} \mathbf{1}\big(Z[v^+] > Z[v^-]\big)$, where v^+ and v^- are the positive and negative predictions, respectively.

For efficiency evaluation, we notice that there is limited consensus due to the great variety in GT training schemes. Therefore, we attempt to employ a comprehensive evaluation considering processing times of different learning phases for a fair comparison. Model speed is represented by the average training time per epoch and the inference time on the testing set. For models with graph precomputation, the time for this process is separately recorded.

5.5.2 Performance Comparison

Tables 5.2 and 5.3 presents the efficacy and efficiency evaluation results on 8 large-scale graphs and 6 smaller graphs, respectively. As an overview, HubGT demonstrates fast computation speed and favorable mini-batch scalability throughout the learning process and is applicable to million-scale graphs. It also achieves top-tier accuracy on 11 out of 14 datasets.

Time Efficiency. Benefiting from the decoupled architecture, HubGT is powerful in achieving competitive learning and inference speeds with existing efficiency-oriented GTs. It consistently showcases the fastest inference, since Section 5.4.3 elaborates that label querying can be pipelined in asynchronous execution, and the process is as simple as Transformer operations without the interference of graph computation.

In comparison to other hierarchical GTs, HubGT excels with the fastest training and overall learning time. Specifically, its precomputation is 250-1000× faster than ANS-GT, which is also relatively fast in model training and inference. Aligned with our complexity

TABLE 5.2: Effectiveness and efficiency results on large-scale graph datasets. "Pre." , "Epoch", and "Infer" are precomputation, training epoch, and inference time (in seconds), respectively. "OOM" implies that the model encounters the out-of-memory error. "TLE" means the learning process exceeds the time limit of 24 hours before convergence. Respective results of the first and second best performances are marked in **bold** and underlined fonts.

Homophilous	PHYSICS				OGBN-ARXIV				REDDIT				OGBN-MAG			
Metrics	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc
DIFFormer*	-	1.7	3.8	96.10±0.11	-	0.89	4.1	55.90±8.23	-	2.4	21	94.96±0.37	-	1.7	9.7	31.13±0.48
PolyNormer*	-	0.76	2.4	**96.59**±0.16	-	0.83	11	**73.24**±0.13	-	5.3	246	**96.64**±0.07	-	20	992	32.42±0.15
SGFormer*	-	0.52	2.6	96.33±0.29	-	0.82	1.9	72.55±0.28	-	2.2	21	95.63±0.29	-	1.7	5.1	33.47±0.61
NAGphormer	33	8.4	2.4	96.52±0.24	18	4.4	2.2	67.85±0.17	280	3.2	2.1	95.77±0.08	89	10.3	2.2	33.23±0.06
ANS-GT	2203	63	35	96.31±0.28	16205	109	2.7	71.06±0.48		(OOM)				(OOM)		
GOAT	45	14	12	96.24±0.15	1823	48	61	69.66±0.73	628	141	104	(TLE)	2673	116	102	(TLE)
HSGT*	12	41	62	96.05±0.50	16	475	142	68.30±0.32	614	453	482	(TLE)	182	582	629	(TLE)
HubGT (ours)	2.0	2.9	0.31	96.38±0.25	30	9.3	1.3	69.17±0.33	192	21	1.6	94.39±0.06	523	62	1.2	**33.74**±0.24
Heterophilous	PENN94				GENIUS				TWITCH-GAMER				POKEC			
Metrics	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc	Pre.	Epoch	Infer	Acc
DIFFormer*	-	0.53	0.65	61.77±3.41	-	0.77	5.5	84.52±0.36	-	0.61	5.1	60.81±0.44	-	4.6	15	73.89±0.35
PolyNormer*	-	0.58	18.4	**79.87**±0.06	-	0.77	28	85.64±0.52	-	1.45	89	64.72±0.65	-	4.2	67	**81.03**±0.08
SGFormer*	-	0.95	0.55	77.52±0.56	-	0.72	2.4	85.01±0.25	-	0.38	3.3	65.93±0.15	-	4.4	29	73.13±0.16
NAGphormer	237	6.1	2.1	74.45±0.60	38	5.4	1.0	83.88±0.13	16	1.9	2.4	61.92±0.19	70	16.1	3.1	73.06±0.05
ANS-GT	3889	42	4.9	67.76±1.32	34092	37	5.0	67.76±1.32	12924	19	6.7	61.55±0.45		(OOM)		
GOAT	1332	33	18	71.42±0.44	2664	28	39	80.12±2.32	3348	37	63	61.38±0.83	3855	760	804	(TLE)
HSGT*	12	115	110	67.77±0.27	21	98	114	84.03±0.24	68	235	253	61.60±0.09	551	1420	1557	(TLE)
HubGT (ours)	7.3	3.4	0.29	78.13±0.43	125	23	2.5	**89.68**±0.48	49	12	1.2	**67.03**±2.17	5422	99	13	76.96±0.44

* Inference of these models is performed on the CPU in a full-batch manner due to their requirement of the whole graph.

TABLE 5.3: Effectiveness and efficiency results on small-scale datasets.

Homophilous	CORA					CITESEER					PUBMED				
	Pre.	Epoch	Infer	Mem.	Acc	Pre.	Epoch	Infer	Mem.	Acc	Pre.	Epoch	Infer	Mem.	Acc
DIFFormer*	-	0.11	0.13	1.2	83.37±0.50	-	0.07	0.07	1.7	74.65±0.67	-	0.37	0.35	2.7	75.77±0.40
PolyNormer*	-	0.11	0.65	1.4	80.43±1.55	-	0.21	0.86	1.6	68.70±0.95	-	0.86	6.07	2.5	75.80±0.46
NAGphormer	0.68	0.01	0.06	0.5	76.96±0.73	1.26	0.01	0.38	0.5	62.26±2.10	3.05	0.01	0.04	0.5	78.46±1.01
ANS-GT	43	2.0	1.12	2.0	85.42±0.52	59.9	11.65	4.25	11.9	73.58±0.98	529	14	3.52	1.9	89.53±0.51
GOAT	10.1	0.25	0.93	2.5	78.26±0.17	11.1	0.31	1.04	2.1	64.69±0.43	57.4	0.34	1.61	5.3	77.76±0.97
HSGT*	0.1	1.81	2.33	0.5	81.73±1.95	0.06	0.87	1.23	0.9	69.72±1.02	5.0	3.89	4.44	24	88.86±0.46
HubGT (ours)	0.42	0.20	0.02	2.5	**85.58**±0.18	0.43	0.24	0.02	2.0	**75.47**±2.22	2.6	1.47	0.16	1.7	**89.80**±0.48
Heterophilous	CHAMELEON					SQUIRREL					TOLOKERS				
	Pre.	Epoch	Infer	Mem.	Acc	Pre.	Epoch	Infer	Mem.	Acc	Pre.	Epoch	Infer	Mem.	ROC AUC
DIFFormer*	-	0.09	0.38	0.50	37.83±4.54	-	0.05	0.05	0.7	35.73±1.37	-	0.16	85.8	0.88	74.88±0.59
PolyNormer*	-	0.03	0.17	1.1	40.70±3.38	-	0.07	0.49	1.2	**38.40**±1.10	-	1.27	15.5	9.4	79.39±0.50
NAGphormer	0.27	0.03	0.03	0.5	33.18±4.30	0.85	0.08	0.08	0.5	32.02±3.93	1.59	0.11	0.02	0.5	79.32±0.39
ANS-GT	11.2	1.98	0.78	2.8	41.19±0.69	28.1	4.48	1.95	6.6	37.15±1.10	716	2.37	3.42	10.7	79.31±0.97
GOAT	1.99	0.34	0.44	0.4	35.02±1.15	6.66	0.37	0.58	0.6	30.78±0.91	36.1	5.49	5.87	5.0	79.46±0.57
HSGT*	0.01	0.34	0.73	0.3	32.28±2.43	0.01	0.42	0.74	0.4	34.32±0.51	2.62	7.76	8.12	17.4	79.24±0.83
HubGT (ours)	0.04	0.08	0.007	1.6	**43.63**±2.34	0.24	0.18	0.01	2.6	37.16±0.57	1.4	0.99	0.02	2.2	**79.86**±0.47

* Inference of these models is performed on the CPU in a full-batch manner due to their requirement of the whole graph.

analysis in Table 2.5, the overhead of HubGT indexing is mainly relevant to the node size n and is less affected by m and F compared to precomputation in other methods, which ensures its efficiency on denser graphs such as REDDIT and PENN94. Baselines models employing graph-altered learning schemes including DIFFormer, PolyNormer, and SGFormer are empirically fast in training due to the simplified model architecture. However, it is noticeable that their inference rely on the full-graph structure and can be only performed on CPU without GPU computation, resulting in slower and less scalable performance.

We additionally note that while some baselines claim to be applicable to million-scale graphs, they exhibit excessive training time in our settings and fail to produce convergent results in one day. Hence, we only record the efficiency evaluations in Table 5.2. In particular, ANS-GT demands a high memory footprint for storing and adjusting its subgraphs, which exceeds the memory limit of our platform for the largest graphs.

Memory Footprint. In modern computing platforms, GPU memory is usually highly constrained and becomes the scalability bottleneck for the resource-intensive graph learning. HubGT exhibits efficient utilization of GPU for training with larger batch sizes while avoiding the out-of-memory issue. In comparison, drawbacks in several model designs prevent them from efficiently performing GPU computation, which stems from the adoption of graph operations. Notably, kernel-based models require full graph message-passing in their inference stage, which is largely prohibitive on GPUs and can only be conducted on CPUs. HSGT faces the similar issue caused by its graph coarsening module. We note that these solutions are less scalable and hinder the GPU utilization during training.

Prediction Efficacy. HubGT successfully achieves top or comparable accuracy on evaluated datasets in Table 5.2 and Table 5.3, with significant accuracy improvement on several graphs such as CHAMELEON and TWITCH-GAMER. We attribute the performance gain to the application of the label graph hierarchy and SPD positional encoding in HubGT, which offers global information and effectively addresses the heterophily issue of certain graphs as analyzed in Section 5.3.2. The label sampling scheme also facilitates learning on hierarchy throughout queries under a controlled overhead. Since the label graph also preserves edges in the raw graph, the performance of HubGT is usually not lower than learning on the latter.

In comparison, baseline methods without hierarchical graph designs, including DIFFormer, NAGphormer, and HSGT, perform relatively worse especially under heterophily. This is because their models tend to rely on the raw adjacency or even promote it with higher modularity. As a consequence, node connections retrieved by GT attention modules are restrained in the local neighborhood and hardly produce accurate classifications. On the other hand, while PolyNormer achieves remarkable accuracy on several graphs thanks to its strong expressivity, its performance is largely suboptimal on small homophilous graphs as we further evaluated in Table 5.3.

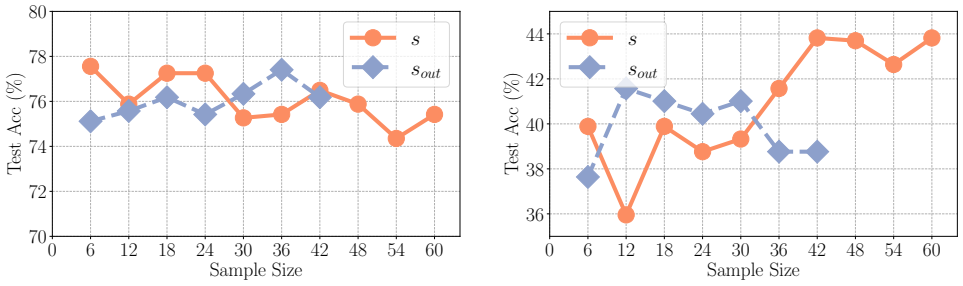
5.5.3 Ablation Study

Table 5.4 examines the respective effectiveness of the hierarchical modules in the HubGT network architecture, where we separately present results on homophilous and heterophilous datasets. It can be observed that the model without SPD bias suffers the greatest accuracy drop, since topological information represented by positional encoding is necessary for GTs to retrieve the relative connection between nodes and gain performance improvement over learning plain node-level features.

In HubGT, the learnable global hub representation is invoked to provide adaptive graph-level context before Transformer layers, while the attention-based node-wise readout module aims to distinguish nodes inside subgraphs and aggregate useful representation after encoder transformation. As shown in Table 5.4, both modules achieve relatively higher accuracy improvements on the heterophilous graph CHAMELEON, which validates that the proposed designs are particularly suitable for addressing the heterophily issue by recognizing hierarchical information.

TABLE 5.4: Ablation study of HubGT model components. The first line shows the accuracy of the complete HubGT architecture. Each subsequent line indicates the performance difference when the specified module is removed.

Dataset	CITESEER	Δ	CHAMELEON	Δ
HubGT	75.47	–	43.63	–
w/o Node Readout	72.21	-3.26	38.76	-4.87
w/o Global Hubs	71.15	-4.32	37.08	-6.55
w/o SPD Bias	68.55	-6.92	36.52	-7.11



(a) CITESEER **(b)** CHAMELEON

FIGURE 5.6: Effect of sample sizes on different datasets.

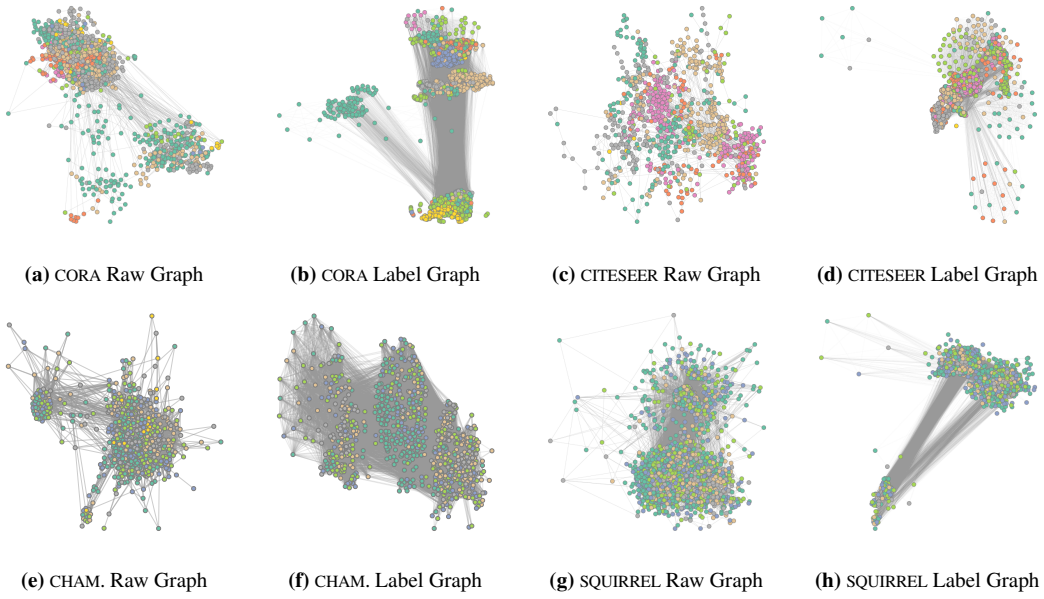
5.5.4 Effect of Hyperparameters

Sample Size. We then study the effectiveness of the label graph hierarchy in HubGT featuring the subgraph generation process in Figure 5.6, which displays the impact of sample sizes s and s_{out} corresponding to Algorithm 5.3. Regarding the total subgraph size s, it can be observed that a reasonably large s is essential for effectively representing graph labels and achieving stable accuracy. In the main experiments, we uniformly adopt a constant $s = 48$ token size across all datasets, as it is large enough to cover the neighborhood of most nodes while maintaining computational efficiency. As a reference, the actual average H-1 index sizes of $|\mathcal{L}(v) + \mathcal{L}'(v)|$ among all nodes are 40.6 on PHYSICS and 47.7 on PENN94, while the average H-0 sizes $|\mathcal{I}(v)|$ are 230.9 and 440.1, respectively. Both are significantly smaller than the theoretical bounds of $2s$ and s^2, which validates the efficiency of our three-level hierarchical labeling.

Within the fixed token length, the relative size of s_{out} indicates the preference of hubs with lower node indices. Comparing Figures 5.6(a) and 5.6(b) of varying s_{out} under $s = 48$, it can be observed that the accuracy tends to increase on the homophilous CITESEER. On the opposite, the performance on CHAMELEON decreases when introducing more out labels under heterophily, showing that our Algorithm 5.3 sampling design is effective in retrieving distinct graph information by adjusting the hyperparameters of s_{out} and s_{in}.

Clustering Effect of Labeling. An exemplary illustration of 4 small graphs is displayed in Figure 5.7. The original CHAMELEON graph is heterophilous, i.e., connected nodes frequently belong to distinct classes. In Figure 5.7(e), different classes are mixed in graph clusters, which pose a challenge for GTs to perform classification based on edge connections. In contrast, nodes in the graph marked by graph labels in Figure 5.7(f) clearly form multiple densely connected clusters, exhibiting a distinct hierarchy. Certain

classes can be intuitively identified from the hierarchy, which empirically demonstrates the effectiveness of our utilization of graph labeling.



(a) CORA Raw Graph **(b)** CORA Label Graph **(c)** CITESEER Raw Graph **(d)** CITESEER Label Graph

(e) CHAM. Raw Graph **(f)** CHAM. Label Graph **(g)** SQUIRREL Raw Graph **(h)** SQUIRREL Label Graph

FIGURE 5.7: Visualization of the hierarchy of original and label graphs on realistic datasets. Color of each node denotes its class.

5.6 Summary and Discussion

In this work, we present HubGT, a novel PE calculation featuring the decoupled graph hierarchy through hub labeling. Our analysis reveals that the label graph exhibits an informative hierarchy and enhances GT attention learning on the interaction between nodes. Regarding efficiency, construction and distance query of the label graph can be accomplished with *linear* complexity and are decoupled from iterative model training. Hence, the model benefits from scalability in computation speed and mini-batch training. Empirical evaluation showcases the superiority of HubGT, including efficacy under both homophily and heterophily, as well as efficient computation especially for inference on large-scale graphs of up to millions of nodes.

Since this study focuses on the efficiency and effectiveness of PE calculation for GTs on large graphs, we employ the classic Transformer architecture as introduced in Section 5.4.3 because of its generality. However, the model architecture can be further improved by advanced techniques, such as linear attention, for better speed and accuracy

performance. The experimental evaluation, particularly the efficiency performance, may be enhanced as well.

Chapter 6

Walk-based Representation for Dynamic Subgraph Graph Neural Network

6.1 Introduction

Subgraph GNN has recently garnered substantial attention for effectively understanding graph-structured data in modeling entities and their relationships, especially for complicated and large-scale graphs in real-world applications, such as recommender systems [188, 257], anomaly detection [195, 258] and network modeling [204, 259]. In many use cases, subgraph GNN aims to utilize neural networks to retrieve information from the graph structure and map it into low-dimensional representations, which is then used to generate predictions for downstream tasks such as node classification and link prediction [260].

From the perspective of subgraph GNN system, its learning process exhibits a joint utilization of CPU and GPU devices as shown in Figure 6.1. Typically, CPU is utilized to compute the subgraph extraction and feature generation. Afterwards, the feature data is learned by a neural network on the GPU in batches. However, most existing subgraph GNN approaches are tailored for static graphs and overlook the dynamic perspective. In

This work is published as: [5] Zihao Yu*, **Ningyi Liao***, Siqiang Luo. "GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs". In *Proceedings of the VLDB Endowment (**VLDB**)*, vol. 17, no. 9, pp. 2269–2278, 2024. DOI: 10.14778/3665844.3665856.

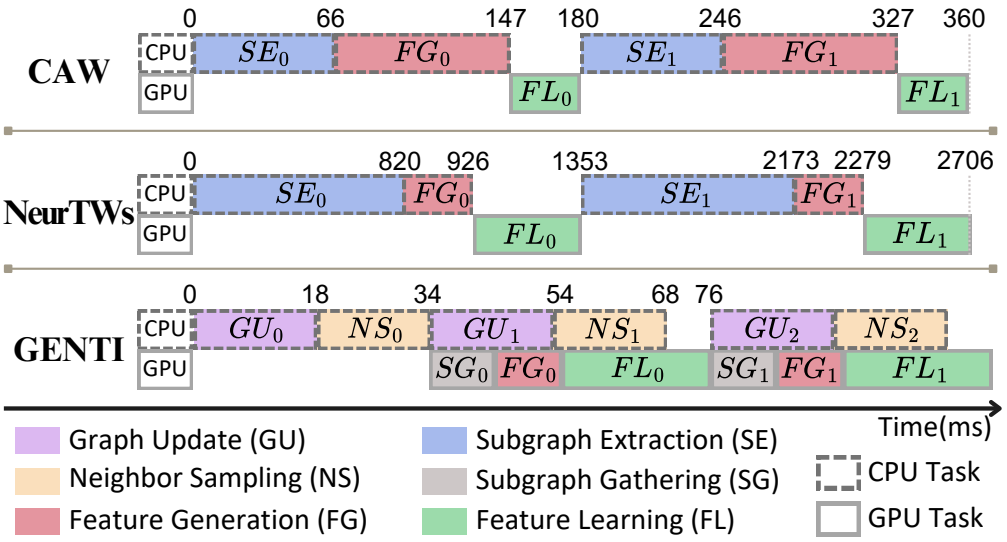


FIGURE 6.1: Experimental comparison on execution time of our GENTI pipeline against SGRL methods CAW and NeurTWs. Each stage is either on the CPU or GPU device. GENTI divides the subgraph extraction bottleneck into asynchronous CPU neighbor sampling and GPU subgraph gathering to facilitate balanced and concurrent workloads. Graph update procedures are not shown in CAW and NeurTWs as they are preprocessed in an extra stage.

realistic scenarios such as financial transactions and recommender systems [261, 257], graphs are actively evolving with frequent updates such as the establishment of new connections. With the aim of transferring to dynamic graphs, recent subgraph GNN models [204, 205] propose to utilize temporal variants of subgraph processing techniques and achieve promising accuracy in the link prediction task.

Despite their prominent algorithmic solutions, the dynamic subgraph GNN scheme calls for dedicated system designs with unique challenges: the subgraph data in extraction requires **streaming updates** under the evolving graph structure, entailing a significant computation and communication cost [262, 261]. As subgraph extraction is prerequisite for feature learning, it becomes the **workload bottleneck** between CPU and GPU devices. Correspondingly, graph **data structure** needs to be specifically redesigned for enhanced efficiency when undergoing dynamic amendments and subgraph sampling operations. These system-level issues pose practical difficulties when applying the advanced algorithms to real-world scenarios, especially when computational resources are limited.

Figure 6.1 illustrates the time consumption of the three-stage pipeline of representative dynamic SGRL methods CAW [204] and NeurTWs [205] in our experiment. It can be observed that subgraph extraction and feature generation executed on CPU constitute the majority of subgraph GNN overhead, and further delays GPU computation. The

sequential execution also results in low CPU and GPU device utilization. Additionally, in our evaluation, both current methods are prohibitive on the largest available dynamic graph MAG with 1.3 billion edges, demanding prolonged learning time over months.

In this work, we propose GENTI, a GPU-efficiENt Subgraph GNN on continuous-TIme dynamic graphs. GENTI highlights the adaptation of GPU to better balance the workload and improve efficiency. We primarily focus on the subgraph extraction procedure in light of its pivotal role in affecting both the SGRL pipeline's efficacy and efficiency. To this end, algorithms, data structures, and operations related to this stage are thoroughly refined for GPU processing and dynamic updates on billion-edge graphs.

To harness GPU acceleration and address imbalanced device workload, we devise a novel subgraph extraction scheme by dissecting it into two phases respectively for neighbor sampling and subgraph gathering. The first phase samples neighborhoods from the graph structure by CPU and loads the data onto GPU. Then, subgraph gathering by performing random walks is completely conducted on GPU thanks to its parallelism. A memory-efficient pool is innovatively crafted to cache the sampled subgraph on GPU and support gathering in batches. As shown in Figure 6.1, our overall subgraph extraction scheme ensures that it is asynchronous to feature processing, effectively distributing the workload between the two devices and preventing the pipeline from being blocked by the CPU computation bottleneck. Moreover, we perform dedicated enhancements on the data structure for graph storage in consideration of the dynamic SGRL pipeline, powering it with capabilities including $O(1)$ updating and search-free sampling.

We conduct comprehensive experiments to evaluate the efficiency and effectiveness of GENTI on 7 real-world dynamic graphs with up to millions/billions of nodes/edges. Concerning the comprehensive pipeline and specifically the subgraph extraction stage, our approach respectively achieves up to 30× and 26× speedups in running time when compared to state-of-the-art SGRL algorithms.

6.2 Temporal Walk-based Sampling

Walk-based SGRL employs the random walk procedure to sample connected nodes and construct the subgraph. The temporal random walk serves as an extension tailored for dynamic graphs, facilitating learning from the temporal dimension. A temporal random walk at time t is a traversal of $\mathcal{G}$ that initiates from the source node $u \in \mathcal{V}$ and, at each

step, advances to a randomly selected out-neighbor of the current node. Generating the subgraph for a node of interest usually requires k independent temporal random walks. At each step of these walks, the destination nodes are selected by the Weighted Neighbor Sampling (WNS) technique [204, 205]. At time t, the weighted neighbor set of node u is defined as $\mathcal{N}_u = \{(v, \tau, w) \mid (u, v) \in \mathcal{E}, \tau < t\}$. Each node v is an out-neighbor of u indicated by a specific $\tau < t$. In other words, the edge (u, v) is added to the graph at time τ and maintained at time t. In GENTI, we employ an exponentially decaying weighting scheme for sampling temporal edges, which is shown to be empirically effective and enjoys efficient computation as introduced below. The associated sampling weight is determined by $w = \exp(\tau/\alpha)$, where $\alpha > 0$ is a hyper-parameter controlling the effect of temporal significance. The current degree of node u is denoted as $d_u = |\mathcal{N}_u|$.

Different from previous GRL methods that directly implement WNS by performing sampling from the entire neighborhood, we adopt an equivalence between WNS and the technique for dynamic weighted set sampling (DWSS) [263, 264, 265, 266]. The DWSS problem aims to perform k independent sampling with replacement from a dynamic set of elements based on corresponding weights. Corresponding to our context, it is equivalent to the WNS problem from the node neighborhood $\mathcal{N}_u$.

Zhang et al. [266] propose a bucketing scheme [263, 265] for efficiently solving the DWSS problem. Candidate elements are allocated into r buckets in total based on their weights, where the b-th bucket is $\mathcal{I}_{u,b} = \{(v, \tau, w) \mid (v, \tau, w) \in \mathcal{N}_u, \exp(b) \leq w < \exp(b+1)\}$, and the bucket index $0 \leq b \leq r$. Fetching an element from the buckets follows a rejection sampling scheme `FETCH`: when assessing an element v from bucket $\mathcal{I}_{u,b}$, the sampler either accepts it with a probability of $w/\exp(b+1)$, or rejects it and repeats the process until an element is accepted. [266] proves that the total complexity for sampling k elements can be bounded by $O(\log d_u + k)$.

6.3 The Framework of GENTI

In this section, we introduce GENTI by highlighting the GPU-oriented and streaming update aspects of our method. First, we present the overall framework of GENTI in Section 6.3.1 by introducing the relationship between different phases and data structures. Then, we describe the designs related to graph storage, decoupled sampling, and subgraph gathering in the following sections.

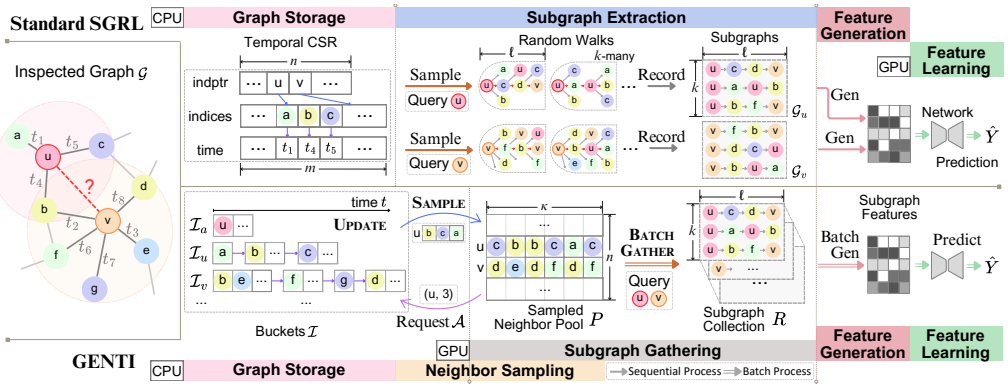


FIGURE 6.2: Framework overview of GENTI and standard SGRL methods. Conventionally, the three-stage SGRL pipeline, including subgraph extraction, feature generation, and feature learning, is conducted in a sequential manner. GENTI divides the critical subgraph extraction stage into asynchronous neighbor sampling and subgraph gathering, which enables the representation learning stages to commence in batches on GPU without blockage. The overlap of neighbor sampling and subgraph gathering stages indicates that they can be performed simultaneously in actual execution. Moreover, GENTI introduces a bucketing scheme to enhance dynamic graph storage by supporting efficient update and sampling.

6.3.1 GPU-powered Learning Pipeline

Figure 6.2 displays the overview of GENTI compared to typical walk-based SGRL methods [204, 205]. Among the three consecutive stages of the SGRL pipeline, i.e., subgraph extraction, feature generation, and feature learning, the last one usually occurs on GPU devices, enjoying efficient batch processing. In contrast, the former two stages must be sequentially computed by the CPU. During subgraph extraction, canonical SGRL performs random walk sampling directly from the stored graph structure. The frequent graph access results in a relatively high cache miss ratio. This stage hence becomes the efficiency bottleneck in the SGRL deployment as shown in Figure 6.1, even though its theoretical time complexity appears to be low.

To mitigate the imbalanced workload, we intend to exploit parallel computation and propose disentangling the subgraph extraction into two independent phases, one for sampling and the other for gathering. By introducing new data structures and addressing their dependency, the two stages can be executed in parallel on CPU and GPU, respectively. In this way, we successfully transfer all subsequent stages onto the GPU for batch processing. Our GPU-efficient pipeline of GENTI is shown in Algorithm 6.1. The tensor P maintains the candidate pool for subgraph extraction, with each row $P[u]$ representing the sampled neighbors of node u. For each timestamp, we construct both the source and

destination nodes of current link prediction queries as a single batch $\mathcal{U}_t$ and perform the batch *subgraph gathering* algorithm BSGATHER to construct the collection of subgraphs R for all queried nodes based on current pool P. The usage of P is recorded in requests $\mathcal{A}$, and the pool is updated on demand by calling *neighbor sampling* algorithm SAMPLE in a separate thread. Simultaneously, the graph storage is updated according to temporal events, and the prediction of the query batch is composed by learning from the generated features.

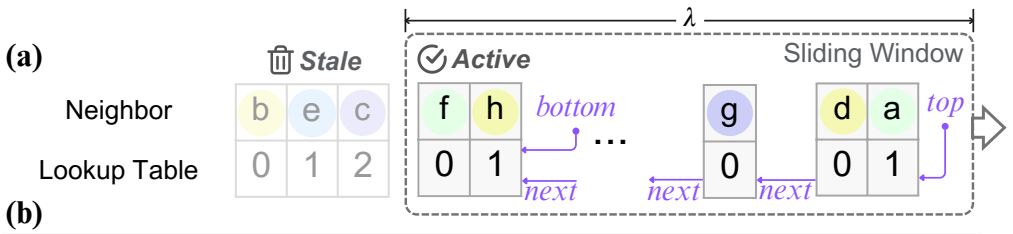
Among the operations, the SAMPLE and UPDATE are processed by CPU, while all other stages are conducted by GPU utilizing the power of batch computation. It is noteworthy that the two major data structures, i.e., the graph storage on RAM and the sampled neighbor pool on GPU, are updated asynchronously. Therefore, the subgraph learning on GPU is ensured to receive the up-to-date data to accommodate the dynamic graph changes. Meanwhile, the workload is considerably balanced to prevent cross-device blockage.

As GENTI focuses on the graph processing stages, it can be applied as the preprocessing for various SGRL feature learning models. In this study, we leverage the backbone of CAW [204] for downstream learning tasks. More specifically, it utilizes an Recurrent Neural Network (RNN) to encode sampled walks, and an attention layer to capture mutual relations.

ALGORITHM 6.1: GENTI

Input: Graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, X)$, link prediction queries $\{(u, v, \tau)\}$
Output: Predictions $\hat{Y}$

for all $u \in \mathcal{V}$ **do**
  $P[u] \leftarrow \text{SAMPLE}(u, \kappa)$
for each timestamp t **do**
  $\mathcal{U}_t \leftarrow \{u, v \mid \tau = t\}$
  Append $\mathcal{U}_t$ with negative queries if required
  $R, \mathcal{A} \leftarrow \text{BSGATHER}(P, \mathcal{U}_t)$
  for each $(u, k_u) \in \mathcal{A}$ **do** ▷ *in separate thread*
    $P[u] \leftarrow \text{SAMPLE}(u, k_u)$
  $\mathcal{G} \leftarrow \text{UPDATE}(\mathcal{U}_t)$
  $\hat{Y} \leftarrow \texttt{PREDICT}(R, X)$
return $\hat{Y}$



Bucket	$\mathcal{I}_{u,0}$	$\mathcal{I}_{u,1}$	⋯	$\mathcal{I}_{u,r-1}$	$\mathcal{I}_{u,r}$
Timestamp	$[0, \alpha)$	$[\alpha, 2\alpha)$	⋯	$[(r-1)\alpha, r\alpha)$	$[r\alpha, (r+1)\alpha)$
Sample Weight	$[0, e^1)$	$[e^1, e^2)$	⋯	$[e^{r-1}, e^r)$	$[e^r, e^{r+1})$
Random Number	$\varnothing$	$\{z_4\}$	⋯	$\{z_3\}$	$\{z_1, z_2\}$
Fetch Amount	0	1	⋯	1	2
Sampled Neighbor	$\varnothing$	f	...	g	a d

FIGURE 6.3: **(a)** Bucket-based graph storage $\mathcal{I}_u$ for maintaining the neighbor set of a node u in temporal order. Two pointers are maintained to track the top and bottom indices of the current sliding window, ensuring that only buckets inside it are active, and stale ones are removed. **(b)** Sampling is performed by successively fetching nodes from corresponding buckets to construct the sampled neighborhood.

6.3.2 Bucket-based Dynamic Graph Storage

Conventional SGRL stores graph on RAM by sparse matrix structures such as the compressed sparse row (CSR) format [201, 93]. We however note that it is not optimized for performing walk-based sampling, as walking on the graph requires iteratively accessing neighboring nodes. This process renders a time complexity of $O(k\ell d_{max})$ when generating k-many ℓ-length random walks for each source node, where d_{max} is the maximum degree of visited nodes [267]. In addition, live update on the graph structure is also of prohibitive overhead due to the continuous layout. As a consequence, dynamic SGRLs [204, 205] have to preprocess all the timestamps and maintain the entire temporal information, which inevitably causes computation and memory redundancy.

We mainly look into two abilities of the dynamic storage scheme: (1) efficient sampling for the neighborhood of given nodes; (2) online update of edge addition and deletion. Inspired by the DWSS process introduced in Section 6.2, we employ the bucketing scheme as our graph storage on RAM. As illustrated in Figure 6.3, we store out-neighbors of each node u by a linked list of non-empty buckets $\mathcal{I}_{u,b}$. Inside each bucket, we also maintain a lookup table for fast accessing the individual element. The index b of a bucket indicates its time interval and determines the temporal sampling weight $w_{u,b} = \sum_{(v,\tau,w)\in\mathcal{I}_{u,b}} w$. The total weight of all buckets with respect to node u is $w_u = \sum_b w_{u,b}$. Each traversal of

the linked list starts from the most recent non-empty bucket with the largest index r. Denote the hyperparameter representing duration length of a bucket as α and the current timestamp as t, there is $r = \lfloor t/\alpha \rfloor$.

Bucket number λ. An inherent merit of the bucketing storage is that, it is naturally sorted in temporal order, and only a certain number of buckets representing most recent updates are frequently accessed for sampling at the current timestamp. According to [266], the desired samples are likely to exist with high probability in buckets with index range $r - 2\lceil \ln d_u \rceil \leq b \leq r$. Therefore, we can apply a sliding window scheme that only maintains $\lambda = 2\lceil \ln d_u \rceil$ buckets in this time frame and progressively drops those stale ones. As a result, the memory usage in RAM remains $O(m + \lambda)$, as each edge is added only once and is deleted permanently upon expiration.

Sampling. As elaborated in Section 6.2, we innovatively utilize DWSS to implement the walk-based sampling. Instead of conducting k independent sampling, our storage scheme is capable of acquiring k samples *at once* as shown in Algorithm 6.2. To achieve this, we generate k random numbers uniformly within the overall weight range $z \sim U(0, w_u)$. The amount of samples drawn from each bucket is decided by the amount of numbers z falling in the corresponding bucket-wise weight range. Therefore, we are able to sample all k neighbors of the given node in a single traverse of buckets, starting from the most recent one and scanning through the maintained pointers. As the number of buckets is $\lambda = 2\lceil \ln d_u \rceil$, the total time complexity of SAMPLE is $O(\log d_u + k)$.

Figure 6.3 illustrates a running example for the sampling request $\mathcal{A} = (u, 4)$ enquiring

ALGORITHM 6.2: SAMPLE

Input: Storage $\mathcal{I}$, source node u, number of samples k
Output: Updated neighbor pool $P[u]$

$\mathcal{Z} \leftarrow \{z_1, \cdots, z_k \mid z_j \sim U(0, w_u),\ 1 \leq j \leq k\}$
$b \leftarrow r$, $w_{sum} \leftarrow w_u$
while $\mathcal{Z} \neq \emptyset$ **do**
    $w_{sum} \leftarrow w_{sum} - w_{u,b}$
    $\mathcal{Z}_b \leftarrow \{z \in \mathcal{Z} \mid z \geq w_{sum}\}$, $\mathcal{Z} \leftarrow \mathcal{Z} \setminus \mathcal{Z}_b$
    for $i \leftarrow 1$ to $|\mathcal{Z}_b|$ **do**
        $v \leftarrow \texttt{FETCH}(\mathcal{I}_b)$
        $c_{u,tail} \leftarrow (c_{u,tail} + 1) \mod \kappa$
        $P[u, c_{u,tail}] \leftarrow v$
    $b \leftarrow \mathcal{I}_{u,b}.next$
return P

$k = 4$ neighbors of source node u. At the current timestamp, the index range of active buckets is $1 \le b \le r$. The time range and weight range of each bucket can be directly derived from its index, and $k = 4$ random numbers are generated accordingly. Among them, both z_1 and z_2 are within the weight range of $\mathcal{I}_{u,r}$, while z_3 and z_4 correspond to $\mathcal{I}_{u,r-1}$ and $\mathcal{I}_{u,1}$, respectively. We perform $2, 1, 1$ times of retrieval `FETCH` from these buckets by successively accessing them in the linked list. Eventually, four neighbors are sampled and loaded into the pool $P[u]$.

Update. The dynamic UPDATE operation can be found in Algorithm 6.3. Thanks to the temporal order of our bucket storage, the target bucket corresponding to incoming and stale edges can be immediately indexed based on the timestamp. The lookup table then locates and manages the specific neighbor within the bucket to perform neighbor addition or deletion. At the end, stale buckets are removed since they are rarely accessed in the future. This is implemented by accessing the bucket with the lowest index in the linked list and setting its pointer to nil, marking its data as deleted. Overall, this scheme completes a single update event in $O(1)$ time.

6.3.3 Sampled Neighbor Pool

The sampled neighbor pool is a novel data structure we design to expedite the subgraph extraction, which serves as an intermediate storage on GPU for the sampled neighbors

ALGORITHM 6.3: UPDATE

Input: Storage $\mathcal{I}$, update events $\mathcal{S} = \{(u, v, t, op)\}$
Output: Up-to-date storage $\mathcal{I}$

for all $(u, v, t, op) \in \mathcal{S}$ **do**
  $b \leftarrow \lfloor t/\alpha \rfloor$
  if op = 'insert' **then**
    if $\mathcal{I}_{u,b} = \emptyset$ **then**
      insert $\mathcal{I}_{u,b}$
    $\mathcal{I}_{u,b} \leftarrow \mathcal{I}_{u,b} \cup \{v\}$
    $w \leftarrow \exp(t/\alpha)$, $w_{u,b} \leftarrow w_{u,b} + w$
  if op = 'delete' **then**
    $\mathcal{I}_{u,b} \leftarrow \mathcal{I}_{u,b} \setminus \{v\}$
    if $\mathcal{I}_{u,b} = \emptyset$ **then**
      remove $\mathcal{I}_{u,b}$
    $w_{u,b} \leftarrow w_{u,b} - \exp(t/\alpha)$
  remove $\mathcal{I}_{u,b'}$ for $b' < r - 2\lceil \ln d_u \rceil$
return $\mathcal{I}$

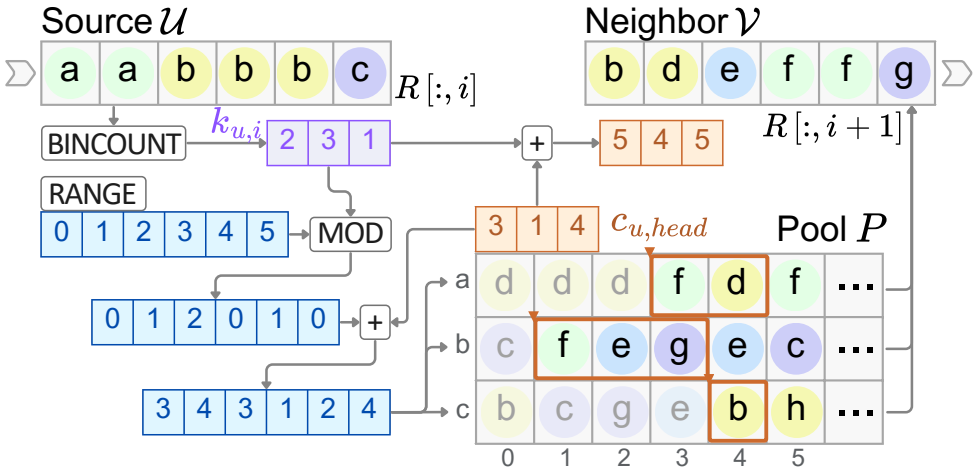


FIGURE 6.4: Example of one step of BSGATHER by computing indices and accessing the sampled neighbor pool. All operations are implemented on the GPU via batch processing.

acquired by Section 6.3.2, and further supports asynchronous gathering and updating. The pool is denoted as tensor $P \in \mathbb{R}^{n \times \kappa}$, where κ is the width of the pool. Each row of the tensor $P[u]$ represents a queue containing out-neighbors with respect to source node u. Two indices $c_{u,head}, c_{u,tail}$ maintain the current access points of BSGATHER and SAMPLE processes, respectively.

From the device perspective, subgraph sampling and gathering are performed on two separated threads, and the CPU and GPU are loosely coordinated by leveraging the pool storage. During subgraph extraction, the separate thread for SAMPLE loads the sampling results onto GPU and append them to the end of the queue in a streaming fashion. In contrast, the GPU process BSGATHER consumes neighbors stored in P to construct subgraphs. Pool elements positioned between the two pointers $c_{u,head}, c_{u,tail}$ are available for the following subgraph construction since they are freshly sampled and have not been accessed by previous gathering. The disentangled read and write operations towards the pool prevent BSGATHER from waiting for neighbor sampling to provide the updated data and thereby enjoys better throughput.

Pool width κ. The width κ of queues in the pool has the following requirements: (1) it should be as small as possible to save GPU memory usage; (2) it should be large enough to accommodate the neighbor consumption by BSGATHER for applicable cases. As described in Section 6.3.4, the gathering procedure carries out k-many ℓ-length random walks for each query node. A naive selection is thence $\kappa = k\ell$, resulting in a memory expense of $O(nk\ell)$ as in [96, 93] for the entire static graph.

By referring to the cache strategy in the two-pass streaming random walk [268], we are able to reduce the overhead with a tightened bound for κ. We distinguish the storage schemes into two types based on the consumption status of pool elements. *Heavy* nodes are source nodes being likely to visit more than κ neighbors. Hence we directly access all the neighbors with size d_u of these nodes during walks. On the contrary, candidate neighbors for *light* nodes are handled by the pool since they will not exhaust the κ-length queue during gathering. [268] derive the following lemma:

Lemma 6.1 ([268])**.** *Under the boundary $\kappa \sim O(\sqrt{\ell})$, the sum of outgoing degrees of all heavy nodes u is bounded by $\sum_u d_u \leq O(n\sqrt{\ell})$.*

In our implementation, we set $\kappa = k\sqrt{\ell}$ as the pool width. Consequently, the total GPU memory usage for all nodes in the graph, including both heavy and light ones, has a complexity of $O(nk\sqrt{\ell})$.

6.3.4 Batch Subgraph Gathering

In this section, we highlight the GPU-efficient batch subgraph gathering BSGATHER as illustrated in Algorithm 6.4. It returns a shape $|\mathcal{U}_t| \times k \times \ell$ tensor R containing subgraphs formed by k-many ℓ-length random walks with respect to $|\mathcal{U}_t|$ query nodes. The tensor R can be directly used by the subsequent feature generation and learning pipeline, thereby boosts the GPU batch processing.

According to the selective caching strategy adapted in Section 6.3.3, in each walk step, we split the current nodes into heavy and light multisets $\mathcal{U}_{heavy}$ and $\mathcal{U}_{light}$ by comparing the pointers $c_{u,head}$ and $c_{u,tail}$. Different from [268] requiring an individual round of walk to identify heavy nodes, our splitting scheme of the multiset can be instantly completed in $O(1)$ time. We then perform discriminative sampling for these nodes as one step of walk. For heavy source nodes, we gather all their neighbors directly from the graph structure $\mathcal{I}_u$. For light nodes, the sampled neighbor pool can be employed to provide $k_{u,i}$ number of active neighboring nodes based on the multiplicity $k_{u,i}$, i.e., occurrence of source node u in the multiset. All the sampled neighbors are combined and recorded as the i-th step random walk result R constituting the subgraph, and initiate the next step. After the ℓ-length walk is finished, nodes accessed during the process are marked as stale and are accordingly updated by sending requests to the SAMPLE thread. This is to ensure each pool element is used once so that the walks for subgraph extraction are mutually

independent. Since there are ℓ iteration steps handling k-many random walks, the total complexity of BSGATHER is $O(k\ell)$.

Note that for each walk step i, all operations regarding the k walks are performed in batches so that the GPU computation power is fully utilized. An example for conducting a BSGATHER step is provided in Figure 6.4. We first calculate the occurrence $k_{u,i}$ of each individual element in current node set $\mathcal{U}$, which is then used to update the head pointer and generate neighbor indices, both by batch addition operation. As all of the nodes belong to the light set, the algorithm constructs $\mathcal{V}$ by accessing entries in the pool P based on the indices corresponding to source nodes.

6.4 Experimental Evaluation

In this section, we empirically evaluate the proposed framework GENTI, targeting the following major questions:

RQ1. Can GENTI provide prediction performance comparable to other GRL methods for CTDGs?

RQ2. What is efficiency gain of GENTI in the subgraph extraction stage and overall training time?

ALGORITHM 6.4: BSGATHER

Input: Storage $\mathcal{I}$, sampled neighbor pool P, walk number k, walk length ℓ, seed nodes $\mathcal{U}_t$

Output: Random walks R of shape $|\mathcal{U}_t| \times k \times \ell$, requests $\mathcal{A}$

Construct multiset $\mathcal{V}$ by repeating k times for each $u \in \mathcal{U}_t$
for $i \leftarrow 1$ to ℓ **do**
  $\mathcal{U}_{heavy} \leftarrow \{u \mid u \in \mathcal{V}, c_{u,head} = c_{u,tail}\}$
  $\mathcal{V}_{heavy} \leftarrow \{v \mid v \in \mathcal{I}_u, u \in \mathcal{U}_{heavy}\}$
  $\mathcal{U}_{light} \leftarrow \mathcal{V} \setminus \mathcal{U}_{heavy}, \mathcal{V}_{light} \leftarrow \emptyset$
  for all each identical $u \in \mathcal{U}_{light}$ **do**
    $k_{u,i} \leftarrow \texttt{BINCOUNT}(u, \mathcal{U}_{light})$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{(u, k_{u,i})\}$
    $\mathcal{V}_{light} \leftarrow \mathcal{V}_{light} \cup P[u, c_{u,head} : (c_{u,head} + k_{u,i})]$
    $c_{u,head} \leftarrow (c_{u,head} + k_{u,i}) \mod \kappa$
  $\mathcal{V} \leftarrow \mathcal{V}_{heavy} \cup \mathcal{V}_{light}, R[:, :, i] \leftarrow \mathcal{V}$
return R, $\mathcal{A}$

RQ3. How does our pipeline design affect the workloads on CPU and GPU, and can they be fully parallelized?

RQ4. How does the performance of GENTI vary with changes in sampling settings including the random walk number, length, and the neighbor pool size?

6.4.1 Experimental Setup

Task and Metric. We evaluate CTDG representation learning by the common task of future link prediction in both transductive and inductive settings [204]. In the *transductive* training, temporal links between all nodes in the graph are observed up to a specific timestamp. The model is tested by predicting the existence of remaining links after the time point. The *inductive* link prediction task involves predicting links associated with nodes that were not observed in the training node set. For both settings, model performance is assessed using average precision (AP) on the test set following previous GRL works [204, 271, 202]. Particularly, GENTI are able to apply for various other tasks such as high-order patterns identification with minor modifications regarding to the input nodes. We exclusively concentrate on the above tasks in this study, as there are currently no dynamic datasets accessible for other tasks.

Datasets. We conduct experiments on 7 real-world dynamic datasets, including 3 small-scale ones: UCI-MSG, WIKIPEDIA, and REDDIT; 4 large-scale ones: SUPERUSER, WIKI-TALK, TGBL-COMMENT and MAG. The statistics of datasets are presented in Table 6.1. We utilize the same chronological split with ratios 70%/15%/15% for train/validation/test sets as [204]. In the inductive setting, we randomly select 10% of nodes and exclude the corresponding temporal links from the training sets.

TABLE 6.1: Statistics of dynamic graph datasets including the number of nodes, temporal edges, the dimension of node features, edge features, and timespan.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	f_v	f_e	T
UCI-MSG [269]	1,899	59,835	0	0	180 days
WIKIPEDIA [204]	9,227	157,474	172	172	30 days
REDDIT [204]	10,984	672,447	172	172	30 days
SUPERUSER [202]	194,085	1,443,339	172	0	2773 days
WIKI-TALK [202]	1,140,149	7,833,140	172	0	2320 days
TGBL-COMMENT [270]	994,790	44,314,507	172	0	1848 days
MAG [201]	121,751,665	1,297,748,926	768	0	1826 days

TABLE 6.2: Comparison of SGRL methods on small CTDGs. We present model performance metrics including transductive average precision (%), inductive average precision (%), total training time (s) , and the number (#) of convergence epochs. The best and second-best results in each column are marked with bold and underlined fonts, respectively. Particularly, "TLE" indicates a time limit exceeded exception that one epoch of model training exceeds 24 hours.

Model	UCI-MSG			WIKIPEDIA			REDDIT		
	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)
JODIE	80.27 ± 0.1	71.64 ± 0.6	431(12)	95.16 ± 0.4	93.13 ± 0.5	1985(18)	95.83 ± 0.3	93.20 ± 0.4	8320.4(12)
TGAT	60.25 ± 0.3	75.27 ± 2.3	689(25)	94.26 ± 0.1	92.88 ± 0.3	2428(29)	97.80 ± 0.2	96.08 ± 0.3	11138(24)
TGN	78.91 ± 0.1	75.47 ± 0.1	507(21)	98.58 ± 0.1	98.05 ± 0.1	1839(26)	98.66 ± 0.1	97.55 ± 0.1	8152(26)
APAN	84.02 ± 0.3	83.14 ± 0.5	**266(25)**	96.41 ± 0.5	96.06 ± 0.4	1352(21)	98.50 ± 0.2	97.62 ± 0.7	7728(9)
Zebra	92.74 ± 0.2	91.16 ± 0.3	483(31)	98.63 ± 0.1	98.65 ± 0.1	1329(32)	98.73 ± 0.1	98.42 ± 0.1	6207(25)
D-DGNN	90.41 ± 0.1	89.72 ± 0.1	14467(30)	99.16 ± 0.3	98.54 ± 0.2	15173(30)	**98.93 ± 0.2**	98.56 ± 0.1	50342(30)
CAW	95.33 ± 0.3	95.19 ± 0.2	1488(8)	99.18 ± 0.1	99.34 ± 0.1	3720(5)	98.80 ± 0.1	98.99 ± 0.1	30912(8)
NeurTWs	**95.46 ± 0.3**	95.70 ± 0.2	44064(12)	99.17 ± 0.1	99.32 ± 0.1	65448(9)	98.32 ± 0.2	98.05 ± 0.1	TLE
GENTI	95.36 ± 0.3	**95.82 ± 0.3**	394(8)	**99.18 ± 0.1**	**99.37 ± 0.1**	**739(8)**	98.87 ± 0.1	**99.18 ± 0.1**	**5890(8)**

TABLE 6.3: Comparison of representative SGRL methods on large CTDGs. We present the same performance metrics as Table 6.2, but the training time are measured in hours. Particularly, “OOM” denotes the out-of-memory error, and "TLE" indicates a time limit exceeded exception that one epoch of model training exceeds 24 hours.

Model	SUPERUSER			WIKI-TALK			TGBL-COMMENT			MAG		
	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)	Trans AP	Induct AP	Time (#)
APAN	89.63 ± 0.5	86.32 ± 0.4	5.18(16)	TLE	TLE	TLE	TLE	TLE	TLE	TLE	TLE	TLE
Zebra	93.34 ± 0.3	97.63 ± 0.2	0.92(21)	95.25 ± 0.1	97.56 ± 0.3	4.65(18)	91.32 ± 0.5	95.15 ± 0.3	18.4(10)	OOM	OOM	OOM
CAW	93.12 ± 0.2	97.36 ± 0.4	10.9(5)	95.37 ± 0.1	98.07 ± 0.3	73.3(10)	TLE	TLE	TLE	TLE	TLE	TLE
GENTI	**94.05 ± 0.2**	**98.27 ± 0.3**	**0.66(8)**	**95.53 ± 0.1**	**98.58 ± 0.2**	**3.70(8)**	**93.51 ± 0.3**	**95.61 ± 0.2**	**12.3(6)**	**94.88 ± 0.4**	**99.45 ± 0.1**	**192(10)**

Baselines. We extensively evaluate 8 state-of-the-art GRL methods applicable on CTDGs with varying categories and designs. They are: (1) GNN-based: JODIE [272], TGAT [273], TGN [206], APAN [271]; (2) Metric-based: Zebra [202], D-DGNN [199]; (3) Temporal walk-based: CAW [204], NeurTWs [205]. We mainly utilize their released source code and training configuration with the best prediction performance, and perform evaluation on our platform. For D-DGNN, we incorporate its preprocessing cost into the training time in our experiment to take update efficiency into account.

Hyperparameters. For a fair comparison, GENTI keeps consistency in network settings with [204, 205], except for fixing the encoder to correctly generate prediction on non-attribute graphs. We employ a bucket length $\alpha = \lfloor T/100 \rfloor$ based on the maximum timestamp T. We set the batch size to 32 for small-scale datasets, 128 for million-scale datasets, and 512 for MAG, as an effort to reproduce baselines without causing the out-of-memory error.

Environment. Experiments are conducted on a server with 192GB RAM, two 28-core Intel Xeon CPUs (2.2GHz), and an NVIDIA RTX A5000 GPU (24GB memory). Our asynchronous SGRL pipeline is implemented by the PyTorch 2.0 multiprocessing interface [274].

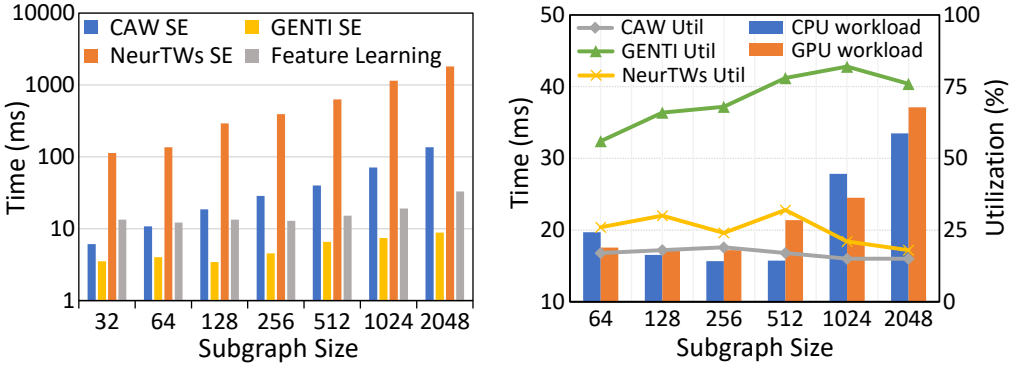
6.4.2 Main Results

Table 6.2 and Table 6.3 present our key results of model performance in the future link prediction task on 7 real-world dynamic graphs. Due to the prolonged training times of most baselines on larger graphs, we evaluate only efficient representatives on large CTDGs in Table 6.3. We conduct a comparative analysis of GENTI against the baselines in terms of both effectiveness and efficiency.

Prediction Accuracy. For GNN-based methods, the average precision is similar to the evaluation in previous studies [202] since we inherit the same experimental settings. It is worth noting that GNN-based methods exhibit relatively inferior performance due to the constraint of their aggregation scheme, which marks the superiority of subgraph-based approaches. Among SGRL methods, models with walk-based designs tend to outperform metric-based ones with fewer epochs in most cases, particularly on non-attribute graphs, where the ability to learn graph structural information is pivotal. This superiority is brought by the employment of their expressive structural encoding technique that utilizes the power of temporal random walk. Moreover, CAW and NeurTWs stand out for requiring fewer epochs to achieve model convergence due to their anonymization encoding strategy, which enhances their generalization capabilities.

In our approach, we focus on enhancing the subgraph extraction stage of walk-based methods while adhering to their random walk encoding techniques. Therefore, we achieve nearly equivalent model convergence speed and prediction accuracy. Similar observations hold for other evaluation metrics, such as accuracy and AUC. As a result, GENTI successfully achieves comparable or better model convergence speed and prediction accuracy as our implementation yields equivalent subgraph extraction and feature learning results and guarantees the model efficacy.

Efficiency and Scalability. According to Table 6.2 and Table 6.3, GENTI consistently achieves either optimal or suboptimal total training time across all datasets compared to other GRL methods. We highlight that GENTI exhibits notable improvements in



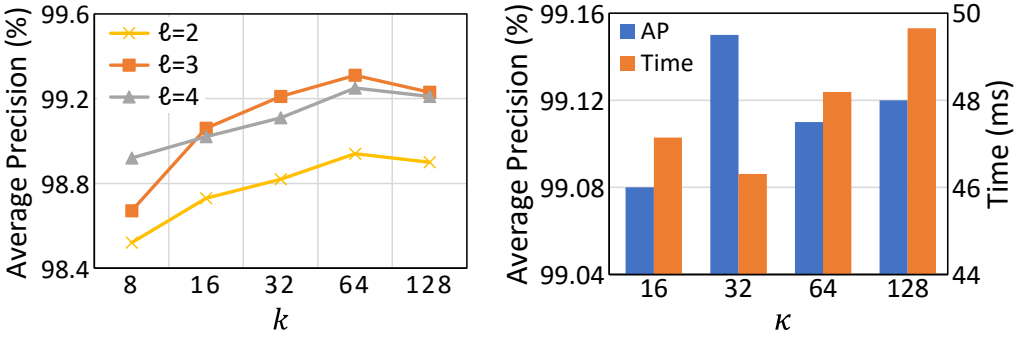
(a) Subgraph Extraction (SE) **(b)** Device Usage

FIGURE 6.5: **(a)** Breakdown running time of the forward subgraph extraction stage of GENTI and walk-based SGRL methods with different subgraph size during single prediction. Overhead of the common feature learning stage is also plotted. Note that the time axis is on a log scale. **(b)** Line chart is the comparison on GPU utilization rate between GENTI and baselines. Corresponding CPU and GPU running times in GENTI forward pipeline are also plotted as bars.

the running time of each training epoch when compared to other walk-based methods, especially on large-scale graphs. Compared to its predecessor CAW, it realizes approximately 3~26× speedups across all datasets. Benefiting from the GPU-oriented design that addresses the pipeline bottleneck and boosts the subgraph extraction, GENTI is the first walk-based method to complete training within 4 hours for the million-size WIKI-TALK and accomplish a single training epoch within 24 hours for the largest MAG. Conversely, other baselines face challenges related to either time constraints or memory issues particularly on large-scale graphs.

Improvement of Subgraph Extraction. We specifically look into the GENTI improvement on the subgraph extraction stage that comprises the majority of SGRL overhead. Figure 6.5(a) illustrates the separate inference time of the subgraph extraction stage among walk-based methods. The feature learning time is also plotted for reference. The result highlights the scalability of GENTI, as it achieves significantly shorter extraction latency and is less sensitive to the increase of subgraph size. When the subgraph size is large, we observe near 30× acceleration compared to the best SGRL counterparts. GENTI is thus more feasible on large-scale graphs thanks to the larger applicable batch size for computation.

Device Usage. We examine the CPU and GPU workload in GENTI pipeline and compare device utilization with other walk-based SGRL methods in Figure 6.5(b). It can be observed that during GENTI execution, workload between the two devices is generally



(a) Walk Parameters **(b)** Pool Width

FIGURE 6.6: **(a)** Impact of the random walk number k and length ℓ on GENTI prediction accuracy on WIKIPEDIA. **(b)** Impact of the pool width κ on GENTI prediction accuracy and single-query prediction time when $k = 32$ and $\ell = 4$.

balanced, and the average device utilization reaches 80% even with a large subgraph size. In comparison, the CPU workload becomes a bottleneck in the pipelines of CAW and NeurTWs when extracting large subgraphs, leading to low device utilization. In summary, our GENTI design contributes to balanced workloads and facilitates parallelism of the streaming pipeline.

6.4.3 Effect of Parameters

In this section, we investigate the impact of important GENTI parameters on both efficiency and prediction performance to evaluate our method design and provide guidance on the parameter choice. Due to the page limit, we mainly discuss representative results of transductive experiments on the WIKIPEDIA dataset.

Walk Number k. The value of k determines the extent of neighborhood information extracted by SGRL for feature generation and learning. Figure 6.6(a) displays the effect of k on GENTI transductive average precision. The prediction precision is positively related to the value of k when $k \leq 64$. We deduce the optimal performance when $k = 64$ indicates that, generating such k-many random walks is adequate for constructing the expressive subgraph of each seed node, while a larger number will further introduce redundancy and negatively affect feature learning. As a general conclusion, choosing a small k is ideal when embedding graphs with low-diversity interaction patterns. Otherwise, a larger value can be applied for sufficiently sampling the nodes and representing the subgraph.

Walk Length ℓ. A larger value of ℓ implies using a longer random walk to form a subgraph, allowing for the exploration of more distant information in the whole graph. According to Figure 6.6(a), when ℓ is set to 3, we achieve the best performance in the future link prediction task, which is a general observation across all datasets in our experiment. Intuitively, this preference indicates that the local neighbors are greater importance compared to those nodes at longer distances. We believe that a generally small ℓ can be employed in scenarios of a similar graph structure and prediction task.

Pool size κ. The pool size κ can impact both prediction accuracy and algorithmic efficiency since it decides the heavy-light splitting scheme in BSGATHER. A small κ can save GPU footprint but results in numerous heavy nodes that cannot be processed efficiently in a single batch. A larger κ tends to introduce more update and sampling workload on CPU and undermines the balanced workload between the two devices. We utilize the prediction time to specifically present the impact of κ on the inference-time efficiency of GENTI. According to our evaluation in Figure 6.6(b), when walking with $k = 32$ and $\ell = 4$, the optimal value with regard to both accuracy and efficiency is achieved by $\kappa = 32$. Comparing to the original selection $\kappa = k\sqrt{\ell}$ in Section 6.3.3, it is considerably smaller with a 4× reduction in memory overhead. The observation suggests that, heavy nodes are uncommon in real-world graphs, and the sampled neighbor pool scheme is competent as random walks rarely run out of its items. In practice, κ can be empirically set to a relatively low value to save GPU memory budget.

6.5 Summary and Discussion

In this work, we propose GENTI, a novel algorithm designed for scalable subgraph-based graph representation learning on dynamic graphs. GENTI decouples the subgraph extraction stage, commonly the bottleneck of SGRL methods, into two asynchronous phases and boosts GPU utilization. In specific, GPU processing incorporates the efficient BSGATHER for subgraph gathering in batches and subsequent feature generation and learning. CPU is solely responsible for maintaining the dynamic graph structure with SAMPLE and UPDATE operations in a streaming manner. We conduct extensive experiments on various datasets to demonstrate GENTI's scalability in subgraph extraction and overall graph learning. GENTI achieves up to 26× faster training time than the state-of-the-art walk-based SGRL methods, without sacrificing prediction performance. Notably, GENTI efficiently processes the billion-scale graph MAG within 24 hours for

each epoch, a significant improvement over existing solutions unable to complete a single training epoch in the same timeframe.

In real-world applications, graphs are stored in distributed machines, potentially introducing additional communication latency to the neighbor sampling stage of GENTI. Therefore, extending GENTI to scenarios with distributed graph storage stands as a potential future direction, enhancing the applicability of GENTI for practical and general purposes.

Part III

Evaluating Graph Neural Network Scalability Performance

Chapter 7

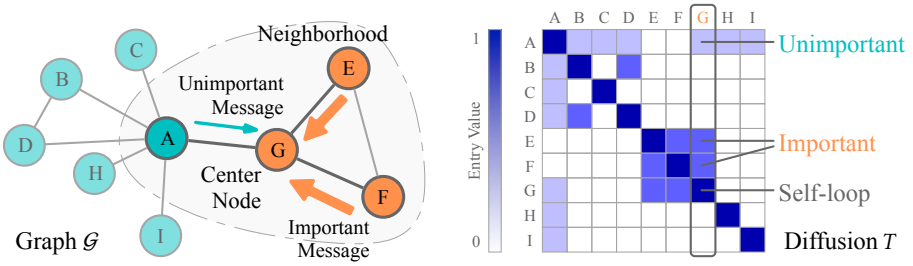
Understanding Graph Neural Networks by Unified Entry-Wise Sparsification

7.1 Introduction

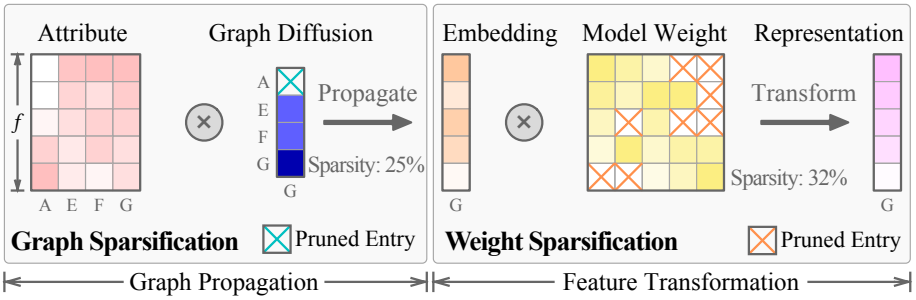
The performance bottleneck arises from the connection with graph size, as both graph propagation and feature transformation can be viewed as multiplications on graph-scale matrices, and lead to computational complexities proportional to the numbers of edges and nodes in the graph, respectively [111]. Hence, the essence of improving GNN learning efficiency lies in reducing the number of operations associated with graph diffusion and model weights [69, 101].

Empirically, a diverse range of strategies have been explored for saving computational cost by sparsifying the graph as well as the model. In efforts to simplify graph computation, prior research utilizes graph sparsification techniques [275]. This typically entails removing graph components based on either predetermined [142, 152, 147] or learned [144, 145, 172] criteria. However, the persistence of a static graph through all propagation hops causes a dilemma: the topology may become overly coarse, thereby omitting crucial information for GNN feature extraction, or the graph structure may be under-sparsified and model efficiency is scarcely improved.

This work is published as: [6] **Ningyi Liao**, Zihao Yu, Ruixiao Zeng, Siqiang Luo. "UNIFEWS: You Need Fewer Operations for Efficient Graph Neural Networks". In *42nd International Conference on Machine Learning (**ICML**)*, vol. 267, pp. 37587–37609, 2025.



(a) An example graph and the diffusion matrix.



(b) UNIFEWS on graph and model entries.

FIGURE 7.1: **(a)** Within one hop of aggregation to the center node, messages from neighbors with larger diffusion values are associated with greater importance. **(b)** UNIFEWS jointly applies sparsification to both graph propagation and feature transformation stages in each GNN layer. Color intensity of an entry denotes its relative magnitude. Unimportant entries of the diffusion and weight matrices are pruned in order to reduce computational operations.

Another direction is to exploit compression on the model architecture by integrating neural network pruning approaches [276], which gradually sparsifies weight elements during GNN training based on performance assessments [158, 164, 162, 165]. Regardless of the improved flexibility, their scalability for large graphs remains limited, as full-graph computation is still necessary at certain stages of the pipeline. Moreover, there is a noticeable gap in the theoretical interpretation of GNN sparsification despite its practical utility. Existing works are either constrained to the layer-agnostic graph approximation [126, 127], or only provide straightforward representation error analysis [146, 147]. The impact of simplified graphs on the GNN learning process, particularly through multiple layers, remains inadequately addressed.

To mitigate the above drawbacks of current compression solutions, we propose to rethink both graph propagation and feature transformation from an entry-wise perspective, i.e., examining individual matrix elements. As illustrated in Figure 7.1, the two operations are performed by employing the diffusion and weight matrices, respectively. Directly sparsifying entries in these two matrices enables us to reduce the number of *entry*

operations. The entry-wise mechanism enjoys flexibility compared to conventional graph-level sparsification techniques, while ensuring a deterministic graph topology for network learning.

In this work, we propose **UNIFEWS**, a UNIFied Entry-Wise Sparsi-fication framework for GNN graph and weights. Theoretically, we establish a quantitative framework featuring UNIFEWS approximation with respect to the graph learning objective, and demonstrate that the amount of reduced operations is at least linear to the sparsity. Our analysis elucidates the effect of sparsification across iterative GNN layers, which differs from the previous approach focusing on a particularized metric for approximation.

In practice, UNIFEWS is capable of jointly removing entry operations in both graph propagation and feature transformation towards improved model efficiency. The simple but effective strategy is adeptly integrated into matrix computations on the fly without incurring additional overhead. The progressive sparsification across layers further fosters an increase in sparsity for both stages, which reciprocally augments their efficiency. Thanks to our unified interpretation, the theoretical and implementation framework of UNIFEWS is applicable to a broad family of message-passing GNNs, covering the representative models of both iterative and decoupled architectures, and benefits the real-world model efficiency during training and inference.

7.2 Graph Smoothing under Sparsification

In this section, we first relate GNN learning with the graph smoothing process, which enables us to characterize the process by our novel approximation bound.

GNN as Graph Smoothing. Generally, a broad scope of both iterative and decoupled GNNs can be interpreted by a spectral graph smoothing process [277]. We adopt the optimization framework as:

Definition 7.1 (**Graph Laplacian Smoothing** [234])**.** Given a weighted graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ with Laplacian matrix $\boldsymbol{L}$. Based on an input signal vector $\boldsymbol{x} \in \mathbb{R}^n$, the Graph Laplacian Smoothing problem aims to optimize vector $\boldsymbol{p} \in \mathbb{R}^n$ with the goal:

$$\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}} \mathcal{L}, \quad \mathcal{L} = \|\boldsymbol{p} - \boldsymbol{x}\|^2 + c \cdot \boldsymbol{p}^\top \boldsymbol{L} \boldsymbol{p}, \tag{7.1}$$

where $\|\cdot\|$ is the vector L_2 norm and the regularization coefficient c is chosen from $[0, 1]$.

In Eq. (7.1), $\boldsymbol{L}$ is the general Laplacian matrix, as normalization only causes a difference in coefficient. The first term of $\mathcal{L}$ reflects the closeness to the *input signal*, i.e., node attributes representing their identity. The second term is associated with the *graph structure*, acting as a regularization that constrains the representation values of connected node pairs to be similar. The closed-form solution $\boldsymbol{p}^* = (\boldsymbol{I} + c\boldsymbol{L})^{-1}\boldsymbol{x}$ can be inferred when the derivative $\partial\mathcal{L}/\partial\boldsymbol{p} = 0$. However, note that it is prohibitive to directly acquire the converged solution due to the inverse calculation on large graph-scale matrix. Hence, GNN models employ an iterative approach to learn the representation under this framework with varying architectural designs. For example, GCN convolution corresponds to $c = 1$.

Approximate Graph Smoothing. Next, we introduce graph sparsifiers as an approximation process of graph smoothing. In order to measure the extent of approximation and its impact on learning outcomes, we consider the spectral similarity:

Definition 7.2 (ϵ**-Spectral Similarity**)**.** The approximate Laplacian matrix $\hat{\boldsymbol{L}}$ is said to be ϵ-spectrally similar to the raw Laplacian matrix $\boldsymbol{L}$ if:

$$\boldsymbol{x}^\top(\hat{\boldsymbol{L}} - \epsilon\boldsymbol{I})\boldsymbol{x} \leq \boldsymbol{x}^\top\boldsymbol{L}\boldsymbol{x} \leq \boldsymbol{x}^\top(\hat{\boldsymbol{L}} + \epsilon\boldsymbol{I})\boldsymbol{x},\ \forall\boldsymbol{x} \in \mathbb{R}^n. \tag{7.2}$$

or, equivalently:

$$\left|\boldsymbol{x}^\top(\boldsymbol{L} - \hat{\boldsymbol{L}})\boldsymbol{x}\right| \leq \epsilon \cdot \|\boldsymbol{x}\|^2, \quad \forall\boldsymbol{x} \in \mathbb{R}^n. \tag{7.3}$$

Graph approximation satisfying Definition 7.2 can be regarded as spectral sparsification [148, 278], which identifies operations that maintains certain spectral properties such as eigenvalues during modification. Compared to the common *multiplicative* spectral similarity [279, 280, 281], Definition 7.2 possesses an *additive* tolerance, which allows for manipulating specific entries of $\boldsymbol{L}$ and suits our scenario.

Then, we are able to characterize the graph smoothing problem for sparsified graphs. Under edge modifications, we intend to bound the optimization goal under approximation:

Lemma 7.1 (**Approximate Graph Laplacian Smoothing**)**.** *Given two graphs $\mathcal{G} = \langle\mathcal{V}, \mathcal{E}\rangle$ and $\hat{\mathcal{G}} = \langle\mathcal{V}, \hat{\mathcal{E}}\rangle$, where $\hat{\mathcal{E}}$ is the sparsified edge set. When the Laplacian $\hat{\boldsymbol{L}}$ of $\hat{\mathcal{G}}$ is ϵ-similar to $\boldsymbol{L}$ of $\mathcal{G}$, the solution $\hat{\boldsymbol{p}}^*$ to the problem Eq. (7.1) w.r.t $\hat{\boldsymbol{L}}$ is called an ϵ-approximation of the solution $\boldsymbol{p}^*$ w.r.t. $\boldsymbol{L}$, and:*

$$\|\hat{\boldsymbol{p}}^* - \boldsymbol{p}^*\| \leq c\epsilon\|\boldsymbol{p}^*\|. \tag{7.4}$$

The proof of Theorem 7.1 can be found in Appendix B.1. It establishes a novel interpretation for characterizing the smoothing procedure under a sparsified graph, that if a sparsifier complies with the spectral similarity Definition 7.2, it is capable of effectively approximating the iterative graph optimization and achieving a close output with bounded error. Compared to approximation bounds specifying feature values in previous GNN sparsification theories [146, 157], our analysis highlights the impact of graph sparsifier throughout the holistic learning process, which enjoys better expressiveness and suitability.

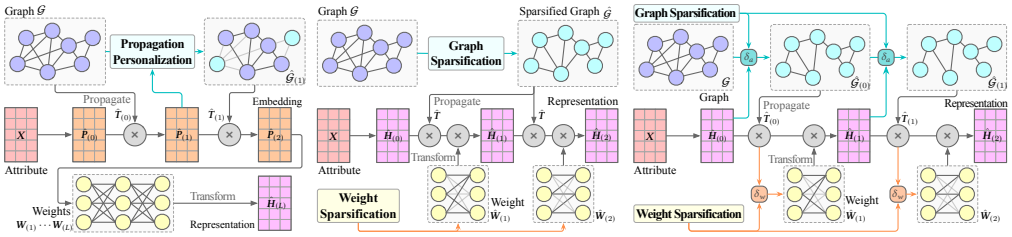
7.3 Method

This section presents our UNIFEWS framework by respectively developing its application to decoupled and iterative GNNs, where graph and weight sparsification are separately performed in the former architecture, and are combined in the latter. Our key theoretical insights are summarized as:

- **Bridging sparsification and smoothing:** UNIFEWS can be described as an spectral sparsifier. Its sparsification strength is determined by the threshold of entry removal.
- **Multi-layer bounds:** UNIFEWS provides an approximation to the graph smoothing optimization. Its representation error to the original objective is effectively bounded.
- **Improvements on efficiency and efficacy:** UNIFEWS is advantageous in enhancing efficiency, mitigating the over-smoothing issue, and facilitating enhanced joint sparsity.

7.3.1 UNIFEWS as Spectral Sparsification

Intuition: Entry Values Denote Importance in Graph Computation. As depicted in Figure 7.2(b), conventional graph sparsification methods for GNNs only provide fixed and uniform *graph-level* adjustments through the entire learning process. This lack of flexibility hinders their performance when employing to the recurrent update design with multiple GNN layers. Contrarily, *node-wise* models in Figure 7.2(a) aim to personalize the graph-based propagation, but come with additional explicit calculations and impaired



(a) Personal Propagation **(b)** Architecture Compression **(c)** UNIFEWS on Iterative GNN

FIGURE 7.2: Comparison of GNN learning pipelines between conventional simplification techniques and our UNIFEWS framework. **(a)** The approach of personalized propagation iteratively simplifies the node-dependent graph diffusion but is only applicable to decoupled GNNs. **(b)** Joint model compression can sparsify both graph and weight, whereas the same diffusion is uniformly utilized across all layers. **(c)** In contrast, our UNIFEWS framework implements on-the-fly unified sparsification of both graph and weights for each layer. The adaptive entry-wise scheme enjoys fine-grained control and improved sparsity.

runtime efficiency. We are hence motivated to design a sparsification approach that (1) enables modifications in a *fine-grained* manner to further simplify the computation; and (2) can be seamlessly integrated into the matrix operation with *negligible overhead*.

To this end, we first focus on the isolated graph propagation stage and express the propagation of decoupled models in an *entry-wise* manner on node u as:

$$\boldsymbol{p}_{(l+1)}[u] = \sum_{v \in \mathcal{N}(u)} \tau_{(l)}[u,v] = \sum_{v \in \mathcal{N}(u)} \boldsymbol{T}_{(l)}[u,v] \cdot \boldsymbol{p}_{(l)}[v], \tag{7.5}$$

where the propagation message $\tau_{(l)}[u,v]$ from node v to u is regarded as an entry. As illustrated in Figure 7.1(a), in a single hop of GNN diffusion, edges carrying propagation messages exert varying impacts on neighboring nodes, whose importance is dependent on the graph topology. Messages with minor significance are usually considered redundant and can be eliminated to facilitate sparsity. From the perspective of matrix computation, this is achieved by omitting the particular message $\tau[u,v]$ in current aggregation based on an importance indicator such as its magnitude.

Edge Pruning for Single Layer. In order to perform pruning on entry $\tau[u,v]$, the diffusion matrix $\boldsymbol{T}$ can be utilized to record the pruning information, which is exactly the concept of graph sparsification. Given a universal magnitude threshold δ_a, zeroing out entries $|\tau[u,v]| < \delta_a$ is equivalent to sparsifying $\boldsymbol{T}$ with a node-wise threshold δ'_a as:

$$\hat{\boldsymbol{T}}[u,v] = \mathrm{thr}_{\delta'_a}\left(\boldsymbol{T}[u,v]\right) \cdot \boldsymbol{T}[u,v], \quad \delta'_a = \delta_a / |\boldsymbol{p}[v]|, \tag{7.6}$$

where the pruning function with an arbitrary threshold δ is $\text{thr}_\delta(x) = 1$ if $|x| > \delta$, and $\text{thr}_\delta(x) = 0$ otherwise. Sparsification by Eq. (7.6) has two concurrent effects: for graph computation, messages with small magnitudes are prevented from propagating to neighbors and composing the output representation; for graph topology, corresponding edges are removed from the diffusion process.

Next, we show that edge pruning in Eq. (7.6) for one layer can be considered as a spectral sparsification following Definition 7.2, which enables us to derive its approximation bound. Practically, the target sparsity $\eta_a \in [0, 1]$ is usually determined by the realistic application. Let $\hat{\mathcal{E}}$ be the edge set achieved by UNIFEWS sparsification and the corresponding Laplacian is $\hat{\boldsymbol{L}}$, there is $\eta_a = 1 - |\hat{\mathcal{E}}|/m$. We first derive the following lemma as a general guarantee associating sparsification and spectral similarity:

Lemma 7.2. *Given graph $\mathcal{G}$ and embedding $\boldsymbol{p}$, let $\hat{\mathcal{E}}$ be the edge set achieved by graph sparsification with threshold δ_a. The sparsified Laplacian matrix $\hat{\boldsymbol{L}}$ is ϵ-similar to $\boldsymbol{L}$ when $q_a\delta_a \leq \epsilon\|\boldsymbol{p}\|$, where q_a is the number of edges removed.*

The proof of Theorem 7.2 is detailed in Appendix B.2.

To derive more specific bounds of q_a, we need to examine the actual distribution of entry values. Regarding the assumption on edge distribution, we particularly investigate the scale-free graph, which is common in realistic large-scale graphs [282]. For input attribute values, we consider the Gaussian distribution, which is also commonly used for depicting the feature distribution of neural networks as an extension of the central limit theorem [283, 284]. With these two assumptions, the following theorem relates the threshold and approximation bound with sparsification:

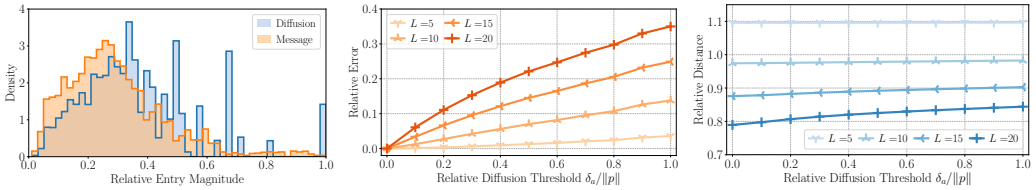
Theorem 7.3 (Bound for UNIFEWS). *Given a graph $\mathcal{G}$, embedding $\boldsymbol{p}$, and required edge sparsity η_a, the threshold δ_a can be decided by*

$$\delta_a = C(1 - \eta_a)^{-t}, \tag{7.7}$$

and the sparsified Laplacian $\hat{\boldsymbol{L}}$ is ϵ-similar to $\boldsymbol{L}$ with the approximation bound:

$$\epsilon = O\left(\eta_a(1 - \eta_a)^{-t}\right), \tag{7.8}$$

where C and t are positive constants. $C > 0$ is determined by the embedding values $\|\boldsymbol{p}\|$, and $0 < t < 1$ depicts the degree distribution.



(a) Distribution of Entries **(b)** Approximation Error **(c)** Over-smoothing Distance

FIGURE 7.3: Empirical evaluation on the SGC graph propagation and the effect of UNIFEWS on CORA dataset. **(a)** Distribution of entries in the diffusion matrix $\boldsymbol{T}$ and the message $\boldsymbol{\tau}_{(0)}$ at hop $l = 0$. **(b)** The relative error margin $\|\hat{\boldsymbol{P}}_{(L)} - \boldsymbol{P}_{(L)}\|_F / \|\boldsymbol{P}_{(L)}\|_F$ of approximation to the raw embedding against the strength of UNIFEWS and propagation hops. **(c)** The relative distance $\|\hat{\boldsymbol{P}}_{(L)} - \boldsymbol{P}^*\|_F / \|\boldsymbol{P}^*\|_F$ to the converged embedding against the strength of UNIFEWS and propagation hops.

The proof of Theorem 7.3 and specific derivations of C and t can be found in Appendix B.3.

The significance of Theorem 7.3 lies in bridging the sparsification technique and Laplacian smoothing with a specific precision bound for the first time to the best of our knowledge. Its implication is that, the adaptive sparsification by UNIFEWS for a single layer qualifies as a graph spectral sparsifier bounded by ϵ. Its pivotal parameter, i.e., the sparsification threshold, and its approximation bound can be determined once given the sparsity η_a. A sparsity closer to 1 results in a larger threshold value as well as a loose error guarantee.

Entry Distribution. An example on the real-word graph CORA is shown in Figure 7.3(a), which demonstrates that the distribution of node degree follows the power law, and the distribution of message values is correlated with the diffusion entries. Hence, message magnitude is effective in representing edge importance and determining entry removal. Setting entries in the diffusion matrix $\boldsymbol{T}$ to zero according to Eq. (7.6) implies removing the corresponding edges from the graph diffusion process. Consequently, messages with small magnitudes are prevented from propagating to neighboring nodes and composing the output embedding.

7.3.2 UNIFEWS for Graph Propagation

Thanks to the adaptive property of the entry-wise pruning scheme, we are able to further perform fine-grained and gradual sparsification across propagation layers. Intuitively, a message deemed of minor significance in the current propagation is unlikely to propagate

further and influence more distant nodes in subsequent layers. UNIFEWS hence offers progressively increasing sparsity throughout GNN layers.

As depicted in Figure 7.2(c), UNIFEWS iteratively applies sparsification to each layer, and the pruned diffusion matrix $\hat{\boldsymbol{T}}_{(l)}$ is inherited to the next layer for further edge removal. Denote the edge set corresponding to $\hat{\boldsymbol{T}}_{(l)}$ as $\hat{\mathcal{E}}_{(l)}$, there is $\hat{\mathcal{E}}_{(l)} \subseteq \hat{\mathcal{E}}_{(l-1)} \subseteq \cdots \subseteq \hat{\mathcal{E}}_{(0)} \subseteq \mathcal{E}$, which indicates a diminishing number of operations for deeper layers. Consequently, Eq. (7.5) is modified as the following to represent entry-level diffusion on the sparsified graph:

$$\hat{\boldsymbol{p}}_{(l+1)}[u] = \sum_{v \in \mathcal{N}(u)} \hat{\boldsymbol{T}}_{(l)}[u,v] \cdot \boldsymbol{p}_{(l)}[v] = \sum_{v \in \hat{\mathcal{N}}_{(l)}(u)} \tau_{(l)}[u,v] = \sum_{v \in \hat{\mathcal{N}}_{(l)}(u)} \boldsymbol{T}_{(l)}[u,v] \cdot \boldsymbol{p}_{(l)}[v], \quad (7.9)$$

where the neighborhood composed by remaining connections is $\hat{\mathcal{N}}_{(l)}(u) = \{v \mid (u,v) \in \hat{\mathcal{E}}_{(l)}\}$.

We illustrate the application of UNIFEWS on decoupled graph propagation in Algorithm 7.1 as highlighted components. Note that the nested loops starting from Line 5 are identical to the canonical graph propagation conducted by sparse-dense matrix multiplication in common GNNs. Compared to the normal propagation Line 11, it refrains the value update and prunes the corresponding edge for insignificant entries in Line 9. Hence, UNIFEWS can be implemented into the GNN computation to enhance efficiency without

ALGORITHM 7.1: UNIFEWS on Decoupled Propagation

Input: Graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, diffusion $\boldsymbol{T}_{(l)}$, attribute $\boldsymbol{X}$, propagation hop L, graph sparsification threshold δ_a

Output: Approximate embedding $\hat{\boldsymbol{P}}_{(L)}$

1 $\hat{\boldsymbol{P}}_{(0)} \leftarrow \boldsymbol{X}$, $\hat{\mathcal{E}}_{(-1)} \leftarrow \mathcal{E}$
2 **for** $l = 0$ to $L-1$ **do**
3 $\hat{\boldsymbol{P}}_{(l+1)} \leftarrow \hat{\boldsymbol{P}}_{(l)}$
4 $\hat{\mathcal{E}}_{(l)} \leftarrow \hat{\mathcal{E}}_{(l-1)}$, $\hat{\boldsymbol{T}}_{(l)} \leftarrow \boldsymbol{T}_{(l)}$
5 **for all** $u \in \mathcal{V}$ **do** ▷ *[matrix op]*
6 $\hat{\mathcal{N}}_{(l)}(u) \leftarrow \{v \mid (u,v) \in \hat{\mathcal{E}}_{(l)}, v \neq u\}$
7 **for all** $v \in \hat{\mathcal{N}}_{(l)}(u)$ **do**
8 **if** $|\hat{\boldsymbol{T}}_{(l)}[u,v]| < \delta_a / \|\hat{\boldsymbol{P}}_{(l)}[v]\|$ **then**
9 $\hat{\mathcal{E}}_{(l)} \leftarrow \hat{\mathcal{E}}_{(l)} \setminus \{(u,v)\}$, $\hat{\boldsymbol{T}}_{(l)}[u,v] \leftarrow 0$
10 **else**
11 $\hat{\boldsymbol{P}}_{(l+1)}[u] \leftarrow \hat{\boldsymbol{P}}_{(l+1)}[v] + \hat{\boldsymbol{T}}_{(l)}[u,v] \cdot \hat{\boldsymbol{P}}_{(l)}[v]$
12 **return** $\hat{\boldsymbol{P}}_{(L)}$

additional matrix-wise overhead. For multi-dimensional feature matrix, the pruning function Eq. (7.6) is performed by replacing $|\boldsymbol{p}[v]|$ with the specific norm $\|\boldsymbol{P}[v]\|$ across feature dimensions. Note that to maintain node identity, we adopt skip connections to each layer in the algorithm by initializing $\boldsymbol{P}_{(l+1)}$ explicitly with the value in the previous hop, in order to preserve at least one meaningful value for each node embedding in case all connected edges are pruned.

Moreover, the feature transformation stage in decoupled GNNs is feasible to the weight pruning described in the following section, which is similar to classical irregular pruning with magnitude-based thresholds for MLP networks [285, 286].

Approximation Bound. From the theoretical perspective, the error introduced by multi-hop propagation is more complicated than the single-layer case in Theorem 7.3, as it is composed of both the diffusion and embedding values in approximation. By exploiting the GNN graph smoothing process in Definition 7.1, we show that:

$$\|\hat{\boldsymbol{p}}_{(l+1)} - \boldsymbol{p}_{(l+1)}\| \leq \|\hat{\boldsymbol{p}}_{(l)} - \boldsymbol{p}_{(l)}\| + O(\epsilon) \cdot \|\hat{\boldsymbol{T}} - \boldsymbol{T}\|_2 \|\boldsymbol{p}_{(l)}\|.$$

Therefore, as long as the sparsification satisfies Theorem 7.3 for each hop, the whole process of pruning and accumulating the sparsified graph across multiple hops can be regarded as the approximate smoothing in Theorem 7.1, as stated in the following proposition:

Proposition 7.1 (Progressive UNIFEWS for graph propagation). *For an L-hop graph propagation under Algorithm 7.1, if the final sparsifier satisfies ϵ-similarity, then the overall process is an ϵ-approximation to the original graph smoothing problem.*

The proof of Proposition 7.1 is detailed in Appendix B.4.

An empirical evaluation of multi-hop UNIFEWS on CORA is shown in Figure 7.3(b), which validates that the approximation error is affected by the sparsification threshold δ_a as well as propagation hops L. Meanwhile, even under aggressive sparsification, the error is still adequately bounded to yield meaningful learning outcomes.

Impact on Over-smoothing. Additionally, we note that the "error" induced in Algorithm 7.1 is not necessarily harmful. Examining Eq. (7.1), excessive propagation of common GNNs can lead to a decline in performance known as the *over-smoothing issue*, where the graph regularization is dominant and meaningful node identity is lost

[149]. By eliminating a portion of graph diffusion, UNIFEWS effectively alleviates the over-smoothing issue.

We showcase the effect of alleviating over-smoothing in UNIFEWS by depicting the embedding difference to optimization convergence in Figure 7.3(c). With an increased number of hops, the output embedding tends towards an over-smoothed state. However, a stronger sparsification prevents the rapid smoothing and hereby contributes to better graph learning performance.

7.3.3 UNIFEWS for Iterative Update

Compared to decoupled designs, graph propagation and feature transformation in iterative GNN models are tightly integrated in each layer. Traditionally, this poses a challenge to GNN sparsification methods depicted in Figure 7.2(b), as specialized schemes are necessary for simultaneously modifying the two components without impairing the performance. On the contrary, our UNIFEWS approach takes advantage of this architecture for jointly sparsifying the model towards a win-win situation in graph learning.

The sparsification of iterative UNIFEWS for the entire message-passing process is presented in Algorithm 7.2, where the difference from the common scheme is also highlighted. Similarly, although presented as nested loops in the algorithm, the multiplication and sparsification are implemented as matrix operations. Its graph propagation stage is identical to Algorithm 7.1, except that the embedding $\hat{\boldsymbol{P}}_{(l)}$ is initialized by the previous representation $\hat{\boldsymbol{H}}_{(l)}$. Meanwhile, sparsification for weights is relatively straightforward compared to graph edges, since weight matrices are structured and their approximation is well-studied as network pruning [285, 286, 276]. Given the embedding of the current layer $\hat{\boldsymbol{P}}_{(l)}$, we similarly rewrite the iterative GNN update as:

$$\boldsymbol{H}_{(l+1)}[:,i] = \sigma\Big(\sum_{j=1}^{f} \boldsymbol{W}_{(l)}[j,i] \cdot \boldsymbol{P}_{(l)}[:,j]\Big), \tag{7.10}$$

where $\boldsymbol{H}_{(l+1)}[:,i]$ denotes the i-th column vector of all nodes in $\boldsymbol{H}_{(l+1)}$, and the weight entry $\boldsymbol{W}_{(l)}[j,i]$ symbolizes the neuron mapping the j-th embedding feature to the i-th representation feature. UNIFEWS sparsification on the weight matrix can thus be presented in the entry-wise manner following weight threshold δ_w:

$$\hat{\boldsymbol{W}}[j,i] = \mathrm{thr}_{\delta'_w}\big(\boldsymbol{W}[j,i]\big) \cdot \boldsymbol{W}[j,i], \;\; \delta'_w = \delta_w / \|\boldsymbol{P}[:,j]\|. \tag{7.11}$$

Approximation Bound. To derive the precision bound for UNIFEWS on iterative models, we investigate the difference of layer representation:

$$\hat{\boldsymbol{H}}_{(l+1)} - \boldsymbol{H}_{(l+1)} = (\hat{\boldsymbol{P}}_{(l)} - \boldsymbol{P}_{(l)})\boldsymbol{W}_{(l)} + \hat{\boldsymbol{P}}_{(l)}(\hat{\boldsymbol{W}}_{(l)} - \boldsymbol{W}_{(l)}).$$

The first term is similar to graph propagation in Proposition 7.1, while the second term can be bounded by the weight pruning scheme regarding δ_w. Hence, we are able to show that the representation under layer-progressive UNIFEWS for iterative networks containing both graph and weight operations satisfies the following proposition:

Proposition 7.2 (Progressive UNIFEWS for iterative update). *For an L-round iterative update under Algorithm 7.2, the approximation error on output $\|\hat{\boldsymbol{H}}_{(L)} - \boldsymbol{H}_{(L)}\|_F$ is bounded by $O(\epsilon\|\boldsymbol{H}_{(L)}\|_F + \delta_w)$.*

The detailed interpretation of Proposition 7.2 is discussed in Appendix B.5.

In brief, the approximation of layer representation is jointly bounded by factors depicting graph and weight sparsification processes. Hence, the unified pruning produces an advantageous approximation of the learned representations across GNN layers, which completes our framework for characterizing the approximation bound of UNIFEWS for general GNN schemes by examining the graph smoothing optimization throughout model learning.

ALGORITHM 7.2: UNIFEWS on Iterative GNN

Input: Graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, diffusion $\boldsymbol{T}_{(l)}$, attribute $\boldsymbol{X}$, network layer L, graph sparsification threshold δ_a, weight sparsification threshold δ_w

Output: Approximate representation $\hat{\boldsymbol{H}}_{(L)}$

$\hat{\boldsymbol{H}}_{(0)} \leftarrow \boldsymbol{X}$, $\hat{\mathcal{E}}_{(-1)} \leftarrow \mathcal{E}$
for $l = 0$ to $L - 1$ **do**
  $\hat{\boldsymbol{H}}_{(l+1)} \leftarrow \boldsymbol{0}$, $\hat{\boldsymbol{P}}_{(l)} \leftarrow \hat{\boldsymbol{H}}_{(l)}$
  Acquire sparsified $\hat{\boldsymbol{P}}_{(l)}, \hat{\mathcal{E}}_{(l)}, \hat{\boldsymbol{T}}_{(l)}$ as in Algorithm 7.1
  for $i \leftarrow 1$ to f **do** ▷ *[matrix op]*
    for $j \leftarrow 1$ to f and $\hat{\boldsymbol{W}}_{(l)}[j,i] \neq 0$ **do**
      if $|\hat{\boldsymbol{W}}_{(l)}[j,i]| < \delta_w / \|\hat{\boldsymbol{P}}_{(l)}[:,j]\|$ **then**
        $\hat{\boldsymbol{W}}_{(l)}[j,i] \leftarrow 0$
      else
        $\hat{\boldsymbol{H}}_{(l+1)}[:,i] \leftarrow \hat{\boldsymbol{H}}_{(l+1)}[:,i] + \hat{\boldsymbol{W}}_{(l)}[j,i] \cdot \hat{\boldsymbol{P}}_{(l)}[j]$
  $\hat{\boldsymbol{H}}_{(l+1)} \leftarrow \sigma\big(\hat{\boldsymbol{H}}_{(l+1)}\big)$
return $\hat{\boldsymbol{H}}_{(L)}$

Complexity Analysis. As shown in Algorithms 7.1 and 7.2, entry-level operations can be naturally inserted into the computation *without* additional overhead. For graph propagation, when the graph sparsity is $\eta_a = q_a/m$, where q_a is the number of removed edges, the computation complexity for propagation in each layer is reduced from $O(m)$ to $O\big((1-\eta_a)m\big)$. In particular, for iterative models, this applies to both time and memory overhead. For weight pruning regarding matrices $\boldsymbol{P}, \boldsymbol{H} \in \mathbb{R}^{n\times f}$ and $\boldsymbol{W} \in \mathbb{R}^{f\times f}$, let the pruning ratio of weight matrix be η_w. The sparsification scheme at least reduces complexity to $O\big((1-\eta_w)nf^2\big)$. The favorable merit of joint graph and weight sparsification by UNIFEWS is that, both the propagation result $\boldsymbol{P}$ and the weight multiplication product $\boldsymbol{H}$ enjoy sparsity from the previous input alternatively. In fact, as proven by [157], the scale of reduced computational operation can be advanced to $(1-\eta_w)^2$ for certain distribution.

7.4 Experimental Evaluation

We implement UNIFEWS for sparsifying various GNN architectures and evaluate its performance against strong competitors in the scope of GNN graph and network simplification. We first respectively highlight the accuracy and efficiency improvement of our approach. Further discussions are provide regarding effects of the joint pruning and key parameters.

7.4.1 Experiment Setting

Dataset. We adopt 3 representative small-scale datasets: CORA, CITESEER, and PUBMED as well as 4 large-scale ones: PHYSICS, ARXIV, PRODUCTS, PAPERS100M. Statistics of the datasets can be found in Table 7.1, where we follow the common settings including the graph preparing and training set splitting pipeline in these benchmarks. In the table, we incorporate self-loop edges and count undirected edges twice to better reflect the propagation overhead.

Metrics. We uniformly utilize transductive node classification accuracy as the evaluation metric of model prediction. To precisely observe and compare the number of operations in GNN learning, the efficiency is assessed by computation time and floating-point operations (FLOPs), and 1FLOPs $\approx$ 2MACs. The graph and weight sparsities are calculated

as portions of pruned entries compared to the original matrices. For layer-dependent methods, the average sparsity across all layers is used. Evaluation are conducted on a server with 32 Intel Xeon CPUs (2.4GHz), an Nvidia A30 (24GB memory) GPU, and 512GB RAM.

Backbone Models. Since UNIFEWS is adaptable to a range of GNN architectures, we select classic GNNs as our backbones, i.e., subject architectures of sparsification methods, due to the consideration that most baselines are only applicable to a couple of classic architectures such as GCN and GAT. In contrast, we demonstrate the generality of UNIFEWS on more backbones including GraphSAGE and GCNII.

We divide them into two categories based on iterative and decoupled GNN designs. Iterative backbone models include:

- GCN [25] is the representative message-passing GNN with a diffusion matrix $\boldsymbol{T} = \tilde{\boldsymbol{A}}$ across all layers.
- GAT [287] learns a variable diffusion $\boldsymbol{T}$ for each layer by multi-head attention. We set the number of heads to 8 for hidden layers.
- GraphSAGE [27] performs more specialized message-passing aggregation and is suitable for large graphs.
- GCNII [67] features residual connections and identity mapping. We use $L = 32$ for demonstrating the effect of sparsification on deep GNNs.

For decoupled designs, we implement the pre-propagation version [112] of the following models:

TABLE 7.1: Statistics of graph datasets. f and N_c are the numbers of input attributes and label classes, respectively. Numbers in the "Split" column are percentages of nodes in training/validation/testing set w.r.t. labeled nodes, respectively

Dataset	Nodes n	Edges m	Features f	Classes N_c	Split
CORA [25]	2,485	12,623	1433	7	50/25/25
CITESEER [25]	3,327	9,228	3703	6	50/25/25
PUBMED [25]	19,717	88,648	500	3	50/25/25
PHYSICS [255]	34,493	495,924	8415	5	50/25/25
ARXIV [43]	169,343	2,315,598	128	40	54/18/29
PRODUCTS [43]	2,400,608	123,718,024	100	47	08/02/90
PAPERS100M [43]	111,059,956	3,228,124,712	128	172	78/08/14

- SGC [125] corresponds to GCN in spectral smoothing, but computes the propagation $\boldsymbol{P} = \tilde{\boldsymbol{A}}^L \boldsymbol{X}$ separately in advance.
- APPNP [123] accumulates and propagates the embedding with a decay factor $\boldsymbol{P} = \sum_{l=0}^{L-1} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l \boldsymbol{X}$.

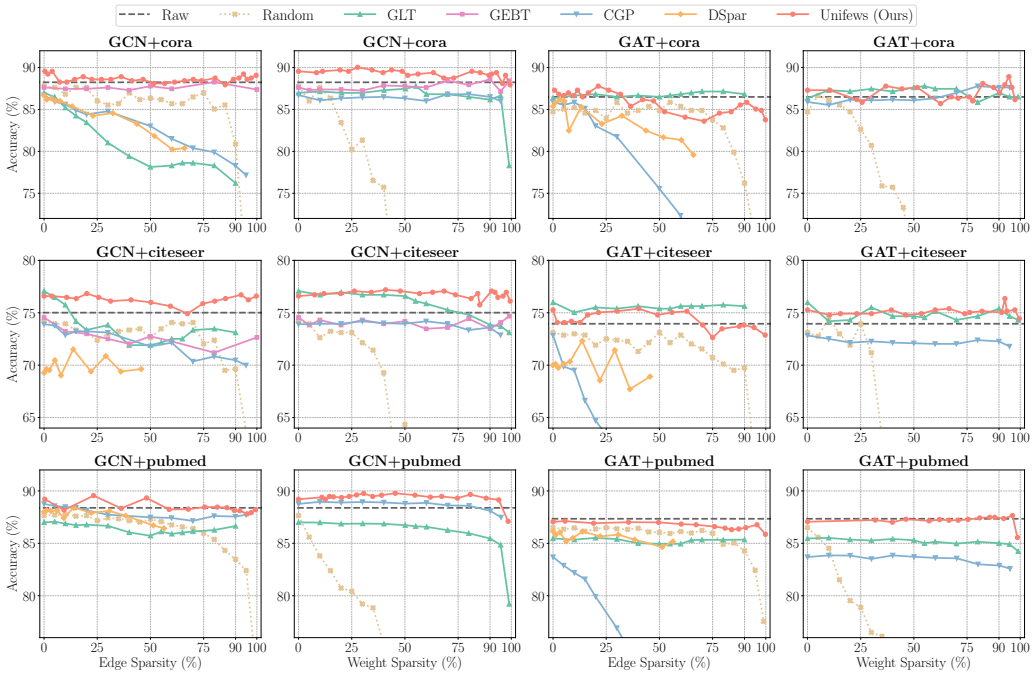
Baseline Methods. Our selection of baselines is dependent on their applicable backbones. We mostly utilize their public source codes and retain original implementations.

For iterative models, we consider state of the arts in graph and joint compression, which are methods with the capability to produced sparsified graphs with smaller edge sets for both GNN training and inference:

- GLT [164] proposes concurrently sparsifying GNN structure by gradient-based masking on both adjacency and weights.
- GEBT [162] gradually discovers the small model during training. Its implementation is limited to the GCN backbone.
- CGP [159] iteratively prunes and regrows graph connections while exploiting irregular compression on weights.
- DSpar [147] employs one-shot graph sparsification according to a degree-based metric, which implies an upper bound for pruning rate. It does not perform weight pruning.
- Random (RAN) refers to the sparsification method that removes entries with uniform probability based on the specified sparsity.

For the decoupled scheme, we mainly evaluate the graph sparsification. There are two propagation personalization techniques, both only available for the SGC backbone:

- NDLS [150] determines the hop number by calculating the distance to convergence, which produces a customized propagation.
- NIGCN [151] offers better scalability by performing degree-based estimation on the node-wise propagation depth.



(a) GCN Edge **(b)** GCN Weight **(c)** GAT Edge **(d)** GAT Weight

FIGURE 7.4: Accuracy results of UNIFEWS and baseline methods applied on iterative backbone models GCN and GAT over three small datasets. Columns of "Edge" and "Weight Sparsity" present the average results of models with solely edge and weight sparsification, respectively. Black dashed lines are the performance of backbone models with full graph and weights.

The graph and weight sparsities are calculated as portions of pruned entries compared to the original matrices. For layer-dependent methods, the average sparsity across all layers is used. For UNIFEWS, the pruning thresholds δ_a and δ_w are determined by Theorem 7.3. Note that as Algorithm 7.1 naturally preserve diagonal elements in $\hat{T}$, a GCN layer with $\eta_a = 100\%$ UNIFEWS pruning is equivalent to an MLP layer.

Hyperparameters. We commonly utilize graph normalization $\rho = 0.5$, model layer depth $L = 2$, and layer width $F_{hidden} = 512$. For decoupled models, the number of propagation hops is 20. We employ full-batch and mini-batch training for iterative and decoupled methods, respectively. The total number of training epochs is uniformly 200 for all models, including pre-training or fine-tuning process in applicable methods. The batch size is 512 for small datasets and 16384 for large ones. We tune the edge and weight sparsity of evaluated models across their entire available ranges, and the pruning ratio of UNIFEWS is controlled by adjusting δ_a and δ_w.

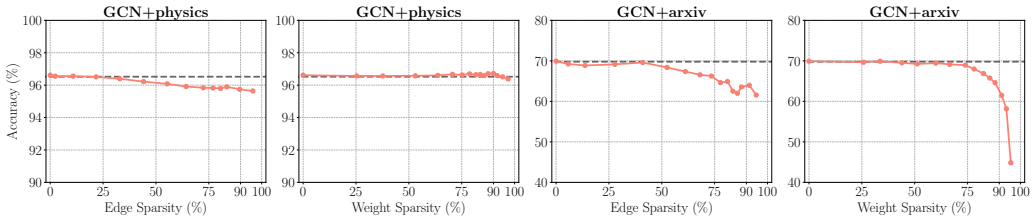
7.4.2 Performance Comparison

We first separately apply only one part of sparsification, i.e., either edge or weight pruning, for better comparison. Figure 7.4 presents accuracy results of iterative backbones and compression methods over representative datasets. GEBT encounters out of memory error on PUBMED. For larger graphs, most of the iterative baselines suffer from the out of memory error due to the expense of trainable adjacency matrix design and full-graph training process.

Edge Sparsification. For the *iterative* architecture in Figures 7.4(a) and 7.4(c), UNIFEWS outperforms state-of-the-art graph and joint compression approaches in most backbone-dataset combinations. Typically, for relatively small ratios $\eta_a < 80\%$, models with UNIFEWS pruning achieve comparable or exceeding accuracy, aligning with our approximation analysis. For higher sparsity, UNIFEWS benefits from the skip connection design, which carries essential information of node identity and therefore retains accuracy no worse than the trivial transformation. On the contrary, most of the competitors experience significant accuracy drop on one or more datasets. CGP and DSpar exhibit poor utility on the GAT backbone since its variable connections are more vulnerable to removal. Additionally, the comparison with random sparsification indicates that, the entry-wise scheme is particularly effective for small ratios, as the randomized pruning even surpasses dedicated methods for some circumstances. For high edge sparsity, the UNIFEWS preservation of dominant edges and identity mapping is critical to accuracy, which validates the advantage of our design.

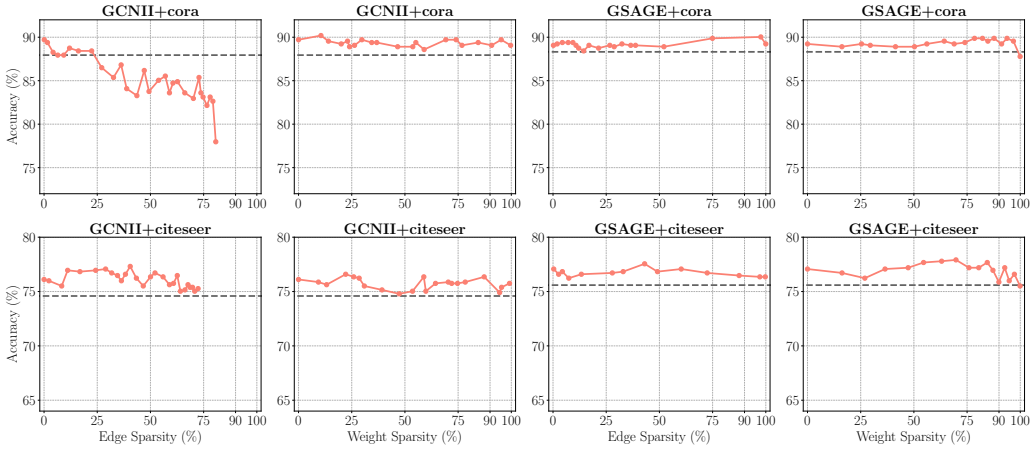
Additionally, we present the experimental results of UNIFEWS on larger datasets and on more backbone architectures in Figure 7.5 and Figure 7.6, respectively. No baseline method is available for these settings. It can be observed that UNIFEWS constantly achieves satisfying performance in a large range of sparsity. The effect of graph sparsification on GCNII is relatively unstable, likely because the deeper layers amplify the approximation error.

Similarly, for *decoupled* propagation in Figure 7.8, UNIFEWS is able to preserve remarkable accuracy and outperform personalized propagation methods. On CITESEER, it raises SGC accuracy by 2% through mitigating the over-smoothing issue. We hence conclude that, compared to heuristic pruning schemes, UNIFEWS successfully removes unimportant graph edges without sacrificing efficacy, thanks to its fine-grained and adaptive scheme.



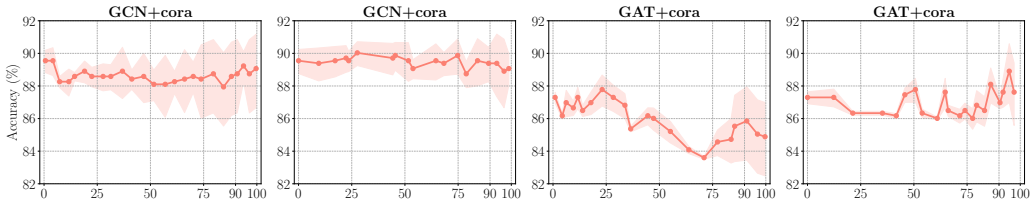
(a) GCN Edge **(b)** GCN Weight **(c)** GCN Edge **(d)** GCN Weight

FIGURE 7.5: Accuracy of *iterative* UNIFEWS over medium-scale PHYSICS and ARXIV, while all baseline methods meet OOM.



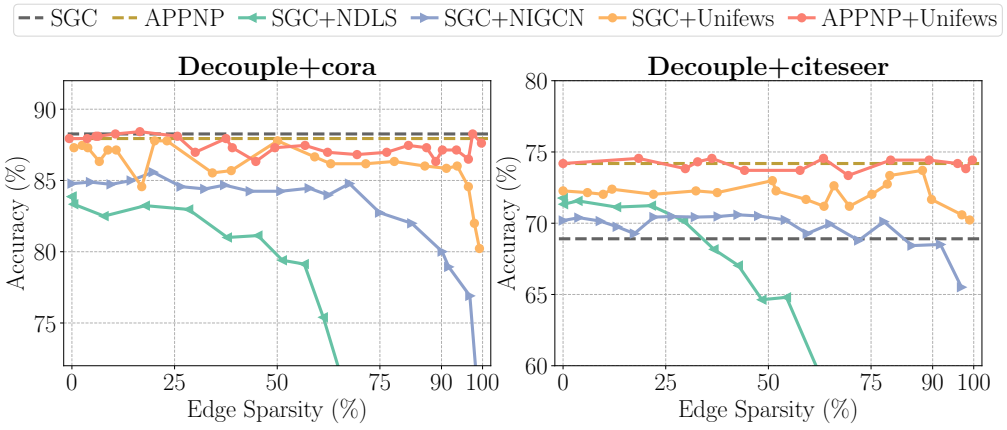
(a) GCNII Edge **(b)** GCNII Weight **(c)** GraphSAGE Edge **(d)** GraphSAGE Weight

FIGURE 7.6: Accuracy of *iterative* UNIFEWS for GCNII and GraphSAGE backbones, while no baseline method is applicable.



(a) GCN Edge **(b)** GCN Weight **(c)** GCN Edge **(d)** GCN Weight

FIGURE 7.7: Accuracy and variance of *iterative* UNIFEWS over CORA.



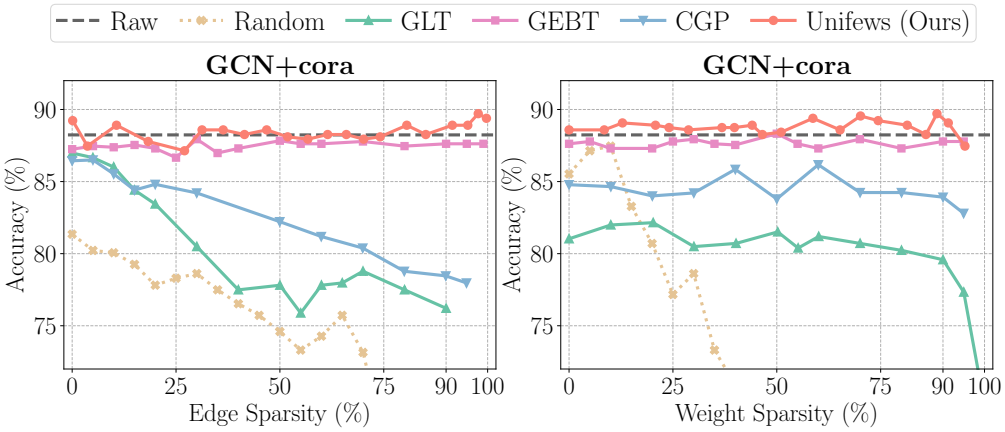
(a) CORA **(b)** CITESEER

FIGURE 7.8: Accuracy results of UNIFEWS and baseline methods applying graph sparsification on decoupled models over CORA and CITESEER. UNIFEWS is employed with solely edge removal and $\eta_w = 0\%$. Black and brown dashed lines are the performance of backbone SGC and APPNP models with full graph and weights, respectively.

Variance. The variance of UNIFEWS is particularly shown in Figure 7.7 corresponding to Figure 7.4. Typically, the variance of iterative UNIFEWS models are between the range of 1-3%, which is slightly larger than the backbone model due to the perturbation of edge and weight sparsification.

Weight Sparsification. Figures 7.4(b) and 7.4(d) display the efficacy of network pruning. For all combinations of models and graphs, UNIFEWS achieves top-tier accuracy with a wide range of weight sparsity. For small ratios, the model produces up to 3% accuracy gain over the bare GNN, resembling the benefit of neural network compression [288]. Most evaluated baselines also maintain low error rate compared to their performance in graph pruning, indicating that the GNN weights are relatively redundant and are suitable for substantial compression. Thanks to the redundancy in GNN architecture, baselines including GEBT and CGP present strong performance in certain cases compared to edge sparsification. Random removal fails for high weight sparsity, which implies that the magnitude-based scheme is the key to maintain model effectiveness.

Joint Sparsification. For unified pruning on graph edges and network weights, accuracy comparison is provided by Figure 7.9 with applicable methods. It is noticeable that UNIFEWS retains comparable or better accuracy than the backbone GCN with up to 3% improvement. Most baseline methods only obtain suboptimal accuracy especially for weight pruning, which is affected by their comparatively poor graph sparsification. While GEBT mostly retains effectiveness on CORA, it is highly limited by the specific



(a) Edge Pruning with $\eta_w = 30\%$ **(b)** Weight Pruning with $\eta_a = 30\%$

FIGURE 7.9: Accuracy results of joint UNIFEWS and baseline methods on GCN over CORA. Columns of "Edge" and "Weight Sparsity" respectively present the results of models with varying edge and weight sparsification, while fixing the other sparsity at 30%. Black dashed lines are the performance of backbone GCN with full graph and weights.

architecture and high training overhead. The evaluation further highlights the advantage of UNIFEWS in considering the two GNN operation stages in a unified manner.

7.4.3 Efficiency Analysis

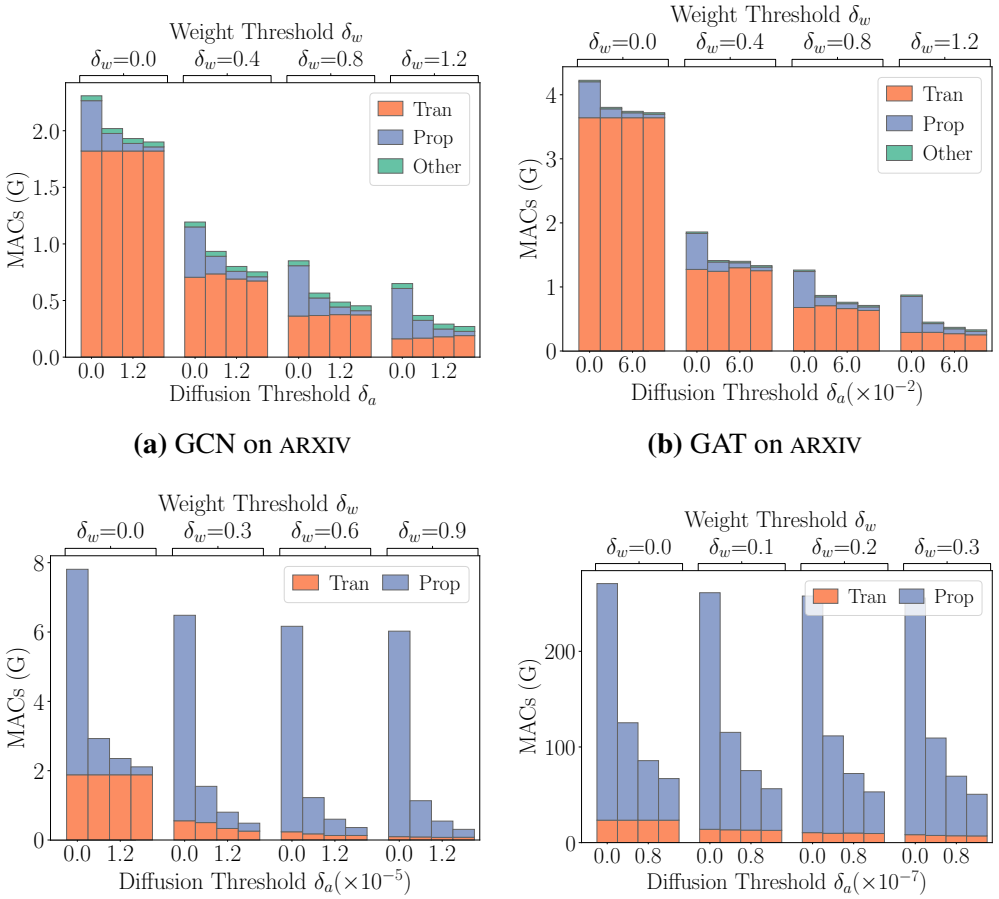
FLOPs. As graph computation is the primary bottleneck for large-scale tasks, we particularly study the efficiency improvement utilizing decoupled models in Table 7.2. Evaluation implies that, UNIFEWS is effective in producing operational reduction proportional to the sparsity, i.e., saving 50% computation FLOPs. It also achieves higher compression such as for APPNP over PRODUCTS, by recognizing more unnecessary edges than required. Contrarily, NDLS suffers from out of memory error on large datasets due to its complex calculation and implementation, while NIGCN does not guarantee decreased computation because of its expansionary propagation design. Table 7.2 also presents the transformation FLOPs for reference, where the sparsified embedding of UNIFEWS also benefits the downstream computation. We remark that, although the FLOPs value for transformation stage appears to be on the same level with propagation, in practice it can be efficiently processed utilizing parallel computation on devices such as GPU [97, 49]. Hence the GNN scalability bottleneck, especially on large datasets, is still the graph propagation. In this context, the ability of UNIFEWS in reducing propagation operations is of unique importance.

Runtime. Within each method in Table 7.2, the propagation time is correlated with its FLOPs, and escalates rapidly with the data size due to the nature of graph operations. Comparison across methods demonstrates that UNIFEWS excels in efficiently conducting graph propagation with regard to both FLOPs and execution time. It realizes significant acceleration on large graphs by favorably reducing the graph scale across all layers, which benefits the system workload such as better entry-level access. The PAPERS100M result highlights the superiority of UNIFEWS with $85-100\times$ improvement over the backbone and $9\times$ speed-up over NIGCN.

Operation Breakdown. To specifically evaluate the efficiency enhancement, in Figure 7.10, we separately assess the FLOPs related to graph propagation and network transformation for different backbone models and datasets under UNIFEWS sparsification. For better presentation, model width is set to 64 in this experiment. Figures 7.10(a)

TABLE 7.2: Evaluation results of $\eta_a = 50\%$ graph sparsification on decoupled models. UNIFEWS is applicable to both SGC and APPNP backbones, while two baseline methods are only available on the former one. "Acc" and "Time" are the prediction accuracy (%) and propagation time (in seconds), respectively. "FLOPs" separately presents the operational complexity (in GMACs) for propagation and transformation on all nodes. "OOM" stands for out of memory error. "Improvement" is the comparison between UNIFEWS and the corresponding backbone model, where improved accuracy is marked in bold fonts.

Dataset	**CORA**			**CITESEER**			**PUBMED**		
	Acc	Time	FLOPs	Acc	Time	FLOPs	Acc	Time	FLOPs
SGC	85.8	0.13	0.36+2.2	67.7	0.44	0.68+2.8	83.0	0.36	0.89+2.3
+NDLS	80.3	1362	0.19+2.7	64.7	1940	0.33+4.3	77.0	4717	0.42+2.0
+NIGCN	84.2	0.45	0.22+3.0	70.4	0.47	0.28+2.1	85.0	9.2	0.44+2.0
+UNIFEWS (Ours)	**86.0**	0.10	0.18+1.2	**73.0**	0.26	0.35+1.7	83.0	0.24	0.47+2.0
Improvement	0.2	1.3×	2.0×, 1.9×	5.3	1.7×	2.0×, 1.7×	0.0	1.5×	1.9×, 1.2×
APPNP	86.2	0.15	0.36+2.2	71.6	0.43	0.68+2.8	87.6	0.33	0.89+2.3
+UNIFEWS (Ours)	**86.5**	0.08	0.18+1.8	**73.7**	0.21	0.31+2.3	**88.0**	0.26	0.43+1.8
Improvement	0.4	1.8×	2.0×, 1.2×	2.1	2.0×	2.2×, 1.2×	0.4	1.3×	2.1×, 1.3×
Dataset	**ARXIV**			**PRODUCTS**			**PAPERS100M**		
	Acc	Time	FLOPs	Acc	Time	FLOPs	Acc	Time	FLOPs
SGC	68.8	3.9	5.9+15.0	79.1	289.6	247.4+434.4	63.3	19212	17.7+253.6
+NDLS		(OOM)			(OOM)			(OOM)	
+NIGCN	63.7	87.6	15.6+14.7	77.9	1026	137.2+182.0	53.7	1770	110.7+238.6
+UNIFEWS (Ours)	**69.4**	1.5	3.0+13.3	78.5	203.1	124.0+186.9	63.1	192.4	5.3+143.8
Improvement	0.6	2.6×	2.0×, 1.1×	-0.5	1.4×	2.0×, 2.3×	-0.2	99.8×	3.3×, 1.8×
APPNP	64.8	2.6	5.9+20.9	72.5	248.5	247.4+269.4	60.9	15305	17.7+247.7
+UNIFEWS (Ours)	**65.0**	0.93	3.0+15.0	**76.9**	58.5	31.7+186.9	**62.8**	178.5	8.9+241.8
Improvement	0.2	2.8×	1.9×, 1.4×	4.5	4.2×	7.8×, 1.4×	1.9	85.7×	2.0×, 1.0×



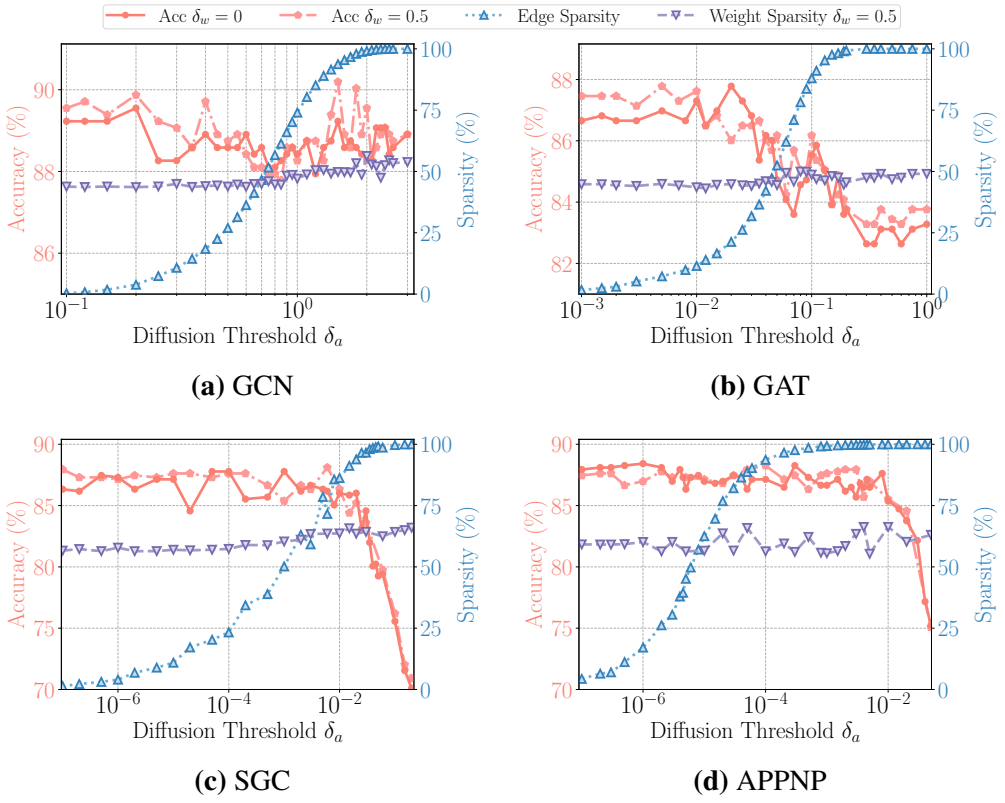
(a) GCN on ARXIV **(b)** GAT on ARXIV

(c) SGC on ARXIV **(d)** SGC on PRODUCTS

FIGURE 7.10: Composition of full-graph inference FLOPs with respect to propagation and transformation operations, under varying UNIFEWS edge and weight thresholds. Iterative models: **(a)** GCN and **(b)** GAT over ARXIV. Decoupled models: SGC over **(c)** ARXIV and **(d)** PRODUCTS datasets. For better presentation, model width is set to 64 in this experiment.

and 7.10(b) imply that the majority of computational overhead of *iterative models* is network transformation, even on graphs as large as ARXIV. Consequently, weight compression is essential for reducing GNN operations.

On the other hand, graph propagation becomes the bottleneck in *decoupled designs*, and is increasingly significant on graphs with greater scales. This is because of the larger number of propagation hops $L = 20$ for these structures. In this case, UNIFEWS is effective in saving computational cost by simplifying the propagation and bypassing unnecessary operations, with over 20× and 5× joint reduction on ARXIV and PRODUCTS, respectively. We also discover the benefit of joint pruning that a higher threshold results in smaller FLOPs of both operations in Figures 7.10(c) and 7.10(d), which signifies the win-win situation brought by increased sparsity. By combining these two sparsifications, we



(a) GCN **(b)** GAT

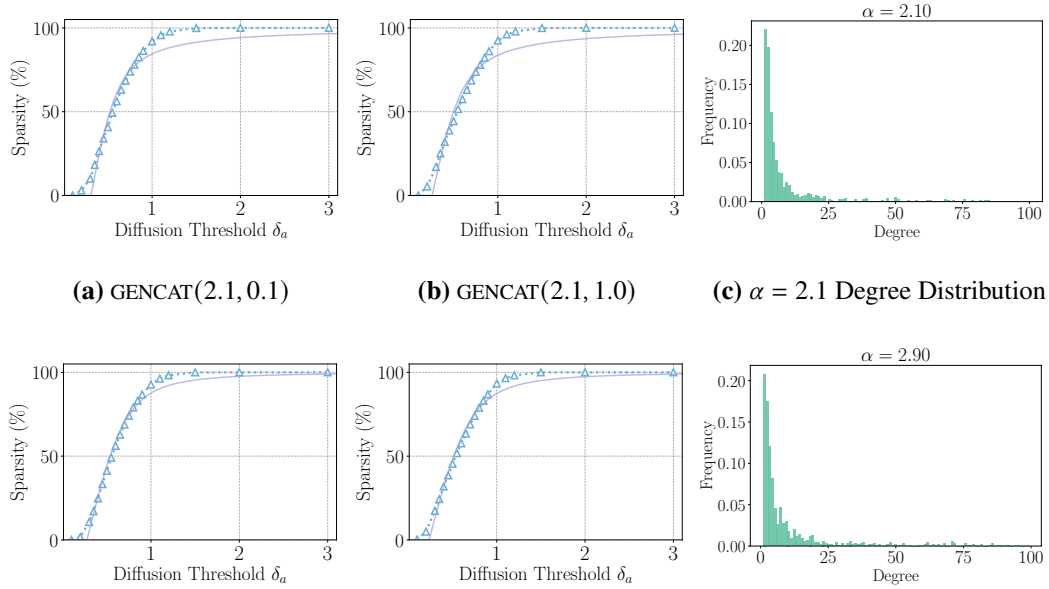
(c) SGC **(d)** APPNP

FIGURE 7.11: Sensitivity of joint sparsification thresholds on four models over CORA. We respectively set the weight ratio δ_w to 0 and 0.5, then evaluate the accuracy, edge sparsity, and weight sparsity of the model. Note that the x-axis representing diffusion threshold is on a logarithmic scale.

summarize that the unified scheme of UNIFEWS is capable of mitigating the computational overhead for both propagation- and transformation-heavy tasks.

7.4.4 Effect of Joint Simplification

Thresholds. The thresholds controlling diffusion and weight sparsification are pivotal to UNIFEWS compression, affecting both efficacy and efficiency of the produced model. Figure 7.11 presents the changes of inference accuracy and dual sparsity of GNN models under graph and joint sparsification with varying adjacency thresholds δ_a. Accuracy in the plot follows the conclusion in previous evaluation, that it only degrades above extreme sparsity $\eta_a > 95\%$. The experiment reveals that, the relation between edge sparsity and adjacency threshold aligns with Theorem 7.3, and decoupled models typically require a larger range of threshold to traverse the sparsity range, which is because of the wider distribution of their entry values throughout the deeper propagation. Interestingly, the



(a) GENCAT(2.1, 0.1) **(b)** GENCAT(2.1, 1.0) **(c)** α = 2.1 Degree Distribution

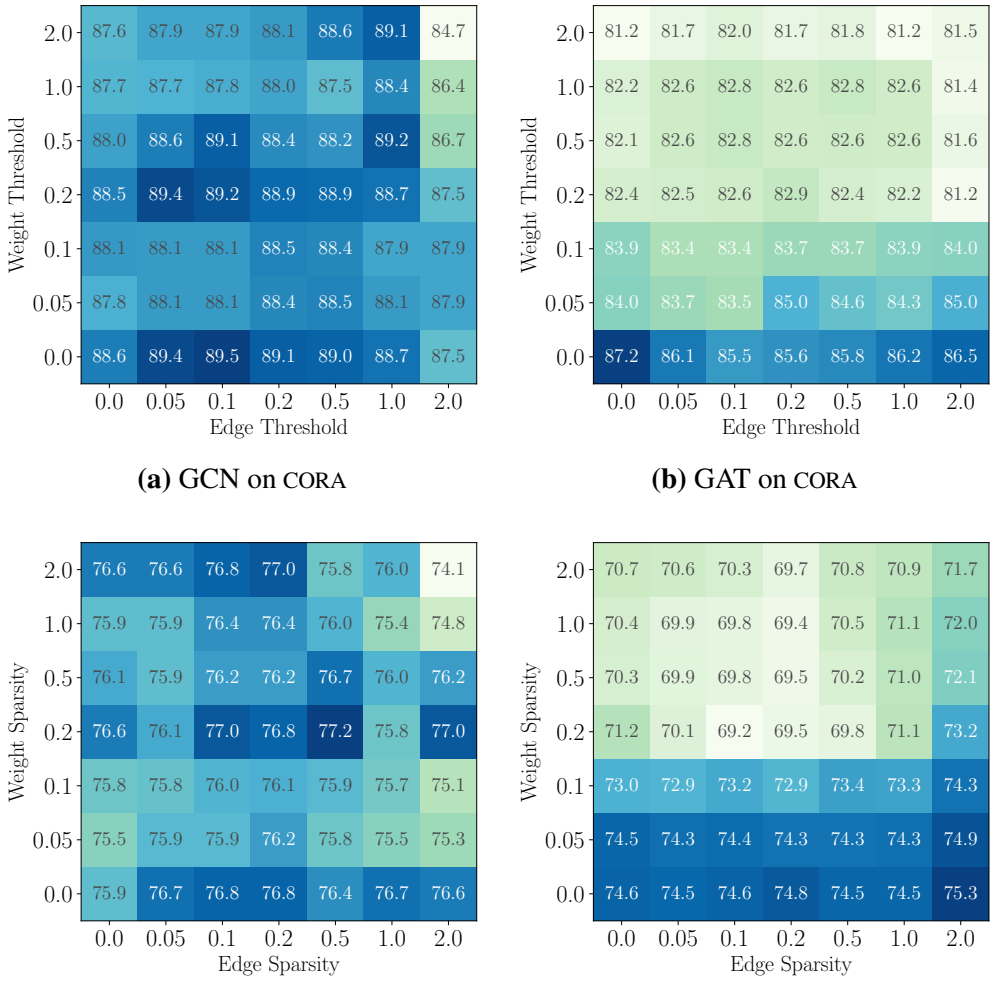
(d) GENCAT(2.9, 0.1) **(e)** GENCAT(2.9, 1.0) **(f)** α = 2.9 Degree Distribution

FIGURE 7.12: Relationship between edge threshold δ_a and sparsity η_a on synthetic graphs generated by GENCAT. We also present the degree distribution of nodes in different GENCAT graphs controlled by α to illustrate the power law.

weight sparsity when $\delta_w = 0.5$ also increases under high edge pruning ratios. The pattern is also observed in the reverse case. We deduce that the reciprocal enhancement in graph and model sparsity is brought by the unified sparsification of UNIFEWS, which conforms to our analysis as well as previous studies [157].

Thresholds on Synthetic Graphs. We further utilize GENCAT [289, 290] to generate synthetic graphs with randomized connections and features by varying its parameters. GENCAT(α, σ) is the state-of-the-art graph generation approach that allows for configuring the edge connections and node attributes synthesis by specifying their distributions, where α is the coefficient of edge power law distribution $P(d) = d^{-\alpha}$, and σ is the deviation of attribute Gaussian distribution $p[v] \sim N(0, \sigma^2)$, both sharing the definitions in our paper. The remaining inputs of GENCAT, including label distribution and attribute-label correlation, are set by mimicking the statistics from CORA. We utilize two groups of α and σ values to generate 4 graphs with different edge and attribute distributions and evaluate UNIFEWS.

Then, we evaluate the relationship between the edge threshold δ_a and the edge sparsity η_a following the settings of Figure 7.11. The results are available in Figure 7.12. As an overview, the pattern is similar to the one in Figure 7.11, while the constants are



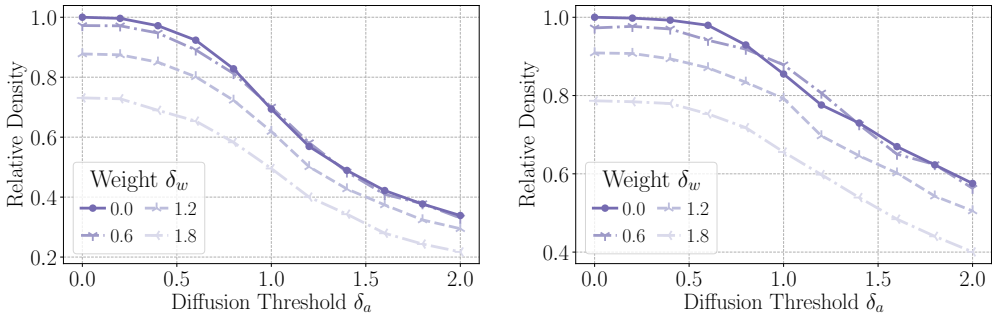
(a) GCN on CORA **(b)** GAT on CORA

(c) GCN on CITESEER **(d)** GAT on CITESEER

FIGURE 7.13: Accuracy of varying joint sparsification thresholds on GCN and GAT over CORA and CITESEER.

effectively affected by the changes of edge and feature distribution in the GENCAT graph. For δ_a not too close to 0, the relation with η_a largely follows Theorem 7.3.

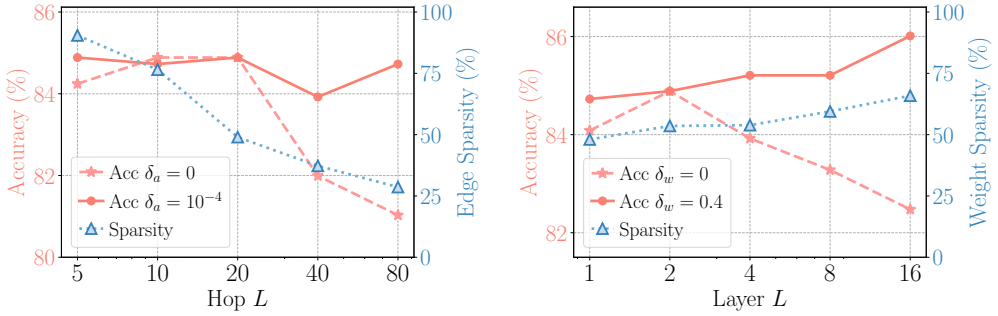
Effectiveness. For effectiveness, the impact of jointly changing δ_a and δ_w is displayed in Figure 7.13. Intuitively, GCN is more resilient to UNIFEWS sparsification, considering its relatively high redundancy of the wide distribution of entry values. In comparison, GAT is more sensitive to weight removal, that beyond a certain threshold δ_w, its accuracy drops significantly. This observation suggests that the value of learnable attention weights in GAT are highly concentrated, and selecting an appropriate sparsity is critical to its performance.



(a) Message $\boldsymbol{\tau}_{(0)}$ **(b)** Representation $\boldsymbol{H}_{(1)}$

FIGURE 7.14: Density of intermediate results during joint edge and weight sparsification at varying threshold values, evaluated by the ratio of matrix norm to the unpruned one on a 2-layer GCN over CORA. **(a)** Density of edge-wise message $\|\hat{\boldsymbol{\tau}}_{(0)}\|/\|\boldsymbol{\tau}_{(0)}\|$ relative to the raw one in the first layer $l = 0$. **(b)** Density of node-wise representation $\|\hat{\boldsymbol{H}}_{(1)}\|_F/\|\boldsymbol{H}_{(1)}\|_F$ relative to the raw one output by the first layer.

Sparsity. Additionally, we investigate the entry-wise sparsity of specific intermediate results in the process of GNN computation. Figure 7.14 displays the relative density of the edge message and node representation matrices, respectively. It can be clearly observed that the entry sparsity is enhanced with the increase of both two pruning ratios, signifying our theoretical analysis that UNIFEWS not only directly shrinks edges and weights, but promotes the sparsity of the product matrix as well. It also supports our claim that UNIFEWS is superior in employing dual sparsification where the two stages benefit each other alternatively and enjoy a win-win situation brought by the increased sparsity.



(a) Propagation Hop **(b)** Network Layer

FIGURE 7.15: Sensitivity of propagation and network transformation layers evaluated on SGC over CORA. **(a)** Impact on prediction accuracy and average edge sparsity of varying numbers of propagation hops. **(b)** Impact on prediction accuracy and average weight sparsity of varying numbers of network layers. Note that the x-axis representing layers is on a logarithmic scale.

7.4.5 Effect of Hyperparameters

Propagation and Network Layer. Since UNIFEWS adopts layer-dependent graphs with incremental sparsity for GNN propagation, we additionally evaluate its performance on models with varying layers. Figure 7.15(a) is the result of iteratively applying UNIFEWS edge sparsification to SGC with different propagation depths. The backbone model without pruning $\delta_a = 0$ typically suffers from the over-smoothing issue under large hop numbers. On the contrary, as elaborated in Section 7.3.2, UNIFEWS is powerful for identifying and eliminating unimportant propagations, especially for larger hops, and thereby prevents information loss. With respect to architectural compression Figure 7.15(b), it is noticeable that UNIFEWS promotes model performance and average weight sparsity at the same time for deeper layers, effectively reducing network redundancy.

7.5 Summary and Discussion

In this work, we present UNIFEWS, an entry-wise GNN sparsification with a unified framework for graph edges and model weights. UNIFEWS adaptively simplifies both graph propagation and feature transformation throughout the GNN pipeline and remarkably improves efficiency with little expense. In theory, we showcase that the layer-dependent UNIFEWS provides an effective estimation on the graph learning process with a close optimization objective, which is favorable for multi-layer GNN updates. Comprehensive experiments underscore the superiority of UNIFEWS in terms of efficacy and efficiency, including comparable or improved accuracy, $90-95\%$ operational reduction, and up to $100\times$ faster computation.

One potential limitation and future direction lies in the consideration of graph heterophily, as the current strategy prunes insignificant messages. However, under heterophily, where connected nodes have dissimilar labels, messages with large magnitudes may not be beneficial to model inference. In this case, the graph smoothing objective Definition 7.1 needs to be refined, and consequently, the sparsification strategy should be adjusted. While there are recent works on the spectral optimization process for heterophilic graphs, how to apply sparsification is largely unexplored.

Chapter 8

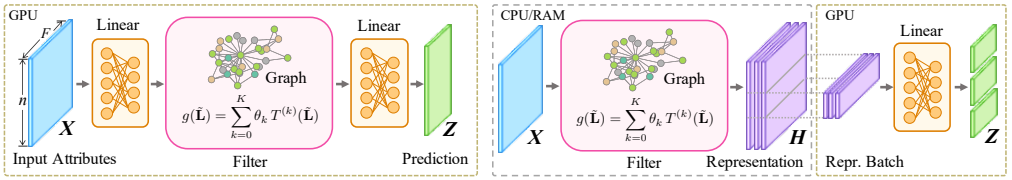
Benchmarking Spectral Graph Neural Networks

8.1 Introduction

Spectral GNNs denote a specialized set of GNN designs that are based on spectral graph theory and apply graph signal filters in the spectral domain [24, 25, 63, 26]. In the graph, such spectral information is powerful for characterizing various patterns, from local edge connections to global clustering structures [243, 291]. Hence, spectral GNNs have garnered exceptional utility for real-world applications encompassing specialized graph topologies, such as multivariate time-series forecasting [292, 293] and point cloud analysis [294, 295].

Particularly, spectral models are superior in addressing the notorious scalability issue of common GNNs [45, 42, 81]. This merit stems from the unique graph data management as illustrated in Figure 8.1. Conventionally, learning GNNs is achieved through the full-batch scheme, where graph data and neural network weights are integrated, rendering iterative computation and high GPU overhead. In comparison, the mini-batch setup is uniquely available for spectral GNNs by dividing learning data into small batches, thereby alleviating both efficiency bottlenecks and memory footprints [78, 47]. The graph data and weights can be processed separately with tailored computations, enabling

This work is published as: [7] **Ningyi Liao**, Haoyu Liu, Zulun Zhu, Siqiang Luo, Laks V.S. Lakshmanan. "A Comprehensive Benchmark on Spectral GNNs: The Impact on Efficiency, Memory, and Effectiveness". In *ACM SIGMOD International Conference on Management of Data (**SIGMOD**)*, vol. 3, no. 4, 2026. DOI: 10.1145/3749156.



(a) Full-batch (b) Mini-batch

FIGURE 8.1: The spectral filter remains the same between **(a)** full-batch learning, which is the common setting of GNN training, and **(b)** mini-batch learning, which is specialized for spectral GNNs. The invariability enables unified spectral filter implementation and evaluation across different model architectures and learning schemes possessing varying efficiency and scalability performance.

spectral GNNs to excel in learning web-scale graphs with hundreds of millions of nodes [296, 118].

The spectral domain of graphs has long been incorporated into graph learning tasks [297, 115]. Previous surveys on spectral GNNs [298, 299] primarily focus on deriving spectral models and evaluating their accuracy on small datasets. Although the spectral design has been shown to be promising in scaling up GNNs, there is a lack of comprehensive study, especially on their efficiency and scalability performance. We note that implementing and evaluating these models for large-scale graphs remains a demanding task considering the following major challenges:

- **Under-explored practical performance on large-scale graphs:** For the interest of fast and useful GNNs in a broader scope, it is crucial to understand the performance of advanced solutions, extensively considering their time efficiency, memory overhead, effectiveness, and the relationships among these aspects. However, it remains unknown of such systematic insights regarding the feasibility and performance characteristics of spectral GNNs at various data scales, rendering it difficult to decide the appropriate model.

- **Limited availability of efficient and scalable computation techniques:** Most existing scalable computations based on spectral GNNs, such as graph operation accelerations [111, 112, 300, 1] and mini-batch training techniques [301, 302, 3] are exclusively available for only a few models. While the spectral GNN design promotes better scalability thanks to compact graph computation, the majority of them have never been implemented or evaluated with these techniques, consequently constraining their practical application in large-scale scenarios.

- **Varied implementations and inconsistent evaluations:** Although a broad scope of models can be regarded as spectral GNNs, their computational designs vary greatly in terms of graph managements, network architectures, and training schemes. The distinct settings and implementations lead to varied time and memory overheads and complicate fair comparison among spectral models. For instance, while many are inspired by Personalized PageRank (PPR), GLP [303] directly acquires the scores by matrix inversion, whereas APPNP [123] employs an iterative process through message passing, and GDC [133] relates the computation to spectral filtering.

In this research, we present a comprehensive benchmark on spectral GNNs, especially targeting the above challenges. We extensively review existing GNN designs with spectral interpretations, ranging from traditional to cutting-edge studies, and reach a total of 35 models. To facilitate fair and versatile evaluations, we abstract the filters, i.e., the specialized graph management process, among these models. A novel taxonomy (Table 8.2) is proposed to identify and categorize these filters into three categories, each exhibiting distinct spectral capabilities and performances.

We implement the spectral filters in a unified manner applicable to a variety of settings and tasks. Our framework embraces scalable graph processing techniques, especially the dedicated mini-batch training scheme, and allows for widely applicable spectral operations on million-scale graphs, which have not been achieved before. On top of our taxonomy and implementation, we perform evaluation on efficiency, memory consumption, and effectiveness of spectral filters. Our findings aim to provide new assessment criteria, insightful observations, and useful guidelines, which are of pragmatic interest for researching and deploying spectral GNNs. Compared to previous GNN evaluations, we highlight the following specialties of this work:

- **Unified Framework:** We offer an open-source, plug-and-play collection for spectral models and filters. The framework realizes unified and efficient implementations covering a variety of spectral GNN designs. Reproducible pipelines are included for training and evaluating the provided models.
- **Spectral-oriented Design:** Our spectral framework is designed in a spectral-oriented manner independent to spatial operations. Such implementation ensures that most filters are easily adaptable to a wide range of model-level options on transformation, architecture, and learning schemes. In this way, spectral GNNs

can be constructed on-demand from various choices of modules and configurations offering different precision, speed, and memory performances.

- **Comprehensive and Fair Evaluation:** We provide extensive study on the effectiveness and efficiency of spectral filters across various model architectures and learning schemes under our framework. Particularly, most filters are applicable to graphs beyond million-scale, which has never been achieved before. We analyze the spectral properties of the filters and conclude practical guidelines for designing and utilizing spectral models as well as open questions for future research.

Our benchmark settings cover an inclusive range of spectral GNN components, graph filter designs, and training schemes. The observations feature diverse graph properties and impact factors on the performance of spectral models, and reveal the intricate relationship between efficiency and effectiveness across different scales of graphs, which is only available by our unified framework highlighting scalability. In specific, we outline the novel findings from benchmarking GNNs at scale as follows:

- While the GNN efficiency bottleneck is commonly attributed to graph computation, our evaluation indicates that it only dominates time and memory overheads on graphs above the million scale.
- Similarly, the efficiency enhancement brought by mini-batch learning is only significant on larger graphs, where the graph computation overhead is considerable.
- The effectiveness and efficiency of spectral GNNs are not mutually exclusive, challenging the prevailing belief. With adequate configurations, simple filters can excel in high accuracy and fast computation on most graphs.
- The effectiveness of spectral filters primarily roots in their alignment with the graph information; that is, preserving useful patterns while attenuating noise. Contrarily, sophisticated designs do not guarantee improved accuracy.
- As the approach to harmonizing effectiveness and efficiency in spectral GNNs, it is recommended to employ simple but suitable filters through a better understanding of the graph data.

8.2 Formulation of Spectral GNN Paradigm

8.2.1 Polynomial Spectral GNN

Spectral GNNs view the graph information as a matrix to perform signal analysis techniques such as eigen-decomposition. Although it is based on the distinct interpretation with spatial GNNs, these two variants can be derived from each other, as revealed by previous research [25, 298]. To demonstrate, we start from the spatial GNN formulation by focusing on the graph-related components $f^{(j)}(\tilde{\boldsymbol{A}})$ in GNN and omitting the non-linear transformation, i.e., let $\varphi(\boldsymbol{x}) = \boldsymbol{x}$. We also utilize the graph Laplacian $\tilde{\boldsymbol{L}}$ instead of adjacency to denote graph information, and use the $F = 1$ attribute vector $\boldsymbol{x}$ as node-wise signal instead of the attribute matrix. In this case, the K-hop graph computation can be characterized by a polynomial $g(\tilde{\boldsymbol{L}})$:

$$g(\tilde{\boldsymbol{L}}) \cdot \boldsymbol{x} = \sum_{k=0}^{K} \theta_k \, T^{(k)}(\tilde{\boldsymbol{L}}) \, \boldsymbol{x}, \tag{8.1}$$

where $T^{(k)}(\tilde{\boldsymbol{L}})$ is the k-th term of the polynomial basis, and θ_k is the corresponding parameter. Combination of these two components determines the actual management of graph signals [243].

In spectral domain, a *filter* $\hat{g} : [0, 2] \to \mathbb{R}$ processes the input *signal* into *response* by amplifying or attenuating certain frequencies. Denote the filtering operation on signal $\boldsymbol{x}$ as $\hat{g} * \boldsymbol{x}$, its relationship with spatial operations is given by the Fourier transform:

$$\hat{g} * \boldsymbol{x} = \boldsymbol{U}\hat{g}(\boldsymbol{\Lambda})\boldsymbol{U}^{\top}\boldsymbol{x} = \sum_{i=0}^{n} \hat{g}(\lambda_i)\boldsymbol{u}_i\boldsymbol{u}_i^{\top}\boldsymbol{x}. \tag{8.2}$$

Eq. (8.2) can be understood by three consecutive stages: the Fourier transform $\boldsymbol{y}_1 = \boldsymbol{U}^{\top}\boldsymbol{x}$ which transfers the input signal to the spectral domain; the modulation $\boldsymbol{y}_2 = \hat{g}(\boldsymbol{\Lambda})\boldsymbol{y}_1$ which applies graph information through the filter; and lastly the inverse Fourier transform $\boldsymbol{x}^{\star} = \boldsymbol{U}\boldsymbol{y}_2$ which transfers the output back from the spectral domain.

Eq. (8.2) implies that the spectral filtering operation can be achieved by iterative spatial multiplications with the signal. Therefore, Eq. (8.1) provides a feasible approximation by only considering $K \ll n$ orders of those relatively important graph spectrum. In practice, this polynomial form dominates the actual implementations in spectral GNNs, bypassing

the prohibitive eigen-decomposition through the well-established propagation based on graph adjacency. As a consequence, spectral and spatial GNNs share the same elementary operations, rendering the possibility to derive and evaluate a large family of GNNs under the interpretation of spectral filters. Eq. (8.1) is said to be a K-order polynomial w.r.t. spectral approximation order, or equivalently a K-hop polynomial w.r.t. the number of graph operations.

8.2.2 Relation to Spatial Convolutions

Spatial methods are identified by their direct operation f on the graph topology $\tilde{\boldsymbol{A}}$. A line of studies have revealed the relationship between spatial and spectral operations in the context of GNN learning, showing that spectral formulation can be derived from the spatial one, and vice versa [25, 298, 299]. Without loss of generality, by expanding $T(\tilde{\boldsymbol{L}})$ and substituting $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$, the *spectral* filter defined in Eq. (8.1) can be written as a polynomial based on $\tilde{\boldsymbol{A}}$ as $f(\tilde{\boldsymbol{A}}) = \sum_{j=0}^{J} \xi_j \tilde{\boldsymbol{A}}^j$, which exactly corresponds to the *spatial* convolution under mean aggregation. In other words, the polynomial spectral filters can be equivalently achieved by a series of recurrent graph propagations. In cases where the spatial and spectral orders align with each other, there is $J = K$. Examples of deriving between $f(\tilde{\boldsymbol{A}})$ and $g(\tilde{\boldsymbol{L}})$ formulations are given in Section 8.3.

In terms of multi-layer GNNs, if for the k-th layer, the single-layer filter in spectral domain is $g^{(k)}$, then the overall spectral filter is as expressive as $g = g^{(k)} * g^{(k-1)} * \cdots * g^{(0)}$ [304, 243]. Conversely, if the explicit polynomial is known, the model can be calculated in the iterative form by recursively multiply the diffusion matrix (or matrices) to acquire the k-order basis, and adding to the result with respective weight parameter. Hence, we use iterative or explicit formulas interchangeably to express filters.

8.2.3 Relation to GNNs Surveys and Benchmarks

The broad area of graph neural networks has been extensively reviewed from different aspects, while the spectral paradigm is a relatively unprecedented topic. We respectively discuss the related benchmark studies:

- Surveys for *general* GNNs examine the algorithmic designs including architectural selections, convolutional and sequential operations, and overall graph learning

TABLE 8.1: Review of GNN frameworks and benchmarks literature having empirical evaluations. "Scheme": available learning schemes. "Eff.": whether efficiency is evaluated. "Scale": largest dataset used in terms of the number of nodes.

Type	Study	Spectral Models	Scheme*	Eff.	Scale
General	PyG [305]	GCN, ChebNet, SGC, APPNP, ARMA	FB & GP	✗	/
	Oleg Platonov et al. [235]	GCN, CPGNN, FAGCN, GPRGNN, JacobiConv	FB	✗	49K
	OGB [43]	GCN, SGC	FB	✗	100M
	LINKX [136]	GCN, SGC, APPNP, GPRGNN	FB & MB	✗	3M
Efficiency	Maekawa et al. [289]	GCN, ChebNet, SGC, GPRGNN	FB	✓	15K
	Dwivedi et al. [83]	GCN	FB & GP	✓	235K
	Duan et al. [78]	SGC	FB & MB	✓	3M
	SGL [79]	SGC, S^2GC, APPNP	FB & MB	✓	3M
Spectral	GAMMA-Spec [299]	14	FB	✗	20K
	Ours	35	FB & MB	✓	3M

* **Scheme**: FB = Full-Batch, GP = Graph Partition, MB = Mini-Batch.

pipelines [16, 61, 76]. Software frameworks [305, 306] and datasets [235, 43, 136] have also been developed. Although a few classic spectral models are covered, they are not distinguished from other spatial models.

- The GNN *efficiency* is especially surveyed in [84, 69, 101] and evaluated in [83, 289, 78, 79], focusing on the empirical performance of algorithmic designs in GNN architectures. However, most of these techniques are limited to spatial interpretations, which are orthogonal to the spectral perspective in our work.

- Two recent surveys target *spectral* GNNs: [298] derives the general connection between typical spatial and spectral operators, advancing the unified interpretation. Our definition of spectral GNNs is primarily based on this work. [299] reviews 14 spectral models identified by the eigen-decomposition. However, both frameworks contain computation-intensive operations, which are prohibitive for large graphs.

Table 8.1 presents the literature containing empirical evaluations. Overall, we note a lack of research encompassing a broad range of up-to-date spectral GNNs, particularly in benchmarking graph operations and scalable implementations. This study hence contributes by offering efficiency comparisons, performance on large-scale datasets, and a comprehensive coverage of spectral filters.

8.2.4 Relation to Spectral GNN Taxonomy

The spectral domain of graphs has long been incorporated into graph learning tasks [297, 115, 107]. In this study, we mainly focus on the form of K-order polynomial $g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k T^{(k)}(\tilde{\boldsymbol{L}})$ as a feasible filter approximation for $\hat{g}$, since directly acquiring eigen-decomposition is prohibitive for graph-scale matrices in practice. Previous studies show that the expressiveness of polynomial filters is satisfactory for estimating arbitrary smooth signals [307, 308]. The formulation is also advantageous in representing filter bank models in the same form.

Other taxonomies are also proposed for categorizing spectral filters. The most inclusive description of spectral graph filters is derived from the full graph spectrum as presented in Eq. (8.2). However, directly acquiring the graph spectrum through eigen-decomposition is prohibitive due to computational cost, rendering this framework distant from the practical graph filtering process in GNNs. As elaborated, Eq. (8.1) serves as a K-order approximation to the full spectrum filtering. Relevant works are discussed in Section 8.2.5.

Three types in [298]. [298] develops a taxonomy containing linear, polynomial, and rational functions. Intuitively, both its linear and polynomial functions can be expressed by our polynomial formulation Eq. (8.1), as the linear function corresponds to the special case where $K = 1$. The rational function is a more powerful filter formulation being capable of learning non-smooth signals, typically discontinuous ones. Nonetheless, computing the inverse of the graph matrix is impractical, and rational filters are also reduced to the polynomial form in practical computation. This can be achieved by the Neumann series on matrix inverse that $(\boldsymbol{I} - \boldsymbol{A})^{-1} = \sum_{k=0}^{\infty} \boldsymbol{A}^k$, and truncate to K-order approximation. In Appendix C, filters such as PPR, HK, and Horner showcase such transformation.

Decomposition-based framework in [299]. [299] mostly identifies spectral GNNs by the eigen-decomposition. In its theoretical framework and code implementation, this is performed by explicitly acquiring the graph spectrum $\tilde{\boldsymbol{L}} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$ and applying graph convolution $\hat{g}(\boldsymbol{\Lambda})$ accordingly. When applied to graph signals, the full n-order eigen-decomposition performs the spectral convolution following Eq. (8.2). In Section 8.2.1, we elaborate that the polynomial form in Eq. (8.1) serves as an effective truncated K-order approximation. In fact, realistic graph information are largely available in the low-frequency domain with small k-s, while high-frequency components are mostly

noise. The spectral information is further compressed during spectral processing on the F-dimension representation. These considerations render our framework based on the polynomial form a practical and effective approach in characterizing spectral filters.

8.2.5 Other Relevant GNN Models

Spectral Decomposition. A range of works are only applicable to the full eigen-decomposition. SpectralCNN [24] employs trainable weight matrices as filters on the eigenspace, while LanczosNet [233] integrates simple spatial operations and appends multi-scale capability. SIGN [309] employs a sign invariant and permutation equivariant on the Laplacian eigenvectors, which are viewed as the positional encodings and added as additional node features in the graphs. Alternatively, Specformer [310] exploits a transformer model to establish expressive spectral filtering. Nonetheless, we note that the full eigen-decomposition is largely prohibitive, resulting in weaker applicability of these models, especially on large graphs. Hence, we do not include them in our benchmark evaluation, while comparison for these models can be found in [299] and [298].

Alternative Spectral Filters. Signal processing techniques beyond the graph Fourier transform have been introduced to build spectral models, such as the wavelet and pyramid transforms [311, 312, 313]. Other research also sets the stage for adapting spectral properties or filtering pipelines to specific tasks and graph variants [314, 315, 316, 242]. Although these methods expand the scope of spectral GNNs, they are less typical considering their reliance on highly customized operations, usually without available implementation for large-scale datasets.

Task-Specific Models. These methods set the stage for adapting the spectral properties or filtering pipelines to particular tasks, instead of focusing on novel filter designs. **GHRN** [314] explores the heterophily in the spatial domain and the frequency in the spectral domain, which efficiently addresses the heterophily issue for graph anomaly detection. **StableGCNN** [315] provides a theoretical spectral perspective of existing GNN models which aims to analyze their stability and establish their generalization guarantees. **SpecGN** [316] applies a smoothing function on the graph spectrum to alleviate correlation among signals. **li2021f** [242] explores the utility of the spectrum of both node and attribute affinity graphs.

8.3 Taxonomy of Spectral GNNs

In this study, we propose a novel taxonomy for spectral GNNs based on their inherent parameterization schemes, and categorize them into three types in Table 8.2. Our taxonomy is primarily based on the formation of the polynomial filter depicted by the basis $T(\tilde{\boldsymbol{L}})$ and the parameter θ, which is the pivotal component of spectral GNNs and determines the actual response to graph signals [243]. We respectively introduce the filter types and their representative models, while detailed derivations of filters within each type are presented in Appendix C. A summary of filters covered in this study is provided in Table 8.2.

Selection and Naming of Filters. We extensively survey existing GNN designs that can be represented in the polynomial form Eq. (8.1). In specific, we focus on works encompassing graph data and propagation operations, which signify the spectral formulations $g(\tilde{\boldsymbol{L}};\theta)$ of the filter. The naming of the filters generally follows the cited original works in Table 8.2, since most of the GNN models are inspired by traditional polynomial functions or signal processing filters. Exceptional cases include the novel filters ChebInterp and OptBasis as well as filter bank models, which are named after the respective works in Table 8.2.

8.3.1 Fixed Filter GNN

For spectral GNNs with polynomial filters in Eq. (8.1), the basis $T(\tilde{\boldsymbol{L}})$ and the parameter θ jointly determine the actual response to graph signals, and consequently dominate the model performance [243]. It is pivotal for designing spectral GNNs to choose the basis and parameter that are suitable for the task. We thereby categorize polynomial spectral GNNs based on the acquisition schemes of bases and parameters into three types, as respectively introduced in the following subsections.

For the first type of spectral GNNs, the basis and parameters are both constant during learning, resulting in fixed filters $g(\tilde{\boldsymbol{L}})$. In practice, these filters are usually predefined simple schemes such as Monomial or PPR summation. Below we formulate several representative fixed filters and corresponding spectral GNN models.

Impulse. SGC [125] adopts a decoupled architecture by removing non-linear transformations in intermediate layers. Its embedding in spatial domain is $\boldsymbol{P} = \tilde{\boldsymbol{A}}^J\boldsymbol{X}$. By selecting

TABLE 8.2: Taxonomy of spectral GNN filters with notable corresponding models included in this work.

Type	Filter*	Filter Function $g(\tilde{L})$	Model†
Fixed	Identity	$\boldsymbol{I}$	MLP
	Linear	$2\boldsymbol{I} - \tilde{\boldsymbol{L}}$	$\mathbb{I}$: GCN [25]
	Impulse	$(\boldsymbol{I} - \tilde{\boldsymbol{L}})^K$	$\mathbb{D}$: SGC [125], gfNN [317], GZoom [169], GRAND+ [302]
	Monomial	$\frac{1}{K+1}\sum_{k=0}^{K}(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{D}$: S^2GC [132], AGP [112], GRAND+ [302]
	PPR	$\sum_{k=0}^{K}\alpha(1-\alpha)^k(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{I}$: GLP [303], GCNII [67]; $\mathbb{D}$: APPNP [123], GDC [133], AGP [112], GRAND+ [302]
	HK	$\sum_{k=0}^{K}\frac{e^{-\alpha}\alpha^k}{k!}(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{D}$: GDC [133], AGP [112], DGC [318]
	Gaussian	$\sum_{k=0}^{K}\frac{\alpha^k}{k!}(2\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{D}$: G^2CN [319]
Variable	Linear	$(1+\theta)\boldsymbol{I} - \tilde{\boldsymbol{L}}$	$\mathbb{I}$: GIN [320], AKGNN [321]
	Monomial	$\sum_{k=0}^{K}\theta_k(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{D}$: DAGNN [322], GPRGNN [134]
	Horner	$\sum_{k=0}^{K}\theta_k(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{I}$: ARMAGNN [323], HornerGCN [324]
	Chebyshev	$\sum_{k=0}^{K}\theta_k T_{\text{Cheb}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{I}$: ChebNet [26]; $\mathbb{D}$: ChebBase [301]
	ChebInterp	$\frac{2}{K+1}\sum_{k=0}^{K}\sum_{\kappa=0}^{K}\theta_\kappa T_{\text{Cheb}}^{(k)}(x_\kappa)T_{\text{Cheb}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{D}$: ChebNetII [301]
	Clenshaw	$\sum_{k=0}^{K}\theta_k T_{\text{Cheb2}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{I}$: ClenshawGCN [324]
	Bernstein	$\sum_{k=0}^{K}\frac{\theta_k}{2^K}\binom{K}{k}(2\boldsymbol{I} - \tilde{\boldsymbol{L}})^{K-k}\tilde{\boldsymbol{L}}^k$	$\mathbb{D}$: BernNet [325]
	Legendre	$\sum_{k=0}^{K}\theta_k\frac{(-1)^k}{k!\binom{2k}{k}}(2\boldsymbol{I} - \tilde{\boldsymbol{L}})^k\tilde{\boldsymbol{L}}^k$	$\mathbb{D}$: LegendreNet [326]
	Jacobi	$\sum_{k=0}^{K}\theta_k T_{\text{Jacobi}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{D}$: JacobiConv [243]
	Favard	$\sum_{k=0}^{K}\theta_k T_{\text{Favard}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{D}$: FavardGNN [327]
	OptBasis	$\sum_{k=0}^{K}\theta_k T_{\text{OptBasis}}^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{D}$: OptBasisGNN [327]
Bank	Linear (channel-wise)	$\Vert_{q=1}^{Q}(\boldsymbol{I} - \gamma_q\tilde{\boldsymbol{L}})$	$\mathbb{D}$: AdaGNN [328]
	Linear (LP, HP)	$\gamma_1(\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2\tilde{\boldsymbol{L}}$	$\mathbb{I}$: FBGCN (I/II) [329]
	Linear (LP, HP, ID)	$\gamma_1(\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2\tilde{\boldsymbol{L}} + \gamma_3\boldsymbol{I}$	$\mathbb{I}$: ACMGNN (I/II) [139]
	Linear (LP, HP)	$\gamma_1((\beta+1)\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2((\beta-1)\boldsymbol{I} + \tilde{\boldsymbol{L}})$	$\mathbb{D}$: FAGNN [137]
	Gaussian (LP, HP)	$\sum_{q=1}^{Q}\sum_{k=0}^{\lfloor K/2\rfloor}\frac{\alpha_q^k}{k!}((1+\beta_q)\boldsymbol{I} - \tilde{\boldsymbol{L}})^{2k}$	$\mathbb{D}$: G^2CN [319]
	PPR (LP, HP)	$\sum_{q=1}^{Q}\sum_{k=0}^{K}\alpha_q(1-\alpha_q)^k(\boldsymbol{I} + \beta_q\tilde{\boldsymbol{L}})(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$	$\mathbb{D}$: GNN-LF/HF [277]
	Mono, Cheb, Bern, ID	$\sum_{q=1}^{Q}\sum_{k=0}^{K}\gamma_q\theta_{q,k}T_q^{(k)}(\tilde{\boldsymbol{L}})$	$\mathbb{D}$: FiGURe [330]

* **Filter**: LP = Low-pass, HP = High-pass, ID = Identity.
† **Model**: $\mathbb{I}$ = Iterative architecture, $\mathbb{D}$ = Decoupled architecture.

TABLE 8.3: Complexity analysis of spectral GNN filters. "Parameters" represents learnable filter parameters acquired during model training, while "Hyperparameters" represents tunable filter hyperparameters acquired through hyperparameter search. "Time" and "Memory" are time and space complexity for filter computation, respectively.

Type	Filter	Parameters	Hyperparameters	Time	Memory
Fixed	Identity	/	/	$O(KnF)$	$O(nF)$
	Linear	/	/	$O(KmF)$	$O(nF)$
	Impulse	/	/	$O(KmF)$	$O(nF)$
	Monomial	/	/	$O(KmF)$	$O(nF)$
	PPR	/	α	$O(KmF)$	$O(nF)$
	HK	/	α	$O(KmF)$	$O(nF)$
	Gaussian	/	α	$O(KmF)$	$O(nF)$
Variable	Linear	θ_k	/	$O(KmF)$	$O(nF)$
	Monomial	θ_k	/	$O(KmF)$	$O(nF)$
	Horner	θ_k	/	$O(KmF)$	$O(2nF)$
	Chebyshev	θ_k	/	$O(KmF)$	$O(2nF)$
	ChebInterp	θ_k	/	$O(KmF + KF^2)$	$O(2nF)$
	Clenshaw	θ_k	/	$O(KmF)$	$O(3nF)$
	Bernstein	θ_k	/	$O(K^2mF)$	$O(nF)$
	Legendre	θ_k	/	$O(KmF)$	$O(2nF)$
	Jacobi	θ_k	α, β	$O(KmF)$	$O(2nF)$
	Favard	θ_k	/	$O(KmF + KnF)$	$O(2nF)$
	OptBasis	θ_k	/	$O(KmF + KnF^2)$	$O(2nF)$
Bank	AdaGNN	γ_q	/	$O(KmF)$	$O(nF)$
	FBGCN	γ_q	/	$O(QKmF + QKnF)$	$O(QnF)$
	ACMGNN	γ_q	/	$O(QKmF + QKnF)$	$O(QnF)$
	FAGNN	γ_q	β	$O(QKmF)$	$O(QnF)$
	G^{2}CN	γ_q	α_q, β_q	$O(QKmF)$	$O(QnF)$
	GNN-LF/HF	γ_q	α_q, β_q	$O(QKmF)$	$O(QnF)$
	FiGURe	$\gamma_q, \theta_{q,k}$	/	$O(QKmF)$	$O(QnF)$

the basis $T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$, the spectral filter is only activated on the K-th term:

$$g(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^K = \sum_{k=0}^{K} \theta_k (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_0 = \theta_1 = \cdots = \theta_{K-1} = 0,\ \theta_K = 1.$$

Personalized PageRank (PPR). APPNP [123] is a pioneering decoupled GNN that iteratively applies a decaying graph propagation $\boldsymbol{H}^{(j+1)} = \varphi((1-\alpha)\tilde{\boldsymbol{A}}\boldsymbol{H}^{(j)} + \alpha\boldsymbol{H}^{(0)})$, where $\alpha \in [0, 1]$ denotes the decay coefficient. It is revealed to be equivalent to the well-studied PPR calculation [129], i.e., iteratively multiplying the signal with $\tilde{\boldsymbol{A}}^k$, and accumulating with a decaying coefficient $\alpha(1-\alpha)^k$. Recall that $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$, the spectral basis for each iteration can be written as $T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$, and the corresponding parameter is $\theta_k = \alpha(1-\alpha)^k$. We name it as the PPR filter:

$$g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \alpha(1-\alpha)^k (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_k = \alpha(1-\alpha)^k.$$

Regarding computational complexity, calculating the K-order filter requires $O(KmF)$ time for the graph computation and $O(nF)$ space for storing the representation matrix.

8.3.2 Variable Filter GNN

The second type of spectral GNN in our taxonomy also features a predetermined basis $T^{(k)}(\tilde{\boldsymbol{L}})$ in Eq. (8.1). The series of parameters θ_k is variable, usually acquired through gradient descent during model training, leading to a variable filter $g(\tilde{\boldsymbol{L}}; \theta)$. Compared to spectral GNNs with fixed filters, these models enjoy better capability, especially for fitting high-frequency signals and capturing non-local graph information. The choice of basis is the primary factor of the model performance on practical tasks [243, 327]. We also present a few exemplars below.

Monomial. GRPGNN [134] is among the first works to invoke a learnable parameter ξ_j for each hop of the decoupled graph propagation, which is equivalent to Generalized PageRank computation with arbitrary hop-wise coefficients ξ_j [331, 332]. It establishes connection between the learned values of parameters and the model adaptability on

heterophilous graphs. We respectively formulate its spatial and spectral expressions as:

$$f(\tilde{\boldsymbol{A}};\xi)=\sum_{j=0}^{J}\xi_j\,\tilde{\boldsymbol{A}}^j;\quad g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k(\boldsymbol{I}-\tilde{\boldsymbol{L}})^k.$$

Chebyshev. ChebNet [26] is a representative spectral GNN that explicitly utilizes the classical Chebyshev polynomial [333] as basis, which possesses orthogonal terms with straightforward computation. Each term $T^{(k)}(\tilde{\boldsymbol{L}})$ is expressed in the recursive form based on the computation results $T^{(k-1)}(\tilde{\boldsymbol{L}})$ and $T^{(k-2)}(\tilde{\boldsymbol{L}})$ of previous hops. The Chebyshev filter can be formulated as:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\quad T^{(k)}(\tilde{\boldsymbol{L}})=2\tilde{\boldsymbol{L}}T^{(k-1)}(\tilde{\boldsymbol{L}})-T^{(k-2)}(\tilde{\boldsymbol{L}}),\ T^{(1)}(\tilde{\boldsymbol{L}})=\tilde{\boldsymbol{L}},\ T^{(0)}(\tilde{\boldsymbol{L}})=\boldsymbol{I}.$$

The computation scheme of ChebNet can be directly inferred from the filter formulation. Initially, it employs $\boldsymbol{X}$ and $\tilde{\boldsymbol{L}}\boldsymbol{X}$ as the 0- and 1-order terms. Then, the recursive equation is followed to apply $T^{(k)}(\tilde{\boldsymbol{L}})$ and results are accumulated by multiplying θ_k. Its complexity for performing graph operations is $O(KmF)$. We denote its memory footprint as $M\sim 2nF$, that the dominant overhead is proportional to the cost of maintaining two $n\times F$ matrices during the iterative filter computation.

8.3.3 Filter Bank GNN

Some studies argue that a single filter is limited in leveraging complex graph signals. Hence, spectral GNNs with filter bank arise as an advantageous approach to provide abundant information for learning. In this study, we innovatively incorporate these models into the spectral GNN taxonomy as a mixture of Q fixed or variable filters $g_q(\tilde{\boldsymbol{L}};\theta)$, each assigned a filter weight parameter γ_q:

$$g(\tilde{\boldsymbol{L}};\gamma,\theta)=\bigoplus_{q=1}^{Q}\gamma_q\cdot g_q(\tilde{\boldsymbol{L}};\theta),\quad g_q(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_{q,k}\,T_q^{(k)}(\tilde{\boldsymbol{L}}),\tag{8.3}$$

where $\bigoplus$ denotes an arbitrary fusion function such as summation or concatenation. By this means, the filter bank is able to cover different channels, i.e., ranges of frequencies, to generate more comprehensive embeddings of the graph. The filter parameter γ can be

either learned separately or along with GNN training, depending on the specific model implementation.

ACMGNN. ACMGNN [139] employs three simple filters as adaptive channel mixing as an effort to address the heterophily issue by solely considering node-wise local information without heavy full-graph computation. For each model layer, the propagation is conducted by the graph adjacency $T_1 = \tilde{\boldsymbol{A}}$, Laplacian $T_2 = \tilde{\boldsymbol{L}}$, and identity matrices $T_3 = \boldsymbol{I}$ separately. It is underscored that these operations respectively correspond to low-pass, high-pass, and all-pass filters that captures spectral signals of diverse frequencies. Filters are combined as the weighted sum, where filter weights γ are learned during training. The single-layer spectral expression can be simplified as:

$$g_{\theta}(\tilde{\boldsymbol{L}}) = \gamma_1(\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2\tilde{\boldsymbol{L}} + \gamma_3\boldsymbol{I}.$$

FiGURe. FiGURe [330] is an exemplar filter bank GNN following exactly Eq. (8.3) with summation fusion $g_{\theta}(\tilde{\boldsymbol{L}}; \gamma, \theta) = \sum_{q=1}^{Q} \gamma_q \cdot g_q(\tilde{\boldsymbol{L}}; \theta)$, where filter-level parameters γ_q are learned to control the channel strength. It presents an unsupervised precomputation process to learn the filter transformation γ_q as an approach to generate expressive graph embeddings. The model is then capable of supervised representation learning for specific tasks by tuning the model weights. Common variable filters including Monomial, Chebyshev, and Bernstein bases are utilized to compose the filter bank. With a straightforward implementation, the time and memory complexity of the combination are Q times that of an individual filter.

8.4 Benchmark Design

Framework Implementation. Based on the paradigm of spectral GNNs, we develop a unified framework oriented towards the invariable spectral filters in particular. Our implementation embraces the modular design principle, offering plug-and-play filter modules that can be seamlessly integrated into mainstream graph learning toolkits. It is also easily extendable to new filter designs, as only the spectral formulation described in Section 8.3 needs to be implemented. Then the filter enjoys a wide range of off-the-shelf blocks for building a complete GNN.

Our framework design is based on the popular graph learning library PyTorch Geometric (PyG) [305] with the same arrangement of components. We list the following highlights of our framework compared to PyTorch Geometric and similar works [78, 79]:

- **Plug-and-play modules:** The model and layer implementation in our framework follows the same design as PyTorch Geometric, implying that they can be seamlessly integrated into PyG-based programs. Our rich collection of spectral models and filters greatly extends the PyG model zoo.
- **Separated spectral kernels:** We decouple non-spectral designs and feature the pivotal spectral filters being consistent throughout different settings. Most of the filters are thence applicable to be combined with various network architectures, learning schemes, and other toolkits, including those provided by PyG and PyG-based frameworks.
- **High scalability:** As spectral GNNs are inherently suitable for large-scale learning, our framework is feasible to common scalable learning schemes and acceleration techniques. Several spectral-oriented approximation algorithms are also supported.

We further elaborate on our code structure in Figure 8.2 by comparing corresponding components in PyG:

- We implement the spectral filters as `nn.conv` modules, which are pivotal and basic blocks for building spectral GNNs. This is similar to the implementation of spatial layers in PyG.

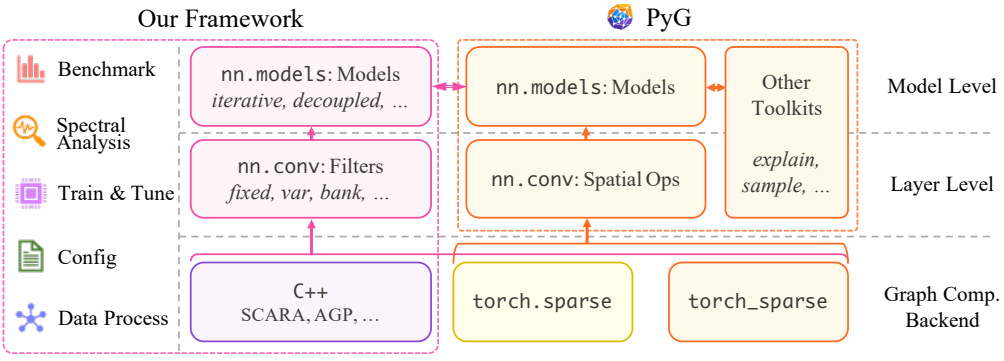


FIGURE 8.2: Code structure of our framework and relation to PyG.

- On the model level, common architectures are included in `nn.models`. Since these models are built in PyG style, they can seamlessly access other utility interfaces offered by PyG and PyG-based toolkits.
- Regarding the underlying backend for sparse matrix multiplication in graph propagation, we support PyTorch-based computations inherited from PyG, as well as extendable interfaces for scalable C++-based algorithms specialized for spectral filtering.
- Additionally, we include our reproducible benchmark pipeline for training and evaluating the models, with a particular focus on spectral analysis as well as scalable mini-batch training.

Tasks and Metrics. In main experiments, we focus on the semi-supervised learning for node-classification task on a single graph, as it is the mainstream task for spectral GNNs. In particular, the evaluation features some of the largest datasets available for graph learning, which is ideal for assessing the scalability of spectral models. We follow the dataset settings for using *efficacy* metrics including accuracy and ROC AUC. *Efficiency* with respect to both running speed and CPU/GPU memory usage is measured for each learning stage, including training, inference, and precomputation. We conduct 10, 10, and 5 runs with different random seeds for small-, medium-, and large-scale evaluations in the main experiment, respectively, and report the average metrics on test split with standard deviation. Evaluation are conducted on a single machine with 32 Intel Xeon CPUs (2.4GHz), an Nvidia A30 (24GB memory) GPU, and 512GB RAM.

Datasets. We comprehensively involve 22 datasets that are widely used in GNN node classification detailed in Table 8.4. In the table, we incorporate self-loop edges and count undirected edges twice to better reflect the graph computation overhead. We mainly categorize the datasets from two aspects. We follow the unified dataset processing protocol for common node classification task including edge direction, degree normalization, and feature transformation in line with the original literature. Random 60%/20%/20% splits for train/validation/test sets are used for graphs without predefined splits.

In **efficacy** evaluations, we separately investigate *homophilous* and *heterophilous* datasets considering they exhibit different patterns of graph signals. For experiments on **efficiency**, we alternatively distinguish the datasets into *small*, *medium*, and *large* categories. We focus on medium and large datasets for efficiency analysis, with large graphs identified by

empirical evaluation that at least some models are prohibitive due to the critical scalability issue. The criteria is mainly as follows:

- *Small-scale datasets:* This set of datasets are mainly utilized for efficacy evaluation of all models and schemes. Since the graph sizes are relatively small, results regarding efficiency are omitted in this study, while they can be similarly produced using the published codebase.
- *Medium-scale datasets:* Experiments on these datasets share the same settings with small-scale ones. However, we include the efficiency study to provide comparable results across different models and training schemes.
- *Large-scale datasets:* This category of graphs is mainly only applicable to the mini-batch scheme, as most full-batch models occur out-of-memory error. Some hyperparameters are also separated to ensure efficient learning performance.

TABLE 8.4: Dataset statistics. d, F, N_c, and $\mathcal{H}_{\text{n}}$ are average degree, input attribute dimension, number of label classed, and node homophily score, respectively.

Scale	Hetero.	Dataset	Nodes n	Edges m	d	F	N_c	$\mathcal{H}_{\text{n}}$	Metric
Small	Homo.	CORA [254]	2,708	10,556	3.90	1,433	7	0.83	Accuracy
		CITESEER [254]	3,327	9,104	2.74	3,703	6	0.72	Accuracy
		PUBMED [254]	19,717	88,648	4.50	500	3	0.79	Accuracy
		MINESWEEPER [235]	10,000	78,804	7.88	7	2	0.68	ROC AUC
		QUESTIONS [235]	48,921	307,080	6.28	301	2	0.90	ROC AUC
		TOLOKERS [235]	11,758	1,038,000	88.28	10	2	0.63	ROC AUC
	Hetero.	CHAMELEON [114]	890	17,708	19.90	2,325	5	0.24	Accuracy
		SQUIRREL [114]	2,223	93,996	42.28	2,089	5	0.19	Accuracy
		ACTOR [114]	7,600	30,019	3.95	932	5	0.22	Accuracy
		ROMAN-EMPIRE [235]	22,662	65,854	2.91	300	18	0.05	Accuracy
		AMAZON-RATINGS [235]	24,492	186,100	7.60	300	5	0.38	Accuracy
Medium	Homo.	FLICKR [121]	89,250	899,756	10.08	500	7	0.32	Accuracy
		OGBN-ARXIV [43]	169,343	1,166,243	6.89	128	40	0.63	Accuracy
	Hetero.	ARXIV-YEAR [136]	169,343	1,166,243	6.89	128	5	0.31	Accuracy
		PENN94 [136]	41,554	2,724,458	65.56	4,814	2	0.48	Accuracy
		GENIUS [136]	421,961	984,979	2.33	12	2	0.83	ROC AUC
		TWITCH-GAMER [136]	168,114	6,797,557	40.43	7	2	0.97	ROC AUC
Large	Homo.	OGBN-MAG [43]	736,389	5,416,271	7.36	128	349	0.32	Accuracy
		OGBN-PRODUCTS [43]	2,449,029	123,718,280	50.52	100	47	0.83	Accuracy
	Hetero.	POKEC [136]	1,632,803	30,622,564	18.76	65	2	0.45	Accuracy
		SNAP-PATENTS [136]	2,923,922	13,972,555	4.78	269	5	0.27	Accuracy
		WIKI [136]	1,925,342	303,434,860	157.60	600	5	0.28	Accuracy

Learning Schemes. Figure 8.1 indicates that the spectral filter is invariable across different training schemes. *Full-batch* training loads all input data onto the GPU, which is the de facto scheme for most GNN models. In contrast, *mini-batch* scheme performs graph-related operations in a precomputation stage on CPU. Then, only the intermediate representations are loaded onto the GPU in batches during training, which prevents the memory footprint to be coupled with the graph size and enjoys better scalability. Our novel implementation enables the application of most filters to both schemes for fair and comprehensive comparison.

Model Architecture. We utilize a unified architecture for all models in our evaluation, which is scalable to larger graphs while maintaining efficacy. This mainly incorporates the decoupled design, i.e., having fixed or learnable scalars θ_k as parameters. In the spatial interpretation, the decoupled architecture implies performing all graph propagations successively, and transformations conducted by learnable weight matrices are only performed before or after all propagation operations, respectively denoted as φ_0 and φ_1.

Hyperparameters. We unify the non-spectral aspects of architectures for all models in our evaluation, as an approach to provide comparable performance while enhancing scalability to larger graphs. This includes the same transformation operations, network depth and width, propagation hops, batch size, and training epochs among all models

TABLE 8.5: Hyperparameter search scheme. Hyperparameters are explored based on the combination of listed ranges, and underlined values in Stage 1 are the universal settings used across main experiments.

Stage 1 (Universal)	**Full-batch Small/medium-scale**	**Mini-batch Small/medium-scale**	**Mini-batch Large-scale**
Propagation hop K	2, 4, $\cdots$, 10, $\cdots$, 30	2, 4, $\cdots$, 10, $\cdots$, 30	2, 4, $\cdots$, 10, $\cdots$, 30
Hidden width F	16, 32, 64, 128, 256	16, 32, 64, 128, 256	16, 32, 64, 128, 256
Linear layer of φ_0	1, 2, 3	0	0
Linear layer of φ_1	1, 2, 3	1, 2, 3	1, 2, 3
Batch size	–	4,096	200,000
Stage 2 (Individual)			
Graph normalization ρ	$[0, 1]$	$[0, 1]$	$[0, 1]$
Learning rate of φ_0, φ_1	$[10^{-5}, 0.5]$	$[10^{-5}, 0.5]$	$[10^{-5}, 0.5]$
Learning rate of θ, γ	$[10^{-5}, 0.5]$	$[10^{-5}, 0.5]$	$[10^{-5}, 0.5]$
Weight decay of φ_0, φ_1	$[10^{-7}, 10^{-3}]$	$[10^{-7}, 10^{-3}]$	$[10^{-7}, 10^{-3}]$
Weight decay of θ, γ	$[10^{-7}, 10^{-3}]$	$[10^{-7}, 10^{-3}]$	$[10^{-7}, 10^{-3}]$

in the main experiment. For other hyperparameters not affecting efficiency, we perform individual hyperparameters tuning for each model and dataset.

In order to ensure a fair comparison of both effectiveness and efficiency, we employ a two-step hyperparameter tuning scheme. The hyperparameters in each stage, search ranges, and used values are listed in Table 8.5:

- *First*, we search for hyperparameters possessing significant impacts on the model efficiency, including the number of hops K and hidden dimension F. To ensure comparable total running times, the epoch number is kept constant at 500.
- *Then*, we utilize a fixed choice of these critical hyperparameters for experiments across all models and datasets, and perform specific search on other hyperparameters for each respective experiment.

8.5 Results: Efficiency and Effectiveness

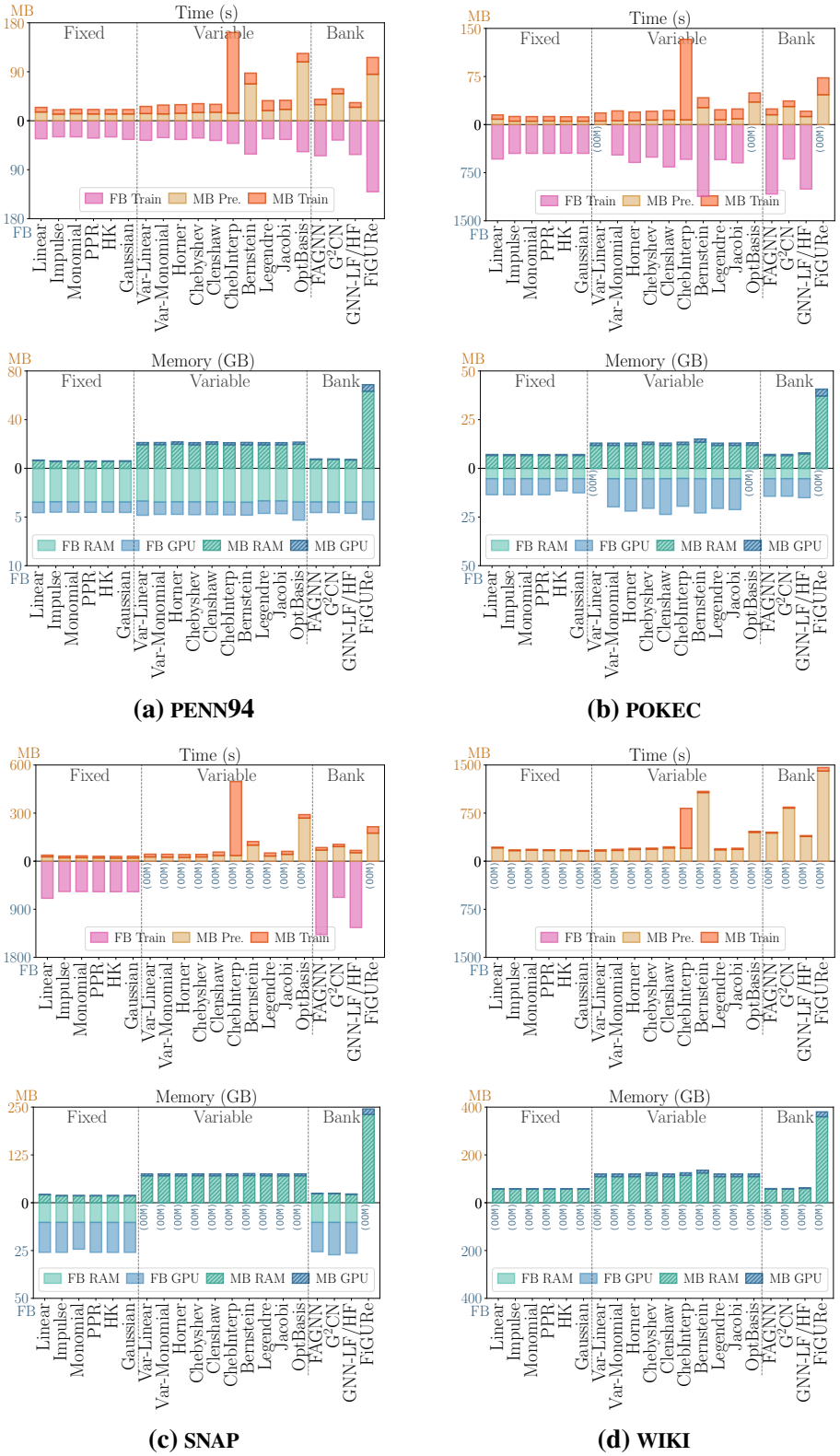
In this section, we present primary research questions (RQs) and analysis on the performance comparison among a total of 27 filters across three types, benchmarking their effectiveness, efficiency, and memory overhead under different datasets and training schemes.

8.5.1 Efficiency

Thanks to our unified implementation framework, the time and memory efficiency of spectral GNNs can be compared across different graph scales under fair criteria. Figure 8.3 provides a detailed breakdown of full- and mini-batch schemes, separating learning stages and devices for the time and memory evaluations, respectively.

Model Operations. We assess the performance from multiple perspectives of filter operations and learning schemes, firstly comparing among the evaluated filters under the same training settings, followed by a discussion on different learning schemes.

RQ1: *What are the efficiency bottlenecks of filters under different graph scales?*



(a) PENN94 **(b) POKEC**

(c) SNAP **(d) WIKI**

FIGURE 8.3: Comparison of filter time and memory efficiency under mini-batch (MB, upper axis) and full-batch (FB, lower axis) training on four medium- to large-scale datasets. Note that FB and MB axes may be on different scales. Empty bars are models with OOM error.

As analyzed in Section 8.2, the key operations of graph propagation and weight transformation exhibit diverse computational overheads. Overall, it can be observed from Figure 8.3 that filter efficiency can be effectively characterized by our taxonomy Table 8.2. The three types of filters exhibit distinct performances, and the empirical time and memory efficiency are in line with our complexity analysis.

Regarding specific operations, variable *transformations* come at the cost of additional computational resources. In the FB scheme, this leads to reduced speed and increased GPU memory footprint. While some simple variable filters (Monomial, Chebyshev) and filter banks consisting of fixed filters (G^2CN, GNN-LF/HF) attain favorable performance when the graph scale is relatively small, models with complex transformations (Favard, OptBasis) experience prolonged training times and poor GPU memory scalability, and even result in out-of-memory (OOM) errors on large graphs. For MB training, the cost is reflected in increased RAM usage. Compared to fixed filters, memory footprint of variable designs is K times larger for separately storing results in each hop. Similarly, a filter bank design with Q variable filters demands up to Q times more memory.

On the other hand, *propagation* becomes more computationally intensive on larger graphs. In Figure 8.3, network transformation represented by MB training entails a large portion of time overhead on medium-scale graphs (PENN94), while propagation in precomputation dominates the MB training on larger graphs (POKEC, SNAP). Additional propagation further increases learning time but barely affects memory scalability, which is evident from variable filters with more graph-related propagations (ChebInterp, Bernstein). Nonetheless, thanks to our scalable implementation, their memory footprints are on par with other variable filters and are applicable to large-scale data even in memory-constrained environments.

Learning Schemes. For investigating the efficiency difference between learning schemes, we compare the full-batch and mini-batch learning performance in Figure 8.3 from the perspective of both memory scalability and learning speed.

RQ2: *What are the efficiency and scalability improvements brought by mini-batch training?*

Figure 8.3 implies that filter *scalability*, i.e., whether filters can be applied to larger graphs, is mainly constrained by GPU memory capacity. With FB, models typically encounter the OOM issue on datasets larger than OGBN-MAG and POKEC due to the representation storage proportional to node size n. In comparison, MB training benefits scalability by shifting

the memory overhead from GPU to RAM, enabling the employment of memory-intensive filters (Favard, OptBasis, FiGURe) on POKEC and even larger graphs. To the best of our knowledge, our implementation is the first attempt to successfully scale up an array of advanced spectral GNNs to these million-scale datasets and comprehensively characterize the performance.

Another advantage of MB lie in *time efficiency*. Even though FB for some filters is applicable to some million-scale graphs (POKEC, SNAP), the training time is excessively long. MB achieves a significant $10-50\times$ speedup by decoupling the computation-intensive propagation operations and simplifying the iterative training. Moreover, the speedup is mainly observable on large-scale datasets, while FB and MB present the same level of overall time on medium-sized graphs (PENN94). This can be explained by RQ1, as the increase in efficiency stems from faster propagation and is only effective when propagation is the bottleneck.

By summarizing RQ1&2, we conclude our guidelines for employing mini-batch training: Firstly, MB is particularly suitable for filters without variable parameters, showcasing faster computation and a smaller memory footprint thanks to better device utilization. Another ideal use case arises when FB for complex filters is prohibitive due to memory constraints. MB is capable of conducting graph filtering operations on CPU and storing full-scale representations in RAM at the cost of increased CPU/RAM usage. Even so, since GPU memory often has stricter constraints on practical platforms, MB with GPU memory complexity independent of the graph scale is favorable for deploying spectral GNNs across a wider range of environments.

8.5.2 Effectiveness

Our benchmark specifically features the filter effectiveness from the varied ancillary settings, aiming to demystify their capability in processing graph signals with fair comparison. Table 8.6 displays the filter accuracy with decoupled architecture and full-batch training. In particular, Identity represents the baseline without graph information. Our results align with previous evaluations involving spectral GNNs such as [136, 235, 299] and are mostly comparable to those in the original papers listed in Table 8.2.

Filter-level Comparison. As an overview, our results suggest that filter efficacy depends on graph patterns, and no single filter consistently achieves dominant accuracy across

TABLE 8.6: Effectiveness results (%) and standard deviations of spectral filters with *full-batch* training on available datasets. For each dataset, results are highlighted based on the relative effectiveness among filters, where green results are better.

Type	Filter	CORA	CITESEER	PUBMED	MINE	QUESTIONS	TOLOKERS	FLICKR	ARXIV	MAG
Fixed	Identity	72.37±2.02	72.24±1.51	87.74±0.62	50.52±2.42	71.29±1.81	72.30±1.09	46.83±0.30	53.46±0.49	25.43±1.21
	Linear	86.14±1.15	74.07±1.91	85.15±0.75	68.50±1.49	71.47±0.70	72.74±1.55	50.55±0.46	65.56±1.86	31.52±0.97
	Impulse	84.43±0.82	74.24±2.19	83.38±0.54	64.06±1.61	69.24±1.37	56.75±2.13	49.92±0.23	63.84±1.58	29.32±1.57
	Monomial	86.28±1.72	75.21±1.62	89.07±0.49	70.21±1.46	66.61±6.57	76.51±1.36	50.27±0.73	59.41±0.55	32.41±0.57
	PPR	86.58±1.96	76.09±2.30	89.12±0.39	67.97±1.95	64.19±3.85	76.82±2.37	49.45±0.72	65.14±1.06	26.18±0.96
	HK	86.02±1.53	74.62±1.96	89.15±0.66	72.95±1.15	70.71±1.64	76.72±1.91	47.15±0.06	69.26±0.37	23.53±1.04
	Gaussian	86.44±1.02	75.36±1.64	88.97±0.73	72.61±1.09	66.25±3.71	78.32±1.69	46.67±0.30	63.72±0.83	27.87±3.45
Variable	Linear	85.72±1.46	75.17±1.95	85.50±0.69	72.41±1.21	68.97±3.02	72.04±6.40	50.53±0.10	63.69±1.22	31.11±0.60
	Monomial	86.85±0.65	76.18±1.86	89.05±0.77	89.26±0.80	65.29±4.63	78.95±0.72	52.23±1.30	65.47±1.37	32.03±1.38
	Horner	86.39±2.06	75.44±1.83	89.71±0.48	88.53±1.06	64.01±1.32	77.40±0.69	52.94±0.46	62.94±2.40	29.71±0.55
	Chebyshev	85.95±1.13	75.38±2.13	89.14±0.61	89.92±0.78	62.65±4.21	78.85±0.83	55.11±0.14	66.31±0.61	30.90±1.42
	Clenshaw	83.90±1.98	73.63±1.83	88.49±2.85	88.70±1.33	62.49±5.24	79.66±0.54	52.46±0.38	47.25±10.56	24.06±5.66
	ChebInterp	83.52±5.59	66.96±6.76	89.72±0.87	89.43±0.86	68.15±5.93	80.27±0.91	54.97±2.55	62.81±1.84	23.25±3.75
	Bernstein	71.03±2.49	62.21±2.39	84.24±0.97	87.38±1.12	62.51±2.89	76.36±1.24	47.95±0.56	48.86±0.73	13.83±3.31
	Legendre	87.68±0.96	74.75±2.13	89.72±0.36	89.86±0.76	66.36±2.00	79.04±2.09	48.86±0.92	59.04±1.84	25.00±2.69
	Jacobi	87.02±0.87	75.25±2.28	89.35±0.95	89.48±0.80	68.08±1.57	78.13±2.62	49.09±0.55	38.92±2.73	28.58±1.78
	Favard	86.14±1.60	73.29±2.77	89.18±1.00	89.74±0.79	58.86±4.25	74.51±4.90	42.26±0.00	64.53±0.35	31.29±0.95
	OptBasis	80.74±2.03	69.65±2.84	89.28±0.75	88.73±0.51	56.31±7.84	79.93±1.25	44.81±1.21	58.67±1.92	(OOM)
Bank	AdaGNN	85.45±1.25	74.52±2.09	89.82±0.65	85.83±4.37	67.48±1.86	79.65±1.29	53.55±0.75	54.62±2.20	28.06±5.25
	FBGNNI	81.43±5.82	73.56±1.36	87.06±0.85	89.60±3.13	61.14±3.56	76.63±2.09	51.08±0.66	63.73±3.91	(OOM)
	FBGNNII	81.18±4.04	73.08±2.08	88.12±0.91	84.37±3.70	62.72±5.60	77.31±1.48	52.84±1.12	66.76±3.93	(OOM)
	ACMGNNI	85.12±1.98	74.62±1.90	88.70±0.94	86.21±4.93	56.29±7.46	80.30±1.85	51.04±1.07	69.41±0.51	(OOM)
	ACMGNNII	84.16±4.09	73.95±2.11	89.16±0.54	86.23±3.50	52.93±5.17	84.51±1.09	52.83±1.65	54.49±2.39	(OOM)
	FAGNN	85.45±0.85	75.53±1.58	87.19±1.36	73.98±2.10	66.95±4.62	74.99±1.20	50.06±1.75	67.51±0.40	30.78±1.89
	G^2CN	85.12±3.32	75.50±1.44	88.67±0.69	67.98±1.24	68.95±2.30	77.91±1.95	51.83±0.21	68.96±0.92	29.72±2.17
	GNN-LF/HF	86.83±0.98	75.53±1.67	89.03±0.66	87.80±0.77	60.29±3.55	78.27±2.02	52.34±0.19	60.97±0.32	27.63±0.48
	FiGURe	87.04±0.99	74.49±1.71	88.22±0.95	89.71±0.56	65.38±8.16	79.76±1.39	53.02±0.65	67.57±0.40	29.43±2.18

Type	Filter	CHAMELEON	SQUIRREL	ACTOR	ROMAN	RATINGS	YEAR	PENN94	GENIUS	TWITCH	POKEC
Fixed	Identity	31.60±2.28	29.18±6.07	35.32±1.31	64.64±0.72	41.87±0.54	35.14±0.03	74.22±0.73	86.21±0.11	97.73±1.45	62.21±0.07
	Linear	40.66±2.70	36.43±2.42	26.88±1.43	37.60±0.68	42.17±0.42	44.52±2.04	63.57±9.48	85.71±3.76	96.93±0.06	51.59±1.54
	Impulse	37.57±3.71	35.48±2.12	25.67±1.13	27.37±3.27	31.67±3.94	40.68±0.46	51.00±2.85	88.02±0.15	96.93±0.06	51.40±0.81
	Monomial	39.82±3.38	38.31±2.64	37.84±1.65	63.44±1.48	44.57±1.12	44.74±2.55	75.27±0.29	88.30±0.07	96.99±0.10	60.06±0.22
	PPR	38.20±3.82	35.73±1.45	36.77±1.29	63.78±1.72	39.38±0.49	41.21±0.48	74.14±0.24	88.31±0.16	97.15±0.35	59.91±0.22
	HK	40.10±4.77	37.89±2.27	37.57±1.04	65.08±1.08	41.40±0.91	44.62±0.98	74.59±0.34	88.31±0.21	96.94±0.08	62.10±1.09
	Gaussian	36.38±4.57	36.35±1.36	37.35±1.22	64.04±0.90	41.68±0.40	46.11±1.94	74.64±0.17	87.93±0.21	97.63±0.06	59.95±0.06
Variable	Linear	40.66±3.57	37.61±2.27	26.81±1.71	31.16±0.86	43.83±1.36	46.58±0.63	70.81±0.09	88.42±0.11	97.18±0.49	(OOM)
	Monomial	40.10±3.84	36.94±1.97	34.95±1.64	64.16±1.68	39.50±1.38	46.19±0.50	75.23±0.99	88.42±0.06	96.91±0.07	59.10±1.49
	Horner	40.59±3.09	33.74±1.24	37.67±1.43	58.27±2.38	41.82±1.81	48.43±0.48	74.56±0.39	88.43±0.99	96.93±0.06	72.02±0.82
	Chebyshev	33.71±3.30	37.11±2.39	35.76±1.03	67.92±2.22	39.70±1.17	42.91±1.42	84.31±0.27	87.47±0.03	96.93±0.06	54.26±0.43
	Clenshaw	30.62±2.91	34.66±0.56	36.11±1.95	70.37±3.27	40.44±3.20	43.72±2.00	82.55±0.33	87.40±0.12	96.98±0.04	53.02±0.29
	ChebInterp	30.41±4.08	35.59±1.74	35.78±2.29	69.38±1.98	36.70±0.14	46.50±0.06	82.26±1.37	79.65±7.86	96.93±0.07	69.15±0.10
	Bernstein	34.97±3.41	35.84±1.81	34.90±0.97	46.77±2.18	27.56±2.78	43.08±0.21	81.36±0.56	87.32±0.10	96.92±0.06	72.03±0.21
	Legendre	36.17±2.48	36.94±2.39	37.69±1.43	59.00±6.53	42.90±0.22	44.64±3.46	84.43±0.51	87.79±0.71	96.95±0.04	57.76±0.21
	Jacobi	37.43±4.48	34.86±1.74	30.28±1.91	64.77±1.74	43.58±0.98	48.38±0.07	84.77±0.48	87.67±0.27	97.16±0.35	56.15±3.20
	Favard	30.83±5.11	36.54±2.69	35.97±1.58	70.82±0.86	40.98±3.44	48.05±0.20	79.59±9.46	86.82±0.24	97.67±0.66	(OOM)
	OptBasis	35.53±2.54	34.13±1.09	36.88±1.13	72.76±1.06	41.65±0.73	48.98±0.08	80.72±4.70	87.80±0.69	97.39±0.44	(OOM)
Bank	AdaGNN	36.10±4.05	38.62±2.19	36.77±1.23	64.62±0.89	39.39±1.48	46.41±0.58	77.02±1.94	87.73±0.36	97.17±0.42	63.99±1.00
	FBGNNI	35.88±3.40	33.20±2.22	34.66±1.72	60.35±1.52	40.36±1.71	47.70±0.48	68.67±2.35	87.62±0.31	97.63±0.14	(OOM)
	FBGNNII	39.97±4.30	34.05±1.66	36.02±1.18	58.82±2.97	42.91±0.41	45.14±2.53	72.22±0.35	82.11±0.36	97.77±0.24	(OOM)
	ACMGNNI	38.83±4.15	22.78±6.72	35.73±0.81	55.79±2.70	42.13±0.25	43.46±2.03	72.50±0.27	85.73±1.86	96.93±0.06	(OOM)
	ACMGNNII	35.04±4.34	34.69±1.42	32.50±3.91	55.20±2.08	40.04±2.83	46.94±0.37	73.96±0.30	87.44±0.37	97.08±0.25	(OOM)
	FAGNN	37.71±2.79	35.34±1.62	26.50±1.77	39.50±0.36	43.28±0.77	48.25±0.28	67.15±2.28	88.13±0.16	97.06±0.21	53.55±0.28
	G^2CN	40.45±2.75	34.63±1.12	35.76±1.22	61.45±1.89	44.35±1.35	48.29±0.46	77.26±0.65	88.37±0.27	97.43±0.15	54.13±0.04
	GNN-LF/HF	36.80±2.40	36.04±2.04	34.29±2.07	62.68±3.63	38.72±0.04	41.28±0.26	74.54±0.68	87.81±0.11	97.02±0.14	63.06±0.23
	FiGURe	34.90±1.63	39.16±1.24	33.45±1.09	62.84±1.45	41.69±0.34	44.14±3.99	83.30±0.31	88.19±0.83	97.15±0.38	(OOM)

TABLE 8.7: Effectiveness results (%) and standard deviations of spectral filters with *mini-batch* training on all datasets. For each dataset, results are highlighted based on the relative effectiveness among filters, where green results are better.

Filter	CORA	CITESEER	PUBMED	MINE	QUESTIONS	TOLOKERS	ARXIV	MAG	PRODUCTS	CHAMELEON	SQUIRREL
Identity	66.54±2.35	67.10±1.79	87.48±0.56	50.18±2.13	66.39±1.49	74.39±0.31	47.05±0.12	55.28±0.31	10.61±nan	26.62±0.04	29.10±5.13
Linear	85.29±1.27	74.66±1.11	84.96±0.50	65.64±2.16	63.13±2.83	78.84±1.44	52.58±0.63	68.24±0.21	13.06±0.92	26.59±0.01	38.09±3.13
Impulse	86.04±1.76	74.14±1.04	84.03±0.51	62.99±0.79	67.53±1.31	66.54±5.00	50.55±0.62	70.75±0.15	22.42±1.69	26.16±0.75	39.83±2.21
Monomial	86.30±1.54	74.66±1.88	89.46±0.47	58.46±5.64	73.70±2.08	81.61±1.30	50.80±0.23	70.66±0.11	32.79±0.84	26.59±0.00	37.70±2.16
PPR	87.19±1.72	75.53±1.50	88.73±0.46	81.53±2.86	65.36±1.91	80.90±1.19	50.31±0.11	70.02±0.40	17.02±3.99	26.91±0.44	40.73±2.78
HK	86.60±1.39	74.76±0.96	89.39±0.41	68.48±1.02	71.99±1.90	76.79±0.89	50.69±0.27	58.91±0.43	25.72±0.67	26.61±0.76	41.63±3.14
Gaussian	67.86±1.81	74.44±1.23	88.70±0.55	82.16±1.65	57.80±1.95	67.78±4.30	51.45±0.24	68.61±0.29	25.70±0.70	26.59±0.00	39.08±2.93
Linear	76.36±0.79	71.79±1.40	87.60±0.47	50.57±1.89	65.34±4.59	71.50±0.35	46.63±0.18	54.85±0.32	26.44±0.84	26.49±0.23	37.36±3.17
Monomial	87.41±1.24	76.26±1.35	89.83±0.53	67.66±3.43	63.67±4.81	75.35±2.85	53.62±0.41	67.05±0.47	33.11±0.37	26.82±0.42	33.99±3.79
Horner	86.64±1.63	75.33±0.80	84.17±0.46	62.14±1.32	63.35±2.36	71.22±1.21	52.47±0.30	65.53±0.63	30.04±0.47	26.59±0.00	36.40±2.74
Chebyshev	87.89±0.87	75.71±1.63	90.06±0.33	88.36±4.03	68.64±4.79	80.33±1.35	47.99±1.14	71.05±0.21	19.54±5.50	26.58±0.01	34.27±3.39
Clenshaw	80.83±2.22	72.60±0.96	75.75±1.07	54.17±2.07	63.99±1.80	75.03±1.64	41.35±0.99	54.34±0.15	29.74±0.94	26.59±0.00	35.06±2.57
ChebInterp	84.70±3.59	74.73±1.89	89.28±0.43	89.48±0.95	58.05±1.94	68.83±0.97	53.79±0.64	65.73±0.89	28.26±0.84	28.11±1.17	28.37±3.09
Bernstein	82.05±1.46	69.18±1.83	85.46±0.45	77.98±1.77	62.56±3.79	74.01±1.19	44.58±2.06	39.92±0.71	16.07±0.32	26.59±0.17	39.61±3.21
Legendre	87.43±1.00	76.17±1.02	89.84±0.29	63.32±1.81	64.14±2.24	75.78±1.02	50.53±0.20	71.65±0.13	34.74±0.45	27.10±0.36	38.70±2.49
Jacobi	87.17±1.50	75.34±1.26	89.75±0.33	89.84±0.91	71.42±3.01	76.80±0.72	53.29±0.17	71.41±0.09	32.96±0.23	24.37±3.14	38.54±3.73
OptBasis	82.20±2.46	73.11±1.23	88.23±1.10	88.56±1.13	57.70±5.70	79.62±2.38	50.81±0.68	70.46±0.24	31.29±1.68	26.56±0.06	38.09±2.54
FAGNN	86.95±1.13	75.07±1.05	85.25±0.28	73.82±0.93	62.01±2.39	77.91±0.59	53.94±0.21	70.11±0.33	35.55±0.37	26.59±0.01	39.66±2.47
G^{2}CN	86.51±1.79	75.05±1.11	84.42±0.51	73.11±1.09	72.65±2.74	69.70±6.19	51.04±0.24	67.34±0.41	29.68±0.27	26.65±0.13	40.79±3.47
GNN-LF/HF	87.36±1.02	75.95±1.35	89.76±0.39	76.77±7.55	73.70±1.52	79.35±0.95	47.65±0.45	70.51±0.06	29.41±0.22	25.80±0.91	40.66±2.14
FiGURe	87.21±1.13	76.67±1.33	89.73±0.49	57.93±3.89	70.88±1.87	77.36±3.03	52.51±0.36	71.87±0.18	34.43±1.13	26.24±1.03	39.33±4.26

Filter	ACTOR	ROMAN	RATINGS	FLICKR	YEAR	PENN94	GENIUS	TWITCH	POKEC	SNAP	WIKI
Identity	24.98±2.41	34.86±1.46	63.64±0.47	44.49±1.27	35.69±0.25	72.27±0.49	85.31±1.29	99.98±0.01	61.71±0.08	30.39±0.03	35.21±0.13
Linear	38.33±2.46	25.99±1.07	31.28±1.14	34.69±2.49	49.07±0.26	72.38±0.34	88.37±0.15	72.77±2.62	54.98±0.38	45.33±0.19	37.69±0.42
Impulse	38.94±1.78	24.32±1.60	28.63±0.87	35.81±2.37	45.23±0.15	60.52±0.49	88.38±0.34	77.50±0.67	57.84±0.11	53.56±0.18	32.88±1.47
Monomial	34.91±0.89	32.64±1.41	64.14±0.51	43.83±0.89	48.93±0.31	75.03±0.32	88.15±0.25	94.36±6.64	63.82±0.10	42.83±1.12	23.07±2.46
PPR	34.89±1.29	32.49±1.22	69.81±0.59	39.72±1.00	48.56±0.31	75.03±0.36	89.06±0.49	97.34±0.32	62.18±0.04	47.83±3.67	22.40±2.00
HK	36.37±1.94	32.88±1.39	67.82±0.49	36.66±1.16	47.50±0.30	72.77±0.76	89.10±0.14	97.79±1.10	62.04±0.08	50.99±0.16	43.06±0.30
Gaussian	35.32±1.60	32.84±1.28	65.38±1.55	34.76±2.36	48.03±0.21	73.93±0.45	88.44±0.40	99.32±0.10	62.20±0.05	52.58±0.04	42.81±0.45
Linear	34.93±1.94	35.72±0.88	63.34±0.79	36.27±0.83	35.03±0.50	74.63±0.67	86.10±0.15	89.90±5.42	62.21±0.16	30.47±0.34	38.98±0.51
Monomial	35.07±2.04	34.81±1.34	70.63±1.80	30.84±4.82	47.92±0.41	82.63±0.54	89.51±0.47	81.87±6.42	78.11±0.70	50.35±0.07	31.34±0.48
Horner	36.62±1.69	25.49±1.47	28.40±1.69	30.91±4.71	44.84±0.19	47.09±0.65	88.38±0.14	77.82±0.67	62.87±0.25	38.73±1.71	31.42±0.37
Chebyshev	34.23±1.21	29.89±1.82	35.06±2.65	37.15±2.22	45.54±0.49	83.40±0.40	86.70±0.19	99.92±0.05	78.95±0.13	42.83±0.96	24.26±2.39
Clenshaw	35.18±0.97	23.93±0.55	21.42±0.60	45.63±1.05	44.64±2.77	48.39±2.59	86.98±0.07	74.34±4.28	52.79±2.66	47.18±0.20	31.25±0.10
ChebInterp	34.62±1.98	30.36±1.22	70.95±0.69	35.58±2.22	44.28±0.29	63.59±0.70	87.65±0.38	99.71±0.45	78.60±0.68	45.77±0.56	23.85±1.84
Bernstein	35.68±2.35	30.68±1.05	53.87±1.09	31.24±2.14	37.34±1.11	79.72±0.34	88.12±0.47	97.43±0.55	73.61±0.02	48.67±0.28	24.54±1.95
Legendre	36.15±2.02	33.05±1.45	61.08±2.63	39.94±1.27	47.73±0.30	76.88±0.41	89.15±0.42	97.32±4.52	72.60±1.49	42.94±0.54	31.16±0.40
Jacobi	35.86±1.80	33.32±2.14	71.05±0.67	37.06±1.47	50.29±0.23	84.06±0.43	89.79±0.10	98.61±2.03	73.05±2.02	53.37±2.23	33.25±3.10
OptBasis	35.00±1.62	29.34±1.61	56.85±2.90	42.15±1.08	40.87±0.89	81.81±0.99	88.00±0.57	86.13±0.50	78.67±0.11	57.49±2.42	30.87±3.11
FAGNN	37.91±2.05	28.23±1.78	46.59±1.97	41.44±0.75	45.85±0.96	68.49±0.88	88.14±0.17	79.92±1.70	62.60±0.12	39.29±0.66	31.23±0.55
G^{2}CN	39.08±2.93	21.47±0.88	65.53±0.74	33.84±1.18	49.85±0.36	80.55±0.45	89.50±0.46	78.72±1.21	50.69±0.05	25.89±0.16	38.66±0.10
GNN-LF/HF	34.75±1.98	33.91±1.61	45.11±5.00	38.04±1.84	49.47±0.38	75.72±0.44	88.79±0.15	99.91±0.07	62.95±0.65	44.82±1.81	35.97±1.65
FiGURe	38.21±2.33	33.45±2.07	63.48±6.27	39.09±1.17	50.26±0.33	83.70±0.38	83.21±0.18	71.26±3.64	73.58±2.53	41.76±0.66	38.57±0.76

all scenarios. In this section, we analyze the effectiveness among the filter variants, particularly focusing on two aspects: filters under varying degrees of heterophily and filters across different categories.

RQ3: *How do the graph pattern of homophily/heterophily affect filter effectiveness?*

As introduced in Section 8.2, under *homophily*, simple filters such as Linear can leverage graph inductive bias in low-frequency components and benefit effectiveness over Identity [314]. On conventional homophilous graphs (CORA and PUBMED), it is common that a large part of filters can achieve top-tier accuracy within the error interval. This observation indicates that the graph signal is relatively easy to learn, and is in line with other recent GNN evaluations [334, 335]. With proper settings, even conventional fixed filters can deliver satisfying performance (PPR and HK). Contrarily, variable filters and filter banks (AdaGNN and FiGURe) are superior on TOLOKERS and MINESWEEPER. Although these graphs are also classified as homophilous, we deduce their graph patterns are more complex than simple low-frequency clustering, and thus demands more adaptive filters.

The scenario of *heterophily* is more challenging, as the local graph structure does not assist graph learning and entails advanced filters featuring high-frequency spectral signals, i.e., filters emphasizing θ_k for larger k-s under our polynomial formulation Eq. (8.1). Empirical results in Table 8.6 support the design intuitive, as low-pass filters (Linear, Impulse) fail on heterophilous graphs, sometimes performing even worse than Identity. Filters incorporating high-pass components are effective in learning under heterophily. This guideline applies to all filter categories: for fixed filters, this can be achieved by increasing the hop number K (Monomial) or reducing the decay factor α (PPR, Gaussian). For variable filters, it is usually beneficial to select bases that facilitate learning for high-frequency parameters (Horner, Bernstein). Meanwhile, there are also critiques that some variable bases, including Monomial and Bernstein, prioritize heterophilous accuracy and sacrifice their capability under homophily [336].

RQ4: *What is the impact of variable and filter bank designs on filter effectiveness?*

In our taxonomy Table 8.2, we identify the different designs of filter parameterization, typically discriminating fixed, variable, and filter bank designs. Our observation in Table 8.6 suggests that, different designs do not guarantee improved accuracy on *certain* graphs, but may benefit generality across *various* datasets.

In specific, the simple and fast *fixed* filters sufficiently achieve top-tier accuracy on a number of datasets, as demonstrated in RQ3. *Variable* filters offer a more flexible

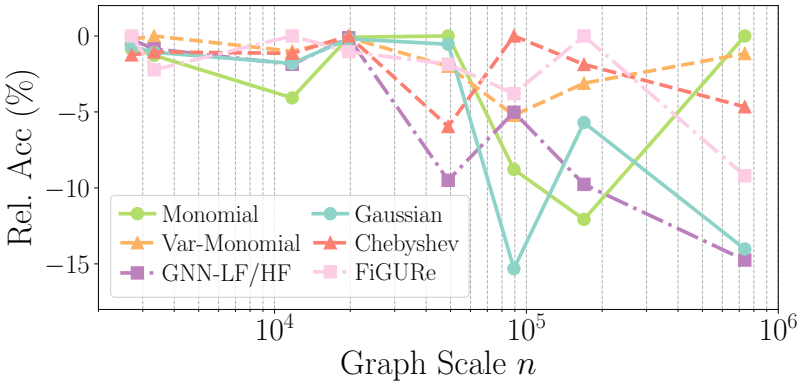


FIGURE 8.4: Shift of filter effectiveness on homophilous datasets across different scales. Effectiveness (y-axis) is presented by the relative accuracy to the highest filter in each dataset. The graph scale (x-axis) is presented by node size n in log scale.

approach for approximating a wider range of the graph spectrum. This allows for dynamically learning weight parameters from input signals, which is advantageous for capturing and leveraging richer information in the frequency domain. In cases where graph information is useful, these filters empirically produce better effectiveness, although the level of improvement varies with different data distributions. The *filter bank* design alternatively enhances model capacity by adopting multiple filters, usually spanning diverse frequency ranges. This approach is effective in mitigating the failure of a single filter and ensures reasonable accuracy in broad scenarios. On the other hand, this design does not necessarily outperform a single filter since there is no inherent improvement in filter expressiveness.

Relation to Efficiency. Then, we relate filter effectiveness and efficiency to deliver a comprehensive discussion on choosing proper filters. We first investigate the impact of learning schemes in RQ5, then discuss the relationship between effectiveness and efficiency in our core research question RQ6.

RQ5: *How do full-batch and mini-batch training schemes affect model efficacy?*

Theoretically, mini-batch computation is identical to the full-batch scheme from the aspect of spectral operations, and the only difference roots in the feature transformation procedure, as MB models do not perform the pre-transformation on input attributes before applying graph filters. *Empirically*, the majority results in Table 8.7 confirm that MB training delivers comparable accuracy to the corresponding FB results in Table 8.6, and our key observations RQ3&4 on filter effectiveness still hold. It also supports our motivation for extracting and benchmarking spectral filters, as they are generally applicable to different training schemes without affecting the capability of GNN learning.

Meanwhile, accuracy drops of MB can be observed on graphs with a small attribute dimension F_i (MINESWEEPER, TOLOKERS, RATINGS). This defect can be explained by the over-squashing phenomenon [337], which stems from the information loss when encoding the comprehensive graph topology into a small F_i during the separate filtering process. Variable filters are relatively prone to this issue since their filter parameters are not sufficiently trained under these circumstances. In contrast, MB on heterophilous datasets such as CHAMELEON and ROMAN achieves higher accuracy than full-batch models. This is precisely the opposite outcome of over-squashing, where malignant graph information is implicitly alleviated by the spectral filtering process on raw attributes.

RQ6: *What is the relationship between filter effectiveness and efficiency? Moreover, how to choose spectral filters that are both effective and efficient?*

A prevailing trend in spectral GNN studies favors more sophisticated filter designs for better effectiveness, which, as indicated by RQ1, compromise time and memory efficiency. However, RQ3 demonstrates that these two techniques lead to distinctive outcomes compared with fixed filters: Improving filter variability through *learnable parameters* can expand model capability in some cases, though the impact largely depends on the usefulness of graph signals. The design also incurs additional efficiency overhead related to the feature transformation. The *filter bank* approach usually contributes to overall filter adaptability across various graph signals, rather than improving accuracy of existing filters. This comes at the cost of multiplying the computation time and memory by the number of filters involved.

In other words, different from the common view of a straightforward trade-off between efficacy and efficiency, our benchmark study uncovers a more intricate nature: these two aspects are not mutually exclusive. The filter effectiveness is determined by its *inherent frequency response*. Even with the same variability and complexity, different filters yield varied accuracy, and filters appropriate to the graph signals results in better accuracy. Alternatively, time and memory efficiency are relevant to the *external designs* of graph computation and feature transformations, which can be explicitly inferred from their complexity. For instance, when processing heterophilous graph topology, a fixed high-order filter is more likely to achieve superior performance compared to a filter with a large parameter size focusing on low-frequency components.

Thus, balancing effectiveness and efficiency preferably requires a comprehensive consideration of graph knowledge and available environments. It is more important to find a

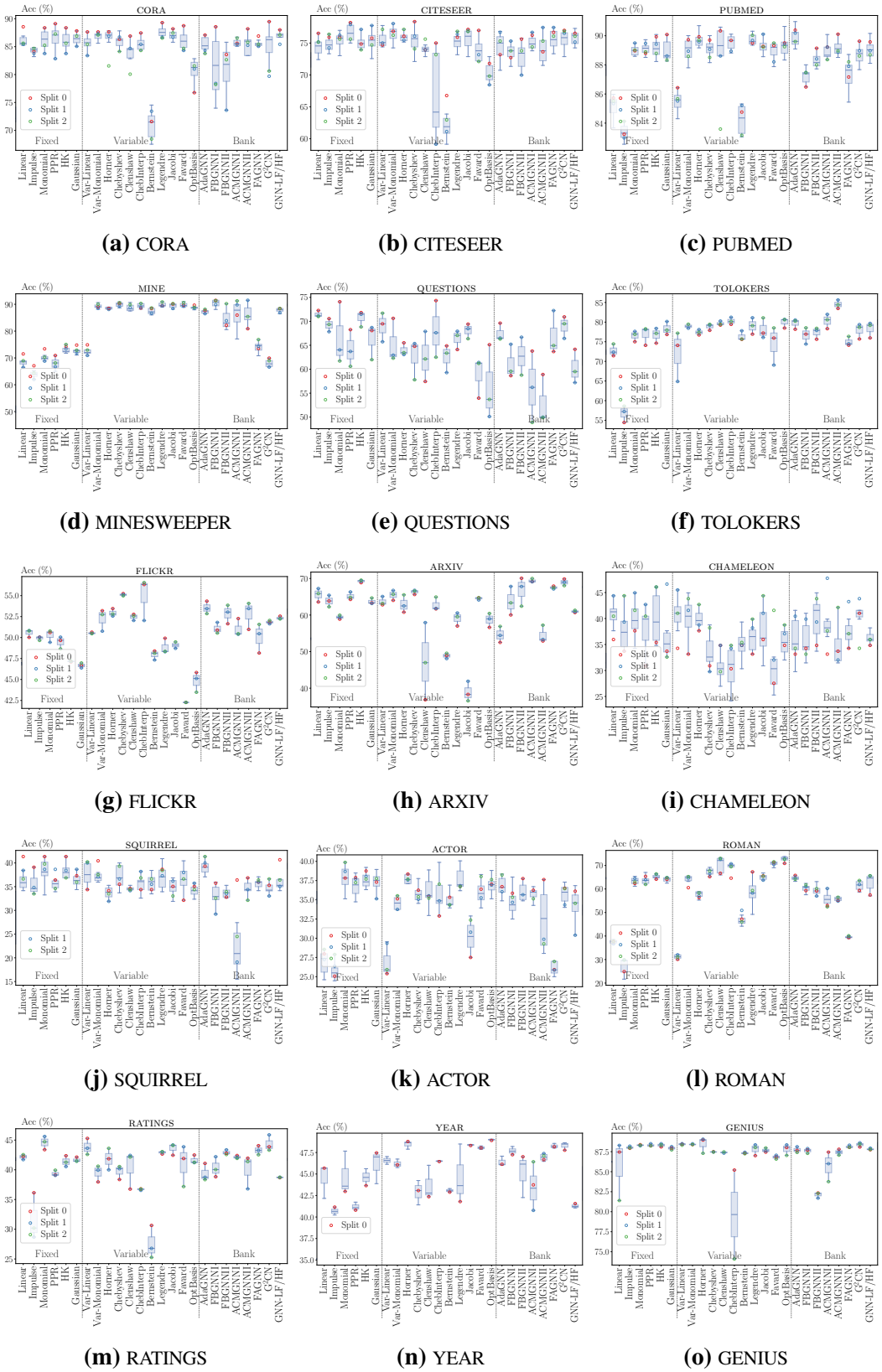
suitable spectral expression by examining the particular graph input, instead of simply exploiting sophisticated designs. Significant factors affecting learning efficacy and spectral expressiveness are thus explored more specifically in Section 8.6. We suggest the following practice as an attempt toward effective and efficient spectral GNNs: When learning on simple and homophilous graphs, one can stick to fixed filters for the best efficiency and comparably superb precision. For tasks with complex graph signals, or when fixed filters are inadequate, it is recommended to carefully choose an appropriate model that fits the graph spectrum and producing satisfactory performance, while ensuring that the time and memory overheads remain feasible in the given environment.

8.5.3 Result Stability

In this section, we especially investigate the statistical significance of our evaluation on filter effectiveness and efficiency, ensuring that our conclusions are widely applicable to general circumstances.

Efficacy Variance and Divergence. We further visualize the confidence intervals of filter accuracy in Table 8.6 by box plot in Figure 8.5 with selected seeds. Particularly, CORA and ARXIV represent the datasets with random and attribute-based splits, respectively, and all filters learn on the same split for the same seed. Figure 8.5(a) implies that the variance in datasets like CORA is largely caused by the split difference, as some seeds lead to high accuracy for most filters while others greatly impede efficacy. This is a common phenomenon in semi-supervised learning, where random splits may not be representative of containing sufficient information, such as lacking minor label groups, which results in large accuracy deviation. On ARXIV, where the splits are more stable, the filter accuracy is more concentrated. Nonetheless, in both cases, the relative effectiveness among filters can be effectively depicted by average accuracy, which supports our RQ3&4 based on Table 8.6.

Our scalable implementation offer an unprecedented opportunity to study filter efficacy across graphs with different scales, which is showcased in Figure 8.4 with representatives in three categories. Together with Figure 8.5, we observe that the relative difference among filters is more significant on larger graphs. On small-scale graphs (Figure 8.5(a)), the accuracy difference among filters are marginal, and a broad range of filters can achieve high accuracy. However, filter effectiveness becomes more divergent on larger graphs where $n > 10^5$ (Figure 8.5(h)). While filters with suitable frequency responses,



(a) CORA **(b)** CITESEER **(c)** PUBMED

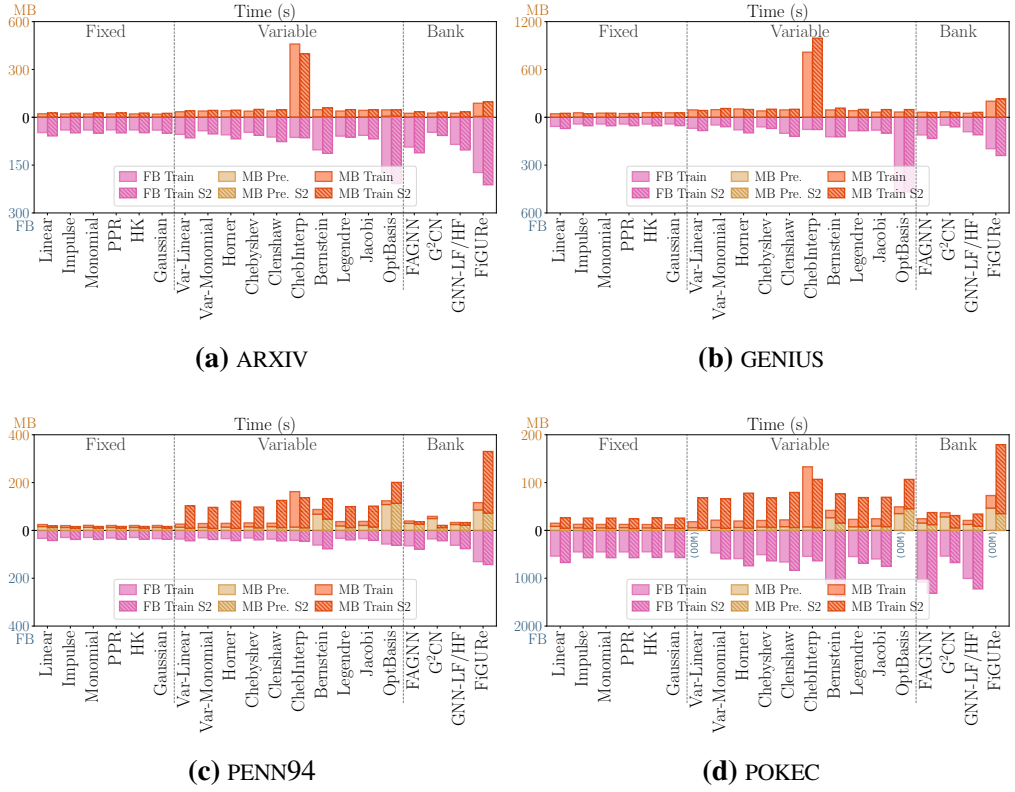
(d) MINESWEEPER **(e)** QUESTIONS **(f)** TOLOKERS

(g) FLICKR **(h)** ARXIV **(i)** CHAMELEON

(j) SQUIRREL **(k)** ACTOR **(l)** ROMAN

(m) RATINGS **(n)** YEAR **(o)** GENIUS

FIGURE 8.5: Statistical significance of filter effectiveness on 8 homophilous and 7 heterophilous datasets. Parameter initialization and dataset split for all filters follow the same set of random seeds. Results from three seeds are marked in the figure.



(a) ARXIV **(b)** GENIUS

(c) PENN94 **(d)** POKEC

FIGURE 8.6: Time efficiency comparison of FB (lower axis) and MB (upper axis) training on medium- and large-scale datasets with different hardware.

e.g. the fixed Monomial, can still achieve leading performance on corresponding graphs, inappropriate ones exhibit larger accuracy gaps from the best filters. The finding further underscores the importance of our conclusion RQ3 especially on large-scale graphs, that choosing filters that fit the graph spectrum is critical for effectiveness.

Efficiency on Different Hardware. To validate our efficiency observations on diverse hardware platforms, we evaluate the filter efficiency on another server marked as S2, which is with slower CPUs (Intel Xeon, 2.2GHz) and a faster GPU (NVIDIA RTX A5000). The time breakdown on the typical dataset PENN94 is in Figure 8.6. Notably, the figure demonstrates varied bottlenecks for different filter types due to the hardware difference. For MB fixed filters with transformation being the bottleneck, the overall learning time on S2 is shorter thanks to faster GPU computation. In comparison, as graph propagation dominates the efficiency of FB training and MB variable filters, the empirical speed is relatively slower. The slowdown is more significant on datasets larger than PENN94. The observation aligns with RQ1&2 regarding the efficiency of model operations and learning schemes, verifying that they are generally applicable irrespective of hardware settings.

8.5.4 Key Conclusions

C1. Model time and memory efficiency are respectively dominated by graph propagation and weight transformation operations as indicated by our taxonomy. On graphs above million-scale, propagation turns into the bottleneck of GNN training. (RQ1)

C2. Mini-batch training is a unique learning scheme of spectral GNN models, offering comparable efficacy and better scalability, especially on large graphs. In turn, it falls short in flexibility and is more sensitive to raw graph attributes. (RQ2&5)

C3. Model effectiveness mainly stems from the inherit filter property of frequency response, i.e., the emphasis on low- and high-frequency signals. With proper configurations, simple filters can also excel on suitable graphs. (RQ3)

C4. While sophisticated filter designs are favored in common consensus, they are less related to accuracy improvement, but potentially benefit generality across datasets. Variable filters increase the capacity to deal complex input signals, while filter banks can assemble the performance of individual filters. (RQ4)

C5. Contrary to the prevailing belief, we reveal that efficacy and efficiency are not mutually exclusive. We provide our practical guidelines for analyzing the graph spectrum and choosing appropriate filters to achieve both efficacy and efficiency. (RQ6)

8.6 Results: Specific Evaluations

In this section, we conduct in-depth evaluations regarding specific properties of individual spectral filters, offering ancillary view on obtaining desired filter effectiveness and efficiency. Further research questions are raised for evaluations leading to new findings.

8.6.1 Extended Tasks

In this section, we highlight the generalizability of our implementation and evaluation by comparing to other graph processing methods and performing tasks other than node classification.

Implementation Comparison. To demonstrate the performance of our implementation, especially regarding GNN efficiency and scalability, we conduct additional evaluation for typical GNN models deployed in other popular frameworks. The baselines include spatial message-passing GNNs (GraphSAGE [27]), spectral message-passing GNNs (GCN [25], ChebNet [26]), as well as scalability-oriented Graph Transformers (NAGphormer [184], ANS-GT [173]).

As comparison with other implementations, we evaluate these spatial, spectral, and sequential models in the PyG framework [305] in Table 8.8. The models are evaluated with iterative architecture $K = J = 4$ under available PyG graph backends. Spatial and spectral models are in full-batch training, as PyG does not provide the mini-batch scheme. Graph Transformers are learned in mini-batch training with random sampling. Especially, the EI backend is the most common backend used in PyG by default with $O(mF)$ space footprint for graph propagation. The SP backend is more efficient and used as the default backend in our main experiments, while only a handful of PyG models are compatible with it.

TABLE 8.8: Effectiveness and efficiency of models outside our framework on representative datasets. "Train" and "Infer" respectively refer to average training time per epoch and inference time (s), while precomputation time is separated in applicable models.

Model (Backend*)	ARXIV				MAG			
	Acc	Train	Infer	GPU	Acc	Train	Infer	GPU
GCN (SP)	53.2	0.05	0.03	1.1	27.4	0.31	0.11	7.8
GraphSAGE (SP)	50.3	0.04	0.005	1.2	24.8	0.25	0.01	7.7
GCN (EI)	53.0	0.06	0.06	3.2		(OOM)		
GraphSAGE (EI)	54.1	0.09	0.03	2.7	28.2	0.33	0.18	10.8
ChebNet (EI)	53.4	0.11	0.05	3.0		(OOM)		
NAGphormer (EI)	67.8	280+3.2	2.1	2.3	33.2	89+10.3	2.2	3.8
ANS-GT (EI)	71.1	16205+109	2.7	11.3		(OOM)		
Model (Backend)	PENN94				POKEC			
GCN (SP)	73.1	0.04	0.03	1.3	60.4	663.3	0.35	0.01
GraphSAGE (SP)	74.2	0.08	0.003	2.3	63.4	0.45	0.009	9.6
GCN (EI)	67.6	0.06	0.06	3.7		(OOM)		
GraphSAGE (EI)		(OOM)				(OOM)		
ChebNet (EI)		(OOM)				(OOM)		
NAGphormer (EI)	74.4	237+6.1	2.1	2.3	73.1	70+16.1	3.1	8.9
ANS-GT (EI)	67.8	34092+37	5.0	8.7		(OOM)		

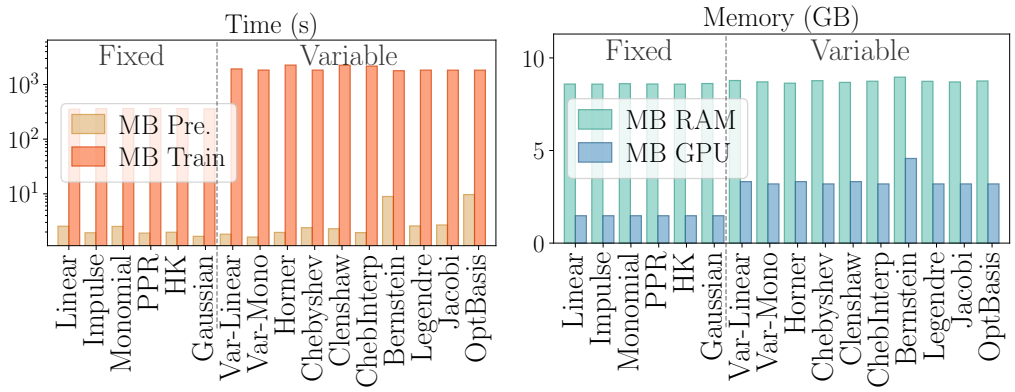
* **Backend**: SP = `torch.sparse`, EI = `torch_geometric.EdgeIndex`.

It can be observed that most accuracy are in line with our results in Table 8.6. Regarding efficiency and scalability, message-passing models with the SP backend is more superior than EI for faster training and less GPU memory footprint, while the most common EI backend encounters OOM on large datasets. Graph Transformers are more computational intensive for demanding significantly long precomputation, slower training speed, and more memory overhead due to the complicated model architecture.

Link Prediction. Link prediction is another popular task that presents substantial challenges regarding GNN scalability [338, 339]. Compared to the node classification task in Section 8.5, link prediction focuses on transformation operations, as identifying node-pair correlation through neural networks is critical for prediction accuracy. Graph information retrieved by the filters only serve as fundamental knowledge for the downstream transformation. Different from Table 8.3, the typical complexity for transformation is $O(\kappa m F^2)$. This is because for a graph with m ground-truth edges, the model needs to processes a total of κm positive and negative samples, where κ is usually $2-10$. Hence, performing

TABLE 8.9: Effectiveness results (%) of mini-batch filters for link prediction. For each dataset, results are highlighted based on the relative effectiveness among filters, where green results are better.

Type	Dataset	DDI	COLLAB	PPA	CITATION2
	Node n	4,267	235,868	576,289	2,926,620
	Edge m	1,334,889	1,285,465	30,326,273	30,561,187
	Metric	Hits@20	Hits@50	Hits@100	MRR
Fixed	Linear	14.39	23.88	7.49	3.31
	Impulse	10.37	22.19	26.77	14.24
	Monomial	13.04	26.47	22.15	45.62
	PPR	11.11	23.91	22.08	20.51
	HK	16.77	23.27	21.15	69.92
	Gaussian	25.65	28.06	4.76	61.09
Variable	Linear	12.26	4.20	1.35	21.63
	Monomial	40.68	31.37	19.18	64.80
	Horner	13.27	30.29	26.75	38.99
	Chebyshev	18.66	17.86	9.04	69.67
	Clenshaw	5.43	18.80	14.92	44.40
	ChebInterp	34.59	35.71	18.02	58.63
	Bernstein	21.06	3.42	1.16	9.97
	Legendre	22.59	19.24	14.53	70.08
	Jacobi	31.16	7.44	12.26	67.56
	OptBasis	24.18	42.62	7.62	3.11



(a) Time **(b)** Memory

FIGURE 8.7: Filter time and memory efficiency of mini-batch link prediction on PPA. Note that the time axis is on log scale.

the link prediction task inevitably entails mini-batch training, as the full-scale memory overhead is prohibitive.

The generalizability of our framework allows for utilizing the same set of spectral filters to acquire graph representation in MB scheme. We do not attempt to design complicated transformation networks and stick to a simple MLP network, while more sophisticated downstream modules can be integrated in a plug-and-play manner as introduced in Section 8.4. The OGBL datasets [43], including some of the largest available link prediction graphs, are employed for evaluation. We similarly unify the experiment settings and tune the hyperparameters as in Section 8.4.

Effectiveness results are shown in Table 8.9, while representative efficiency evaluations are in Figure 8.7. Due to the intricate transformation parameters, some filters typically fail to achieve reasonable results on most datasets (Linear, Bernstein). Our RQ3&4 still hold, that filters suitable for graph signals, even some simple filters (HK, Gaussian) can achieve satisfying performance. On COLLAB and CITATION2, this suggests low-frequency components are effective, whereas the opposite is true for the other two datasets. Different from RQ1, link prediction efficiency is dominated by the transformation operation on larger graphs due to the considerable amount of iterative edge-wise computations, while GPU memory can be controlled by the batch size. Nonetheless, such computational bottleneck is less related to the topic of graph processing, and a range of dedicated accelerations are available [91, 92].

Signal Regression. Spectral filtering also offers a regression task for fitting a given signal encoded in the graph, which is intuitive in characterizing the frequency response to

signals spanning diverse frequency components. We define the fully-supervised graph regression task [325] by learning from the predefined pair of input $\mathbf{x}$ and the objective $z = g^\star * x$, as an approach to approximate the given simple spectral filter function $g^\star$. Table 8.10 presents the average R^2 score of five typical signal functions, where larger scores indicate better regression precision.

The observation implies that most filters still highly focus on low-frequency components, demonstrating higher precision on LOW and BAND REJECT. This is in line with our findings in RQ4 that introducing variable bases does not necessarily strengthen filter capability, as the performance is principally determined by its spectral formulation. On the contrary, filters enforcing higher attention on high-frequency domains, including Monomial, Horner, and OptBasis, achieve strong performance on high-frequency signals (BAND, COMBINE, and HIGH), although Monomial and Horner sacrifice low-frequency ability to certain degrees. OptBasis outperforms on all signals thanks to its flexible parameter acquisition. However, it should be noted that the regression precision of simple signals does not guarantee node classification effectiveness in our main experiment, as better utilization of graph signals is not always beneficial due to the complex grounding of the realistic task.

TABLE 8.10: Average R^2 scores of graph regression on five signal functions. For each dataset, results are highlighted based on the relative effectiveness among filters, where green results are better.

Type	Dataset / Signal	BAND $e^{-10(\lambda-1)^2}$	COMBINE $\lvert\sin(\pi\lambda)\rvert$	HIGH $1-e^{-10\lambda^2}$	LOW $e^{-10\lambda^2}$	REJECT $1-e^{-10(\lambda-1)^2}$
Fixed	PPR	21.40	32.13	35.42	77.58	91.24
	Linear	6.74	10.12	8.87	94.90	83.22
	Impulse	6.73	9.99	8.79	93.27	82.25
	Monomial	21.40	32.13	35.42	77.58	91.24
	HK	7.57	11.77	10.23	96.93	86.94
	Gaussian	7.62	11.96	10.26	96.92	86.93
Variable	Monomial	6.87	10.21	8.87	94.80	79.75
	Horner	48.98	69.10	78.87	89.14	78.96
	Chebyshev	6.74	8.40	6.87	91.15	81.12
	Clenshaw	6.77	8.06	8.14	76.68	81.49
	ChebInterp	6.72	9.58	8.88	90.42	80.95
	Bernstein	13.03	17.79	22.10	97.56	86.08
	Legendre	6.68	9.82	8.59	94.00	72.86
	Jacobi	6.69	9.73	8.62	93.99	82.14
	Favard	5.23	8.89	7.46	67.57	67.68
	OptBasis	82.88	79.73	93.69	99.19	99.06

8.6.2 Spectral Capabilities

As analyzed in RQ3, the effectiveness of graph learning with different filters is related to the compatibility between their spectral expressions and the graph data. Therefore, we delve into the spectral properties to provide a thorough understanding of the spectral capabilities concerning different filter formulations.

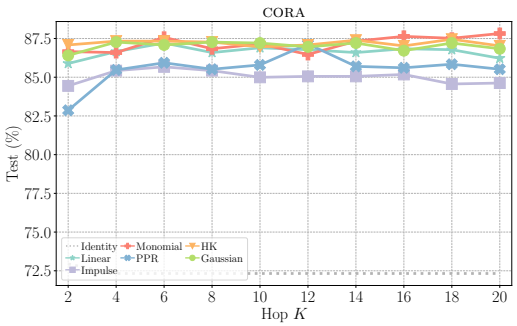
RQ7: *What inherent spectral properties are impactful in determining filter effectiveness?*

Effect of Propagation Hop. The number of propagation hops K is a critical hyperparameter governing both spectral expressiveness and empirical efficiency. Deciding the value of K requires careful consideration in the spectral domain. A small K indicates a limited number of polynomial terms in Eq. (8.1), leading to restricted filter capability. Conversely, a large number of propagations may introduce excessive graph information, which also implies linearly increased computational overhead.

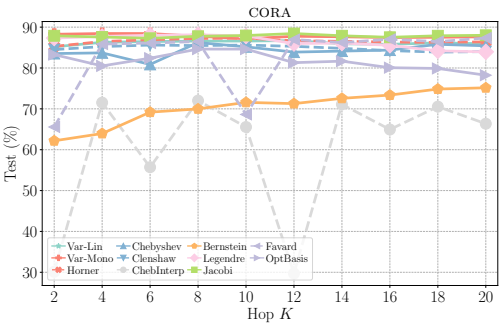
The value of K in existing literature varies significantly due to non-spectral designs, complicating the assessment of the filter capability. Our framework offers a unified architecture for spectral GNNs, enabling a more thorough examination of the impact of propagation hops on graph filters. Figures 8.8 and 8.9 illustrate the model accuracy patterns when varying the number of hops in common the range $K \in [2, 20]$ on homophilous and heterophilous graphs. Generally, a limited K is more favorable, especially for homophilous graphs. Overall, our selection of $K = 10$ in the main experiments is reasonable for maintaining the performance of most models across various datasets.

For low-pass filters (Linear and Impulse), the efficacy gradually decreases when the hop number increases for both homophilous and heterophilous graphs. This corresponds to the over-smoothing issue [149, 340, 138], where the signal is overwhelm by non-local noise and loses useful information. Filters such as PPR and Gaussian can alleviate this issue by tuning the decaying factor, while filters with high-frequency components, especially those with orthogonal basis, are also more stable by manipulating those signals.

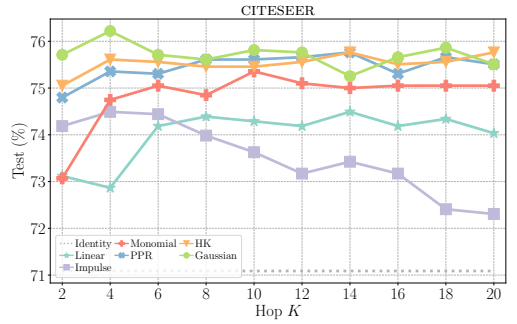
Clustering Visualization. The t-SNE method provides an intuitive way to understand the decision boundary learned by spectral GNNs [341], which is essentially related to their prediction performance in Table 8.6. Figures 8.10 and 8.11 showcases visualizations of representative filters. Generally, embeddings for most filters on the homophilous CORA are well-clustered with sharp boundaries, explaining the similar classification accuracy.



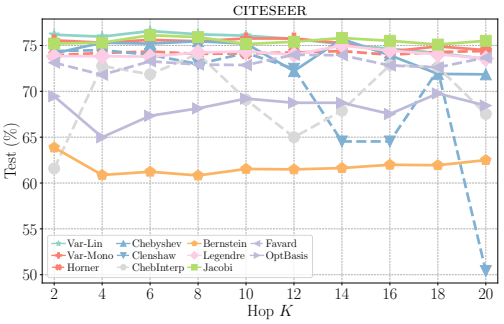
(a) Fixed filters on CORA



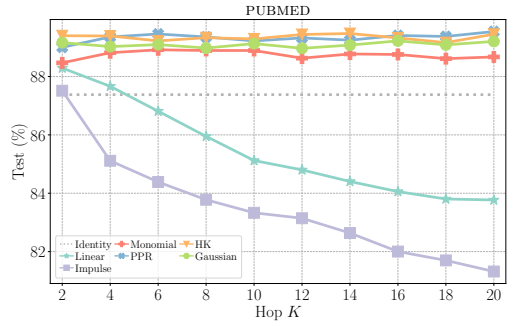
(b) Variable filters on CORA



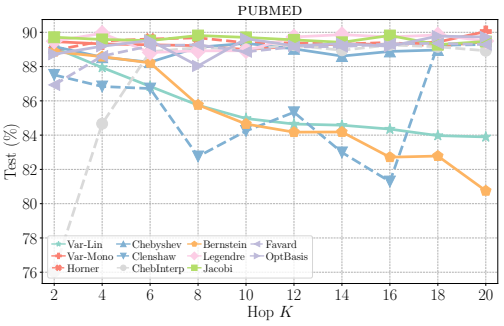
(c) Fixed filters on CITESEER



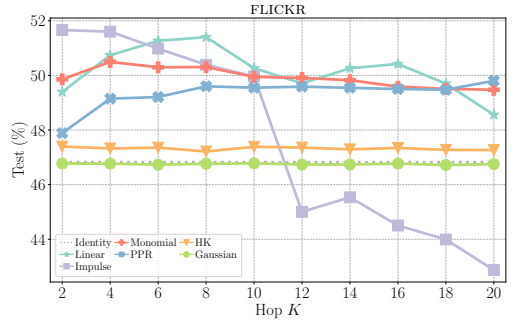
(d) Variable filters on CITESEER



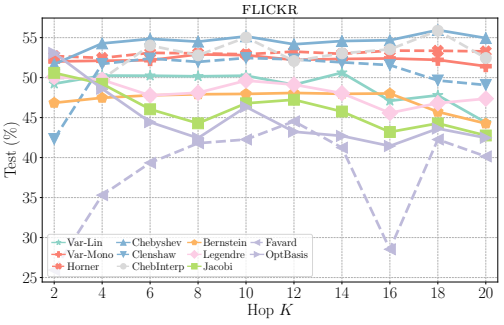
(e) Fixed filters on PUBMED



(f) Variable filters on PUBMED

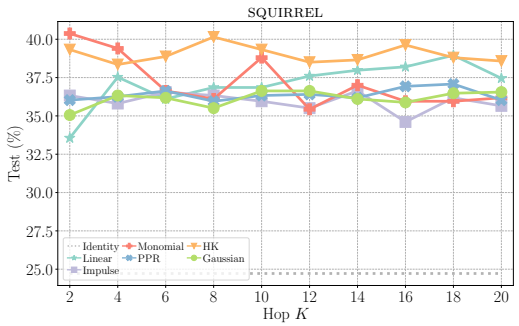


(g) Fixed filters on FLICKR

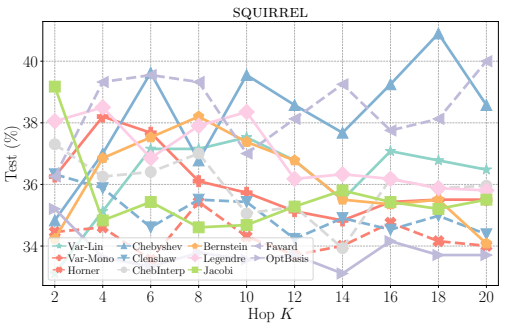


(h) Variable filters on FLICKR

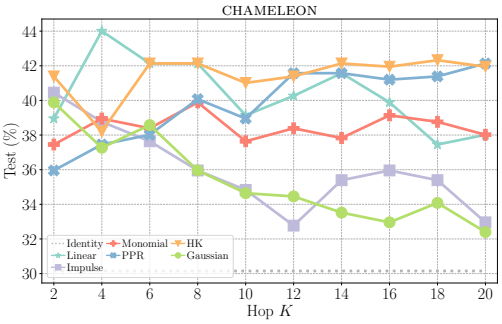
FIGURE 8.8: Effect of *propagation hops* K of full-batch fixed and variable filters on 4 homophilous datasets.



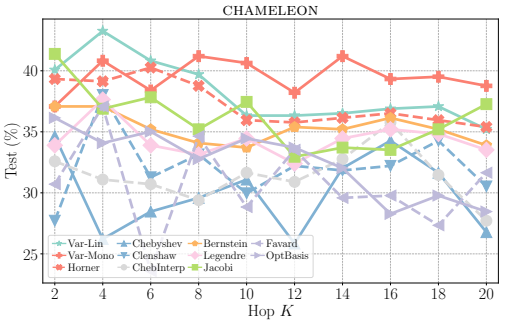
(a) Fixed filters on SQUIRREL



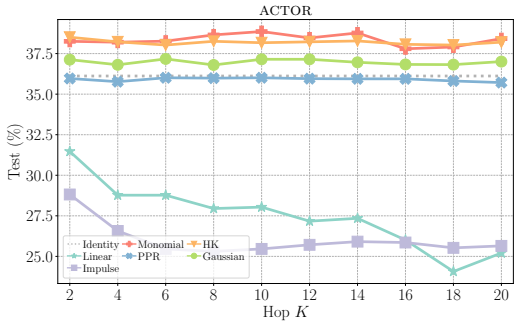
(b) Variable filters on SQUIRREL



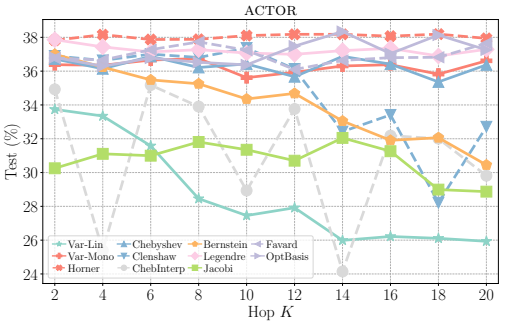
(c) Fixed filters on CHAMELEON



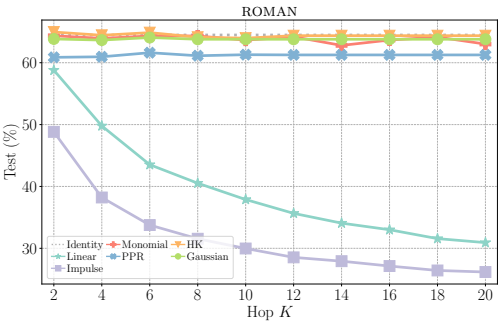
(d) Variable filters on CHAMELEON



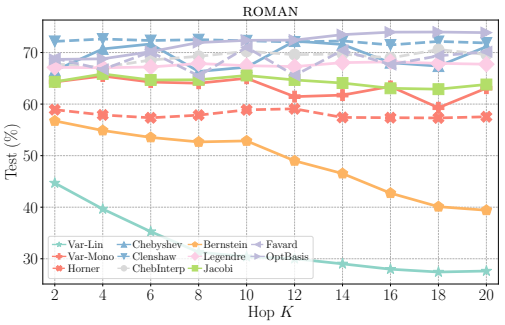
(e) Fixed filters on ACTOR



(f) Variable filters on ACTOR

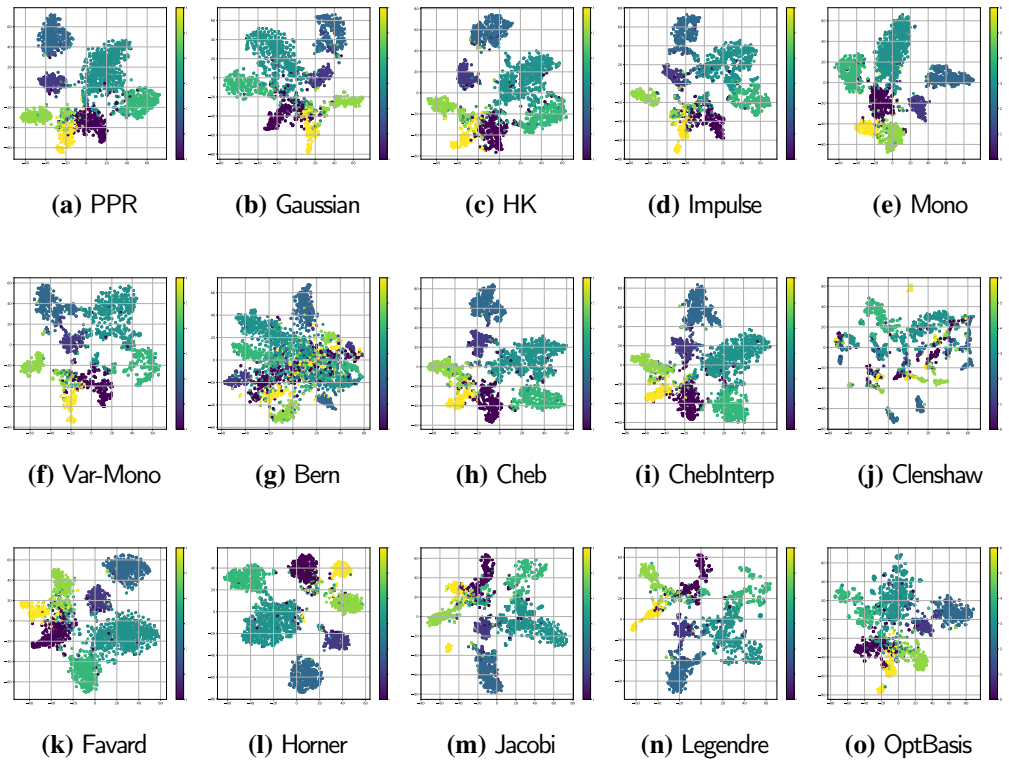


(g) Fixed filters on ROMAN



(h) Variable filters on ROMAN

FIGURE 8.9: Effect of *propagation hops K* of full-batch fixed and variable filters on 4 heterophilous datasets.

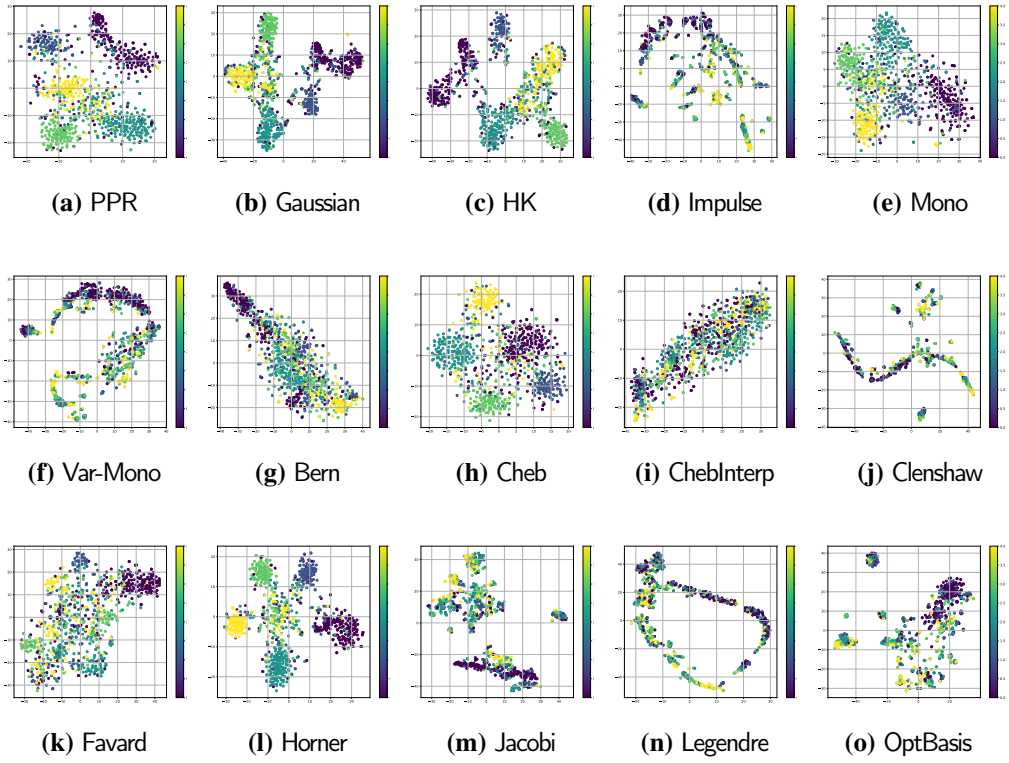


(a) PPR **(b)** Gaussian **(c)** HK **(d)** Impulse **(e)** Mono

(f) Var-Mono **(g)** Bern **(h)** Cheb **(i)** ChebInterp **(j)** Clenshaw

(k) Favard **(l)** Horner **(m)** Jacobi **(n)** Legendre **(o)** OptBasis

FIGURE 8.10: Clusters of different filters on CORA.



(a) PPR **(b)** Gaussian **(c)** HK **(d)** Impulse **(e)** Mono

(f) Var-Mono **(g)** Bern **(h)** Cheb **(i)** ChebInterp **(j)** Clenshaw

(k) Favard **(l)** Horner **(m)** Jacobi **(n)** Legendre **(o)** OptBasis

FIGURE 8.11: Clusters of different filters on CHAMELEON.

On the heterophilous CHAMELEON, a few filters are able to maintain the ability to form clear clusters and consequently satisfy prediction performance (Monomial, Chebyshev). In comparison, the dispersed and overlapping clusters of PPR and ChebInterp correspond to the noise introduced by heterophilous graph topology, which undermines their effectiveness. Filters such as Impulse and Jacobi generate scattered clusters with notable outliers, rendering difficulties in fitting graph signals and producing meaningful representations. In summary, the evaluation also implies that spectral expressiveness outweighs variable designs in terms of filter efficacy when conducting classification tasks under different graph patterns.

8.6.3 Degree-specific Effectiveness

A recent trend examines node-wise performance and discovers that the learning effectiveness of vanilla GNNs varies with respect to nodes of differing degree levels [342, 138, 343, 344, 345, 346]. In this benchmark, we are motivated to offer a preliminary investigation into the degree-specific performance of spectral GNNs, including the causative factors and impacts of the phenomenon.

Relation with Filter Effectiveness. Unlike previous investigations assuming homophily, we extend the degree-wise evaluation to heterophilous graphs. Figure 8.12 displays the difference between prediction accuracy on high- and low-degree nodes across representative datasets. The extended experimental evaluation enables us to relate the degree-specific performance to overall model accuracy in Section 8.5.2.

RQ8: *What is the degree-wise effectiveness under homophily and heterophily, and how does it affect the filter efficacy?*

It can be observed in Figure 8.12 that most filters behave distinctively on homophilous and heterophilous datasets. On *homophilous* graphs (CITESEER in Figure 8.13(a)), the performance of high-degree nodes is generally on par with or higher than low-degree ones, which echoes earlier studies. From the spectral domain view, high-degree nodes are commonly located in clusters, which is more related to low frequency in the graph spectrum and is preferable for homophilous GNNs throughout learning. Contrarily, the accuracy of high-degree nodes is usually significantly lower under *heterophily*, as shown in Figure 8.13(b) for ROMAN. In this case, the hypothesis that higher degrees are naturally favored by GNNs no longer holds true. Although these nodes aggregate more information from the neighborhood, it is not necessarily beneficial, and heterophilous connections

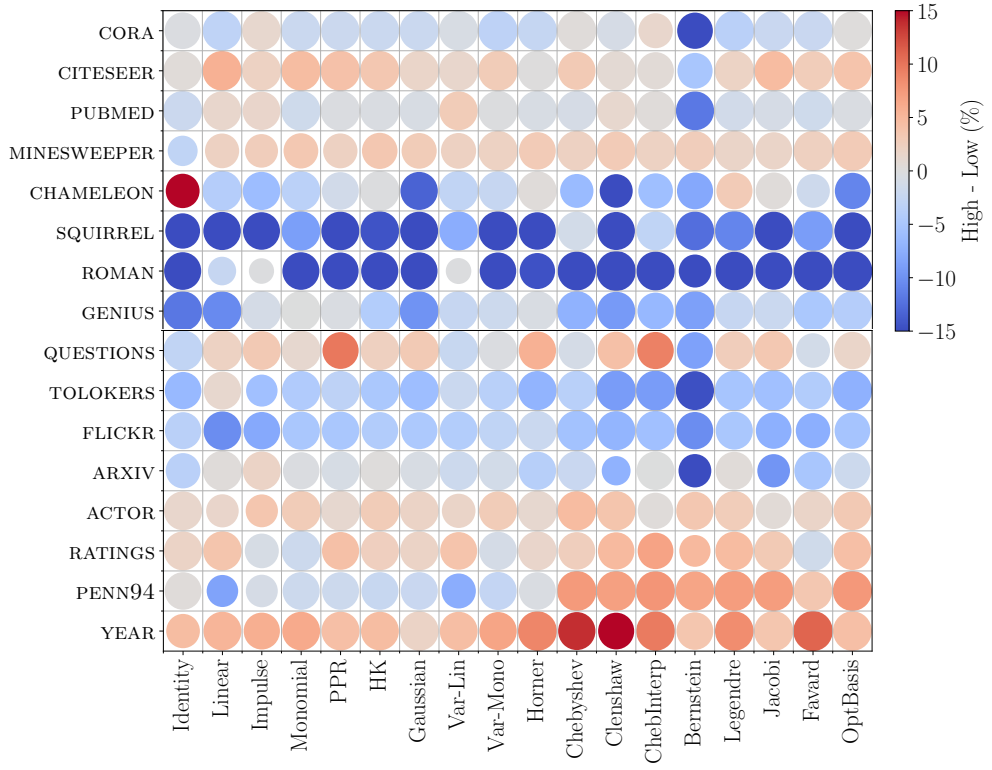
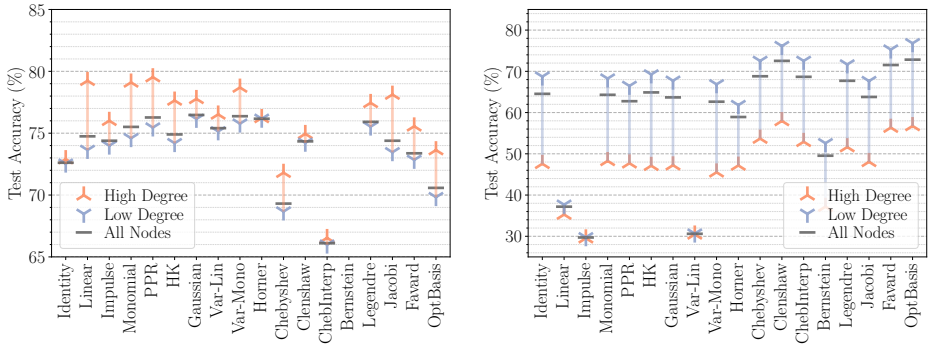


FIGURE 8.12: Accuracy gap between high- and low-degree nodes across filters on homophilous and heterophilous graphs. Color of each data point indicates the difference value, while size of the circle represents the relative overall accuracy among all filters in each dataset.



(a) CITESEER **(b)** ROMAN

FIGURE 8.13: Respective accuracy of high- and low-degree nodes on **(a)** CITESEER and **(b)** ROMAN.

may carry destructive bias and hinder the prediction. As a more precise amendment to the conclusion in prior works, we state that the degree-wise bias is more *sensitive*, but not necessarily beneficial, to high-degree nodes with regard to varying graph conditions.

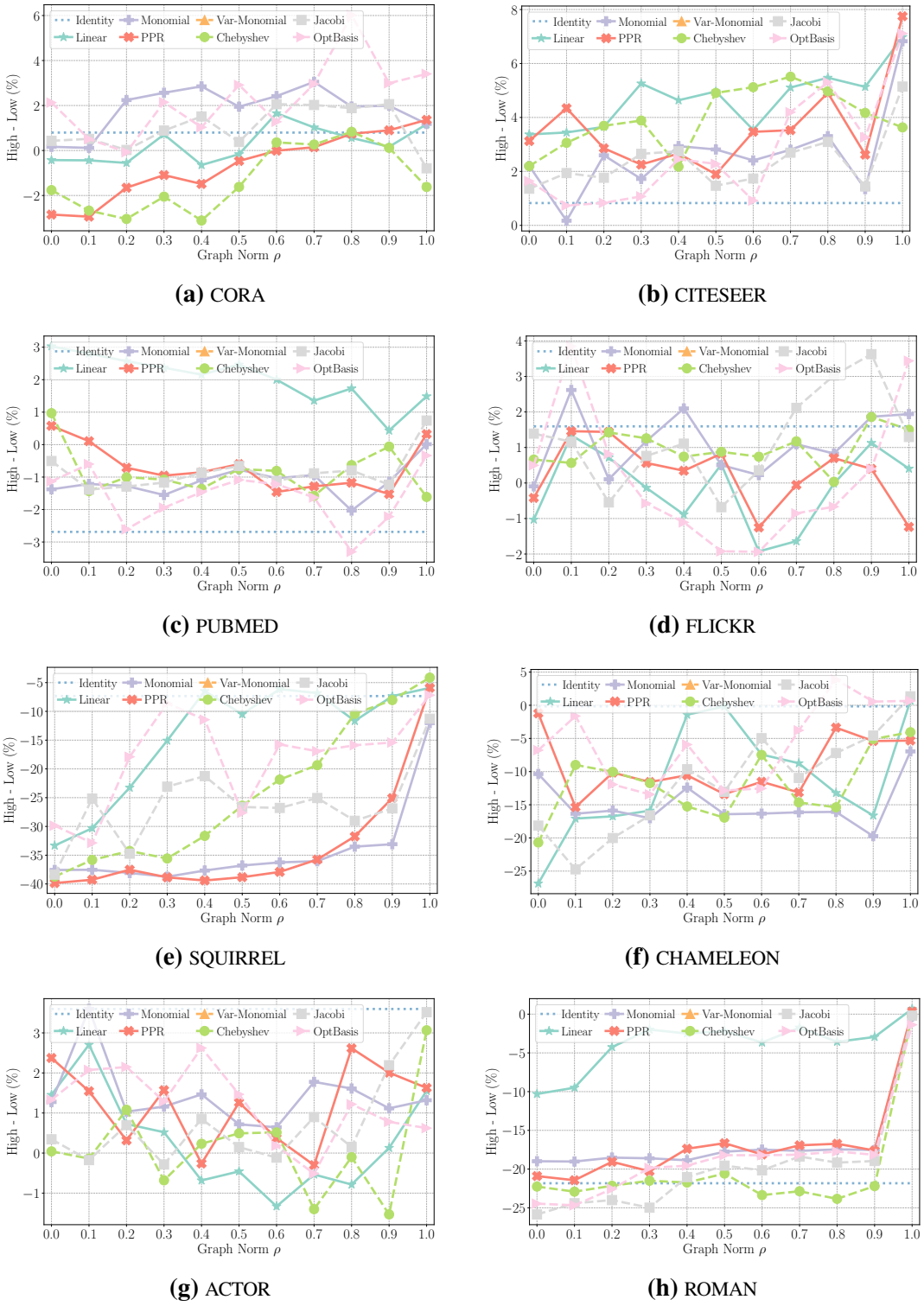
It is also notable in Figure 8.12 that the degree-specific difference is correlated with the test accuracy of models within each dataset, especially under heterophily. For example, in Figure 8.13(b) ROMAN, filters with greater bias achieve relatively better overall accuracy. In comparison, filters such as Linear and Impulse fail to distinguish between high- and low-degree nodes, resulting in lower test accuracy. From the observation, we deduce that GNNs tend to recognize and adapt to the majority of low-degree nodes in order to achieve higher performance. This preference is potentially exaggerated by the heterophily-oriented filter design, which pays more attention to the high-frequency components correlating to low-degree nodes while compromising the performance of high-degree nodes. It is thus advisable to explore filter formulations balancing different frequency ranges, which are promising for addressing the current drawback and further advancing overall model accuracy.

Effect of Graph Normalization. Recall that the normalized adjacency $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{D}}^{\rho-1}\bar{\boldsymbol{A}}\bar{\boldsymbol{D}}^{-\rho}$ explicitly encodes degree information during propagation, we are thence motivated to use it to impact filter effectiveness.

RQ9: *How to utilize the graph information to control degree-wise effectiveness, and therefore affect overall accuracy?*

In previous studies, it is revealed that the normalization in form of $\tilde{\boldsymbol{A}} = \bar{\boldsymbol{D}}^{-1}\bar{\boldsymbol{A}}$ affects the node-wise performance on homophilous graphs [347, 345], while we extend to the continuous range of $\rho \in [0, 1]$. Specially, $\rho = 1/2$ is the symmetric normalization with equal contributions from both in- and out-edges.

We present experimental result of the accuracy gap between high- and low-degree nodes when varying the graph normalization hyperparameter ρ in Figure 8.14. It can be observed that a larger ρ increases the difference value, i.e., improves the relative accuracy of high-degree nodes for both fixed and variable filters on CITESEER and ROMAN. This suggests that inbound information is preferred for model inference on high-degree nodes, and adjusting the aggregation through ρ could be an practical attempt to alleviate degree-wise differences and achieve favorable performance. On graphs such as CHAMELEON and ACTOR, where connection utility is hindered by heterophily, this pattern is weaker, and the performance gap becomes more unstable due to the complexity of graph conditions.



(a) CORA **(b)** CITESEER

(c) PUBMED **(d)** FLICKR

(e) SQUIRREL **(f)** CHAMELEON

(g) ACTOR **(h)** ROMAN

FIGURE 8.14: Effect of *graph normalization* ρ on the accuracy difference between high- and low-degree nodes of selected full-batch fixed and variable filters on 4 homophilous and 4 heterophilous datasets.

8.6.4 Key Conclusions

C6. Filter effectiveness can be explained by their spectral properties, especially the response on different frequencies. Sophisticated designs are only effective when their frequency components match the graph signal. (RQ7)

C7. Spectral models exhibit dissimilar effectiveness on nodes of high and low degrees under different graph conditions, challenging the assumptions in previous studies. The efficacy on high-degree nodes degrades under heterophily, potentially indicating greater sensitivity to the graph learning process on these nodes. (RQ8)

C8. The degree-wise bias is also relevant to the overall effectiveness of filters and can be controlled by altering the influence of graph topology. Dedicated filtering designs that benefit both low- and high-degree performance remain underexplored. (RQ9)

8.7 Summary and Discussion

In this study, we conduct extensive evaluations of spectral GNNs concerning both effectiveness and efficiency. We first provide a thorough analysis of the graph kernels in existing GNNs and categorize them by spectral designs. These filters are then implemented under a unified framework with highly efficient learning schemes. Comprehensive experiments are conducted to assess the model performance, with discussions covering heterophily, graph scales, spectral properties, and degree-wise bias.

Our benchmark observations also identify several open questions worthy of further research toward better spectral GNNs:

- *How to properly obtain effective and efficient spectral filters?* As analyzed in RQ6, it is possible to achieve both effectiveness and efficiency for some graphs, while finding the suitable filter is largely empirical. Our observation suggests that despite heterophily, latent patterns of graph data distribution also affect the spectral performance. Identifying and characterizing these factors is beneficial for designing more powerful filters based on graph knowledge.
- *How to further improve filter efficiency and scalability?* In RQ1, the spectral GNN design outlines a solution for performing learning on large graphs, although

existing variable and composed filter computations are far from optimized. The GNN scalability issue calls for a persistent pursuit of more efficient designs, which is of great practical interest. Our study is helpful in uncovering impact factors and bottlenecks of large-scale deployment, and underscores the potential of spectral GNNs for achieving efficiency without compromising efficacy.

- *How to address the degree-specific bias to enhance overall effectiveness?* In RQ8, we discover that the degree-wise difference is related to graph heterophily and affects model precision. As current endeavors to mitigate GNN bias largely assume homophily, it is promising to search for specialized schemes that improve the accuracy of both low- and high-degree nodes, considering various graph conditions, which also benefits overall model performance.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

Recent advances in data processing have stimulated the demand for learning graphs of very large scales. In this work, we present 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 different GNN variants.

Chapters 3 and 4 constitutes the first part of this thesis, where we investigate and enhance convolutional GNNs with novel graph algorithms. In Chapter 3, we propose SCARA, a scalable Graph Neural Network algorithm with feature-oriented optimizations. Our theoretical contribution includes showing SCARA has a sub-linear complexity that efficiently scales-up the graph propagation by two algorithms, namely FEATURE-PUSH and FEATURE-REUSE. In Chapter 4, we propose LD^2, a scalable GNN design for heterophilous graphs, that 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 the 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 a complexity linear to $O(n)$, and GPU memory independence from the graph size n and m.

In the second part encompassing Chapters 5 and 6, we incorporate the idea of scaling up graph computations to broader graph learning variants. In Chapter 5, we present HubGT for leveraging decoupled graph hierarchy by hub labeling. Our analysis reveals that the label graph exhibits an informative hierarchy and enhances attention learning on the interaction between nodes. Regarding efficiency, construction and distance query of the label graph can be accomplished with *linear* complexity and are decoupled from iterative model training. Hence, the model benefits from scalability in computation speed and mini-batch training. In Chapter 6, we propose GENTI, a novel design for scalable subgraph GNN on dynamic graphs. GENTI decouples the subgraph extraction stage into two asynchronous phases and boosts GPU utilization. In specific, GPU processing incorporates the efficient BSGATHER for subgraph gathering in batches and subsequent feature generation and learning. CPU is solely responsible for maintaining the dynamic graph structure with SAMPLE and UPDATE operations in a streaming manner.

We consolidate our approach in the third part, where we offer comprehensive theoretical and empirical analysis regarding the GNN scalability issue. In Chapter 7, our UNIFEWS framework unifies edge and weight sparsification as entry-wise GNN operations. By bridging spectral graph smoothing and GNN sparsification, we showcase in theory that the layer-progressive UNIFEWS provides an effective approximation on the graph learning process with a close optimization objective, which is effective in depicting a range of approximate GNN updates in both iterative and decoupled architectures. In Chapter 8, we conduct extensive evaluations of spectral GNNs concerning both effectiveness and efficiency. We first provide a thorough analysis of the graph kernels in existing GNNs and categorize them by spectral designs. These filters are then implemented under a unified framework with highly efficient learning schemes. Comprehensive experiments are conducted to assess the model performance, with discussions covering heterophily, graph scales, spectral properties, and degree-wise bias.

9.2 Future Works

Supporting Large Model Inference and Retrieval. Large Language Models (LLMs) have come into eclectic aspects, from assisting people's daily lives to reshaping schemas in academic and industrial production. The capability of processing more modalities of data and the reliability of retrieving precise and faithful knowledge have been two important topics where graph learning can be integrated to further empower and enhance LLMs. For instance, Graph-powered Retrieval-Augmented Generation (RAG) [348, 349, 350] has emerged as a powerful approach by organizing essential information as an external graph database, where domain knowledge is provided as edge links between entities. This information in graph form largely enriches the domain reasoning capabilities of LLMs.

However, the current development of graph-powered RAG remains challenging when integrating graph data processing into LLM execution, as efficiency and scalability are prominent issues. The heavy dependence on the full-scale graph structure of mainstream graph-based RAGs becomes a critical efficiency bottleneck for deploying the technique at scale. Due to the high cost of executing LLM on GPUs, it is also essential to plan the utilization of LLM reasonably and schedule the computation on GPU devices. These challenges are respectively tackled for GNN learning in this thesis. Therefore, enhancing the graph-based RAG pipeline by integrating scalable GNN techniques is a promising approach for reducing computational costs and expanding the deployment of RAG on massive data.

Addressing Fine-grained Performance. As demonstrated in Chapter 8, GNN models exhibit biased prediction on graph nodes of different degrees. In particular, while the majority of models are advantageous under homophily, the performance of high-degree nodes significantly drops for heterophilous datasets. This phenomenon reveals a more general consideration of GNN performance: while GNN capability is commonly evaluated by the accuracy among nodes in the whole graph, the learning outcome of specific graph nodes may be unsatisfactory. This direction is meaningful when some graph elements are of particular interest, entailing more careful processing and more precise prediction.

Performing fine-grained operations for graph data at scale comes with diverse scalability issues. Due to neighbor explosion, the computational overhead for representing a single node intensifies to the graph scale, rendering no less time and memory resources than full-graph learning. Additionally, the local, fine-grained information is easily overwhelmed

by the large amount of data. These under-explored challenges call for specific solutions on top of current scalable techniques, which may lead to further advances in better overall GNN capability as well as more effective graph manipulations.

Exploring Graph Data Variants While canonical GNNs assume simple and labeled graphs to perform learning, realistic conditions are more complicated and present new requirements for designing proper scalable GNNs. This direction can be related to the topics of data efficiency and elasticity in other realms of machine learning. For instance, the issue of label insufficiency is common in real-world graph data mining due to the lack of ground-truth labels, especially for large-scale graphs where the expense of manual labeling is excessive. Certain graph processing and model embedding strategies are uniquely useful in these challenging scenarios.

Practical deployments also mark other data variants, such as dynamic graphs characterized by changes in topology over time, which pose a persistent challenge for many GNN designs due to the additional temporal dimension. While a number of advantageous techniques have been proposed for handling sequential data in both graph management and neural network regimes, it deserves further investigation into how these algorithms can integrate and accommodate the paradigm of scalable GNNs. Studies aimed at scaling up these graphs are therefore welcomed by applications based on these variants.

Another widely used graph variant is the knowledge graph, in which graph elements are associated with semantic type information. Typically, applying GNN operations such as message passing requires distinct processing for each type of relation, which is similar to processing respective homogeneous graphs. Then, the heterogeneous data is represented by a fusion transformation across all types. This approach indicates more specialized and data-specific design and implementation of graph learning methods, potentially expanding the applicability of scalable GNN techniques, since the fundamental principles of scalable algorithmic design are preserved while detailed implementations can be tailored for diverse scenarios. Similar strategies are applicable to hypergraphs, where hyperedges connect multiple nodes, enabling more complex message-passing mechanisms.

Appendix A

Supplementary for Chapter 3 SCARA

A.1 Proof of Lemma 3.2

Proof. Similar to the theory in [129], feature PPR can also be interpreted as the solution of the following linear system:

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

which can be transformed to

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

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

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

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

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

According to the associative law, the combination PPR $\check{\boldsymbol{\pi}}(\boldsymbol{x})$ expressed in Eq. (3.8) satisfies

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

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

$$\mathbb{E}\left[\sum_{j=0}^{N_W(\boldsymbol{b}_i)} \frac{r_{sum}(\boldsymbol{b}_i)}{N_W(\boldsymbol{b}_i)} X_j(\boldsymbol{b}_i; t)\right] = \sum_{v \in V} r(\boldsymbol{b}_i; v) \cdot \pi(v, t). \tag{A.1}$$

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:

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

By referring to Lemma 3.2 in [217], we can further acquire the precision guarantee of the PPR as:

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

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

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

and

$$v = \frac{1}{N_{sum}} \sum_{i=0}^{F_B} \sum_{j=0}^{N_W(\boldsymbol{b}_i)} \left(\frac{\theta_i r_{sum}(\boldsymbol{b}_i) N_{sum}}{N_W(\boldsymbol{b}_i)} \right)^2 \mathbb{E}[X_j(\boldsymbol{b}_i; t)] + \frac{1}{N_{sum}} \sum_{j=0}^{N_W(\boldsymbol{z})} \left(\frac{r_{sum}(\boldsymbol{z}) N_{sum}}{N_W(\boldsymbol{z})} \right)^2 \mathbb{E}[X_j(\boldsymbol{z}; t)]. \quad \text{(A.3)}$$

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 v, we substitute Eq. (A.1) into Eq. (A.3) as:

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

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 v, we are able to derive Eq. (A.2) 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

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

□

Appendix B

Supplementary for Chapter 7 Unifews

B.1 Proof of Theorem 7.1

Proof. By using the closed-form solution $\boldsymbol{p}^* = (\boldsymbol{I} + c\boldsymbol{L})^{-1}\boldsymbol{x}$ and the fact that $\boldsymbol{A}^{-1} - \boldsymbol{B}^{-1} = \boldsymbol{B}^{-1}(\boldsymbol{B} - \boldsymbol{A})\boldsymbol{A}^{-1}$, we have:

$$\begin{aligned}\hat{\boldsymbol{p}}^* - \boldsymbol{p}^* &= \left((\boldsymbol{I} + c\hat{\boldsymbol{L}})^{-1} - (\boldsymbol{I} + c\boldsymbol{L})^{-1}\right)\boldsymbol{x} \\ &= (\boldsymbol{I} + c\hat{\boldsymbol{L}})^{-1}(c\hat{\boldsymbol{L}} - c\boldsymbol{L})(\boldsymbol{I} + c\boldsymbol{L})^{-1}\boldsymbol{x} = c(\boldsymbol{I} + c\hat{\boldsymbol{L}})^{-1}(\hat{\boldsymbol{L}} - \boldsymbol{L})\boldsymbol{p}^*.\end{aligned}$$

From Eq. (7.2), we can acquire the difference between the raw and approximate Laplacian matrices based on the spectral property:

$$\|\boldsymbol{L} - \hat{\boldsymbol{L}}\|_2 = \sup_{\|\boldsymbol{x}\|=1} \boldsymbol{x}^\top(\boldsymbol{L} - \hat{\boldsymbol{L}})\boldsymbol{x} = \boldsymbol{x}_0^\top(\boldsymbol{L} - \hat{\boldsymbol{L}})\boldsymbol{x}_0 \leq \epsilon \boldsymbol{x}_0^\top \boldsymbol{I}\boldsymbol{x}_0 = \epsilon, \tag{B.1}$$

where $\|\cdot\|_2$ is the matrix spectral norm, and the supremum is achieved when $\boldsymbol{x} = \boldsymbol{x}_0$.

The distance between $\boldsymbol{p}^*$ and $\hat{\boldsymbol{p}}^*$ follows the consistency of spectral norm $\|\boldsymbol{A}\boldsymbol{x}\| \leq \|\boldsymbol{A}\|_2\|\boldsymbol{x}\|$. By substituting Eq. (B.1) and utilizing the property of spectral norm, we have:

$$\begin{aligned}\|\hat{\boldsymbol{p}}^* - \boldsymbol{p}^*\| &\leq c\|(\boldsymbol{I} + c\hat{\boldsymbol{L}})^{-1}\|_2 \cdot \|\hat{\boldsymbol{L}} - \boldsymbol{L}\|_2 \cdot \|\boldsymbol{p}^*\| \\ &= c\epsilon\|\boldsymbol{p}^*\| \cdot \max_i \left\{\frac{1}{\lambda_i(\boldsymbol{I} + c\hat{\boldsymbol{L}})}\right\} = \frac{c\epsilon\|\boldsymbol{p}^*\|}{1 + c\lambda_1(\hat{\boldsymbol{L}})} = c\epsilon\|\boldsymbol{p}^*\|,\end{aligned}$$

where $\lambda_i(\boldsymbol{A})$ denotes the i-th smallest eigenvalue of matrix $\boldsymbol{A}$. □

Given the additive nature of the graph specifier, the bound for graph smoothing problem Theorem 7.1 is dissimilar to the approximation setting with multiplicative similarity bounded by the quadratic form $O(c\epsilon \boldsymbol{p}^\top \boldsymbol{L}\boldsymbol{p})$ [279, 280], but instead correlates with the embedding vector norm $\|\boldsymbol{p}^*\|$. This correlation arises from the bias introduced by the pruned entries in the diffusion matrix, which is associated with the embedding value.

B.2 Proof of Theorem 7.2

Proof. We outline the diffusion by the general graph adjacency $\boldsymbol{T} = \boldsymbol{A}$. The entry-wise difference matrix for the sparsified diffusion can be derived as $\Upsilon = \boldsymbol{A} - \hat{\boldsymbol{A}} = \hat{\boldsymbol{L}} - \boldsymbol{L}$. Additionally, the pruned edges form the complement set $\mathcal{E}_\Upsilon = \mathcal{E} \setminus \hat{\mathcal{E}}$, and the number of removed edges is $q_a = |\mathcal{E}_\Upsilon|$. If an edge is pruned $(u, v) \in \mathcal{E}_\Upsilon$, the entry $\Upsilon[u, v] = \boldsymbol{A}[u, v]$.

For a current embedding $\boldsymbol{p}$, its product with the difference matrix $\Upsilon\boldsymbol{p}$ only correlates with entries that have been pruned. Based on the Minkowski inequality and sparsification scheme Eq. (7.6), the L_1 norm of the product vector satisfies:

$$\begin{aligned}\|\Upsilon\boldsymbol{p}\|_1 &= \sum_{u\in\mathcal{V}} \Big| \sum_{v\in\mathcal{N}_\Upsilon(u)} \boldsymbol{A}[u,v]\boldsymbol{p}[v] \Big| = \sum_{u\in\mathcal{V}} \Big| \sum_{v\in\mathcal{N}_\Upsilon(u)} \tau[u,v] \Big| \\ &\leq \sum_{u\in\mathcal{V}} \sum_{v\in\mathcal{N}_\Upsilon(u)} \Big| \tau[u,v] \Big| \leq \sum_{u\in\mathcal{V}} \sum_{v\in\mathcal{N}_\Upsilon(u)} \delta_a = q_a\delta_a.\end{aligned}$$

Employing the relationship between vector norms given by the Cauchy-Schwarz inequality $\|\boldsymbol{x}\| \leq \|\boldsymbol{x}\|_1$, the difference of quadratic forms with regard to graph Laplacian can be bounded as:

$$|\boldsymbol{p}^\top \boldsymbol{L}\boldsymbol{p} - \boldsymbol{p}^\top \hat{\boldsymbol{L}}\boldsymbol{p}| = |\boldsymbol{p}^\top \Upsilon\boldsymbol{p}| \leq \|\boldsymbol{p}\| \cdot \|\Upsilon\boldsymbol{p}\| \leq q_a\delta_a\|\boldsymbol{p}\|.$$

Referring to Eq. (7.3), when the condition $q_a\delta_a \leq \epsilon\|\boldsymbol{p}\|$ is met, the sparsified $\hat{\boldsymbol{L}}$ is spectrally similar to $\boldsymbol{L}$ with approximation rate ϵ. □

B.3 Proof of Theorem 7.3

Proof. Recall in Eq. (7.6) pruning, there is $\mathcal{E}_\Upsilon = \{(u,v) \mid |\tau[u,v]| \leq \delta_a\}$. The fraction of entries below the threshold is thence expressed by the probability:

$$\begin{aligned} q_a = |\mathcal{E}_\Upsilon| &= m \cdot P(\mathbf{A}[u,v] \cdot |\mathbf{p}[v]| \leq \delta_a) \\ &= 2m \int_0^\infty P(\mathbf{A}[u,v] \leq \frac{\delta_a}{x}) \cdot f_{p[v]}(x) dx, \end{aligned}$$

where $f_{p[v]}(x)$ represents the probability distribution function of embedding values. As $\mathbf{A}[u,v] \geq x_{min}$, we have $P(\mathbf{A}[u,v] \leq \frac{\delta_a}{x}) = 0$, when $x > \frac{\delta_a}{x_{min}}$. The integral upper bound changes to $\frac{\delta_a}{x_{min}}$,i.e.

$$q_a = 2m \int_0^{\frac{\delta_a}{x_{min}}} P(\mathbf{A}[u,v] \leq \frac{\delta_a}{x}) \cdot f_{p[v]}(x) dx.$$

Acknowledging the assumptions, in scale-free graphs, the distribution of nodes follows the power law $P(d) = d^{-\alpha}$, where $P(d)$ is the fraction of nodes of degree d for large values, and α is a constant typically within range $2 < \alpha < 3$. For the normalized adjacency matrix $\tilde{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\bar{\boldsymbol{A}}\boldsymbol{D}^{-1/2}$, its entry distribution also follows the power law as $x^{-\alpha}$. The embedding value $p[v] \sim N(0, \sigma^2)$. By substituting the distributions we have:

$$\begin{aligned} q_a &= 2m \int_0^{\frac{\delta_a}{x_{min}}} \left[1 - \left(\frac{\delta_a}{x x_{min}}\right)^{-\alpha+1}\right] \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx \\ &= 2m\Big[\tfrac{1}{2}\,\mathrm{erf}\Big(\frac{\delta_a/x_{\min}}{\sqrt{2}\,\sigma}\Big) - \int_0^{\frac{\delta_a}{x_{min}}} \left(\frac{\delta_a}{x x_{min}}\right)^{-\alpha+1} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx\Big] \\ &= 2m(\frac{1}{2} - \frac{1}{\sqrt{2\pi}\sigma}\left(\frac{\delta_a}{x_{min}}\right)^{-\alpha+1} \int_0^{\frac{\delta_a}{x_{min}}} x^{\alpha-1} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx), \end{aligned}$$

$$\text{where} \quad \mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \mathrm{e}^{-t^2} \mathrm{d}t, \quad \gamma(s,x) = \int_0^x t^{\,s-1}\, \mathrm{e}^{-t}\, \mathrm{d}t.$$

The last term can be given by the lower incomplete Gamma function:

$$\int_0^{\frac{\delta_a}{x_{min}}} x^{\alpha-1} \exp\left(-\frac{x^2}{2\sigma^2}\right) dx = \sigma^\alpha\, 2^{\frac{\alpha-2}{2}}\, \gamma\left(\frac{\alpha}{2}, \frac{(\delta_a/x_{\min})^2}{2\sigma^2}\right).$$

Therefore,

$$q_a = m\Big[\mathrm{erf}\Big(\frac{\delta_a/x_{\min}}{\sqrt{2}\,\sigma}\Big) \;-\; \Big(\frac{x_{\min}}{\delta_a}\Big)^{\alpha-1} \;\cdot\; \frac{\sigma^{\alpha}\,2^{\frac{\alpha}{2}}}{\sqrt{2\pi}\,\sigma}\,\gamma\Big(\frac{\alpha}{2},\,\frac{(\frac{\delta_a}{x_{\min}})^2}{2\,\sigma^2}\Big)\Big]. \tag{B.2}$$

To understand Eq. (B.2), we discuss it in two extreme cases.

Case 1. When $\delta_a \to 0$, only a small number of edges are affected. Considering the Taylor Expansion with respect to δ_a, there is:

$$q_a = c_1\,\delta + c_3\,\delta^3 + \cdots,$$

where $c_1 = \frac{m\sqrt{2/\pi}}{x_{\min}\,\sigma}\,\frac{\alpha-1}{\alpha}$.

Case 2. When $\delta_a > 0$ is sufficiently large, this is the main case we concerned where a portion of edges are pruned. The formula will be approximated by:

$$q_a = m - 2m\frac{2^{\frac{\alpha-3}{2}}\Gamma\left(\frac{\alpha}{2}\right)}{\sqrt{\pi}\sigma^{\alpha+1}}\left(\frac{\delta_a}{x_{min}}\right)^{1-\alpha},$$

or equivalently, $\eta_a = q_a/m = 1 - C\delta_a^{1-\alpha}$. Therefore, the relative strength of sparsification represented by the threshold $\delta_a/\|\boldsymbol{p}\|$ and the relative sparsity η_a can be represented by each other.

Referring to Theorem 7.2, the approximation bound is given as:

$$\epsilon = O\left(\eta_a(1-\eta_a)^{\frac{1}{1-\alpha}}\right), \tag{B.3}$$

which corresponds to Theorem 7.3 for giving the approximation bound. □

B.4 Proof Sketch of Proposition 7.1

Consider consecutively sparsifying the diffusion for each hop by Eq. (7.9). Intuitively, as the edges are gradually removed from the original graph, there is $q_{a,(l_1)} < q_{a,(l_2)}$ for $l_1 < l_2$ with the same relative threshold. If the sparsest graph $\boldsymbol{T}_{(L)}$ satisfies Definition 7.1, then the multi-layer update is also bounded.

Recall that common GNN learning can be expressed by the graph smoothing process Definition 7.1. Such optimization problem can be iteratively solved by employing a gradient descent scheme [234], where each iteration derives the l-th hop of graph propagation as in Eq. (7.1):

$$\boldsymbol{p}_{(l+1)} = \boldsymbol{p}_{(l)} - \frac{b}{2} \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{p}}\bigg|_{\boldsymbol{p}=\boldsymbol{p}_{(l)}} = (1-b)\boldsymbol{p}_{(l)} - bc\boldsymbol{L}\boldsymbol{p}_{(l)} + b\boldsymbol{x}. \tag{B.4}$$

where $b/2$ is the step size and initially there is $\boldsymbol{p}_{(0)} = \boldsymbol{x}$. Eq. (B.4) is expressive to represent various propagation operations in decoupled GNN models. For example, APPNP [123] can be achieved by letting $b = \alpha$ and $c = (1-\alpha)/\alpha$, while SGC [125] is the edge case with only the graph regularization term.

Now consider the layer-wise graph sparsification under Eq. (B.4) updates. For the initial state, there is $\boldsymbol{p}_{(0)} = \hat{\boldsymbol{p}}_{(0)} = \boldsymbol{x}$. If in the l-th hop, Eq. (7.6) edge sparsification is applied to the graph $\boldsymbol{L}$, then the approximation gap is:

$$\boldsymbol{p}_{(l+1)} - \hat{\boldsymbol{p}}_{(l+1)} = (1-b)(\boldsymbol{p}_{(l)} - \hat{\boldsymbol{p}}_{(l)}) + bc(\hat{\boldsymbol{L}}\hat{\boldsymbol{p}}_{(l)} - \boldsymbol{L}\boldsymbol{p}_{(l)}). \tag{B.5}$$

To demonstrate that $\hat{\boldsymbol{p}}_{(l+1)}$ is an ϵ-approximation, we use induction by assuming $\hat{\boldsymbol{p}}_{(l)} - \boldsymbol{p}_{(l)} = \Delta_p$, $\|\Delta_p / \boldsymbol{p}_{(l)}\| \sim O(\epsilon)$. Then the bound for approximation follows:

$$\begin{aligned}
\|\boldsymbol{p}_{(l+1)} - \hat{\boldsymbol{p}}_{(l+1)}\| &= \| -(1-b)\Delta_p + bc(\hat{\boldsymbol{L}}\boldsymbol{p}_{(l)} + \hat{\boldsymbol{L}}\Delta_p - \boldsymbol{L}\boldsymbol{p}_{(l)})\| \\
&\leq \|(bc\boldsymbol{L} - (1-b)\boldsymbol{I})\|_2 \|\Delta_p\| + bc\|\hat{\boldsymbol{L}} - \boldsymbol{L}\|_2 \|\boldsymbol{p}_{(l)} + \Delta_p\| \\
&\leq O(\epsilon) \cdot \|\boldsymbol{p}_{(l)}\| + bc\epsilon(1 + O(\epsilon))\|\boldsymbol{p}_{(l)}\|,
\end{aligned}$$

where the inequalities follows from the property of matrix spectral norm and Eq. (B.1). Hence, the relative error of approximate representation $\hat{\boldsymbol{p}}_{(l+1)}$ is constrained by $O(\epsilon)$.

B.5 Proof Sketch of Proposition 7.2

To apply the approximation analysis to iterative GNNs, we first extend the analysis to multi-feature input matrix. When there are f input vectors, i.e., the input matrix is $\boldsymbol{X} \in \mathbb{R}^{n \times f}$, then the matrix form of graph Laplacian smoothing corresponding to Eq. (7.1) is:

$$\boldsymbol{P}^* = \arg\min_{\boldsymbol{P}} \|\boldsymbol{P} - \boldsymbol{X}\|_F^2 + c \cdot \mathrm{tr}(\boldsymbol{P}^\top \boldsymbol{L}\boldsymbol{P}), \tag{B.6}$$

where $\|\cdot\|_F$ is the matrix Frobenius norm and the closed-form solution is $\boldsymbol{P}^* = (\boldsymbol{I}+c\boldsymbol{L})^{-1}\boldsymbol{X}$.

Since graph operations among feature dimensions are mutually independent, conclusion from Theorem 7.3 and Proposition 7.1 are still valid in their matrix forms. In iterative models, the gradient update similar to Eq. (B.4) is instead employed to the representation matrix $\boldsymbol{H}_{(l)}$ and derives each layer as:

$$\boldsymbol{P}_{(l)} = (1-b)\boldsymbol{H}_{(l)} - bc\boldsymbol{L}\boldsymbol{H}_{(l)} + b\boldsymbol{X}.$$

As detailed in [234, 277], the update scheme above is able to describe an array of iterative GNNs. For instance, when $\boldsymbol{H}_{(l)} = \boldsymbol{X}$ and $b = 1/c$, it yields the GCN propagation $\boldsymbol{P}_{(l)} = \tilde{\boldsymbol{A}}\boldsymbol{H}_{(l)}$. GAT convolution can be imitated by a node-wise c depending on the attention values [28, 287].

To interpret the effect of UNIFEWS sparsification, we first investigate the entry-wise graph pruning in Algorithm 7.2 and its outcome, i.e., the embedding matrix $\hat{\boldsymbol{P}}_{(l)}$. For simplicity, we assume the sparsification is only employed upon the $(l+1)$-th layer and $\hat{\boldsymbol{H}}_{(l)} = \boldsymbol{H}_{(l)}$. Invoking Eq. (B.1) and the fact that $\|\boldsymbol{A}\boldsymbol{B}\|_F \leq \|\boldsymbol{A}\|_2\|\boldsymbol{B}\|_F$, the margin of approximate embeddings can be written as:

$$\|\hat{\boldsymbol{P}}_{(l)} - \boldsymbol{P}_{(l)}\|_F \leq bc\|\hat{\boldsymbol{L}} - \boldsymbol{L}\|_2 \, \|\boldsymbol{H}_{(l)}\|_F \leq bc\epsilon\|\boldsymbol{H}_{(l)}\|_F.$$

Then, consider the weight pruning Eq. (7.11). The entry-wise error is a composition of joint embedding and weight approximation:

$$\hat{\boldsymbol{\omega}}_{(l+1)}[j,i] - \boldsymbol{\omega}_{(l+1)}[j,i] = \hat{\boldsymbol{W}}_{(l)}[j,i]\hat{\boldsymbol{P}}_{(l)}[:,j] - \boldsymbol{W}_{(l)}[j,i]\boldsymbol{P}_{(l)}[:,j].$$

Let $\boldsymbol{M}_{(l)} = \boldsymbol{W}_{(l)} - \hat{\boldsymbol{W}}_{(l)}$. Recalling the iterative update scheme Eq. (7.10), the total difference on linear transformation $\boldsymbol{H}' = \boldsymbol{P}_{(l)}\boldsymbol{W}_{(l)}$ is built up by:

$$\hat{\boldsymbol{H}}' - \boldsymbol{H}' = \sum_{i,j=1}^{f} \big(\hat{\boldsymbol{P}}_{(l)}[:,j] - \boldsymbol{P}_{(l)}[:,j]\big)\boldsymbol{W}_{(l)}[j,i] + \hat{\boldsymbol{P}}_{(l)}[:,j]\boldsymbol{M}_{(l)}[j,i].$$

Its first term corresponds to the embedding approximation:

$$\Delta_P \leq \|\hat{\boldsymbol{P}}_{(l)} - \boldsymbol{P}_{(l)}\|_F \, \|\boldsymbol{W}\|_F \leq bc\epsilon\|\boldsymbol{H}_{(l)}\|_F\|\boldsymbol{W}\|_F,$$

and the second term adheres to weight sparsification as per Eq. (7.11):

$$\Delta_W \leq \sum_{i=1}^{f}\sum_{j=1}^{f} \left\| \hat{\boldsymbol{P}}_{(l)}[:, j]\boldsymbol{M}_{(l)}[j, i] \right\| \leq q_w \delta_w,$$

where q_w is the number of pruned weight entries. Finally, the representation matrix in the $(l+1)$-th layer can be bounded by:

$$\|\hat{\boldsymbol{H}}_{(l+1)} - \boldsymbol{H}_{(l+1)}\|_F \leq \ell_\sigma bc\epsilon \|\boldsymbol{H}_{(l)}\|_F \|\boldsymbol{W}\|_F + \ell_\sigma q_w \delta_w, \tag{B.7}$$

where ℓ_σ is the Lipschitz constant representing the non-linearity of activation function σ [351]. The above equation corresponds to the one in Proposition 7.2.

The above analysis shows that UNIFEWS with unified graph and weight pruning produces a good approximation of the learned representations across GNN layers, and the margin of output representations is jointly bounded by the graph sparsification rate ϵ and weight threshold δ_w. A recent work [157] offers a theoretical evaluation specifically on model weight pruning throughout training iterations, under more narrow assumptions and the particular GCN scheme. We believe their results could be supplemental to our theory whose focus is the graph perspective.

Appendix C

Supplementary for Chapter 8 Spectral GNN Benchmark

C.1 Paradigm of Fixed Filters

Linear. A layer of **GCN** [25] propagation $f(\tilde{\boldsymbol{A}}) = \boldsymbol{I} + \tilde{\boldsymbol{A}}$ is equivalent to a single-hop of spectral convolution, which represents a linear filter without high-order terms. Recall that $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \tilde{\boldsymbol{A}}$, the filter can be expressed as:

$$g(\tilde{\boldsymbol{L}}) = 2\boldsymbol{I} - \tilde{\boldsymbol{L}}.$$

Impulse. **SGC** [125] and **gfNN** [317] adopt a pre-propagation decoupled architecture, while **GZoom** [169] applies post-propagation to achieve expansion of the closed-form filter $(\boldsymbol{I} + \tilde{\boldsymbol{L}})^{-1}$. All these models result in a J-hop spatial diffusion operation as $f(\boldsymbol{A}) = \tilde{\boldsymbol{A}}^J$, which corresponds to the impulse signal with only the J-th term. By respectively examining bases $T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k$ and $T^{(k)}(\tilde{\boldsymbol{L}}) = \tilde{\boldsymbol{L}}^k$, we have two equivalent formulations of the filter:

$$g(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^K, \quad T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \theta_0 = \theta_1 = \cdots = \theta_{K-1} = 0, \ \theta_K = 1; \text{ or}$$
$$g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k \tilde{\boldsymbol{L}}^k, \quad T^{(k)}(\tilde{\boldsymbol{L}}) = \tilde{\boldsymbol{L}}^k, \qquad \theta_k = \binom{K}{k}(-1)^k,$$

where $\binom{K}{k}$ is the binomial coefficient.

Monomial. **S^2GC** [132] summarizes K-hop propagation results with uniform weights in decouple precomputation, which is classifies as the monomial propagation $f(\boldsymbol{A}) = \sum_{j=1}^{J} \xi_j \tilde{\boldsymbol{A}}^j$ with equal parameters $\xi_j = 1/(J+1)$. **GRAND+** [302] examines an approximate propagation to acquire the filter. Similarly, there are two commonly used spectral interpretations based on two bases:

$$g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k \, (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \theta_0 = \cdots = \theta_K = \frac{1}{K+1}; \text{ or}$$

$$g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k \, \tilde{\boldsymbol{L}}^k, \qquad T^{(k)}(\tilde{\boldsymbol{L}}) = \tilde{\boldsymbol{L}}^k, \qquad \theta_k = \frac{1}{K+1} \sum_{j=k}^{J} \binom{j}{k} (-1)^k,$$

Personalized PageRank (PPR). **GLP** [303] derives a closed-form $\hat{f}(\boldsymbol{A}) = (\boldsymbol{I} + \alpha \boldsymbol{L})^{-1}$ from the auto regressive (AR) filter [352], while **PPNP** [123] solves PPR [129] as $\hat{f}(\tilde{\boldsymbol{A}}) = \alpha\big(\boldsymbol{I} + (1-\alpha)\tilde{\boldsymbol{A}}\big)^{-1}$. While targeting at different problems, these two graph processing techniques are equivalent in essence. $\alpha \in [0, 1]$ is the coefficient for balancing the strength of neighbor propagation, that a larger α results in stronger node identity and weaker neighboring impact, and vice versa. In both works, the filter is approximated by a recursive calculation $\boldsymbol{H}^{(j+1)} = \varphi((1-\alpha)\tilde{\boldsymbol{A}}\boldsymbol{H}^{(j)} + \alpha \boldsymbol{H}^{(0)})$, which is widely accepted in later studies such as **GCNII** [67]. The explicit spatial and spectral interpretations of the polynomial approximation are respectively:

$$f(\tilde{\boldsymbol{A}}) = \sum_{j=0}^{J} \alpha(1-\alpha)^j \tilde{\boldsymbol{A}}^j; \quad g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_k = \alpha(1-\alpha)^k.$$

In addition, approximate computations have been proposed by representative works including GDC [133], PPRGo [124], GRAND+ [302]. GBP [111] and **AGP** [112] explored the adjacency aggregation under graph normalization.

Heat Kernel (HK). **GDC** [133] inspects the heat kernel PageRank (HKPR) [353] replacing the PPR calculation by an exponential parameter. Equivalent HK filters are also studied in DGC [318] and AGP [112]. Let $\alpha > 0$ be the temperature coefficient, the filter $\hat{f}(\tilde{\boldsymbol{A}}) = e^{-\alpha \tilde{\boldsymbol{L}}}$ is expanded in spatial and spectral forms as:

$$f(\tilde{\boldsymbol{A}}) = \sum_{j=0}^{J} \frac{e^{-\alpha}\alpha^j}{j!} \tilde{\boldsymbol{A}}^j; \quad g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_k = \frac{e^{-\alpha}\alpha^k}{k!}.$$

Gaussian. **G^{2}CN** [319] uses the Gaussian filter [354] for better flexibility on capturing local information. When concentrating on low frequency, the closed-form propagation is $\hat{f}(\tilde{\boldsymbol{A}}) = e^{-\alpha(2\boldsymbol{I}-\tilde{\boldsymbol{L}})}$. Invoking Taylor expansion for the filter leads to:

$$f^{(j)}(\tilde{\boldsymbol{A}}) = \boldsymbol{I} - \frac{\alpha}{J}(2\boldsymbol{I} - \tilde{\boldsymbol{L}}); \quad g(\tilde{\boldsymbol{L}}) = \sum_{k=0}^{K} \theta_k (2\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_k = \frac{\alpha^k}{k!}.$$

C.2 Paradigm of Variable Filter

Linear. **GIN** [320] alters the iterative ridged adjacency propagation with a learnable scaling parameter $\theta > 0$ controlling the strength of self loops, i.e., skip connections. It proves that its propagation $f(\tilde{\boldsymbol{A}}) = (1+\xi)\boldsymbol{I} + \tilde{\boldsymbol{A}}$ is more expressive with regard to the Weisfeiler-Lehman (WL) test. **AKGNN** [321] elaborates the expression in spectral domain, that the scaling parameter adaptively balances the threshold between high and low frequency. We thereby obtain the linear spectral function for each layer as:

$$g(\tilde{\boldsymbol{L}}; \theta) = (1+\theta)\boldsymbol{I} - \tilde{\boldsymbol{L}}.$$

Monomial. The monomial scheme signifies the same operations on parameters of all terms. **DAGNN** [322] studies the scheme of assigning learnable parameters to each hop of the propagation, but implements a costly concatenation-based scheme. **GPRGNN** [134] considers the iterative generalized PageRank computation [332] under heterophily, which also produces the same variable monomial spectral filter formulated as $f(\boldsymbol{A}) = \sum_{j=0}^{J} \xi_j \tilde{\boldsymbol{A}}^j$. We present two bases and corresponding relationship between spatial and spectral parameters when $J = K$:

$$g(\tilde{\boldsymbol{L}}; \theta) = \sum_{k=0}^{K} \theta_k \, (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad T^{(k)}(\tilde{\boldsymbol{L}}) = (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \, \theta_k = \xi_j; \text{ or}$$

$$g(\tilde{\boldsymbol{L}}; \theta) = \sum_{k=0}^{K} \theta_k \, \tilde{\boldsymbol{L}}^k, \qquad T^{(k)}(\tilde{\boldsymbol{L}}) = \tilde{\boldsymbol{L}}^k, \qquad \theta_k = \sum_{j=k}^{J} \binom{j}{k} (-1)^k \xi_j.$$

[134] also inspects the effect of initialization of the parameters ξ_j, θ_k on fitting heterophilous graph signals.

Horner. Horner's method [355] is a recursive algorithm to compute the summation of monomial bases with residual connections. Consider adding a residual term in the layer-wise propagation corresponding to the Monomial filter $\boldsymbol{H}^{(j+1)} = \varphi(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(j)} + \xi_j \boldsymbol{H}^{(0)})$. When the balancing parameter is fixed $\xi_j = \alpha/(1-\alpha)$, it is equivalent to the PPR computation in Appendix C.1. When ξ_j is variable, the model is regarded as **HornerGCN** which is introduced by [324]. Alternatively, **ARMAGNN** [323] utilizes the Auto Regressive Moving Average (ARMA) filter $\hat{f}(\boldsymbol{A}) = \beta(\boldsymbol{I} - \alpha\tilde{\boldsymbol{A}})^{-1}$ [352] as an approach to describe the residual connection learned by respective weights in iterative architecture. Their spectral filter is:

$$g(\tilde{\boldsymbol{L}};\theta) = \sum_{k=0}^{K} \theta_k \, (\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_k = \xi_{K-k}.$$

Although it shares an identical spectral interpretation with the Monomial filter, the explicit residual connection proves beneficial in guiding the learnable parameters to recognize node identity and alleviate over-smoothing throughout propagation.

Chebyshev. The Chebyshev basis is widely accepted for graph signal processing, which is powerful in producing a minimax polynomial approximation for the analytic functions [356, 333]. **ChebNet** [26] utilizes it to replace the adjacency propagation in iterative network architecture so that $f^{(j)}(\tilde{\boldsymbol{A}}) = T^{(j)}(\boldsymbol{I} - \tilde{\boldsymbol{A}})$. To adapt the basis to decoupled propagation with explicit variable parameters, [301] proposes **ChebBase**. The spectral expressiveness of these two models are the same, and the filter is:

$$g(\tilde{\boldsymbol{L}};\theta) = \sum_{k=0}^{K} \theta_k T^{(k)}(\tilde{\boldsymbol{L}}), \quad T^{(k)}(\tilde{\boldsymbol{L}}) = 2\tilde{\boldsymbol{L}}T^{(k-1)}(\tilde{\boldsymbol{L}}) - T^{(k-2)}(\tilde{\boldsymbol{L}}),$$
$$T^{(1)}(\tilde{\boldsymbol{L}}) = \tilde{\boldsymbol{L}}, \; T^{(0)}(\tilde{\boldsymbol{L}}) = \boldsymbol{I}.$$

The Chebyshev polynomial is expressed in the three-term recurrence relation, which is favorable for the GNN iterative propagation. One can also write the Chebyshev basis of the first kind as the closed-form expression $T^{(k)}(\lambda) = \cos(k \arccos \lambda)$.

Chebyshev Interpolation (ChebInterp). **ChebNetII** [301] utilizes Chebyshev interpolation [357] to modify the Chebyshev filter parameter for better approximation with generally decaying weights. For each Chebyshev basis $T^{(k)}(\tilde{\boldsymbol{L}})$, it appends the basis with K-order

interpolation:

$$g(\tilde{\boldsymbol{L}};\theta)=\frac{2}{K+1}\sum_{k=0}^{K}\sum_{\kappa=0}^{K}\theta_\kappa T^{(k)}(x_\kappa)\,T^{(k)}(\tilde{\boldsymbol{L}}),\quad x_\kappa=\cos\left(\frac{\kappa+1/2}{K+1}\pi\right),$$

where $T^{(k)}(x_\kappa), T^{(k)}(\tilde{\boldsymbol{L}})$ follow the Chebyshev basis, and x_κ are the Chebyshev nodes of $T^{(K+1)}$.

Clenshaw. Similar to Horner's method, **ClenshawGCN** [324] applies Clenshaw algorithm on top of the Chebyshev filter to incorporate explicit residual connections. Its spatial convolution is obtained as $\boldsymbol{H}^{(j+1)}=\varphi\big(2\tilde{\boldsymbol{A}}\boldsymbol{H}^{(j)}-\boldsymbol{H}^{(j-1)}+\xi_j\boldsymbol{H}^{(0)}\big), \boldsymbol{H}^{(-1)}=\boldsymbol{H}^{(-2)}=\boldsymbol{O}$. The form of spectral filter is related with Chebyshev polynomials of the second kind:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\quad T^{(k)}(\tilde{\boldsymbol{L}})=2\tilde{\boldsymbol{L}}T^{(k-1)}(\tilde{\boldsymbol{L}})-T^{(k-2)}(\tilde{\boldsymbol{L}}),$$
$$T^{(1)}(\tilde{\boldsymbol{L}})=2\tilde{\boldsymbol{L}},\ T^{(0)}(\tilde{\boldsymbol{L}})=\boldsymbol{I}.$$

Alternatively, the closed-form definition of the Chebyshev basis of the second kind is $T^{(k)}(\cos\lambda)=\frac{\sin((k+1)\lambda)}{\sin\lambda}$. The relation to spatial parameters is $\theta_k=\xi_{K-k}$.

Bernstein. **BernNet** [325] pursues more interpretable spectral filters by thte Bernstein polynomial approximation [358] and invokes constraints form prior knowledge to avoid ill-posed variable parameters. The spatial propagation is special as it applies two graph matrices instead of one. The filter with regard to Bernstein basis $T^{(k)}$ is:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\frac{\theta_k}{2^K}T^{(k)}(\tilde{\boldsymbol{L}}),\quad T^{(k)}(\tilde{\boldsymbol{L}})=\binom{K}{k}(2\boldsymbol{I}-\tilde{\boldsymbol{L}})^{K-k}\,\tilde{\boldsymbol{L}}^k,$$

where learnable parameters are initialized as $\theta_k=T^{(k)}(k/K)$.

Legendre. **LegendreNet** [326] exploits the Legendre polynomials in an accumulation form of calculation similar to BernNet:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\quad T^{(k)}(\tilde{\boldsymbol{L}})=\frac{(-1)^k}{k!\binom{2k}{k}}(2\boldsymbol{I}-\tilde{\boldsymbol{L}})^k\,\tilde{\boldsymbol{L}}^k.$$

Based on the relation of Bernstein bases and Legendre polynomials [359], we can also express it in the recurrence form:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\quad T^{(0)}(\tilde{\boldsymbol{L}})=\boldsymbol{I},\ T^{(1)}(\tilde{\boldsymbol{L}})=\tilde{\boldsymbol{L}},$$
$$T^{(k)}(\tilde{\boldsymbol{L}})=\frac{2k-1}{k}\tilde{\boldsymbol{L}}T^{(k-1)}(\tilde{\boldsymbol{L}})-\frac{k-1}{k}T^{(k-2)}(\tilde{\boldsymbol{L}}).$$

Jacobi. **JacobiConv** [243] utilizes the more general Jacobi basis, whereas Chebyshev and Legendre polynomials can be regarded as special cases. Intuitively, it provides more flexible weight functions with two hyperparameters α,β to adapt to different signals of the spectral graphs. Each polynomial term of JacobiConv can be formulated by the three-term recurrence as following:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\ T^{(0)}(\tilde{\boldsymbol{L}})=\boldsymbol{I},\ T^{(1)}(\tilde{\boldsymbol{L}})=\frac{\alpha-\beta}{2}\boldsymbol{I}+\frac{\alpha+\beta+2}{2}(\boldsymbol{I}-\tilde{\boldsymbol{L}}),$$
$$T^{(k)}(\tilde{\boldsymbol{L}})=\delta_k(\boldsymbol{I}-\tilde{\boldsymbol{L}})T^{(k-1)}(\tilde{\boldsymbol{L}})+\delta'_k T^{(k-1)}(\tilde{\boldsymbol{L}})-\delta''_k T^{(k-2)}(\tilde{\boldsymbol{L}}),$$

where

$$\delta_k=\frac{(2k+\alpha+\beta)(2k+\alpha+\beta-1)}{2k(k+\alpha+\beta)},\ \delta'_k=\frac{(2k+\alpha+\beta-1)(\alpha^2-\beta^2)}{2k(k+\alpha+\beta)(2k+\alpha+\beta-2)},$$
$$\delta''_k=\frac{(k+\alpha-1)(k+\beta-1)(2k+\alpha+\beta)}{k(k+\alpha+\beta)(2k+\alpha+\beta-2)}\ \text{for } k\geq 2.$$

Favard. **FavardGNN** [327] exploits the Favard's Theorem [360] to learn the polynomial basis from available space and ensure orthonormality. It is achieved by a three-term recurrence form with multiple series of hop-dependent variable parameters θ,α,β:

$$g(\tilde{\boldsymbol{L}};\theta)=\sum_{k=0}^{K}\theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\ T^{(-1)}(\tilde{\boldsymbol{L}})=\boldsymbol{O},\ T^{(0)}(\tilde{\boldsymbol{L}})=\frac{1}{\sqrt{\alpha_0}}\boldsymbol{I},$$
$$T^{(k)}(\tilde{\boldsymbol{L}})=\frac{1}{\sqrt{\alpha_k}}\left((\boldsymbol{I}-\tilde{\boldsymbol{L}})T^{(k-1)}(\tilde{\boldsymbol{L}})-\beta_k T^{(k-1)}(\tilde{\boldsymbol{L}})-\sqrt{\alpha_{k-1}}\,T^{(k-2)}(\tilde{\boldsymbol{L}})\right).$$

OptBasis. **OptBasisGNN** [327] considers a Basis to be Optimal with regard to convergence rate in the graph signal denoising problem. By replacing the learnable parameters in FavardGNN with parameters derived from the current input signal $\boldsymbol{h}^{(k)}$, it can approach

the optimal basis without occurring additional overhead.

$$
\begin{aligned}
g(\tilde{\boldsymbol{L}};\theta) &= \sum_{k=0}^{K} \theta_k T^{(k)}(\tilde{\boldsymbol{L}}),\ T^{(-1)}(\tilde{\boldsymbol{L}}) = \boldsymbol{O},\ T^{(0)}(\tilde{\boldsymbol{L}}) = \frac{1}{\|\boldsymbol{h}^{(0)}\|}\boldsymbol{I}, \\
T^{(k)}(\tilde{\boldsymbol{L}}) &= \frac{1}{\|\boldsymbol{h}^{(k)}\|}\left((\boldsymbol{I}-\tilde{\boldsymbol{L}})T^{(k-1)}(\tilde{\boldsymbol{L}}) - \beta_{k-1}T^{(k-1)}(\tilde{\boldsymbol{L}}) - \|\boldsymbol{h}^{(k-1)}\|T^{(k-2)}(\tilde{\boldsymbol{L}})\right), \\
\beta_{k-1} &= \langle(\boldsymbol{I}-\tilde{\boldsymbol{L}})\boldsymbol{h}^{(k-1)}, \boldsymbol{h}^{(k-1)}\rangle.
\end{aligned}
$$

C.3 Paradigm of Filter Banks

AdaGNN. AdaGNN [328] designs $Q = F$ adaptive filters by assigning feature-specific parameters to the Linear filter $g_q(\tilde{\boldsymbol{L}}) = \boldsymbol{I} - \gamma_q\tilde{\boldsymbol{L}}, 1 \leq q \leq F$. The representation update is performed in an iterative manner as $\boldsymbol{H}^{(j+1)} = \boldsymbol{H}^{(j)} - \tilde{\boldsymbol{L}}\boldsymbol{H}^{(j)}\boldsymbol{\Gamma}^{(j)}$, where $\boldsymbol{\Gamma}^{(j)} = \mathrm{diag}(\gamma_1^{(j)}, ..., \gamma_F^{(j)})$ containing the learnable feature-wise parameter for the j-th layer. Hence, considering a single layer, the corresponding aggregated filter in spectral domain is:

$$
g(\tilde{\boldsymbol{L}};\gamma) = \Big\|_{q=1}^{F} (\boldsymbol{I} - \gamma_q\tilde{\boldsymbol{L}}),
$$

where each filter $g_q(\tilde{\boldsymbol{L}})$ is only applied to the q-th feature, and $\|$ is the concatenation operator that combines filtering result tensors among all features.

FBGNN. FBGNN [329] introduces the concept of filter bank for combining multiple filters [231] in spectral GNN under the context of graph heterophily. It designs a two-channel scheme using graph adjacency $T_1 = \tilde{\boldsymbol{A}}$ as Linear low-pass filter (LP) and Laplacian $T_2 = \tilde{\boldsymbol{L}}$ for Linear high-pass filter (HP) to learn the smooth and non-smooth components together. FBGNN adopts an iterative form with scalar parameters $\gamma_1, \gamma_2 \in [0, 1]$ for weighted sum. By omitting components including transformation weights and non-linear activation functions, we give the equivalent spectral function for each layer as:

$$
g(\tilde{\boldsymbol{L}};\gamma) = \gamma_1(\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2\tilde{\boldsymbol{L}}.
$$

ACMGNN. ACMGNN [139] extends FBGNN to three filters with the additional identity (ID) diffusion matrix $T_3 = \boldsymbol{I}$, which corresponds to an all-pass filter maintaining node identity throughout propagation. It has two variants of applying different relative

order between transformation and propagation, which are denoted as ACMGNN-I and ACMGNN-II. Both of their single-layer spectral expressions can be simplified as:

$$g_\theta(\tilde{\boldsymbol{L}};\gamma) = \gamma_1(\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2\tilde{\boldsymbol{L}} + \gamma_3\boldsymbol{I}.$$

FAGCN. FAGCN [137] combines two Linear filters with bias for capturing low- and high-frequency signals as $T_1 = (\beta+1)\boldsymbol{I} - \tilde{\boldsymbol{L}}$ and $T_2 = (\beta-1)\boldsymbol{I} + \tilde{\boldsymbol{L}}$, where $\beta \in [0,1]$ is the scaling coefficient. Since both filters are linear to $\tilde{\boldsymbol{L}}$, the fused representation of each layer can be computed using only one propagation by employing attention mechanism. We generally write the channel-wise parameters as $\gamma_1, \gamma_2 \in [0,1]$ and $\gamma_1 + \gamma_2 = 1$. Then the spectral expression for one layer is:

$$g_\theta(\tilde{\boldsymbol{L}};\gamma) = \gamma_1((\beta+1)\boldsymbol{I} - \tilde{\boldsymbol{L}}) + \gamma_2((\beta-1)\boldsymbol{I} + \tilde{\boldsymbol{L}}).$$

G^{2}CN. G^{2}CN [319] derives the Gaussian filters for high- and low-frequency concentration centers. Specifically, it adopts 2-hop propagation in each layer. Utilizing the decay coefficient $\alpha \in [0,1]$ and the scaling coefficient $\beta \in [0,1]$, we rewrite the layer-wise propagation as $f^{(j)}(\tilde{\boldsymbol{A}}) = \boldsymbol{I} - \alpha((1\pm\beta)\boldsymbol{I} - \tilde{\boldsymbol{L}})^2/J$. The model integrates the $Q = 2$ channels after the decoupled propagation, hence:

$$g_\theta(\tilde{\boldsymbol{L}};\gamma) = \gamma_1 \sum_{k=0}^{\lfloor K/2 \rfloor} \theta_{1,k}\, T_1^{(k)} + \gamma_2 \sum_{k=0}^{\lfloor K/2 \rfloor} \theta_{2,k}\, T_2^{(k)}, \quad T_1^{(k)} = ((1+\beta_1)\boldsymbol{I} - \tilde{\boldsymbol{L}})^{2k}, \quad \theta_{1,k} = \frac{\alpha_1^k}{k!},$$
$$T_2^{(k)} = ((1-\beta_2)\boldsymbol{I} - \tilde{\boldsymbol{L}})^{2k}, \quad \theta_{2,k} = \frac{\alpha_2^k}{k!}.$$

GNN-LF/HF. GNN-LF/HF [277] proposes a pair of generalized GNN propagations based on low- and high-passing filtering (LF/HF) on top of its unified optimization framework depicting a range of GNN designs. Intuitively, its filter formulation is similar to the PPR scheme, except a $(\boldsymbol{I} \pm \beta\tilde{\boldsymbol{L}})$ factor applying to the input signal for distinguishing low- and high-frequency components. We transform the original dual filters into one model with $Q = 2$ channels by learning the balancing coefficients $\gamma_1, \gamma_2 \in [0,1]$ which adjust the relative strength between node identity and low/high frequency features. Consequently, the channels can be implemented in a shared adjacency-based propagation. We formulate

the filter bank version of GNN-LF/HF as:

$$g_{\!\!\!-}(\tilde{\boldsymbol{L}};\gamma) = \gamma_1 \sum_{k=0}^{K} \theta_{1,k}\, T_1^{(k)} + \gamma_2 \sum_{k=0}^{K} \theta_{2,k}\, T_2^{(k)}, \quad \begin{aligned} T_1^{(k)} &= (\boldsymbol{I} - \beta_1 \tilde{\boldsymbol{L}})(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_{1,k} = \alpha_1(1-\alpha_1)^k, \\ T_2^{(k)} &= (\boldsymbol{I} + \beta_2 \tilde{\boldsymbol{L}})(\boldsymbol{I} - \tilde{\boldsymbol{L}})^k, \quad \theta_{2,k} = \alpha_2(1-\alpha_2)^k, \end{aligned}$$

where there are $\alpha_1, \alpha_2 \in [0,1], \beta_1 \in [0,1/2], \beta_2 \in (0,+\infty)$ according to the optimization objective.

FiGURe. FiGURe [330] suggests using filter bank to adapt the unsupervised settings where the graph information can assist filter formation. It considers up to $Q = 4$ filters, i.e., Identity $\boldsymbol{I}$, Monomial $\boldsymbol{I} - \tilde{\boldsymbol{L}}$, Chebyshev, and Bernstein bases for the filter bank. In the first unsupervised stage, an embedding function $\gamma_q : \mathbb{R}^{n\times F} \to \mathbb{R}^{n\times F}$ is learned for each filter by maximizing the mutual information across all channels. The second stage of supervised graph representation learning fine-tunes another scalar weight γ'_q to tailor for the downstream task, along with other trainable model parameters. Formulation of the eventual FiGURe filter can be written as:

$$g_{\!\!\!-}(\tilde{\boldsymbol{L}};\gamma,\theta) = \sum_{q=1}^{Q} \gamma'_q \gamma_q \cdot g_q(\tilde{\boldsymbol{L}};\theta), \quad g_q(\tilde{\boldsymbol{L}};\theta) = \sum_{k=0}^{K} \theta_{q,k}\, T_q^{(k)}(\tilde{\boldsymbol{L}}),$$

Bibliography

[1] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. SCARA: Scalable graph neural networks with feature-oriented optimization. In *Proceedings of the VLDB Endowment*, volume 15, number 11, pages 3240–3248. VLDB Endowment, July 2022. DOI:10.14778/3551793.3551866. 35, 164

[2] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. Scalable decoupling graph neural networks with feature-oriented optimization. *The VLDB Journal*, 33(3):667–683, May 2024. ISSN 0949-877X. DOI:10.1007/s00778-023-00829-6. 35

[3] Ningyi Liao, Siqiang Luo, Xiang Li, and Jieming Shi. LD^2: Scalable heterophilous graph neural network with decoupled embeddings. In *Advances in Neural Information Processing Systems*, volume 36, pages 10197–10209, December 2023. 61, 164

[4] Ningyi Liao, Zihao Yu, Siqiang Luo, and Gao Cong. HubGT: Fast graph Transformer with decoupled hierarchy labeling. In *Advances in Neural Information Processing Systems*, volume 38, December 2025. arXiv:2412.04738. 87

[5] Zihao Yu, Ningyi Liao, and Siqiang Luo. GENTI: GPU-powered walk-based subgraph extraction for scalable representation learning on dynamic graphs. In *Proceedings of the VLDB Endowment*, volume 17, number 9, pages 2269–2278. VLDB Endowment, May 2024. DOI:10.14778/3665844.3665856. 113

[6] Ningyi Liao, Zihao Yu, Ruixiao Zeng, and Siqiang Luo. Unifews: You need fewer operations for efficient graph neural networks. In *42nd International Conference on Machine Learning*, May 2025. arXiv:2403.13268. 135

[7] Ningyi Liao, Haoyu Liu Liu, Zulun Zhu, Siqiang Luo, and Laks V.S. Lakshmanan. A comprehensive benchmark on spectral GNNs: The impact on efficiency, memory, and effectiveness. In *ACM SIGMOD International Conference on Management of Data*, volume 3, number 4. ACM, June 2026. DOI:10.1145/3749156. 163

[8] Ningyi Liao, Siqiang Luo, Xiaokui Xiao, and Reynold Cheng. Advances in designing scalable graph neural networks: The perspective of graph data management. In *Companion of the 2025 International Conference on Management of Data*, pages 844–850, Berlin, Germany, June 2025. ACM. ISBN 9798400715648. DOI:10.1145/3722212.3725634. 31

[9] Haoyu Liu, Ningyi Liao, and Siqiang Luo. SIGMA: An efficient heterophilous graph neural network with fast global aggregation. In *IEEE 41st International Conference on Data Engineering*, pages 1924–1937, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. DOI:10.1109/ICDE65448.2025.00147. 24

[10] Kai Siong Yow, Ningyi Liao, Siqiang Luo, and Reynold Cheng. Machine learning for subgraph extraction: Methods, applications and challenges. In *Proceedings of the VLDB Endowment*, volume 16, number 12, pages 3864–3867, Vancouver, Canada, September 2023. VLDB Endowment. DOI:10.14778/3611540.3611571.

[11] Jun Xuan Yew, Ningyi Liao, Dingheng Mo, and Siqiang Luo. Example Searcher: A spatial query system via example. In *IEEE 39th International Conference on Data Engineering*, pages 3635–3638, Anaheim, CA, USA, April 2023. IEEE. ISBN 979-8-3503-2227-9. DOI:10.1109/ICDE55515.2023.00286.

[12] Ningyi Liao, Weiping Yu, Siqiang Luo, and Junfeng Liu. RAGDoll: Efficient offloading-based online rag system on a single GPU, 2025. arXiv:2504.15302.

[13] Kai Siong Yow, Ningyi Liao, Siqiang Luo, Reynold Cheng, Chenhao Ma, and Xiaolin Han. A survey on machine learning solutions for graph pattern extraction, April 2022. arXiv:2204.01057.

[14] Márton Pósfai and Albert-László Barabási. *Network Science*. Cambridge University Press, Cambridge, UK, 2016. 1

[15] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, July 2017. ISSN 1053-5888. DOI:10.1109/MSP.2017.2693418. 1, 3

[16] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 14(8), 2020. 1, 3, 19, 169

[17] Geoffrey E. Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew W. Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29:82, 2012. URL https://api.semanticscholar.org/CorpusID:7230302. 1

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012. 1

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations*, 2015. 1

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27, 2014. 1

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015. DOI:10.1038/nature14539. 1

[22] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014. ISSN 1932-8346. DOI:10.1561/2000000039. 1

[23] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT Press, Cambridge, MA, USA, 2017. 1

[24] 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. 1, 3, 163, 171

[25] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations*, 2017. 1, 3, 15, 16, 35, 51, 148, 163, 167, 168, 173, 195, 225

[26] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, volume 29, pages 3844–3852, 2016. 1, 3, 163, 173, 176, 195, 228

[27] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning in large attributed graphs. In *Advances in Neural Information Processing Systems*, volume 30, Long Beach, CA, USA, October 2017. 1, 3, 18, 35, 36, 51, 148, 195

[28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations*, 2018. 1, 3, 24, 35, 36, 222

[29] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. DeepInf: Social influence prediction with deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '18, page 2110–2119. ACM, 2018. ISBN 9781450355520. DOI:10.1145/3219819.3220077. 1

[30] Lokesh Jain, Rahul Katarya, and Shelly Sachdeva. Opinion leaders for information diffusion using graph neural network in online social networks. *ACM Transactions on the Web*, 17(2), April 2023. ISSN 1559-1131. 1

[31] Kartik Sharma, Yeon-Chang Lee, Sivagami Nambi, Aditya Salian, Shlok Shah, Sang-Wook Kim, and Srijan Kumar. A survey of graph neural networks for social recommender systems. *ACM Computing Surveys*, 56(10), June 2024. ISSN 0360-0300. 1

[32] Tian Xie and Jeffrey C. Grossman. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Physical Review Letters*, 120:145301, April 2018. DOI:10.1103/PhysRevLett.120.145301. 1

[33] Victor Fung, Jiaxin Zhang, Eric Juarez, and Bobby G. Sumpter. Benchmarking graph neural networks for materials chemistry. *npj Computational Materials*, 7(1): 84, 2021. DOI:10.1038/s41524-021-00554-0. 1

[34] Patrick Reiser, Marlen Neubert, André Eberhard, Luca Torresi, Chen Zhou, Chen Shao, Houssam Metni, Clint van Hoesel, Henrik Schopmans, Timo Sommer, and Pascal Friederich. Graph neural networks for materials science and chemistry. *Communications Materials*, 3(1):93, 2022. DOI:10.1038/s43246-022-00315-6. 1

[35] Jeremy Kawahara, Colin J. Brown, Steven P. Miller, Brian G. Booth, Vann Chau, Ruth E. Grunau, Jill G. Zwicker, and Ghassan Hamarneh. BrainNetCNN: Convolutional neural networks for brain networks; towards predicting neurodevelopment. *NeuroImage*, 146:1038–1049, 2017. ISSN 1053-8119. DOI:10.1016/j.neuroimage.2016.09.046. 1

[36] Xiaoxiao Li, Yuan Zhou, Nicha Dvornek, Muhan Zhang, Siyuan Gao, Juntang Zhuang, Dustin Scheinost, Lawrence H. Staib, Pamela Ventola, and James S. Duncan. BrainGNN: Interpretable brain graph neural network for fMRI analysis. *Medical Image Analysis*, 74:102233, 2021. ISSN 1361-8415. DOI:10.1016/j.media.2021.102233. 1

[37] Alaa Bessadok, Mohamed Ali Mahjoub, and Islem Rekik. Graph neural networks in network neuroscience. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5833–5848, May 2023. ISSN 1939-3539. DOI:10.1109/TPAMI.2022.3209686. 1

[38] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 06 2018. ISSN 1367-4803. DOI:10.1093/bioinformatics/bty294. 1

[39] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. Graph neural networks and their current applications in bioinformatics. *Frontiers in Genetics*, 12, 2021. ISSN 1664-8021. DOI:10.3389/fgene.2021.690049. 1

[40] Michelle M. Li, Kexin Huang, and Marinka Zitnik. Graph representation learning in biomedicine and healthcare. *Nature Biomedical Engineering*, 6(12):1353–1369, 2022. DOI:10.1038/s41551-022-00942-x. 1

[41] 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. 1, 35

[42] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer O¨zsu. The ubiquity of large graphs and surprising challenges of graph processing.

Proceedings of the VLDB Endowment, 11(4):420–431, 2018. ISSN 21508097. DOI:10.1145/3164135.3164139. 1, 163

[43] 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. In *Advances in Neural Information Processing Systems*, volume 33, 2020. 1, 51, 103, 148, 169, 180, 197

[44] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9): 62–71, September 2021. ISSN 0001-0782, 1557-7317. DOI:10.1145/3434642. 1

[45] 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 and Data Mining*, pages 974–983, 2018. 1, 35, 163

[46] 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. 1, 35, 51

[47] Isaac Ronald Ward, Jack Joyner, Casey Lickfold, Yulan Guo, and Mohammed Bennamoun. A practical tutorial on graph neural networks. *ACM Computing Surveys*, 54(10s):1–35, January 2022. ISSN 0360-0300, 1557-7341. DOI:10.1145/3503043. 1, 3, 163

[48] 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 *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 13–23. IEEE, March 2021. DOI:10.1109/ISPASS51385.2021.00013. 2, 6

[49] 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. 2, 17, 62, 154

[50] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. Comprehensive evaluation of GNN training systems: A data management perspective. *Proceedings of the VLDB Endowment*, 17(6):1241–1254, February 2024. ISSN 2150-8097. DOI:10.14778/3648160.3648167. 2, 6

[51] William L Hamilton. *Graph representation learning*. Morgan & Claypool Publishers, 2020. 2

[52] Yao Ma and Jiliang Tang. *Deep learning on graphs*. Cambridge University Press, 2021. 2

[53] 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. 2, 61

[54] 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, October 2018. arXiv:1806.01261. 2, 61

[55] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence*, 2(2): 109–127, 2021. DOI:10.1109/tai.2021.3076021. 3

[56] Ilya Makarov, Dmitrii Kiselev, Nikita Nikitinsky, and Lovro Subelj. Survey on graph embeddings and their applications to machine learning problems on graphs. *PeerJ Computer Science*, 7:1–62, 2021. ISSN 23765992. DOI:10.7717/peerj-cs.357. 3

[57] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. *Geometric deep learning: Grids, groups, graphs, geodesics, and gauges*. arXiv:2104.13478, 2021. 3

[58] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23:1–64, 2022. ISSN 15337928. arXiv:2005.03675. 3

[59] Yu Zhou, Haixia Zheng, Xin Huang, Shufeng Hao, Dengao Li, and Jumin Zhao. Graph neural networks: Taxonomy, advances, and trends. *ACM Transactions on Intelligent Systems and Technology*, 13(1):1–54, February 2022. ISSN 2157-6904, 2157-6912. DOI:10.1145/3495161. 3

[60] Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, and Xiaojie Guo. *Graph neural networks: foundation, frontiers and applications*. Springer, 2022. 3

[61] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. 3, 169

[62] Nurul A. Asif, Yeahia Sarker, Ripon K. Chakrabortty, Michael J. Ryan, Md. Hafiz Ahamed, Dip K. Saha, Faisal R. Badal, Sajal K. Das, Md. Firoz Ali, Sumaya I. Moyeen, Md. Robiul Islam, and Zinat Tasneem. Graph neural network: A comprehensive review on non-Euclidean space. *IEEE Access*, 9:60588–60606, 2021. ISSN 2169-3536. DOI:10.1109/ACCESS.2021.3071274. 3

[63] James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 29, pages 2001–2009, 2016. 3, 36, 163

[64] 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, pages 2009–2019, Melbourne, Australia, July 2018. Association for Computational Linguistics. 3

[65] 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. 3, 16, 22, 62, 72

[66] John Boaz Lee, Ryan A. Rossi, Sungchul Kim, Nesreen K. Ahmed, and Eunyee Koh. Attention models in graphs: A survey. *ACM Transactions on Knowledge Discovery from Data*, 13(6):1–25, December 2019. ISSN 1556-4681, 1556-472X. DOI:10.1145/3363574. 3

[67] 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. arXiv:2007.02133. 3, 67, 148, 173, 226

[68] Guohao Li, Matthias Müller, Ali Thabet, and Bernard Ghanem. DeepGCNs: Can GCNs go as deep as CNNs? In *IEEE/CVF International Conference on Computer Vision*, pages 9267–9276, October 2019. arXiv:1904.03751. 3

[69] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, Dongrui Fan, Shirui Pan, and Yuan Xie. Survey on graph neural network acceleration: An algorithmic perspective. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence*, April 2022. arXiv:2202.04822. 3, 24, 135, 169

[70] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. Sampling methods for efficient training of graph convolutional networks: A survey. *IEEE/CAA Journal of Automatica Sinica*, 9(2):205–234, February 2022. ISSN 2329-9266, 2329-9274. DOI:10.1109/JAS.2021.1004311. 3, 31

[71] 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 *Advances in Neural Information Processing Systems*, volume 33, page 12, 2020. 3, 21, 62, 65, 68

[72] 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. arXiv:2205.07308. 3, 19, 22, 62, 67

[73] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data, June 2015. arXiv:1506.05163. 3

[74] Fan RK Chung. *Spectral graph theory*, volume 92. American Mathematical Soc., 1997. 3

[75] Daniel Spielman. *Spectral and algebraic graph theory*. Yale lecture notes, 2019. 3

[76] 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, January 2021. 4, 169

[77] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Accurate, efficient and scalable training of graph neural networks. *Journal of Parallel and Distributed Computing*, 147:166–183, January 2021. ISSN 07437315. DOI:10.1016/j.jpdc.2020.08.011. 4

[78] Keyu Duan, Zirui Liu, Peihao Wang, Wenqing Zheng, Kaixiong Zhou, Tianlong Chen, Xia Hu, and Zhangyang Wang. A comprehensive study on large-scale graph training: Benchmarking and rethinking. In *Advances in Neural Information Processing Systems*, volume 35, pages 5376–5389, 2022. 4, 17, 163, 169, 178

[79] Lu Ma, Zeang Sheng, Xunkai Li, Xinyi Gao, Zhezheng Hao, Ling Yang, Wentao Zhang, and Bin Cui. Acceleration algorithms in GNNs: A survey, May 2024. arXiv:2405.04114. 4, 17, 169, 178

[80] Rui Xue, Haoyu Han, Tong Zhao, Neil Shah, Jiliang Tang, and Xiaorui Liu. Large-scale graph neural networks: The past and new frontiers. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5835–5836, Long Beach, CA, USA, August 2023. ACM. DOI:10.1145/3580305.3599565. 4, 31

[81] Yanyan Shen, Lei Chen, Jingzhi Fang, Xin Zhang, Shihong Gao, and Hongbo Yin. Efficient training of graph neural networks on large graphs. *Proceedings of the VLDB Endowment*, 17(12):4237–4240, 2024. ISSN 2150-8097. DOI:10.14778/3685800.3685844. 4, 31, 163

[82] Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Rong Gu, and Yihua Huang. Empirical analysis of performance bottlenecks in graph neural network training and inference with GPUs. *Neurocomputing*, 446:165–191, 2021. ISSN 18728286. DOI:10.1016/j.neucom.2021.03.015. 4

[83] 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. 4, 169

[84] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys*, 54(9):1–38, December 2022. ISSN 0360-0300, 1557-7341. 4, 169

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

[86] 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. 4

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. 4, 26, 87

[88] 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. arXiv:2412.04738. 4, 26

[89] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do Transformers really perform badly for graph representation? In *Advances in Neural Information Processing Systems*, volume 34, pages 28877–28888, 2021. 4, 26, 27, 76, 87, 89, 90, 101, 102

[90] Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. In *Advances in Neural Information Processing Systems*, volume 34, pages 13266–13279, 2021. 4, 26, 87

[91] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, volume 31, pages 5165–5175, 2018. 5, 29, 35, 197

[92] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Labeling trick: A theory of using graph neural networks for multi-node representation learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 9061–9073, 2021. 5, 29, 197

[93] Haoteng Yin, Muhan Zhang, Jianguo Wang, and Pan Li. SUREL+: Moving from walks to sets for scalable subgraph-based graph representation learning. *Proceedings of the VLDB Endowment*, 16(11):2939–2948, 2023. 5, 29, 119, 122

[94] Balasubramaniam Srinivasan and Bruno Ribeiro. On the equivalence between positional node embeddings and structural graph representations. In *8th International Conference on Learning Representations*, 2020. 5

[95] Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. In *Advances in Neural Information Processing Systems*, volume 33, 2020. 5

[96] Haoteng Yin, Muhan Zhang, Yanbang Wang, Jianguo Wang, and Pan Li. Algorithm and system co-design for efficient subgraph-based graph representation learning. *Proceedings of the VLDB Endowment*, 15(11):2788–2796, 2022. 5, 29, 122

[97] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. G3: When graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment*, 13(12):2813–2816, August 2020. 6, 154

[98] Yunjae Lee, Jinha Chung, and Minsoo Rhu. SmartSAGE: Training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 932–945. ACM, June 2022. DOI:10.1145/3470496.3527391. 6

[99] Yeonhong Park, Sunhong Min, and Jae W. Lee. Ginex: SSD-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. *Proceedings of the VLDB Endowment*, 15(11):2626–2639, July 2022. ISSN 2150-8097. DOI:10.14778/3551793.3551819. 6

[100] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the 17th European Conference on Computer Systems*, pages 417–434, Rennes, France, March 2022. ACM. DOI:10.1145/3492321.3519557. 6

[101] Shichang Zhang, Atefeh Sohrabizadeh, Cheng Wan, Zijie Huang, Ziniu Hu, Yewen Wang, Yingyan, Lin, Jason Cong, and Yizhou Sun. A survey on graph neural network acceleration: Algorithms, systems, and customized hardware, June 2023. arXiv:2306.14052. 6, 135, 169

[102] Meng Wu, Mingyu Yan, Wenming Li, Xiaochun Ye, Dongrui Fan, and Yuan Xie. A comprehensive survey on GNN characterization, August 2024. arXiv:2408.01902. 6

[103] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019*, pages 443–457, 2019. 6

[104] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, August 2019. ISSN 2150-8097. DOI:10.14778/3352063.3352127. 6, 29

[105] Alok Tripathy, Katherine Yelick, and Aydin Buluc. Reducing communication in graph neural network training. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2020-Novem, 2020. ISSN 21674337. DOI:10.1109/SC41405.2020.00074. 6

[106] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment*, 13(12):3125–3137, August 2020. ISSN 2150-8097. DOI:10.14778/3415478.3415539. 6

[107] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation*, pages 551–568, 2021. 6, 170

[108] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. HET: Scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment*, 15(2):312–320, 2021. ISSN 21508097. DOI:10.14778/3489496.3489511. 6

[109] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4582–4591, Washington, DC, USA, August 2022. ACM. DOI:10.1145/3534678.3539177. 6

[110] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph GNN training for large graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–23, June 2023. ISSN 2836-6573. DOI:10.1145/3589288. 6

[111] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji Rong Wen. Scalable graph neural networks via bidirectional propagation. In *Advances in Neural Information Processing Systems*, volume 33, 2020. 13, 16, 17, 21, 24, 36, 37, 38, 39, 51, 52, 55, 135, 164, 226

[112] 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 and Data Mining*, pages 1686–1696. ACM, August 2021. DOI:10.1145/3447548.3467243. 13, 21, 36, 39, 55, 66, 71, 148, 164, 173, 226

[113] 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. 14, 61, 64

[114] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. Geom-GCN: Geometric graph convolutional networks. *8th International Conference on Learning Representations*, 2020. 14, 22, 62, 64, 66, 72, 89, 180

[115] Xiaowen Dong, Dorina Thanou, Laura Toni, Michael Bronstein, and Pascal Frossard. Graph signal processing for machine learning: A review and new perspectives. *IEEE Signal processing magazine*, 37(6):117–127, 2020. 14, 164, 170

[116] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research*, 21(1):2648–2720, 2020. 15, 30

[117] Shubham Gupta and Srikanta Bedathur. A survey on temporal graph representation learning and generative modeling, 2022. arXiv:2208.12126. 15

[118] 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. 16, 62, 68, 164

[119] Sudhanshu Chanpuriya and Cameron Musco. Simplified graph convolution with heterophily. In *Advances in Neural Information Processing Systems*, volume 35, pages 27184–27197, 2022. 16

[120] 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 and Data Mining*, pages 257–266, 2019. 16, 17, 18, 24, 35, 36, 51

[121] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based learning method. In *7th International Conference on Learning Representations*, 2019. 16, 18, 36, 51, 72, 180

[122] 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*, volume 139. PMLR, 2021. 16, 18, 36, 51

[123] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized PageRank. In *7th International Conference on Learning Representations*, pages 1–15, 2019. 16, 20, 36, 38, 54, 66, 149, 165, 173, 175, 221, 226

[124] 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 and Data Mining*, pages 2464–2473, Virtual Event, CA, USA, August 2020. ACM. DOI:10.1145/3394486.3403296. 16, 20, 51, 54, 55, 66, 72, 226

[125] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *36th International Conference on Machine Learning*, volume 97, pages 6861–6871. PMLR, June 2019. 16, 21, 36, 37, 39, 66, 72, 149, 172, 173, 221, 225

[126] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *6th International Conference on Learning Representations*, pages 1–15, 2018. 18, 35, 36, 136

[127] 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 *Advances in Neural Information Processing Systems*, volume 33, 2019. 18, 136

[128] Haipeng Ding, Zhewei Wei, and Yuhang Ye. Large-scale spectral graph neural networks via Laplacian sparsification. In *31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025. 19

[129] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, 1999. 20, 36, 66, 175, 213, 226

[130] 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, 2007. DOI:10.1007/978-3-540-77004-6_12. 20

[131] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using PageRank vectors. In *47th Annual IEEE Symposium on Foundations of Computer Science*, pages 475–486. IEEE, 2006. 20, 40, 43, 49

[132] Hao Zhu and Piotr Koniusz. Simple spectral graph convolution. In *9th International Conference on Learning Representations*, 2021. 21, 66, 68, 173, 226

[133] Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In *Advances in Neural Information Processing Systems*, volume 32, 2019. 21, 66, 165, 173, 226

[134] 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. arXiv:2006.07988. 22, 23, 62, 173, 175, 227

[135] 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, CA, USA, 2019. PMLR. 22, 62, 67, 72

[136] 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 *Advances in Neural Information Processing Systems*, volume 34, 2021. 22, 23, 62, 64, 66, 72, 73, 76, 103, 169, 180, 185

[137] 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. 23, 62, 69, 173, 232

[138] 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. arXiv:2102.06462. 23, 62, 66, 199, 203

[139] 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 *Advances in Neural Information Processing Systems*, volume 35, 2022. 23, 62, 63, 67, 173, 177, 231

[140] 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. arXiv:2202.07082. 23, 61

[141] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. DropEdge: Towards deep graph convolutional networks on node classification. In *8th International Conference on Learning Representations*, March 2020. 24

[142] Cheng Zheng, Bo Zong, Wei Cheng, Dongjin Song, Jingchao Ni, Wenchao Yu, Haifeng Chen, and Wei Wang. Robust graph representation learning via neural sparsification. In *37th International Conference on Machine Learning*, volume 119, pages 11458–11468. PMLR, 2020. 24, 135

[143] Eitan Kosman, Joel Oren, and Dotan Di Castro. LSP: Acceleration of graph neural networks via locality sensitive pruning of graphs. In *IEEE International Conference on Data Mining Workshops*, pages 690–697, Orlando, FL, USA, November 2022. IEEE. 24

[144] Dongyue Li, Tao Yang, Lun Du, Zhezhi He, and Li Jiang. AdaptiveGCN: Efficient GCN through adaptively sparsifying graphs. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 3206–3210, Virtual Event, Queensland, Australia, October 2021. ACM. 24, 135

[145] Jiayu Li, Tianyun Zhang, Hao Tian, Shengmin Jin, Makan Fardad, and Reza Zafarani. Graph sparsification with graph convolutional networks. *International Journal of Data Science and Analytics*, 13(1):33–46, January 2022. 24, 135

[146] Rakshith S. Srinivasa, Cao Xiao, Lucas Glass, Justin Romberg, and Jimeng Sun. Fast graph attention networks using effective resistance based graph sparsification, October 2020. 24, 136, 139

[147] Zirui Liu, Kaixiong Zhou, Zhimeng Jiang, Li Li, Rui Chen, Soo-Hyun Choi, and Xia Hu. DSpar: An embarrassingly simple strategy for efficient GNN training and inference via degree-based sparsification. *Transactions on Machine Learning Research*, July 2023. 24, 135, 136, 149

[148] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, January 2011. 24, 138

[149] Qimai Li, Zhichao Han, and Xiao Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018. arXiv:1801.07606. 24, 145, 199

[150] Wentao Zhang, Mingyu Yang, Zeang Sheng, Yang Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. Node dependent local smoothing for scalable graph learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 20321–20332, 2021. 25, 149

[151] Keke Huang, Jing Tang, Juncheng Liu, Renchi Yang, and Xiaokui Xiao. Node-wise diffusion for scalable graph learning. In *Proceedings of the ACM Web Conference 2023*, pages 1723–1733, Austin, TX, USA, April 2023. ACM. DOI:10.1145/3543507.3583408. 25, 149

[152] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. Decoupling the depth and scope of graph neural networks. In *Advances in Neural Information Processing Systems*, volume 34, pages 19665–19679, 2021. 25, 135

[153] Kwei-Herng Lai, Daochen Zha, Kaixiong Zhou, and Xia Hu. Policy-GNN: Aggregation optimization for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 461–471, Virtual Event, CA, USA, August 2020. ACM. 25

[154] Indro Spinelli, Simone Scardapane, and Aurelio Uncini. Adaptive propagation graph convolutional network. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10):4755–4760, October 2021. 25

[155] Xupeng Miao, Wentao Zhang, Yingxia Shao, Bin Cui, Lei Chen, Ce Zhang, and Jiawei Jiang. Lasagne: A multi-layer graph convolutional network framework via node-aware deep architecture. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021. 25

[156] Wentao Zhang, Ziqi Yin, Zeang Sheng, Yang Li, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. Graph attention multi-layer perceptron. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4560–4570. ACM, August 2022. 25

[157] Shuai Zhang, Meng Wang, Pin-Yu Chen, Sijia Liu, Songtao Lu, and Miao Liu. Joint edge-model sparse learning is provably efficient for graph neural networks. In *11th International Conference on Learning Representations*, 2023. 25, 139, 147, 158, 223

[158] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. Accelerating large scale real-time GNN inference using channel pruning. *Proceedings of the VLDB Endowment*, 14(9):1597–1605, 2021. 25, 136

[159] Chuang Liu, Xueqi Ma, Yibing Zhan, Liang Ding, Dapeng Tao, Bo Du, Wenbin Hu, and Danilo P. Mandic. Comprehensive graph gradual pruning for sparse training in graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2023. 25, 149

[160] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations*, pages 1–42, 2019. 25

[161] Bo Hui, Da Yan, Xiaolong Ma, and Wei-Shinn Ku. Rethinking graph lottery tickets: Graph sparsity matters. In *11th International Conference on Learning Representations*, 2023. 25

[162] Haoran You, Zhihan Lu, Zijian Zhou, Yonggan Fu, and Yingyan Lin. Early-bird GCNs: Graph-network co-optimization towards more efficient GCN training and inference via drawing early-bird lottery tickets. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(8):8910–8918, June 2022. 25, 136, 149

[163] Yong-Duo Sui, Xiang Wang, Tianlong Chen, Meng Wang, Xiang-Nan He, and Tat-Seng Chua. Inductive lottery ticket learning for graph neural networks. *Journal of Computer Science and Technology*, 39(6):1223–1237, 2024. 25

[164] Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang. A unified lottery ticket hypothesis for graph neural networks. In *38th International Conference on Machine Learning*, volume 139, pages 1695–1706. PMLR, 2021. 25, 136, 149

[165] Kun Wang, Yuxuan Liang, Pengkun Wang, Xu Wang, Pengfei Gu, Junfeng Fang, and Yang Wang. Searching lottery tickets in graph neural networks: A dual perspective. In *11th International Conference on Learning Representations*, 2023. 25, 136

[166] 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. In *Advances in Neural Information Processing Systems*, volume 34, 2021. 25, 36

[167] Mehdi Bahri, Gaetan Bahl, and Stefanos Zafeiriou. Binary graph neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9492–9501, 2021. 25

[168] Junfu Wang, Yunhong Wang, Zhen Yang, Liang Yang, and Yuanfang Guo. Bi-GCN: Binary graph convolutional network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1561–1570, 2021. 25

[169] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. GraphZoom: A multi-level spectral approach for accurate and scalable graph embedding. In *8th International Conference on Learning Representations*, February 2020. 25, 173, 225

[170] Chen Cai, Dingkang Wang, and Yusu Wang. Graph coarsening with neural networks. In *9th International Conference on Learning Representations*, February 2021. 25

[171] 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 and Data Mining*, volume 1, pages 675–684, Virtual Event, Singapore, August 2021. ACM. 25, 36

[172] Wei Jin, Lingxiao Zhao, Shichang Zhang, Yozen Liu, Jiliang Tang, and Neil Shah. Graph condensation for graph neural networks. In *10th International Conference on Learning Representations*, September 2022. 25, 135

[173] Zaixi Zhang, Qi Liu, Qingyong Hu, and Chee-Kong Lee. Hierarchical graph Transformer with adaptive node sampling. In *Advances in Neural Information Processing Systems*, volume 35, pages 21171–21183, 2022. 26, 27, 28, 88, 89, 90, 102, 104, 195

[174] Cong Chen, Chaofan Tao, and Ngai Wong. LiteGT: Efficient and lightweight graph Transformers. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 161–170, 2021. 26

[175] Devin Kreuzer, Dominique Beaini, Will Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph Transformers with spectral attention. In *Advances in Neural Information Processing Systems*, volume 34, pages 21618–21629, 2021. 26

[176] Md Shamim Hussain, Mohammed J Zaki, and Dharmashankar Subramanian. Global self-attention as a replacement for graph convolution. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 655–665, 2022. 26, 87

[177] Wonpyo Park, Woong-Gi Chang, Donggeon Lee, Juntae Kim, et al. GRPE: Relative positional encoding for graph Transformer. In *ICLR2022 Machine Learning for Drug Discovery*, 2022. 26, 27

[178] Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. Structure-aware Transformer for graph representation learning. In *10th International Conference on Machine Learning*, pages 3469–3489. PMLR, 2022. 26

[179] Haiteng Zhao, Shuming Ma, Dongdong Zhang, Zhi-Hong Deng, and Furu Wei. Are more layers beneficial to graph Transformers? *11th International Conference on Learning Representations*, 2023. 26, 89

[180] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph Transformer. In *Advances in Neural Information Processing Systems*, volume 35, pages 14501–14515, 2022. 26, 27, 87, 89, 90

[181] Qitian Wu, Wentao Zhao, Zenan Li, David P Wipf, and Junchi Yan. Nodeformer: A scalable graph structure learning Transformer for node classification. In *Advances in Neural Information Processing Systems*, volume 35, pages 27387–27401, 2022. 26, 27, 28, 87, 89

[182] Qitian Wu, Chenxiao Yang, Wentao Zhao, Yixuan He, David Wipf, and Junchi Yan. DIFFormer: Scalable (graph) Transformers induced by energy constrained diffusion. In *11th International Conference on Learning Representations*, 2023. 26, 27, 28, 87, 104

[183] Chenhui Deng, Zichao Yue, and Zhiru Zhang. Polynormer: Polynomial-expressive graph Transformer in linear time. *12th International Conference on Learning Representations*, 2024. 26, 27, 28, 88, 104

[184] Jinsong Chen, Kaiyuan Gao, Gaichao Li, and Kun He. NAGphormer: A tokenized graph Transformer for node classification in large graphs. In *11th International Conference on Learning Representations*, February 2023. 27, 28, 88, 89, 104, 195

[185] Jiahong Ma, Mingguo He, and Zhewei Wei. PolyFormer: Scalable graph Transformer via polynomial attention, 2023. OpenReview:vUgeBN7F9l. 27, 28, 88

[186] Kezhi Kong, Jiuhai Chen, John Kirchenbauer, Renkun Ni, C Bayan Bruss, and Tom Goldstein. GOAT: A global Transformer on large-scale graphs. In *11th International Conference on Machine Learning*, pages 17375–17390. PMLR, 2023. 27, 29, 88, 89, 104

[187] Wenhao Zhu, Tianyu Wen, Guojie Song, Xiaojun Ma, and Liang Wang. Hierarchical Transformer for scalable graph learning. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence*, pages 4702–4710, 2023. 27, 29, 88, 89, 90, 104

[188] Muhan Zhang and Yixin Chen. Inductive matrix completion based on graph neural networks. In *7th International Conference on Learning Representations*, 2019. 29, 113

[189] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1959–1969, 2020. 29

[190] Yunyu Liu, Jianzhu Ma, and Pan Li. Neural predicting higher-order patterns in temporal networks. In *Proceedings of the ACM Web Conference 2022*, page 1340–1351, 2022. 29

[191] Kexin Huang and Marinka Zitnik. Graph meta learning via local subgraphs. In *Advances in Neural Information Processing Systems*, volume 33, pages 5862–5874, 2020. 29

[192] Lecheng Kong, Yixin Chen, and Muhan Zhang. Geodesic graph neural network for efficient graph representation learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 5896–5909, 2022. 29

[193] Pei-Kai Yeh, Hsi-Wen Chen, and Ming-Syan Chen. Random walk conformer: learning graph representation from long and short range. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, number 9, pages 10936–10944, 2023. 29

[194] Gaspard Michel, Giannis Nikolentzos, Johannes F Lutzeyer, and Michalis Vazirgiannis. Path neural networks: Expressive and accurate graph neural networks. In *11th International Conference on Machine Learning*, pages 24737–24755. PMLR, 2023. 29

[195] Emily Alsentzer, Samuel Finlayson, Michelle Li, and Marinka Zitnik. Subgraph neural networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 8017–8029, 2020. 29, 113

[196] Komal Teru, Etienne Denis, and Will Hamilton. Inductive relation prediction by subgraph reasoning. In *8th International Conference on Machine Learning*, pages 9448–9457. PMLR, 2020. 29

[197] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, pages 271–279, 2003. 29

[198] Jiaxuan You, Tianyu Du, and Jure Leskovec. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2358–2366, 2022. 30

[199] Yanping Zheng, Zhewei Wei, and Jiajun Liu. Decoupled graph neural networks for large dynamic graphs. *Proceedings of the VLDB Endowment*, 2023. 30, 126

[200] Long Short-Term Memory. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 2010. 30

[201] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. TGL: A general framework for temporal gnn training on billion-scale graphs. *Proceedings of the VLDB Endowment*, 2022. 30, 119, 125

[202] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. Zebra: When temporal graph neural networks meet temporal personalized PageRank. *Proceedings of the VLDB Endowment*, 16(6):1332–1345, 2023. ISSN 2150-8097. DOI:10.14778/3583140.3583150. 30, 125, 126, 127

[203] Yuhong Luo and Pan Li. Neighborhood-aware scalable temporal network representation learning. In *Learning on Graphs Conference*. PMLR, 2022. 30

[204] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. Inductive representation learning in temporal networks via causal anonymous walks. *9th International Conference on Learning Representations*, 2021. 30, 113, 114, 116, 117, 118, 119, 125, 126

[205] Ming Jin, Yuan-Fang Li, and Shirui Pan. Neural temporal walks: Motif-aware representation learning on continuous-time dynamic graphs. In *Advances in Neural Information Processing Systems*, volume 35, pages 19874–19886, 2022. 30, 114, 116, 117, 119, 126

[206] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs, 2020. arXiv:2006.10637. 30, 126

[207] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. Do we really need complicated model architectures for temporal networks? In *11th International Conference on Learning Representations*, 2023. 30

[208] Yuxing Tian, Yiyan Qi, and Fan Guo. FreeDyG: Frequency enhanced continuous-time dynamic graph model for link prediction. In *12th International Conference on Learning Representations*, 2024. 30

[209] 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 and Data Mining*, 2018. 35

[210] 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. ACM, 2017. 35

[211] 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. 35

[212] Zhengdao Chen, Joan Bruna, and Lisha Li. Supervised community detection with line graph neural networks. In *7th International Conference on Learning Representations*, 2019. 35

[213] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. In *35th International Conference on Machine Learning*, volume 3, pages 1503–1532, 2018. ISBN 9781510867963. arXiv:1710.10568. 36

[214] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning, 2018. arXiv:1803.03735. 36

[215] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-BERT: Only attention is needed for learning graph representations, 2020. arXiv:2008.08617. 36

[216] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. FORA: Simple and effective approximate single-source personalized PageRank. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 505–514, 2017. 39, 40, 41, 43, 49

[217] 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, December 2019. ISSN 0362-5915. arXiv:1908.10583. 39, 42, 214

[218] 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 and Data Mining*, pages 2847–2856, 2018. 40

[219] Lichao Sun, Yingtong Dou, Carl Yang, Kai Zhang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. Adversarial attack and defense on graph data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(8):7693–7711, 2023. DOI:10.1109/TKDE.2022.3201243. 40

[220] 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 ACM SIGMOD 2021 International Conference on Management of Data*, volume 1, pages 1996–2008, June 2021. 40

[221] 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 *IEEE 36th International Conference on Data Engineering*, pages 913–924. IEEE, 2020. 41

[222] 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, January 2005. ISSN 1542-7951. 42

[223] Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM*, 58(3):1–37, 2011. 45, 47

[224] 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. 45

[225] Zhouchen Lin, Risheng Liu, and Zhixun Su. Linearized alternating direction method with adaptive penalty for low-rank representation. In *Advances in Neural Information Processing Systems*, volume 24, 2011. 45

[226] Matthew Coudron and Gilad Lerman. On the sample complexity of robust PCA. In *Advances in Neural Information Processing Systems*, volume 25, 2012. 47

[227] Yanping Zheng, Zhewei Wei, and Jiajun Liu. Decoupled graph neural networks for large dynamic graphs. *Proceedings of the VLDB Endowment*, 16(9):2239–2247, May 2023. ISSN 2150-8097. DOI:10.14778/3598581.3598595. 59

[228] Yueyang Wang, Ziheng Duan, Yida Huang, Haoyan Xu, Jie Feng, and Anni Ren. MTHetGNN: A heterogeneous graph embedding framework for multivariate time series forecasting, 2020. arXiv:2008.08617. 59

[229] 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, page 1287–1297. ACM, 2020. DOI:10.1145/3366423.3380204. 61

[230] 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 and Data Mining*, 2021. 62, 66

[231] 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 and Data Mining*, KDD '20, pages 1243–1253. ACM, 2020. ISBN 9781450379984. DOI:10.1145/3394486.3403177. 63, 231

[232] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003. DOI:10.1162/089976603321780317. 66

[233] Renjie Liao, Zhizhen Zhao, Raquel Urtasun, and Richard S Zemel. LanczosNet: Multi-scale deep graph convolutional networks. In *7th International Conference on Learning Representations*, 2019. 66, 171

[234] 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 and Knowledge Management*, pages 1202–1211, Virtual Event, Queensland, Australia, October 2021. ACM. DOI:10.1145/3459637.3482225. 69, 137, 221, 222

[235] 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. 72, 103, 104, 105, 169, 180, 185

[236] Aditya Grover and Jure Leskovec. Node2Vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864. ACM, 2016. 76

[237] Wenhao Zhu, Tianyu Wen, Guojie Song, Liang Wang, and Bo Zheng. On structural expressive power of graph Transformers. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3628–3637, Long Beach, CA, USA, August 2023. ACM. DOI:10.1145/3580305.3599451. 87, 90

[238] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, pages 1601–1606, 2013. 88, 92, 96

[239] Takuya Akiba, Yoichi Iwata, Ken ichi Kawarabayashi, and Yuki Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. *Proceedings of the Meeting on Algorithm Engineering and Experiments*, pages 147–154, 2014. 88

[240] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, December 2017. ISSN 2150-8097. 88

[241] Wendong Bi, Lun Du, Qiang Fu, Yanlin Wang, Shi Han, and Dongmei Zhang. Make heterophilic graphs better fit GNN: A graph rewiring approach. *IEEE Transactions on Knowledge and Data Engineering*, 36(12):8744–8757, December 2024. ISSN 1558-2191. DOI:10.1109/TKDE.2024.3441766. 89

[242] Qimai Li, Xiaotong Zhang, Han Liu, Quanyu Dai, and Xiao-Ming Wu. Dimensionwise separable 2-D graph convolution for unsupervised and semi-supervised learning on graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 953–963, Virtual Event, Singapore, August 2021. ACM. DOI:10.1145/3447548.3467413. 89, 171

[243] Xiyuan Wang and Muhan Zhang. How powerful are spectral graph neural networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pages 23341–23362. PMLR, 17–23 Jul 2022. arXiv:2205.11172. 89, 163, 167, 168, 172, 173, 175, 230

[244] Yao Ma, Xiaorui Liu, Neil Shah, and Jiliang Tang. Is homophily a necessity for graph neural networks? In *10th International Conference on Learning Representations*, 2022. arXiv:2106.06134. 89

[245] Grégoire Mialon, Dexiong Chen, Margot Selosse, and Julien Mairal. GraphiT: Encoding graph structure in Transformers, June 2021. arXiv:2106.05667. 90

[246] Liheng Ma, Chen Lin, Derek Lim, Adriana Romero-Soriano, Puneet K. Dokania, Mark Coates, Philip Torr, and Ser-Nam Lim. Graph inductive biases in Transformers without message passing. In *40th International Conference on Machine Learning*, volume 202, pages 23321–23337. PMLR, May 2023. arXiv:2305.17589. 90

[247] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, pages 230–241. Springer Berlin Heidelberg, 2011. 91

[248] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In Leah Epstein and Paolo Ferragina, editors, *Algorithms – ESA 2012*, pages 24–35. Springer Berlin Heidelberg, 2012. 91

[249] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, June 2013. DOI:10.1145/2463676.2465315. 91, 93, 95, 97, 100

[250] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, number 1, 2015. 91

[251] Yuxuan Shi, Gong Cheng, and Evgeny Kharlamov. Keyword search over knowledge graphs via static and dynamic hub labelings. In *Proceedings of The Web Conference 2020*, pages 235–245. ACM, April 2020. DOI:10.1145/3366423.3380110. 92, 96

[252] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Xuemin Lin, and Ying Zhang. When hierarchy meets 2-hop-labeling: efficient shortest distance and path queries on road networks. *The VLDB Journal*, 32(6):1263–1287, November 2023. DOI:10.1007/s00778-023-00789-x. 92, 96, 100

[253] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003. DOI:10.1137/S0097539702403098. 94

[254] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93, Sep. 2008. 103, 180

[255] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. In *Advances in Neural Information Processing Systems - Relational Representation Learning Workshop*, volume 32, Montréal, Canada, June 2019. 103, 148

[256] Qitian Wu, Wentao Zhao, Chenxiao Yang, Hengrui Zhang, Fan Nie, Haitian Jiang, Yatao Bian, and Junchi Yan. Simplifying and empowering Transformers for large-graph representations. In *Advances in Neural Information Processing Systems*, volume 36, 2023. 104

[257] Xiao Liu, Shunmei Meng, Qianmu Li, Lianyong Qi, Xiaolong Xu, Wanchun Dou, and Xuyun Zhang. SMEF: Social-aware multi-dimensional edge features-based graph representation learning for recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 2023. 113, 114

[258] Lei Cai, Zhengzhang Chen, Chen Luo, Jiaping Gui, Jingchao Ni, Ding Li, and Haifeng Chen. Structural temporal graph neural networks for anomaly detection in dynamic graphs. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 3747–3756, 2021. 113

[259] Austin R Benson, David F Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016. 113

[260] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2017. 113

[261] Cheng Wu, Chaokun Wang, Jingcao Xu, Ziwei Fang, Tiankai Gu, Changping Wang, Yang Song, Kai Zheng, Xiaowei Wang, and Guorui Zhou. Instant representation learning for recommendation over large dynamic graphs. *IEEE 39th International Conference on Data Engineering*, 2023. 114

[262] Jiasheng Zhang, Jie Shao, and Bin Cui. StreamE: Learning to update representations for temporal knowledge graphs in streaming scenarios. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 622–631, 2023. 114

[263] Torben Hagerup, Kurt Mehlhorn, and J Ian Munro. Maintaining discrete probability distributions optimally. In *Automata, Languages and Programming: 20th International Colloquium, ICALP 93 Lund, Sweden, July 5–9, 1993 Proceedings 20*, pages 253–264. Springer, 1993. 116

[264] Frank Olken and Doron Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5:25–42, 1995. 116

[265] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory of Computing Systems*, 36:329–358, 2003. 116

[266] Fangyuan Zhang, Mengxu Jiang, and Sibo Wang. Efficient dynamic weighted set sampling and its extension. *Proceedings of the VLDB Endowment*, 17(1):15–27, 2023. 116, 120

[267] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. A deep dive into understanding the random walk-based temporal graph learning. In *IEEE International Symposium on Workload Characterization*, 2021. 119

[268] Lijie Chen, Gillat Kol, Dmitry Paramonov, Raghuvansh Saxena, Zhao Song, and Huacheng Yu. Near-optimal two-pass streaming algorithm for sampling random walks over directed graphs. *International Colloquium on Automata, Languages and Programming*, 2021. 123

[269] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932, 2009. 125

[270] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. Temporal graph benchmark for machine learning on temporal graphs. In *Advances in Neural Information Processing Systems*, volume 36, 2024. 125

[271] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2628–2638, 2021. 125, 126

[272] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1269–1278, 2019. 126

[273] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *8th International Conference on Learning Representations*, 2020. 126

[274] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. 127

[275] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys*, 51(3):1–34, May 2019. 135

[276] By Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, May 2020. 136, 145

[277] Meiqi Zhu, Xiao Wang, Chuan Shi, Houye Ji, and Peng Cui. Interpreting and unifying graph neural networks with an optimization framework. In *Proceedings of the Web Conference 2021*, pages 1215–1226. ACM, April 2021. 137, 173, 222, 232

[278] Joshua Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM*, 56(8):87–94, August 2013. 138

[279] Veeranjaneyulu Sadhanala, Yu-Xiang Wang, and Ryan J Tibshirani. Graph sparsification approaches for Laplacian smoothing. In *19th International Conference on Artificial Intelligence and Statistics*, pages 1250–1259, Cadiz, Spain, May 2016. PMLR. 138, 218

[280] Daniele Calandriello, Ioannis Koutis, Alessandro Lazaric, and Michal Valko. Improved large-scale graph learning through ridge spectral sparsification. In *35th International Conference on Machine Learning*, volume 80, Stockholm, Sweden, 2018. PMLR. 138, 218

[281] Neophytos Charalambides and Alfred O. Hero. Graph sparsification by approximate matrix multiplication. In *IEEE Statistical Signal Processing Workshop*, pages 180–184, Hanoi, Vietnam, July 2023. IEEE. 138

[282] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. 141

[283] Yash Deshpande, Subhabrata Sen, Andrea Montanari, and Elchanan Mossel. Contextual stochastic block models. In *Advances in Neural Information Processing Systems*, volume 31, 2018. 141

[284] Aleksandar Bojchevski and Stephan Günnemann. Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking. In *6th International Conference on Learning Representations*, 2018. 141

[285] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Advances in Neural Information Processing Systems*, volume 28, pages 1135–1143, June 2015. 144, 145

[286] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *Procedings of the British Machine Vision Conference 2015*, pages 31.1–31.12, Swansea, UK, 2015. British Machine Vision Association. 144, 145

[287] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *10th International Conference on Learning Representations*, January 2022. 148, 222

[288] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, January 2021. 153

[289] Seiji Maekawa, Koki Noda, Yuya Sasaki, et al. Beyond real-world benchmark datasets: An empirical study of node classification with GNNs. In *Advances in Neural Information Processing Systems*, volume 35, pages 5562–5574, 2022. 158, 169

[290] Seiji Maekawa, Yuya Sasaki, George Fletcher, and Makoto Onizuka. GenCAT: Generating attributed graphs with controlled relationships between classes, attributes, and topology. *Information Systems*, 115:102195, 2023. 158

[291] Chenghua Gong, Yao Cheng, Xiang Li, Caihua Shan, Siqiang Luo, and Chuan Shi. Towards learning from graphs with heterophily: Progress and future, January 2024. arXiv:2401.09769. 163

[292] Defu Cao, Yujing Wang, Juanyong Duan, Ce Zhang, Xia Zhu, Congrui Huang, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, et al. Spectral temporal graph neural network for multivariate time-series forecasting. In *Advances in Neural Information Processing Systems*, volume 33, pages 17766–17778, 2020. 163

[293] Yijing Liu, Qinxian Liu, Jian-Wei Zhang, Haozhe Feng, Zhongwei Wang, Zihan Zhou, and Wei Chen. Multivariate time-series forecasting with temporal polynomial graph neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pages 19414–19426, 2022. 163

[294] Yawei Li, He Chen, Zhaopeng Cui, Radu Timofte, Marc Pollefeys, Gregory S Chirikjian, and Luc Van Gool. Towards efficient graph convolutional networks for point cloud handling. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3752–3762, 2021. 163

[295] Wei Hu, Jiahao Pang, Xianming Liu, Dong Tian, Chia-Wen Lin, and Anthony Vetro. Graph signal processing for geometric data and beyond: Theory and applications. *IEEE Transactions on Multimedia*, 24:3961–3977, 2021. 163

[296] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. SIGN: Scalable inception graph neural networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, November 2020. 164

[297] Antonio Ortega, Pascal Frossard, Jelena Kovačević, José MF Moura, and Pierre Vandergheynst. Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, 106(5):808–828, 2018. 164, 170

[298] Zhiqian Chen, Fanglan Chen, Lei Zhang, Taoran Ji, Kaiqun Fu, Liang Zhao, Feng Chen, Lingfei Wu, Charu Aggarwal, and Chang-Tien Lu. Bridging the gap between spatial and spectral domains: A unified framework for graph neural networks. *ACM Computing Surveys*, page 3627816, October 2023. ISSN 0360-0300, 1557-7341. DOI:10.1145/3627816. 164, 167, 168, 169, 170, 171

[299] Deyu Bo, Xiao Wang, Yang Liu, Yuan Fang, Yawen Li, and Chuan Shi. A survey on spectral graph neural networks, February 2023. arXiv:2302.05631. 164, 168, 169, 170, 171, 185

[300] Chenchen Feng, Yu He, Shiyang Wen, Guojun Liu, Liang Wang, Jian Xu, and Bo Zheng. DC-GNN: Decoupled graph neural networks for improving and accelerating large-scale E-commerce retrieval. In *Companion Proceedings of the ACM Web Conference 2022*, 2022. DOI:10.1145/3487553.3524203. 164

[301] Mingguo He, Zhewei Wei, and Ji-Rong Wen. Convolutional neural networks on graphs with Chebyshev approximation, revisited. In *Advances in Neural Information Processing Systems*, volume 35, December 2022. 164, 173, 228

[302] Wenzheng Feng, Yuxiao Dong, Tinglin Huang, Ziqi Yin, Xu Cheng, Evgeny Kharlamov, and Jie Tang. GRAND+: Scalable graph random neural networks. In *Proceedings of the ACM Web Conference 2022*, pages 3248–3258. ACM, April 2022. DOI:10.1145/3485447.3512044. 164, 173, 226

[303] Qimai Li, Xiao-Ming Wu, Han Liu, Xiaotong Zhang, and Zhichao Guan. Label efficient semi-supervised learning via graph filtering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9574–9583, Long Beach, CA, USA, June 2019. IEEE. DOI:10.1109/CVPR.2019.00981. 165, 173, 226

[304] Jiarui Feng, Yixin Chen, Fuhai Li, Anindya Sarkar, and Muhan Zhang. How powerful are K-hop message passing graph neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pages 4776–4790, 2022. 168

[305] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 169, 178, 195

[306] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, August 2020. URL http://arxiv.org/abs/1909.01315. 169

[307] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE signal processing magazine*, 30(3):83–98, 2013. 170

[308] Zhiqian Chen, Feng Chen, Rongjie Lai, Xuchao Zhang, and Chang-Tien Lu. Rational neural networks for approximating graph convolution operator on jump discontinuities. In *IEEE International Conference on Data Mining*, pages 59–68, November 2018. 170

[309] Derek Lim, Joshua David Robinson, Lingxiao Zhao, Tess E. Smidt, Suvrit Sra, Haggai Maron, and Stefanie Jegelka. Sign and basis invariant networks for spectral graph representation learning. In *11th International Conference on Learning Representations*, 2023. 171

[310] Deyu Bo, Chuan Shi, Lele Wang, and Renjie Liao. Specformer: Spectral graph neural networks meet Transformers. In *11th International Conference on Learning Representations*, 2023. 171

[311] Bingbing Xu, Huawei Shen, Qi Cao, Yunqi Qiu, and Xueqi Cheng. Graph wavelet neural network. In *7th International Conference on Learning Representations*, 2019. 171

[312] Xuebin Zheng, Bingxin Zhou, Junbin Gao, Yu Guang Wang, Pietro Lió, Ming Li, and Guido Montúfar. How framelets enhance graph neural networks. In *38th International Conference on Machine Learning*, volume 139. PMLR, 2021. 171

[313] Haoyu Geng, Chao Chen, Yixuan He, Gang Zeng, Zhaobing Han, Hua Chai, and Junchi Yan. Pyramid graph neural network: A graph sampling and filtering approach for multi-scale disentangled representations. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pages 518–530. ACM, 2023. ISBN 9798400701030. DOI:10.1145/3580305.3599478. 171

[314] Yuan Gao, Xiang Wang, Xiangnan He, Zhenguang Liu, Huamin Feng, and Yongdong Zhang. Addressing heterophily in graph anomaly detection: A perspective of graph spectrum. In *Proceedings of the ACM Web Conference 2023*, pages 1528–1538, Austin, TX, USA, April 2023. ACM. DOI:10.1145/3543507.3583268. 171, 188

[315] Saurabh Verma and Zhi-Li Zhang. Stability and generalization of graph convolutional neural networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1539–1548, 2019. 171

[316] Mingqi Yang, Yanming Shen, Rui Li, Heng Qi, Qiang Zhang, and Baocai Yin. A new perspective on the effects of spectrum in graph neural networks. In *Proceedings of the 39th International Conference on Machine Learning*, pages 25261–25279. PMLR, June 2022. 171

[317] Hoang Nt and Takanori Maehara. Revisiting graph neural networks: All we have is low-pass filters, 2019. arXiv:1905.09550. 173, 225

[318] Yifei Wang, Yisen Wang, Jiansheng Yang, and Zhouchen Lin. Dissecting the diffusion process in linear graph convolutional networks. In *Advances in Neural Information Processing Systems*, volume 34, pages 5758–5769, 2021. 173, 226

[319] Mingjie Li, Xiaojun Guo, Yifei Wang, Yisen Wang, and Zhouchen Lin. G^2CN: Graph Gaussian convolution networks with concentrated graph filters. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, Proceedings of Machine Learning Research, pages 12782–12796. PMLR, 17–23 Jul 2022. 173, 227, 232

[320] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *7th International Conference on Learning Representations*, 2019. 173, 227

[321] Mingxuan Ju, Shifu Hou, Yujie Fan, Jianan Zhao, Yanfang Ye, and Liang Zhao. Adaptive kernel graph neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, number 6, pages 7051–7058, 2022. 173, 227

[322] Meng Liu, Hongyang Gao, and Shuiwang Ji. Towards deeper graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 338–348, Virtual Event, CA, USA, August 2020. ACM. DOI:10.1145/3394486.3403076. 173, 227

[323] Filippo Maria Bianchi, Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Graph neural networks with convolutional arma filters. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3496–3507, 2021. 173, 228

[324] Yuhe Guo and Zhewei Wei. Clenshaw graph neural networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pages 614–625. ACM, August 2023. DOI:10.1145/3580305.3599275. 173, 228, 229

[325] Mingguo He, Zhewei Wei, Zengfeng Huang, and Hongteng Xu. BernNet: Learning arbitrary graph spectral filters via Bernstein approximation. In *Advances in Neural Information Processing Systems*, volume 34, 2021. 173, 198, 229

[326] Jiali Chen and Liwen Xu. Improved modeling and generalization capabilities of graph neural networks with legendre polynomials. *IEEE Access*, 11:63442–63450, 2023. DOI:10.1109/ACCESS.2023.3289002. 173, 229

[327] Yuhe Guo and Zhewei Wei. Graph neural networks with learnable and optimal polynomial bases. In *40th International Conference on Machine Learning*, volume 202, Honolulu, HI, USA, June 2023. PMLR. arXiv:2302.12432. 173, 175, 230

[328] Yushun Dong, Kaize Ding, Brian Jalaian, Shuiwang Ji, and Jundong Li. AdaGNN: Graph neural networks with adaptive frequency response filter. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pages 392–401, Virtual Event, Queensland, Australia, October 2021. ACM. DOI:10.1145/3459637.3482226. 173, 231

[329] Sitao Luan, Mingde Zhao, Chenqing Hua, Xiao-Wen Chang, and Doina Precup. Complete the missing half: Augmenting aggregation filtering with diversification for graph convolutional networks. In *NeurIPS 2022 Workshop: New Frontiers in Graph Learning*, November 2022. 173, 231

[330] Chanakya Ekbote, Ajinkya Pankaj Deshpande, Arun Iyer, Ramakrishna Bairi, and Sundararajan Sellamanickam. FiGURe: Simple and efficient unsupervised node representations with filter augmentations. In *Advances in Neural Information Processing Systems*, volume 36, December 2023. 173, 177, 233

[331] Isabel M Kloumann, Johan Ugander, and Jon Kleinberg. Block models and personalized PageRank. *Proceedings of the National Academy of Sciences*, 114(1): 33–38, 2017. 175

[332] Pan Li, I Chien, and Olgica Milenkovic. Optimizing generalized PageRank methods for seed-expansion community detection. In *Advances in Neural Information Processing Systems*, volume 32, 2019. 175, 227

[333] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011. 176, 228

[334] Yuankai Luo, Lei Shi, and Xiao-Ming Wu. Classic GNNs are strong baselines: Reassessing GNNs for node classification. In *Advances in Neural Information Processing Systems*, volume 37, June 2024. arXiv:2406.08993. 188

[335] Kaan Sancak, Muhammed Fatih Balin, and Umit Catalyurek. Do we really need complicated graph learning models? – a simple but effective baseline. In *The 3rd Learning on Graphs Conference*, November 2024. 188

[336] Sitao Luan, Chenqing Hua, Qincheng Lu, Liheng Ma, Lirong Wu, Xinyu Wang, Minkai Xu, Xiao-Wen Chang, Doina Precup, Rex Ying, Stan Z. Li, Jian Tang, Guy Wolf, and Stefanie Jegelka. The heterophilic graph learning handbook: Benchmarks, models, theoretical analysis, applications and challenges, July 2024. arXiv:2407.09618. 188

[337] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *9th International Conference on Learning Representations*, March 2021. arXiv:2006.05205. 190

[338] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. OGB-LSC: A large-scale challenge for machine learning on graphs, 2021. arXiv:2103.09430. 196

[339] Juanhui Li, Harry Shomer, Haitao Mao, Shenglai Zeng, Yao Ma, Neil Shah, Jiliang Tang, and Dawei Yin. Evaluating graph neural networks for link prediction: Current pitfalls and new benchmarking. In *Advances in Neural Information Processing Systems*, volume 36, 2023. 196

[340] Liang Yang, Chuan Wang, Junhua Gu, Xiaochun Cao, and Bingxin Niu. Why do attributes propagate in graph convolutional neural networks? *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):4590–4598, May 2021. DOI:10.1609/aaai.v35i5.16588. 199

[341] Lu Wang, Wenchao Yu, Wei Wang, Wei Cheng, Wei Zhang, Hongyuan Zha, Xiaofeng He, and Haifeng Chen. Learning robust representations with graph denoising policy network. In *IEEE International Conference on Data Mining*, pages 1378–1383, Beijing, China, November 2019. IEEE. DOI:10.1109/ICDM.2019.00177. 199

[342] Xianfeng Tang, Huaxiu Yao, Yiwei Sun, Yiqi Wang, Jiliang Tang, Charu Aggarwal, Prasenjit Mitra, and Suhang Wang. Investigating and mitigating degree-related biases in graph convoltuional networks. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*, pages 1435–1444, 2020. 203

[343] Zhengyu Chen, Teng Xiao, and Kun Kuang. BA-GNN: On learning bias-aware graph neural network. In *IEEE 38th International Conference on Data Engineering*, pages 3012–3024, 2022. 203

[344] Jie Liao, Jintang Li, Liang Chen, Bingzhe Wu, Yatao Bian, and Zibin Zheng. SAILOR: Structural augmentation based tail node representation learning. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 1389–1399, 2023. 203

[345] Xunkai Li, Jingyuan Ma, Zhengyu Wu, Daohan Su, Wentao Zhang, Rong-Hua Li, and Guoren Wang. Rethinking node-wise propagation for large-scale graph learning. In *Proceedings of the ACM Web Conference 2024*, February 2024. 203, 205

[346] Yurui Lai, Xiaoyang Lin, Renchi Yang, and Hongtao Wang. Efficient topology-aware data augmentation for high-degree graph neural networks. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, June 2024. arXiv:2406.05482. 203

[347] Ziwei Zhang, Peng Cui, Jian Pei, Xin Wang, and Wenwu Zhu. Eigen-GNN: a graph structure preserving plug-in for GNNs. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021. ISSN 1041-4347, 1558-2191, 2326-3865. DOI:10.1109/TKDE.2021.3112746. 205

[348] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization, 2024. arXiv:2404.16130. 211

[349] Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. GRAG: Graph retrieval-augmented generation, 2024. arXiv:2405.16506. 211

[350] Costas Mavromatis and George Karypis. GNN-RAG: Graph neural retrieval for large language model reasoning, 2024. arXiv:2405.20139. 211

[351] Aladin Virmaux and Kevin Scaman. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*, volume 31, 2018. 223

[352] Nicolas Tremblay, Paulo Gonçalves, and Pierre Borgnat. Design of graph filters and filterbanks. In *Cooperative and Graph Signal Processing*, pages 299–324. Elsevier, 2018. DOI:10.1016/B978-0-12-813677-5.00011-0. 226, 228

[353] Fan Chung. The heat kernel as the PageRank of a graph. *Proceedings of the National Academy of Sciences*, 104(50):19735–19740, 2007. DOI:10.1073/pnas.0708838104. 226

[354] Craig Calcaterra and Axel Boldt. Approximating with Gaussians, 2008. arXiv:0805.3795. 227

[355] WG Horner. A new method of solving numerical equations of all orders, by continuous approximation. In *Abstracts of the Papers Printed in the Philosophical Transactions of the Royal Society of London*, volume 2, pages 117–117. JSTOR, 1815. 228

[356] David I Shuman, Pierre Vandergheynst, and Pascal Frossard. Chebyshev polynomial approximation for distributed signal processing. In *International Conference on Distributed Computing in Sensor Systems and Workshops*, pages 1–8. IEEE, 2011. 228

[357] Amparo Gil, Javier Segura, and Nico M Temme. *Numerical methods for special functions*. SIAM, 2007. 228

[358] Rida T. Farouki. The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design*, 29(6):379–419, 2012. 229

[359] Rida T. Farouki. Legendre–Bernstein basis transformations. *Journal of Computational and Applied Mathematics*, 119(1):145–160, 2000. ISSN 0377-0427. DOI:10.1016/S0377-0427(00)00376-9. 230

[360] Walter Gautschi. *Orthogonal polynomials: computation and approximation*. OUP Oxford, 2004. 230