Deep Learning on Graphs: Tackling Scalability, Privacy, and Multimodality

Minji Yoon

CMU-CS-24-139

July 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Christos Faloutsos, Co-chair
Ruslan Salakhutdinov, Co-chair
Tom M. Mitchell
Jure Leskovec (Stanford University)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Kwanjeong Educational Foundation Scholarship and the Amazon Graduate Research Fellowship.

Keywords: Deep Learning, Graph Mining, Deep Learning on Graphs, Graph Representation Learning, Graph Neural Networks, Graph Convolution Networks, Graph Neural Architecture Search, Message Passing, Importance Neighborhood Sampling on Graphs, Transfer Learning on Graphs, Heterogeneous Graph Neural Networks, Graph Generative Models, Graph Transformer, Multimodal Graphs, Multimodal Learning, Multimodal Graph Learning, Language Models

To my shining star, Théo, who believed in me more than I believed in myself.

Abstract

Graphs are everywhere, from e-commerce to knowledge graphs, abstracting interactions among individual data entities. Various real-world applications running on graph-structured data require effective representations for each part of the graph — nodes, edges, subgraphs, and the entire graph — that encode its essential characteristics.

In recent years, **Deep Learning on Graphs (DLG)** has broken ground across diverse domains by learning graph representations that successfully capture the underlying inductive bias in graphs. However, these groundbreaking DLG algorithms sometimes face limitations when applied to real-world scenarios. First, as graphs can be built on any domain that has interactions among entities, real-world graphs are diverse. Thus, for every new application, domain expertise and tedious work are required for hyperparameter tuning to find an optimal DLG algorithm. Second, scales of real-world graphs keep increasing to billions with unfiltered noise. This requires redundant preprocessing such as graph sampling/noise filtering in advance of DLG to be realized in applications. Next, real-world graphs are mostly proprietary, while many DLG algorithms often assume they have full access to external graphs to learn their distributions or extract knowledge to transfer to other graphs. Finally, the advent of single-modal foundation models in language and vision fields has catalyzed the assembly of diverse modalities, resulting in the formulation of multimodal graphs with diverse modalities on nodes and edges. However, learning on multimodal graphs while exploiting the generative capabilities of each modality's foundation models is an open question in DLG.

In this thesis, I propose to make DLG more practical across four dimensions: 1) **automation**, 2) **scalability**, 3) **privacy**, and 4) **multimodality**. First, we automate algorithm search and hyperparameter tuning under the message-passing framework. Then, we propose to sample each node's neighborhood to regulate the computation cost while filtering out noisy neighbors adaptively for the target task to handle scalability issues. For privacy, we redefine conventional problem definitions, including graph generation and transfer learning, to be aware of the proprietary and privacy-restricted nature of real-world graphs. Finally, I proposed a new multimodal graph learning algorithm that is built on unimodal foundation models and generates content based on multimodal neighbor information.

As the data collected by humanity increases in scale and diversity, the relationships among individual elements increase quadratically in scale and complexity. By making DLG more scalble, privacy-certified, and multimodal, we hope to enable better processing of these relationships and positively impact a wide array of domains.

Acknowledgments

My first and biggest thanks go out to my advisors, Christos Faloutsos and Ruslan Salakhutdinov, for their advice and support. Christos has shown the deepest support and trust in me throughout my Ph.D., which has been the cornerstone of my academic journey. Ruslan has inspired me to explore new horizons and provided invaluable guidance and encouragement that fueled my passion for research. I would also like to thank my other thesis committee members, Tom Mitchell and Jure Leskovec, for their insightful questions and constructive feedback.

I was fortunate to be mentored by Bryan Perozzi during my internship at Google Research. Two chapters of this thesis are based on the work I collaborated with him. I have had the opportunity to work with an amazing group of coauthors, each of whom deserves my gratitude: Kijung Shin, Bryan Hooi, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, Jaewon Yang, John Palowitch, Dustin Zelle, Ziniu Hu, Yue Wu, and Jing Yu Koh. I would especially like to thank Bryan Hooi for mentoring me on how to conduct research by guiding me through my first two papers, and for continuing to broaden my horizons as we collaborated on my final paper in my last year.

I am grateful to all my lab mates for being great friends, coworkers, and mentors: Tiffany, JY, Brandon, Murtaza, Kelly, Paul, and Ben from Ruslan's lab, and Jeremy, Catalina, and Saranya from Christos's lab. I also thank my office mates, Chunkai and Tian, for sharing all the ups and downs in the same office throughout 5 years.

I have been fortunate to have wonderful friends who supported me throughout my Ph.D.: Ojash, Viraj, Sammy, Lauren, Kush, and Sam. Also, my Korean friends from CMU, Daye, Chanyoung, Jimin Sun, Jimin Moon, Emily, Tiffany, and Haewon, and my old friends from Korea who always believed in me, Seobin and Hyomin. Special thanks to my first and last roommate, Daye, who shared all the happiness and frustrations with me in Pittsburgh. I strongly believe I could not have completed my Ph.D. journey without any of them. I would like to express my deepest gratitude to my parents, Woosik and Sunhwa, and my parents-in-law, Odile and Eric, for their infinite love and support that made my Ph.D. journey successful. Last but not least, many thanks go out to my husband, Théophile Gervet, the best thing I got from CMU.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Amidst the recent successes in computer vision and natural language processing, a critical aspect of real-world data—relational information—remains underexplored in AI models. Traditional models primarily process single data entities, such as an image or a sentence, individually during training and inference. In contrast, many real-world applications inherently involve data with rich relational structures, naturally represented as graphs, where nodes symbolize data entities and edges encode relationships among them. Understanding each data entity within a graph provides a holistic view of how it is relevant/related to other entities. For instance, in e-commerce, a product's context is discerned not just through its description but through its neighbor nodes in an e-commerce graph that incorporates user reviews, merchant information, or co-purchased products.

Deep Learning on Graphs (DLG) has proposed various Deep Learning methodologies to learn effective representations for node, edge, subgraph, and graph by capturing the underlying inductive biases on graphs [20, 75]. DLG has broken grounds across various domains, from traditional graph applications such as product/friend recommendations in e-commerce/social platforms [88, 176], misinformation detection on social networks [10], and fraud detection in financial transaction networks [156] to newly introduced graph applications, including ETA prediction in navigation applications [29], pandemic forecasting in epidemiology [22, 112], and drug development in biology [70].

1.1 Challenges

DLG aims to learn from this interconnected world and improve understanding of each data entity using graph structures and neighboring information. However, when we try to realize DLG in practice, we face notable challenges arising from the characteristics of real-world graphs.

- **Diversity:** As graphs can be built on any domain that has interactions among entities, real-world graphs are diverse, from e-commerce graphs to knowledge graphs. These diverse graphs require different optimal hyperparameter sets for each DLG algorithms.
- **Scale:** Scales of real-world graphs keep increasing to billions or trillions with unfiltered noise. This requires redundant graph sampling/noise filtering in advance of DLG to be realized in applications.
- **Privacy:** The rise of privacy concerns and the enactment of relevant laws have constrained the sharing of real-world graphs derived from various industries. This introduces unprecedented

challenges to DLG research, including restricted access to graph datasets of interest and disrupted research assumptions regarding accessibility to external graphs.

- **Multimodality:** The advent of single-modal foundation models in language and vision fields has catalyzed the assembly of diverse modalities across domains, resulting in the formulation of multimodal graphs with diverse modalities on nodes and edges. Learning from multimodal graphs while exploiting the potent generative capabilities of each modality's foundation models is an open question in DLG.

Due to these challenges, an array of DLG research could not fully deliver their impact shown in the academic setting to newly emerged graphs in the industrial setting.

1.2 Contributions

Given the above four challenges — *hyperparameter tuning, scalability, privacy, multimodality* — that hamper the broad adoption of DLG to real-world applications, I define new problems that pave new ways to solve these challenges and propose practical solutions that can be deployable on real-world graphs.

- **Automation:** To democratize DLG for practitioners by removing redundant works, I automate neural architecture search (i.e., hyperparameter tuning) and find the optimal message-passing algorithms given a graph, a task, and a resource budget (Chapter 3).
- **Scalability:** Instead of using the full neighborhoods given in the original graphs, I sample neighbors for each node to regulate the computation cost of DLG algorithms. I adaptively sample neighbors that are informative for a given task, filtering out noisy neighbors automatically (Chapter 4).
- **Privacy:** Instead of resorting to external (likely to be proprietary) graphs, I propose new transfer learning that transfers knowledge within a fully-owned heterogeneous graph, avoiding any access to external graphs (Chapter 5). Additionally, I define a novel graph generation problem that generates substitute graphs following distributions of proprietary graphs in a privacy-enhanced way and diversify benchmark graphs for DLG research (Chapter 6).
- **Multimodality:** I proposed a new multimodal graph learning algorithm that is built on unimodal foundation models and generates content based on multimodal neighbor information. This paradigm holds potential as a foundational approach for applications necessitating intricate multimodal data processing, including decision-making, planning, and recommendation systems (Chapter 7).

Based on this array of works, we help DLG to be easily applied to broader domains, thus bringing a larger impact to the real world.

1.3 Thesis Organization

The rest of the thesis proposal is organized as follows. See Table 1.1 for the main problems of each chapter in the form of questions. In Chapter 2, we give preliminaries on different types of graphs,

graph convolutional networks, and an end-to-end pipeline for DLG. In Chapter 3, we present our work on the automation of DLG hyperparameter search. In Chapters 4, 5 and 6, we present our work on scalable and privacy-enhanced modeling. In Chapter 7, we present our work on multimodal graph learning using pretrained foundation models. Finally, in Chapter 8, we provide conclusions and discuss future research directions.

Table 1.1: **Organization of the thesis.**

Challenge	**Research Problem**	**Chapter**
Diversity	How can we find the optimal hyperparameter sets for diverse types of real-world graphs automatically?	3
Scalability	How can we process large-scale graphs with noise	4
Privacy	How can we learn on proprietary graphs without breaching privacy?	5, 6
Multimodality	How can we learn on multimodal graphs while exploiting each modality's foundation models?	7

Chapter 2

Background

In this chapter, we introduce some key concepts and notations that are used throughout this thesis. The notations are also listed in Table 2.1, which is at the end of this chapter.

2.1 Graph-related Concepts and Notations

Homogeneous graphs denote graphs composed of nodes and edges of the same type/modality. When we state a graph without any additional adjective, it commonly denotes homogeneous graphs. Most of the recent DLG models consume either 1) one single large-scale graph or 2) a set of small/medium-scale graphs. Denoting $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a graph with n nodes $v_i \in \mathcal{V}$ and edges $(v_i, v_j) \in \mathcal{E}$, each graph is given with three components as follows:

- **Adjacency matrix** $\mathcal{A} = (a(v_i, v_j)) \in \mathbb{R}^{n \times n}$ where $a(v_i, v_j)$ is set to 1 when there is an edge from v_i to v_j, otherwise 0.
- **Node attribute matrix** $\mathcal{X} \in \mathbb{R}^{n \times d}$ where x_i denotes the d-dimensional attribute vector of v_i.
- **Node label matrix** $\mathcal{Y} \in \mathbb{R}^n$ where y_i denotes the label of v_i.

Sometimes, instead of node labels, a single graph label is provided for each graph (e.g., molecule graphs). However, in this thesis, we focus on graphs with node labels.

Heterogeneous graphs model the relational data of multi-modal systems. Formally, a heterogeneous graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$ where:

- **Node set** $\mathcal{V}$ consisting of nodes in $\mathcal{G}$.
- **Edge set** $\mathcal{E}$ consisting of ordered tuples $e_{ij} := (i, j)$ with $i, j \in \mathcal{V}$, where $e_{ij} \in \mathcal{E}$ iff an edge exists from i to j.
- **Node type set** $\mathcal{T}$ with associated map $\tau : \mathcal{V} \mapsto \mathcal{T}$.
- **Relation type set** $\mathcal{R}$ with associated map $\phi : \mathcal{E} \mapsto \mathcal{R}$.

This flexible formulation allows directed multi-type edges. Similarly to homogeneous graphs, we additionally assume the existence of node attributes and labels as follows:

- **A set of node attribute matrices** $\{\mathcal{X}_t : t \in \mathcal{T}\}$ where $\mathcal{X}_t$ is a node attribute matrix specific to nodes of type t and a node attribute vector $x_i \in \mathcal{X}_{\tau(i)}$ for each $i \in \mathcal{V}$.
- **A set of node label matrices** $\{\mathcal{Y}_t : t \in \mathcal{T}\}$ where $\mathcal{Y}_t$ is a node label matrix specific to nodes of type t and a node label $y_i \in \mathcal{Y}_{\tau(i)}$ for each $i \in \mathcal{V}$.

Note that the data modality of each node type is not necessarily exclusive (e.g., two node types s, t can share either the same or different attribute spaces).

2.2 Graph Neural Networks

First, we briefly review Graph Convolutional Networks (GCNs) [75], one of the most popular DLG algorithms, which is also repeatedly mentioned throughout this thesis. GCNs stack layers of first-order spectral filters followed by nonlinear activation functions to learn node embeddings. When $h_i^{(l)}$ denotes the hidden embeddings of node v_i in the l-th layer, the simple and general form of GCNs is as follows:

$$h_i^{(l+1)} = \sigma(\frac{1}{n(i)} \sum_{j=1}^{n} a(v_i, v_j) h_j^{(l)} W^{(l)}), \quad l = 0, \ldots, L-1 \tag{2.1}$$

where $n(i) = \sum_{j=1}^{n} a(v_i, v_j)$ is the degree of node v_i; $\sigma(\cdot)$ is a nonlinear function; $W^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the learnable transformation matrix in the l-th layer with $d^{(l)}$ denoting the hidden dimension at the l-th layer. $h_i^{(0)}$ is set with the input node attribute x_i. The last layer embeddings $h_i^{(L)}$ are then fed into the downstream task.

GCNs require the full expansion of neighborhoods across layers, leading to high computation and memory costs. To circumvent this issue, GraphSage [55] adds sampling operations to GCNs to regulate the size of neighborhood. We first recast Equation 2.1 as follows:

$$h_i^{(l+1)} = \alpha_{W^{(l)}}(\mathbb{E}_{j \sim p(j|i)}[h_j^{(l)}]), \quad l = 0, \ldots, L-1 \tag{2.2}$$

where we combine the transformation matrix $W^{(l)}$ into the activation function $\alpha_{W^{(l)}}(\cdot)$ for concision; $p(j|i) = \frac{a(v_i, v_j)}{n(i)}$ defines the probability of sampling v_j given v_i. Then we approximate the expectation by Monte-Carlo sampling as follows:

$$h_i^{(l+1)} = \alpha_{W^{(l)}}(\frac{1}{s} \sum_{j \sim p(j|i)}^{s} h_j^{(l)}), \quad l = 0, \ldots, L-1 \tag{2.3}$$

where s is the number of sampled neighbors for each node. Now, we can regulate the size of neighborhood using s, in other words, the computational footprint for each minibatch.

2.3 Heterogeneous Graph Neural Networks

We briefly describe heterogeneous graph neural networks (HGNN) models that extend GNN to apply on heterogeneous graphs. MPNN (message passing neural networks) [49] is originally designed for homogeneous graphs. We extend MPNN to process heterogeneous graphs by adding projection matrices that project input attributes of different node types into the same feature space before running the original MPNN. R-GCN [125] extends MPNN by specializing message matrices in each edge type, while HMPNN specializes all transformation and message matrices in each node/edge

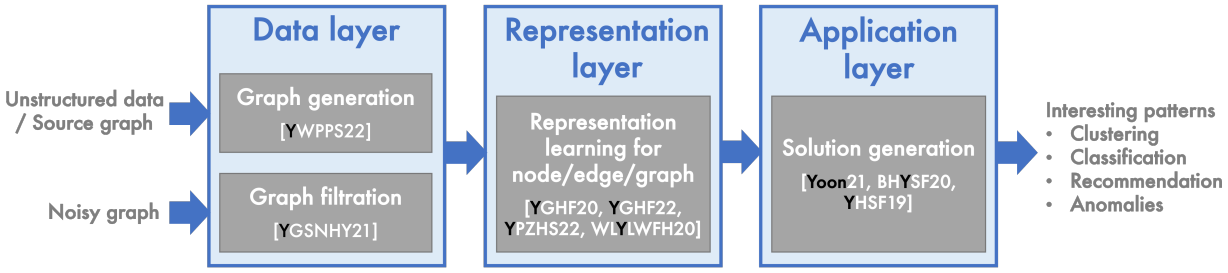


Figure 2.1: **DLG pipeline.** DLG can be decomposed into data, representation, and application layers.

type in MPNN. HGT [65] extends HMPNN by adding attention modules. The attention modules have node-type-specific key/query projection matrices and edge-type-specific key-query similarity matrices, following the transformer architecture. HAN [158] is a meta-path-based model who specializes parameters in each meta-path. HAN exploits meta-path-specific attention modules to aggregate features of neighboring nodes connected by each meta-path. Then HAN aggregates embeddings of different meta-paths with semantic-level attention modules. MAGNN [46] is another meta-path-based HGNN model. MAGNN aggregates features of all nearby nodes sitting on each meta-path using intra-meta-path attention modules. Then MAGNN aggregates features of different meta-paths using inter-meta-path attention modules.

2.4 Graph Neural Networks on Multimodal Graphs

Heterogeneous Graph Neural Networks (HGNNs) extend Graph Neural Networks (GNNs) [193] to learn from multimodal heterogeneous graphs. This is done through precomputing input node embeddings using frozen encoders, and training the GNN to map different modality embeddings either at the input layer [125], intermediate [65], or late layers [182]. However, most HGNN models focus on node classification, and are difficult to adapt for generative tasks. Recently, various approaches have been proposed to fine-tune Large Language Models (LLMs) with GNNs on text-attributed graphs [26, 57, 196]. These methods specialize in node/edge classification tasks by putting GNN models after LLMs, making them difficult to adapt for use in generative tasks.

2.5 End-to-End Pipeline in Deep Learning on Graphs (DLG)

A broad array of problems are studied in DLG, including graph generative models, graph sampling strategies, or node/edge/graph embedding algorithms. These diverse problem domains could puzzle practitioners with questions like "How are those problems related to each other? How could I use them together?". Here, we dissect an end-to-end pipeline of DLG and introduce where our proposed methods are located in the pipeline.

The DLG pipeline receives noisy graphs or unstructured data and outputs interesting patterns hidden in the graphs to solve real-world problems, including clustering, classification, recommendation, and anomaly detection. As shown in Figure 2.1, we dissect this pipeline into three layers, data

layer, representation layer, and application layer, as follows:

- **Data layer**: We generate graphs from users' unstructured data or generate synthetic graphs following distributions of any source graphs. We also improve a given graph by filtering noisy edges. Importance neighborhood sampling described in Chapter 4 and graph generation models described in Chapter 6 belong to the data layer.
- **Representation layer**: We assign scores or embeddings to nodes/edges/subgraphs that encode the graph structure. Chapter 3 describes how to find the optimal representation learning algorithms under the message-passing framework. Chapter 5 describes how to transfer a representation learning model trained on one node type to another node type with different modalities. Chapter 7 presents how to encode multimodal graph context into sequences of node embeddings.
- **Application layer**: We extract patterns from the representations, as required by the users' application (e.g., classify nodes with high scores as anomalies).

I have also conducted several works in the representation and application layers during my Ph.D. studies, which are excluded from this thesis. In the representation layer, we proposed robust representation learning based on a low-pass message-passing mechanism [159]. In the application layer, we proposed [177] and [180] to detect anomalies on static and dynamic graphs, respectively, using Personalized PageRank scores. We also proposed to track edge statistics and detect anomalies on streaming graphs[12].

Chapter 3

Automation

The pervasiveness of graphs today has raised the demand for algorithms to answer various questions: Which products would a user like to purchase given her order list? Which users are buying fake followers? Myriads of new graph algorithms are proposed every year to answer such questions — each with a distinct problem formulation, computational time, and memory footprint. This lack of unity makes it difficult for practitioners to compare different algorithms and pick the most suitable one for their application. These challenges create a gap in which state-of-the-art techniques developed in academia fail to be optimally deployed in real-world applications.

To bridge this gap, we propose AUTOGM, an automated system for graph mining algorithm development. We first define a unified framework UNIFIEDGM for message-passing-based graph algorithms. UNIFIEDGM defines a search space in which five parameters are required to determine a graph algorithm. Under this search space, AUTOGM explicitly optimizes for the optimal parameter set of UNIFIEDGM using Bayesian Optimization. AUTOGM defines a novel budget-aware objective function for the optimization to find the best speed-accuracy trade-off in algorithms under a computation budget. On various real-world datasets, AUTOGM generates novel graph algorithms with the best speed/accuracy trade-off compared to existing models with heuristic parameters.

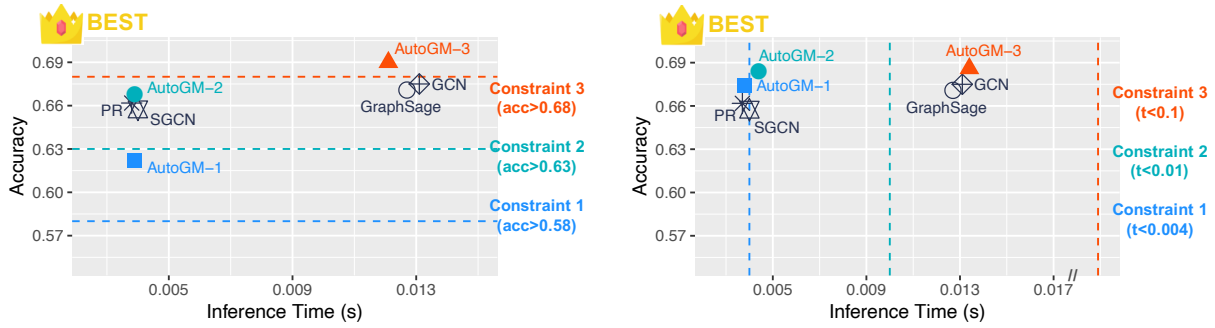
3.1 Motivation

Many real-world problems are naturally modeled using graphs: who-buys-which-products in online marketplaces [180], who-follows-whom in social networks [110, 179], and protein relationships in biological networks [16, 153]. Graph mining provides solutions to practical problems such as classification of web documents [174, 187], clustering in market segmentation [137], recommendation in streaming services [11], and fraud detection in banking [32, 104].

A dizzying array of new graph mining algorithms is introduced every year to solve these real-world problems, giving rise to the question: Which algorithm should we choose for a specific application? Graph mining algorithms designed to solve the same task often have distinct conceptual formulations. Concretely, to estimate the similarity between two nodes — in social recommender systems for example — classical graph mining algorithms (like Personalized PageRank [6]) compute similarity scores by iterating a closed-form expression, while graph neural network algorithms [166] first learn node embeddings using deep learning, then estimate similarity scores with a distance metric in this embedding space. This lack of unity makes it hard for practitioners to determine which aspect of a method contributes to differences in computation time, accuracy, and memory

footprint — significantly complicating the choice of the algorithm. Currently, selecting a graph mining algorithm suitable for a specific task among dozens of candidates is a resource-intensive process requiring expert experience and brute-force search.

To mitigate the cost and complexity of the algorithm selection process, the machine learning community has developed AutoML [72, 95] — which automates the process of algorithm selection and hyperparameter optimization. The success of AutoML depends on the size of the search space: it should be small enough to be tractable in a reasonable amount of time. However, AutoML techniques cannot be directly applied to graph mining because the hyperparameter search space is not even defined due to the lack of unity among graph mining algorithms.



(a) Accuracy constraints on the Citeseer dataset

(b) Time constraints on the Citeseer dataset

Figure 3.1: **AUTOGM finds novel graph algorithms with the best accuracy/inference time trade-off on the node classification task.** (a) Given three accuracy lower bounds (i.e., $0.58, 0.63, 0.68$), AUTOGM generates three novel graph algorithms minimizing inference time. (b) Given three inference time upper bounds (i.e., $0.004, 0.01, 0.1$ seconds), AUTOGM generates three novel graph algorithms maximizing accuracy.

Hence, in this paper, we first unify various graph mining algorithms under our UNIFIEDGM framework, then propose an automated system for graph algorithm development, AUTOGM. We target graph algorithms that pass messages — propagate scores in the PageRank terminology [78, 110] — along edges to summarize the graph structure into nodes statistics. UNIFIEDGM manipulates five parameters of the message passing mechanism: the dimension of the communicated messages, the number of neighbors to communicate with (width), the number of steps to communicate for (length), the nonlinearity of the communication, and the message aggregation strategy. Different parameter settings yield novel graph algorithms, as well as existing algorithms, ranging from conventional graph mining algorithms (like PageRank) to graph neural networks.

Additionally, we introduce UNIFIEDGM-EXT that extends UNIFIEDGM to embrace various attention and sampling methodologies in the message aggregation step. Recently, graph neural networks have adopted attention and importance sampling methodologies to improve their performance and scalability. The attention methodology computes importance/relevance scores of each neighbor with regard to a source node, then uses those scores as weights when we aggregate messages from the neighbors. Importance sampling goes one step further from attention and samples only neighbors with high relevance scores. By extending UNIFIEDGM to embrace attention and importance sampling concepts, we can apply techniques used by graph neural networks to the

conventional graph mining field.

Based on UNIFIEDGM (and UNIFIEDGM-EXT), we propose an automated system for graph algorithm development, AUTOGM. AUTOGM leverages the parameter search space defined in UNIFIEDGM to address a practical problem: given a real-world scenario, what is the graph mining algorithm with the best speed/accuracy trade-off? In real-world scenarios, practitioners optimize performance under a computational budget [83, 90]. AUTOGM defines a novel budget-aware objective function capturing the speed/accuracy trade-off, then maximizes the objective function to find the optimal parameter set of UNIFIEDGM, resulting in a novel graph mining algorithm tailored for the given scenario.

The goal of our work is to empower practitioners without much expertise in graph mining to deploy algorithms tailored to their specific scenarios. The main contributions of this paper are as follows:

- **Unification:** UNIFIEDGM unifies various message-passing based graph algorithms as instantiations of a message-passing framework with five parameters: dimension, width, length, nonlinearity, and aggregation strategy. UNIFIEDGM-EXT extends UNIFIEDGM with attention and sampling options in the message aggregation step.
- **Design space for graph mining algorithms:** UNIFIEDGM provides the parameter search space necessary to automate graph mining algorithm development.
- **Automation:** AUTOGM is an automated system for graph mining algorithm development. Based on the search space defined by UNIFIEDGM, AUTOGM finds the optimal graph algorithm using Bayesian optimization.
- **Budget awareness:** AUTOGM maximizes accuracy of an algorithm under a computational time budget, or minimizes the computational time of an algorithm under a lower bound constraint on accuracy.
- **Effectiveness:** AUTOGM finds novel graph mining algorithms with the best speed/accuracy trade-off compared to existing models with heuristic parameters (Figure 3.1).

Table 3.1 gives a list of symbols and definitions.

Reproducibility: Our code is publicly available [1].

3.2 Unified Graph Mining Framework

In this section, we first motivate the message passing scheme (Section 3.2.1). We then propose our unified framework UNIFIEDGM (Section 3.2.2), explain how existing algorithms fit in the framework (Section 3.2.3), and further analyze how UNIFIEDGM bridges the conceptual gap between conventional graph mining and graph neural networks (Section 3.2.4). Finally, we outline how to choose parameters of UNIFIEDGM given a specific scenario (Section 3.2.5).

[1] https://github.com/minjiyoon/ICDM20-AutoGM

Table 3.1: **Commonly used notation in AUTOGM.**

Symbol	Definition
G	input graph
n, m	numbers of nodes and edges in G
A	$(n \times n)$ binary adjacency matrix of G
d_0	dimension of input feature vectors
d	dimension of communicated messages
k	number of message passing steps
w	number of neighbors sampled per node
l	binary indicator for nonlinearity
a	categorical aggregation strategy
X_0	$(n \times d_0)$ input feature vectors
X_i	$(n \times d)$ i-th layer message vectors $(i = 1 \dots k)$
W_1	$(d_0 \times d)$ 1st layer transformation matrix
W_i	$(d \times d)$ i-th layer transformation matrix $(i = 2 \dots k)$
$\phi(x)$	$\begin{cases} ReLU(x) \text{ if } l = \text{True} \\ x \text{ otherwise} \end{cases}$

3.2.1 Message Passing

A goal common to many graph mining algorithms is to answer queries at the node level (e.g., node clustering, classification, or recommendation) based on global graph information (e.g., edge structure and feature information from other nodes). To transmit the information necessary to answer such queries, in classical graph mining algorithms, nodes *propagate scalar scores to* their neighbors, while in graph neural networks, nodes *aggregate feature vectors from* their neighbors. In short, both families of algorithms pass messages among neighbors: scalars or vectors, inbound or outbound. The intuition behind these message passing algorithms is that whatever the task at hand, connectivity/locality matters: connected/nearby nodes are more similar (clustering), informative (classification), or relevant (recommendation) to each other than disconnected/distant nodes. Our unified framework targets graph algorithms that use the message passing mechanism.

3.2.2 UNIFIEDGM

We propose a unified framework UNIFIEDGM for graph mining algorithms that employ the message passing scheme. UNIFIEDGM defines the message passing mechanism based on five parameters:

- **Dimension** $d \in \mathbb{Z}_{>0}$ of passed messages. If $d = 1$, messages are scalar scores, otherwise they are d-dimensional embedding vectors.
- **Width** $w \in \mathbb{Z} \cup \{-1\}$ decides the number of neighbors each node communicates with. If $w = -1$, nodes communicate with all their neighbors.

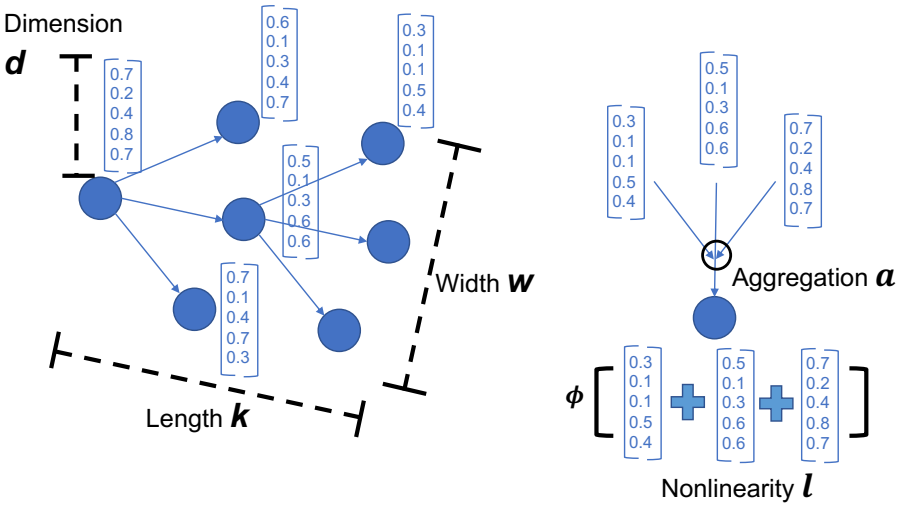


Figure 3.2: **Unified Graph Mining framework.** UNIFIEDGM defines the message passing mechanism based on five parameters: the dimension d, length k, width w, nonlinearity l, and aggregation strategy a.

- **Length** $k \in \mathbb{Z}$ decides the number of message passing steps.
- **Nonlinearity** $l \in \{\text{True}, \text{False}\}$ decides whether to use nonlinearities in the message passing or not.
- **Aggregation strategy** a decides if a node sends a message to itself and how to normalize the sum of incoming messages.

Figure 3.2 shows how each parameter regulates message passing under UNIFIEDGM.

The input of UNIFIEDGM is a graph $G = (V, E)$ and a matrix X_0 of size $(n \times d_0)$ containing d_0-dimensional initial node statistics for all n nodes — either scalar scores or feature vectors. Note that d_0 could be different from d, the dimension of the passed messages. The output of UNIFIEDGM is a set of d-dimensional node embeddings. These embeddings contain information from the node's neighborhood and can be exploited in an output layer which is specialized to a given application (e.g., a logistic regression for node classification.)

Algorithm 1 outlines how UNIFIEDGM passes messages across a graph based on a set of five parameters (d, k, w, l, a). UNIFIEDGM first initializes node statistics (line 1), then iteratively passes messages among neighboring nodes k times. In the i-th message passing step, UNIFIEDGM randomly samples w neighbors to communicate with for each node (line 3) and aggregates messages from sampled neighbors with a strategy decided by the parameter a (lines 4 and 5). Then UNIFIEDGM transforms the aggregated messages linearly with a matrix W_i (line 6) and finally passes them through a function ϕ decided by the parameter l (line 7).

Let us explain in further detail the neighbor sampling and message aggregation steps. Neighbor sampling (line 3) can be expressed as generating a matrix $A_{\text{samp}} = \textit{Sample}(A)$ by randomly zeroing out entries of the binary adjacency matrix A. Message aggregation (lines 4 and 5) is defined by the aggregation strategy $a \in \{\text{SA, SS, SN, NA, NS, NN}\}$. The first letter in {S, N} determines whether a node sends a message to itself or not (Self-loop or No-self-loop). The second

Algorithm 1: UNIFIEDGM Algorithm

Require: initial node statistics X_0, binary adjacency matrix A
Ensure: node embeddings X_k
1: Initialize node statistics X_0
2: **for** message passing step $i = 1$; $i \leq k$; i++ **do**
3: Sample neighbors for each node: $A_{\text{samp}} \leftarrow \textit{Sample}(A)$
4: Generate aggregation matrix: $A_{\text{agg}} \leftarrow \textit{Aggregate}(A_{\text{samp}})$
5: Aggregate messages $X_i \leftarrow A_{\text{agg}} X_{i-1}$
6: Multiply with transformation matrix: $X_i \leftarrow X_i W_i$
7: Pass through nonlinear function: $X_i \leftarrow \phi(X_i) = \begin{cases} ReLU(X_i) \text{ if } l = \text{True} \\ X_i \text{ otherwise} \end{cases}$
8: **end for**
9: **return** X_k

letter in {A, S, N} determines how to normalize the sum of incoming messages (Asymmetric, Symmetric, or No-normalization). Each aggregation strategy a results in an aggregation matrix $A_{\text{agg}} = \textit{Aggregate}(A_{\text{samp}})$, explained in Table 3.2. Multiplying messages X_{i-1} from the previous step by the matrix A_{agg} corresponds to aggregating messages from neighboring nodes.

Letting f_i denote the ith layer of message passing, we can summarize UNIFIEDGM as follows:

$$\begin{aligned} A_{\text{samp}} &= \textit{Sample}(A) \\ A_{\text{agg}} &= \textit{Aggregate}(A_{\text{samp}}) \\ X_k &= f_k(X_{k-1}) = \phi(A_{\text{agg}} X_{k-1} W_k) \\ &= f_k(f_{k-1}(\dots f_1(X_0))) \end{aligned}$$

X_0 is the $(n \times d_0)$ matrix of initial statistic vectors, X_i is the $(n \times d)$ matrix of statistic vectors at step i for $(i = 1 \dots k)$. W_1 and W_i are $(d_0 \times d)$ and $(d \times d)$ transformation matrices respectively $(i = 2 \dots k)$.

3.2.3 Reproduction of Existing Algorithms

In this section, we introduce the most popular graph mining algorithms exploiting the message passing scheme and show how they can be presented under UNIFIEDGM. Table 3.3 shows how to set initial node statistics and parameters (d, k, w, l, a) of UNIFIEDGM to reproduce the original graph algorithms.

PageRank [110] scores nodes in a graph based on their global relevance/importance, and was initially used by Google for webpage recommendation. PageRank initializes all n nodes in the graph with a score of $\frac{1}{n}$. Then, every node iteratively propagates its score across the graph with a decay coefficient $0 < c < 1$ to ensure convergence. Under UNIFIEDGM, PageRank propagates scalar scores $(d = 1)$ to all neighbors $(w = -1)$ with no nonlinear unit $(l = \text{False})$ until scores have converged $(k = \infty)$, and aggregates messages with no self-loop and asymmetric normalization $(a =$

Table 3.2: **Various aggregation strategies in UNIFIEDGM.** The aggregation strategy a decides if a node sends a message to itself or not (Self-loop or No-self-loop) and how to normalize the sum of incoming messages (Asymmetric, Symmetric, or No-normalization). Each combination corresponds to an aggregation matrix $A_{\text{agg}} = \textit{Aggregate}(A)$ in the table. Notation: n is the number of nodes in a graph, A is a $(n \times n)$ binary adjacency matrix, D is a $(n \times n)$ diagonal matrix where $D_{ii} = \sum_j A_{ij}$, and I_n is an identity matrix of size n.

	Self-loop (S)	No-self-loop (N)
Asymmetric (A)	$D^{-1}(A + I_n)$	$D^{-1}A$
Symmetric (S)	$D^{-1/2}(A + I_n)D^{-1/2}$	$D^{-1/2}AD^{-1/2}$
No-normalization (N)	$(A + I_n)$	A

NA). Note that the $(d \times d)$ transformation matrix W in UNIFIEDGM becomes a scalar value and corresponds to the decay coefficient c.

Personalized PageRank (PPR) [6] and **Random Walk with Restart** (RWR) [178, 179] build on PageRank to estimate the relevance of nodes in the perspective of a specific set of seed nodes thus enable personalized recommendation. Under UNIFIEDGM, the only difference of PPR/RWR from PageRank is the initial node scores: RWR/PPR place varying positive scores on the set of seed nodes and zero scores on others. PPR/RWR have the same set of (d, k, w, l, a) as PageRank.

Pixie [35], introduced by Pinterest, complements the ideas of PPR and RWR with neighbor sampling to deal with billions of nodes in real-time. Pixie fixes the number of message passing operations and stays within a computation budget. To reproduce this under UNIFIEDGM, Pixie fixes the product of k and w to a constant number (e.g., $2,000$ from [35]): after k is sampled, w is decided as $\frac{2,000}{k}$. Pixie has the same initial node statistics and parameter $d = 1, l =$ False, and $a =$ NA with PPR/RWR.

Graph Convolutional Networks (GCNs) [75] are a variant of Convolutional Neural Networks that operates directly on graphs. GCNs stack layers of first-order spectral filters followed by a nonlinear activation function to learn node embeddings. Under UNIFIEDGM, given node feature vectors as initial node statistics, GCN passes message vectors ($d = 64$) to all neighbors ($w = -1$) with nonlinear units ($l =$ True) across two-layered networks ($k = 2$) and aggregates messages with a self-loop and symmetric normalization ($a =$ SS).

GraphSAGE [55] extends GCN with neighbor sampling. GraphSage with a mean aggregator averages statistics of a node and its sampled neighbors. Under UNIFIEDGM, GraphSAGE-mean has the same parameters as GCN except w and a. GraphSAGE-mean samples a fixed number of neighbors to communicate with ($w = 25$) and normalizes the aggregated messages asymmetrically ($a =$ SA).

Simplified GCN (SGCN) [162] reduces the excess complexity of GCN by removing the nonlinearities between GCN layers and collapsing the resulting function into a single linear transformation. With fewer parameters to train, SGCN is computationally more efficient than GCN but shows comparable performance on various tasks. Under UNIFIEDGM, SGCN has the same parameters with GCN except l. SGCN does not use any nonlinear unit ($l =$ False).

Table 3.3 presents the original message passing equations of the existing graph algorithms.

Those equations can be fully reproduced from Algorithm 1 with the proper inital node statistics and parameter sets listed in Table 3.3.

Here, we introduce two more graph algorithms that are unified under UNIFIEDGM with slight modifications: K-cores [143] and Belief Propagation [114] — two of the most popular graph mining algorithms. Although these algorithms do not contain trainable parameters and thus do not benefit from AUTOGM, they fit under UNIFIEDGM's message-passing framework. This shows that UNIFIEDGM is general enough to cover various graph mining algorithms.

- **K-core** [143] is the maximal subgraph in which every node is adjacent to at least k nodes. The most straightforward algorithm to compute k-cores is the so-called "shaving" method [131]: repeatedly deleting nodes with a degree less than k until no such node is left. The shaving method is presented in an iterative equation as follows:

$$x_{k+1} = \phi(A_k x_k - k\vec{1})$$

  where x_k is an indicator vector for k-cores where $x_k(i)$ is 1 when i-th node is part of k-cores, otherwise set to 0; A_k is the binary adjacency matrix where only edges among $x_k(i) = 1$ are set to 1, otherwise 0. When we multiply A_k with x_k, the output vector contains the degree of each node in k-cores. $\phi(x)$ is a nonlinear operation where $\phi(x) = 1$ when $x > 0$ else $\phi(x) = 0$. In $A_k x_k - k\vec{1}$, only nodes whose degree is higher than k have positive values. Thus, by passing $A_k x_k - k\vec{1}$ to $\phi(x)$, we output x_{k+1}, the indicator vector for $k+1$-cores. Under UNIFIEDGM, k-cores propagates scalar scores ($d = 1$) to all neighbors ($w = -1$) with a nonlinear unit ($l =$ True) k times, and aggregates messages with no self-loop and no normalization ($a = NN$). The slight modifications to UNIFIEDGM are that the adjacency matrices A_k are iteratively updated, and we perform a subtraction operation ($-k\vec{1}$) instead of the transformation operation (W). Note that the $(d \times d)$ transformation matrix W in UNIFIEDGM becomes the constant value 1.

- **Belief Propagation** (BP) [114] calculates the marginal belief distribution for unobserved nodes, conditional on any observed nodes' belief. FastBP [81] is one of the most widely used approximation algorithms for BP. While BP does not guarantee convergence, FastBP provides convergence in addition to speed and accuracy improvement. FastBP linearizes BP as follows:

$$[I + a'D - b'A]x = \phi_{BP}$$

  where I, D, and A denotes $n \times n$ identity, diagonal, and adjacency matrices, respectively; a' and b' are hyperparameters decided by the BP propagation matrix; x is the final belief vector and ϕ_{BP} is the prior beliefs. The equation is presented in an iteration equation as follows:

$$x = [b'A - a'D]x + \phi_{BP}$$

  This iteration equation has the same form as PageRank. Under UNIFIEDGM, the only difference between FastBP and PageRank is the initial node scores and the aggregation strategy: FastBP sets the initial node scores X_0 with the prior beliefs ϕ_{BP} and does not normalize the aggregation but adds self-loop with coefficients ($a'D - b'A$). FastBP has the same set of parameters (d, k, w, l) as PageRank.

3.2.4 Conventional GM vs. GNNs

As shown, conventional graph algorithms (e.g., PPR, RWR, Pixie) and recent GNNs are unified under UNIFIEDGM. However, before this work, these algorithms were not analyzed in the same framework. What has prevented them from being combined? Two main differences — the use of node feature information and trainability — are the culprits. While GNNs exploit additional node feature information and labels with semi-supervised learning, conventional graph algorithms do not. We analyze this apparent gap and show how UNIFIEDGM reconciles both families of algorithms.

Node feature information: Conventional graph algorithms do not exploit node features, but instead, choose a set of seed nodes to initialize with scores suitable for a given application. Under UNIFIEDGM, these algorithms are also applicable with node features by maintaining the same values for parameters $(d = 1, k, w, l, a)$, but setting initial input dimension d_0 to be the input feature dimension and using a 1st layer tranformation matrix W_1 of size $(d_0 \times 1)$. This would yield a new version of PageRank or PPR that exploits feature information.

Semi-supervised learning: In GNNs, the transformation matrix W is trained with semi-supervised learning using node labels. On the other hand, conventional graph algorithms do not have a training phase in advance of an inference phase. However, conventional algorithms are trainable: the decay coefficient c in PageRank, PPR, and RWR corresponds to an (1×1) transformation matrix W under UNIFIEDGM. Because of its low dimension, the (1×1) transformation matrix could be set heuristically (e.g., $c = 0.85$ in PageRank). But we could use label information to train this (1×1) matrix W with gradient descent as we train it in GNNs.

In our experiments, we show how to train conventional algorithms (PageRank and Pixie) with feature information.

3.2.5 Parameter Selection

We explain the effects of parameters (d, k, w, l, a) on the performance of graph algorithms and how to choose the proper parameters by illustrating the existing algorithm design.

- **Dimension d:** High dimensions of messages enrich the expressiveness of graph algorithms by sacrificing speed. If an application prioritizes fast and simple algorithms, scalar messages (e.g., $d = 1$ in Pixie) are suitable. In contrast, when applications prioritize rich expressiveness of messages and accuracy, high dimensional vectors (e.g., $d = 64$ in GNNs) are more appropriate.
- **Length k:** By deciding the number of message passing steps, k regulates the size of neighborhoods where a graph algorithm assumes locality — where nearby nodes are considered informative. For instance, GCNs assume that a small neighborhood is relevant ($k = 2$). However, when there are label sparsity issues, GNNs propagate toward large scopes ($k = 7$) to transmit label information from distant nodes. Large k results in a long computation time but does not guarantee a high accuracy.
- **Width w:** Large w lets algorithms aggregate information from more neighbors, leading to a possible increase in accuracy. At the same time, large w requires more message passing operations, resulting in longer computation time. In graphs with billions of nodes, like the Pinterest social network, small w is necessary to answer queries in real-time (as done by Pixie).

- **Nonlinearity l:** Nonlinearities enhance the expressiveness of graph algorithms at the cost of speed. They are suitable for anomaly detection systems that require high accuracy (e.g., GNNs for infection detection in medical applications). In contrast, omitting nonlinearity is appropriate for fast recommender systems in social networks (e.g., Pixie in Pinterest).
- **Aggregation strategy a:** The self-loop decides whether a node processes its own embedding during message passing. GNNs include a self-loop to complement a node's features with information from its neighborhood. Conversely, PageRank and RWR do not include a self-loop as they want to spread information from a source node to the rest of the graph to figure out the graph structure. Normalization prevents numerical instabilities and exploding/vanishing gradients in GNNs.

In our experiments, we explore how the five parameters affect the performance of graph algorithms empirically.

3.3 Extended UnifiedGM

We introduce UNIFIEDGM-EXT that extends the message-aggregation step in UNIFIEDGM with two additional building blocks: attention and importance sampling. Recently, graph neural networks have been improved in various ways to improve their performance and scalability. These improvements focus on the aggregation step in the message passing mechanism. In the original form of the message passing mechanism, nodes pass/receive messages uniformly from their directly connected neighbors, assuming that these neighbors are informative. Questions have been raised about this assumption: are all neighbors informative enough to communicate with? In real-world graphs, few connections are made by mistake, and some are valid only for a specific application. For instance, in member-to-member networks in LinkedIn, connections could be made not only among colleagues but also among personal friends and families. When we apply the message passing mechanism on the LinkedIn network to make job recommendations, we aggregate information not only from the colleagues who are crucial information for the job recommendation task, but also from personal friends and families who are from different areas and often irrelevant. The motivations are summarized as follows: which neighbors are informative to pass/receive messages? and how trustworthy are they?

Attention and importance sampling methodologies are proposed on graph neural networks to handle these problems. The attention-based GNNs compute importance/relevance scores of each neighbor with regard to a source node, then use those scores as weights when they aggregate messages from the neighbors and compute a weighted sum of the aggregated messages. Importance sampling goes one step further from attention and samples only neighbors with high relevance scores. Importance sampling does not only handle different importance scores among neighbors, but also solves scalability issues by reducing the size of graphs through sampling.

As we described in Section 3.2.4, UNIFIEDGM unifies the conventional graph mining algorithms and graph neural networks. By extending UNIFIEDGM to embrace attention and importance sampling concepts, UNIFIEDGM-EXT allows applying techniques used by graph neural networks to the conventional graph mining field. To adopt attention into UNIFIEDGM-EXT, we add additional options to the aggregation parameter a. To apply importance sampling on UNIFIEDGM-EXT,

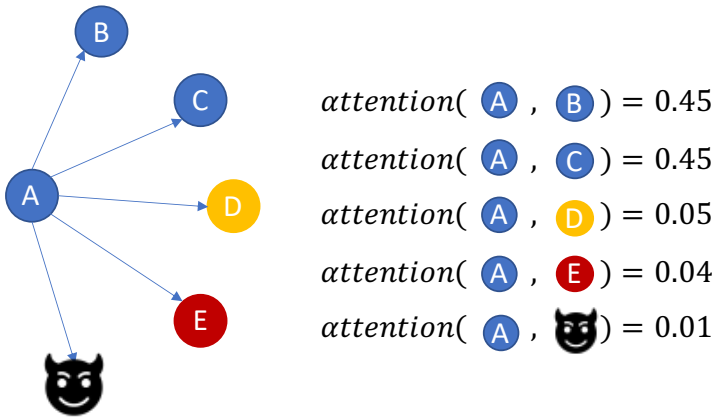


Figure 3.3: **Attention module in UNIFIEDGM.** UNIFIEDGM computes attention between a source node A and its neighbors based on their relevance to node A. Unrelated or adversarial neighbors have small attentions.

we add a new parameter s that decides the sampling strategy. The following section shows how UNIFIEDGM-EXT embraces the concepts of attention and importance sampling concretely.

3.3.1 Attention

Graph attention networks (GAT) [155] is the first attention-based graph neural network model. It estimates the relevance between a source node and its neighbors using their hidden embeddings. Then the computed relevance scores (attentions) are used as weights in a weighted sum in the aggregation step. How to estimate relevance between two nodes or how to design the attention model varies across different methods. UNIFIEDGM-EXT extends the aggregation parameter a with new options: concatenation-based attention, dot-product-based attention, and low-pass attention. We describe how each attention model works under UNIFIEDGM-EXT.

- **Concatenation-based attention** proposed in [155] computes a relevance score $\alpha_l(i,j)$ between node i and j at the l-th layer as follows:

$$\alpha_l(i,j) = \frac{\sigma(a_{att} \cdot [h_l(i)||h_l(j)])}{\sum_{k \in N(i)} \sigma(a_{att} \cdot [h_l(i)||h_l(k)])}$$

  where a_{att} denotes a $(1 \times 2d)$ learnable parameter, $\sigma(x) = exp(LeakyReLU(x))$ is a nonlinear operation for attention computation, $h_l(i) = x_l(k)W_l$ denotes hidden embedding of node i at the l-th layer after multiplying with the transformation matrix W_l, and $N(i)$ denotes the neighbors of node i. Since we concatenate the hidden embeddings ($[h_l(i)||h_l(j)]$), we name it as a concatenation-based attention model. Then we aggregate messages ($h_l(j)$) from neighbors using the computed attention $\alpha_l(i,j)$ as follows:

$$x_{l+1}(i) = \phi(\sum_{j \in N(i)} \alpha_l(i,j) h_l(j))$$

where $\phi(x)$ is the operation decided by the nonlinearity parameter l in UNIFIEDGM (refer to Table 3.1). Then x_{l+1} is used as $(l+1)$-th layer hidden embeddings.

- **Dotproduct-based attention** calculates a relevance score $\alpha_l(i,j)$ between node i and j as follows:

$$\alpha_l(i,j) = \frac{\sigma(W_{att}h_l(i) \cdot W_{att}h_l(j))}{\sum_{k \in N(i)} \sigma(W_{att}h_l(i) \cdot W_{att}h_l(k))}$$

  where W_{att} denotes a $(d_{att} \times d)$ learnable parameter which maps hidden embedding $h_l(i)$ from d-dimensional space to d_{att}-dimensional space. On the d_{att}-dimensional space, we compute the relevance score between node i and j by dot-producing their hidden embeddings.

- **Low-pass attention** reduces the impact of adversarial edge additions/deletions on graphs. To filter out adversarial nodes, [159] introduces a low-pass filter to GCNs that decrease weights of neighbors who are excessively different from a source node in the hidden embedding space. They define a low-pass attention as follows:

$$\beta_l(i,j) = \frac{R}{max(R, ||h_l(i) - h_l(j)||)}$$

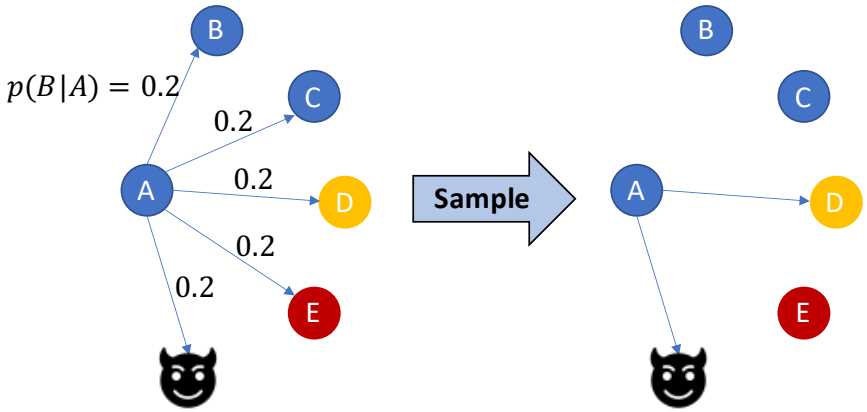
$$\alpha_l(i,j) = \begin{cases} \beta_l(i,j)/d_i & \text{if } j \in N(i) \setminus \{i\} \\ 1 - \sum_{j \in N(i) \setminus \{i\}} \beta_l(i,j)/d_i & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

  where $R > 0$ is a threshold for controlling the low-pass message passing and $d_i = |N(i)|$ denotes the number of neighbors of node i. $\beta_l(i,j)$ assigns a weight of 1 if $h_l(i)$ and $h_l(i)$ are less than R apart, while gradually reducing the weight as the distance between them exceeds R. More distant from the source node i in the embedding space, the neighbor node j has a smaller weight $\beta_l(i,j)$. Then, the final attention $\alpha_l(i,j)$ acts as a low-pass filter to prevent the source node from being excessively affected by suspicious neighbors by giving small attentions.

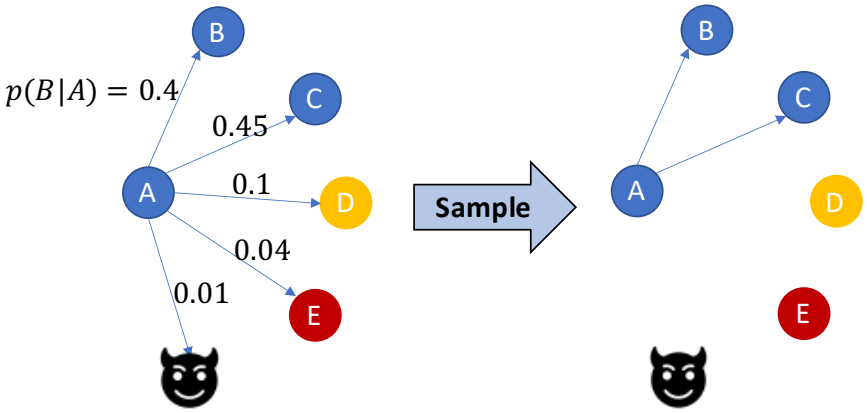
In Section 3.2, the aggregation parameter a has six options (NN, NS, NA, SN, SS, SA from Table 3.2). By merely adding three more options (concatenation, dot product, low-filter attentions) to the parameter a, UNIFIEDGM-EXT successfully embraces attention-based models.

3.3.2 Importance sampling

In Section 3.2, we introduce uniform sampling in UNIFIEDGM-EXT where the sampling number is decided by the sampling parameter w. While uniform sampling resolves high computation and memory footprints problems by reducing the graph's size, it leads to a possible loss of crucial information. Uniform sampling does not discriminate informative neighbors from uninformative ones in the sampling process. Thus it could sample only uninformative or irrelevant neighbors for aggregation, resulting in low-quality embeddings. To deal with this limitation of uniform sampling, importance sampling has been adopted in GCNs. Importance sampling samples each neighbor



(a) Uniform sampling



(b) Importance sampling

Figure 3.4: **Various sampling strategies in UNIFIEDGM.** Uniform sampling samples informative and uninformative neighbors with an equal probability. It could sample only uninformative or irrelevant neighbors. On the other hand, importance sampling samples each neighbor following their distinct sampling probabilities, and the sampling probabilities are computed based on their relevance with regard to a source node.

following their distinct sampling probabilities, and the sampling probabilities are computed based on their relevance with regard to a source node.

We expand UNIFIEDGM-EXT with new parameter s that regulates the message passing mechanism's sampling strategy. The sampling strategy decides 1) how to define sampling probabilities, 2) whether the sampling probabilities are learnable or not, and 3) when to sample neighbors. We describe each component of the sampling strategy and its effect on the sampling process in UNIFIEDGM-EXT.

- **How to define sampling probabilities:** Sampling probability $p(j|i)$ of a neighbor node j given a source node i could be computed individually or by a shared function with other edges. *Individual* sampling probability $p(j|i)$ is given as a scalar value that is decided independently from other edges' sampling probabilities. The only requirement is the sum should be 1 ($\sum_{j \in N(i)} p(j|i)$). The *individual* sampling probabilities could be all same (uniform sampling, $p(j|i) = 1/|N(i)|$) or

proportional to their degree as follows:

$$p(j|i) = \frac{|N(j)|}{\sum_{k \in N(i)} |N(k)|}$$

where $N(i)$ denotes the degree of node i. The *shared function* $p(j|i)$ could have various forms including the three attention models we described above. While *individual* sampling probabilities are intuitive and easy to interpret, *shared* sampling probability functions share a small number of parameters, thus less likely to be overfitted to the training set.

- **Learnability of sampling probabilities:** The sampling probabilities defined as either individual scalar values or a shared function could be set heuristically and fixed to the initial values. They could also be trained by gradient back-propagation. While *heuristic* sampling probabilities are cost-efficient without additional sampling probability training, *learnable* sampling probabilities are customized to a given application, leading to a performance improvement.
- **When to sample neighbors:** *Static sampling* samples neighbors of each node before the message passing mechanism. In *static sampling*, a set of sampled neighbors is fixed during the message passing mechanism. Thus nodes keep interacting with the same set of their sampled neighbors in every layer. On the other hand, *dynamic sampling* samples a new set of neighbors every time a source node is engaged in the message passing mechanism as described in Algorithm 1. In *dynamic sampling*, nodes receive/pass messages with a new set of the sampled neighbors at each layer. While *static sampling* reduces the computation time by running the sampling process only once, *dynamic sampling* increases accuracy. Randomness in the *dynamic sampling* brings a regularization effect, which helps with generalization.

In Table 3.4, the sampling parameter s has 5 (two heuristics and three shared models) $\times 2$ (heuristic/learnable) $\times 2$ (static/dynamic) $= 20$ options. With the addition of the parameter s, UNIFIEDGM-EXT now has six parameters (d, w, k, l, a, s) to define a graph mining algorithm based on a message-passing mechanism. Users decide whether to append new attention options to the parameter a and add a sampling parameter s to UNIFIEDGM. This section showed UNIFIEDGM-EXT is general enough to embrace new approaches with very few modifications. The following section about an automated system for graph algorithm development is based on the original UNIFIEDGM from Section 3.2. However, the number of parameters or the number of options for each parameter does not affect the algorithm we describe in the next section.

3.4 Automation of Graph Mining Algorithm Development

With the proper parameter selection, UNIFIEDGM could output a graph algorithm tailored for a specific application. However, the parameter selection process still relies on the intuition and domain knowledge of practitioners, which would prevent non-experts in graph mining from fully exploiting UNIFIEDGM. How can we empower practitioners without much expertise to deploy customized algorithms? We introduce AUTOGM, which generates an optimal graph algorithm autonomously given a user's scenario.

When designing an algorithm for an application, we need to consider two primary metrics: computation time and accuracy, which usually trade off each other. Take, for example, a developer who aims to develop an online recommender system that makes personalized recommendations to a large number of users at the same time. At first, she employs a state-of-the-art GNN model (in terms of accuracy) but finds that the computation time is too long for her application. Then the developer seeks an alternative simple graph algorithm that runs faster than a time budget by sacrificing accuracy. AUTOGM incorporates this practical issue of finding the best speed-accuracy trade-off into the graph algorithm generation problem. AUTOGM answers two questions: 1) given the maximum acceptable computation time, which graph algorithm maximizes accuracy? 2) given minimum accuracy requirements, which graph algorithm minimizes computation time?

We first formalize our budget-aware graph algorithm generation problem as a constrained optimization problem. Then we replace the constrained problem with an unconstrained optimization problem using barrier methods (Section 3.4.1). We explain why Bayesian optimization is well-suited for this unconstrained problem (Section 3.4.2). Then we describe how AUTOGM solves the optimization problem using Bayesian optimization (Section 3.4.3). Finally, we analyze the time complexities of AUTOGM (Section 3.4.4).

3.4.1 Budget-aware objective function

Letting x denote a graph algorithm, $g(x)$ and $h(x)$ indicate the computation time and accuracy of x, respectively. Then an optimal graph algorithm generation problem with an accuracy lower bound $h_{\min}$ is presented as a constrained optimization as follows:

$$x_{opt} = \text{argmin}_x \; g(x) \text{ subject to } h(x) - h_{\min} \geq 0 \tag{3.1}$$

One of the common ways to solve a constrained optimization problem is using a barrier method [151], replacing inequality constraints by a penalizing term in the objective function. We re-formulate the original constrained problem in Equation 3.1 as an equivalent unconstrained problem as follows:

$$x_{opt} = \text{argmin}_x \; g(x) + I_{h(x)-h_{\min}\geq 0}(x) \tag{3.2}$$

where the indicator function $I_{h(x)-h_{\min}\geq 0}(x) = 0$ if $h(x) - h_{\min} \geq 0$ and ∞ if the constraint is violated. Equation 3.2 eliminates the inequality constraints, but introduces a discontinuous objective function, which is challenging to optimize. Thus we approximate the discontinuous indicator function with an optimization-friendly log barrier function. The log barrier function, defined as $-\log(h(x) - h_{\min})$ is a continuous function whose value on a point increases to infinity ($-\log 0$) as the point approaches the boundary $h(x) - h_{\min} = 0$ of the feasible region. Replacing the indicator function with the log barrier function yields the following optimization problem:

$$f_{GM}(x) = g(x) - \lambda \log(h(x) - h_{\min}) \tag{3.3}$$

$$x_{opt} = \text{argmin}_x \; f_{GM}(x) \tag{3.4}$$

f_{GM} is our novel budget-aware objective function and $\lambda > 0$ is a penalty coefficient. Equation 3.4 is not equivalent to our original optimization problem, Equation 3.1. However, as λ approaches zero, it

Algorithm 2: AUTOGM Algorithm

Require: minimum accuracy (or maximum inference time) constraint, target dataset, BO search budget
Ensure: a graph algorithm (i.e. five parameters of UNIFIEDGM)
1: **for** iteration $i = 1$; $i <$ BO search budget; i++ **do**
2: Choose a point (d, k, w, l, a) to evaluate
3: Generate a graph mining algorithm A from (d, k, w, l, a)
4: Train A on the training set
5: Evaluate A and measure acc, $time$ on the validation set
6: Evaluate $f_{GM}(acc, time)$ and update posterior of f_{GM}
7: **end for**
8: **return** a parameter set with the minimum f_{GM}

becomes an ever-better approximation (i.e., $-\lambda \log(h(x) - h_{\min})$ approaches $I_{h(x)-h_{\min}\geq 0}(x)$) [151]. The solution of Equation 3.4 ideally converges to the solution of the original constrained problem. Now, our budget-aware graph algorithm generation problem is formulated as a minimization problem of f_{GM}.

Given a minimum accuracy constraint $acc_{\min}$, we set $g(x) = time$ to minimize and $h(x) - h_{\min} = acc - acc_{\min} \geq 0$ as a constraint. On the other hand, given a maximum inference time constraint $time_{\max}$, we want to maximize accuracy while observing the time constraint. Then we set $g(x) = -acc$ to minimize and $h(x) - h_{\min} = time_{\max} - time \geq 0$ as a constraint.

3.4.2 Bayesian optimization

Under UNIFIEDGM, a graph algorithm x is defined by a set of parameters (d, k, w, l, a). Then search space $\mathcal{X}$ for the optimization problem becomes a five-dimensional space of parameters (d, k, w, l, a). Suppose we set cardinalities for each parameter as $300, 30, 50, 2$, and 6, respectively (i.e., $0 < d \in \mathbb{Z} \leq 300, 0 < k \in \mathbb{Z} \leq 30, 0 < w \in \mathbb{Z} \leq 50, l \in \{True, False\}, a \in \{NA, NS, NN, SA, SS, SN\}$). Then the number of unique architectures within our search space is $300 \times 30 \times 50 \times 2 \times 6 = 5.4 \times 10^6$, which is quite overwhelming. Moreover, training and validating a graph algorithm, especially on large datasets, takes significant time. Thus it is impractical to search the space $\mathcal{X}$ exhaustively. Most importantly, even if we could measure the computation time and accuracy ($g(x)$ and $h(x)$) of a graph algorithm and calculate the objective function $f_{GM}(x) = g(x) - \lambda \log(h(x) - h_{\min})$, we do not know the exact closed-form of $f_{GM}(x) = f_{GM}(d, k, w, l, a)$ in terms of the parameters (d, k, w, l, a) nor its derivatives. Thus, we cannot exploit classical optimization techniques that use derivative information. To cope with these problems — expensive evaluation and no closed-form expression nor derivatives — which optimization technique is appropriate?

Bayesian optimization (BO) [15] is the most widely-used approach to find the global optimum of a black-box cost function — a function that we can evaluate but for which we do not have a closed-form expression or derivatives. Also, BO is cost-efficient with as few expensive evaluations as possible (more details in Section 3.6.2). Therefore BO is well-suited to our problem to find the best parameter set (d, k, w, l, a) given the expensive black-box objective function $f_{GM}(x)$.

3.4.3 AUTOGM

Users supply three inputs to AUTOGM: 1) a budget constraint (the minimum accuracy or maximum computation time), 2) a target dataset on which they want an optimized algorithm — containing a graph, initial node scores, and labels for supervised learning — and 3) a search budget for Bayesian Optimization. The search budget is given as the total number of evaluations in BO. Then AUTOGM outputs the optimal graph mining algorithm (i.e., parameter set of UNIFIEDGM).

Algorithm 2 outlines how AUTOGM works. Until it has exhausted its search budget, AUTOGM repeats the process: 1) Pick a point $x = (d, k, w, l, a) \in \mathcal{X}$ to evaluate using an acquisition function of BO (line 2) then generate a graph algorithm A from parameters (d, k, w, l, a) (line 3). 2) Train A on the training set (line 4) and measure accuracy and inference time of A on the validation set (line 5). 3) Evaluate the objective function f_{GM} given the accuracy and inference time of A, then update a posterior model for f_{GM} in BO (line 6). After all iterations, AUTOGM returns the parameter set $x = (d, k, w, l, a)$ with the minimum f_{GM} among the evaluated points.

The search space of AUTOGM is not affected by the input but fixed to a five-dimensional space of parameters (d, k, w, l, a). The search time of AUTOGM is determined by the BO search budget (total number of evaluations) and evaluation time. Since the evaluation time of a graph algorithm is often proportional to the input dataset's size, the total search time of AUTOGM is decided by the dataset. BO's minimization of the number of evaluations is especially efficient for large datasets which result in the long evaluation time. Our main contribution is defining the graph algorithm generation problem as an optimization problem on a novel search space.

3.4.4 Time Complexity Analysis

We analyze the time complexities of UNIFIEDGM and AUTOGM.

Theorem 1 (Time Complexity of UNIFIEDGM)**.** *A graph mining algorithm A generated from* UNIFIEDGM *with a parameter set (d, k, w, l, a) takes $O(kdwn)$ time where n is the number of nodes in a given graph.*

Proof. Under UNIFIEDGM, matrix-vector multiplication operations represent the bulk of the computation time. In the matrix-vector multiplication operations, the matrix corresponds to the adjacency matrix whose number of nonzeros is $O(wn)$. Under UNIFIEDGM, every node samples w neighbors, summing up to $O(wn)$ edges in the sampled graph. In the matrix-vector multiplication operations, the vector corresponds to node embeddings $X \in \mathbb{R}^{n \times d}$. Then one matrix-vector multiplication operation takes $O(dwn)$. Under UNIFIEDGM, the matrix-vector multiplication operation is executed k times across k layers. The nonlinear operation decided by l and the normalization operation decided by a take $O(n)$ time each. Thus the algorithm A takes $O(kdwn)$ time in total. □

In Theorem 1, we show the time complexity of the graph algorithm generated from UNIFIEDGM in terms of the parameters d, k, w, l, a. However, in Theorem 2 below, we express the time complexity of AUTOGM in different terms. Intuitively, the computation time of AUTOGM is proportional to the number of times we train a graph algorithm — i.e., evaluate a configuration (d, k, w, l, a) — times the time it takes to train the algorithm. To represent the time it takes to train an algorithm,

we cannot rely on the parameters (d, k, w, l, a) as they keep changing while AUTOGM searches the space. Instead, we note that the computation time of one epoch of training is mainly decided by the size of the graph, which we represent by the number of edges m. The training time is then proportional to Em, where E is the number of training epochs. The total time to BEm where B is the number of times we train the algorithm (the number of evaluations allowed by the BO search budget).

Theorem 2 (Time Complexity of AUTOGM)**.** AUTOGM *takes $O(BEm)$ for searching the optimal graph mining algorithm where m is the number of edges in a given graph, E is the number of epochs for the training, and B is the BO search budget.*

Proof. The computation time of AUTOGM is proportional to the number of times we train a graph algorithm (evaluate a hyper-parameter configuration) times the time it takes to train the algorithm. The graph algorithm executes several adjacency matrix-embedding vector multiplication operations in the forward and backward pass, which take $O(m)$. This is repeated for every batch (E times), for a total training time Em. Finally, AUTOGM trains the algorithm B times as described in Algorithm 2. Thus the overall computation time of AUTOGM is $O(BEm)$. □

3.5 Experiments

In this section, we evaluate the performance of AUTOGM compared to existing models with heuristic parameters. We aim to answer the following questions:

- **Q1. Effectiveness of AUTOGM:** Do algorithms found by AUTOGM outperform their state-of-the-art competitors? Given an upper bound on inference time/a lower bound on accuracy, does AUTOGM find the algorithm with the best accuracy/the fastest inference time? (Section 3.5.2)
- **Q2. Search efficiency of AUTOGM:** How long does AUTOGM take to find the optimal graph algorithm? How efficient it is compared to random search? (Section 3.5.3)
- **Q3. Effect of UNIFIEDGM parameters:** How do parameters (d, k, w, l, a) affect the accuracy and inference time of a graph mining algorithm? (Section 3.5.4)

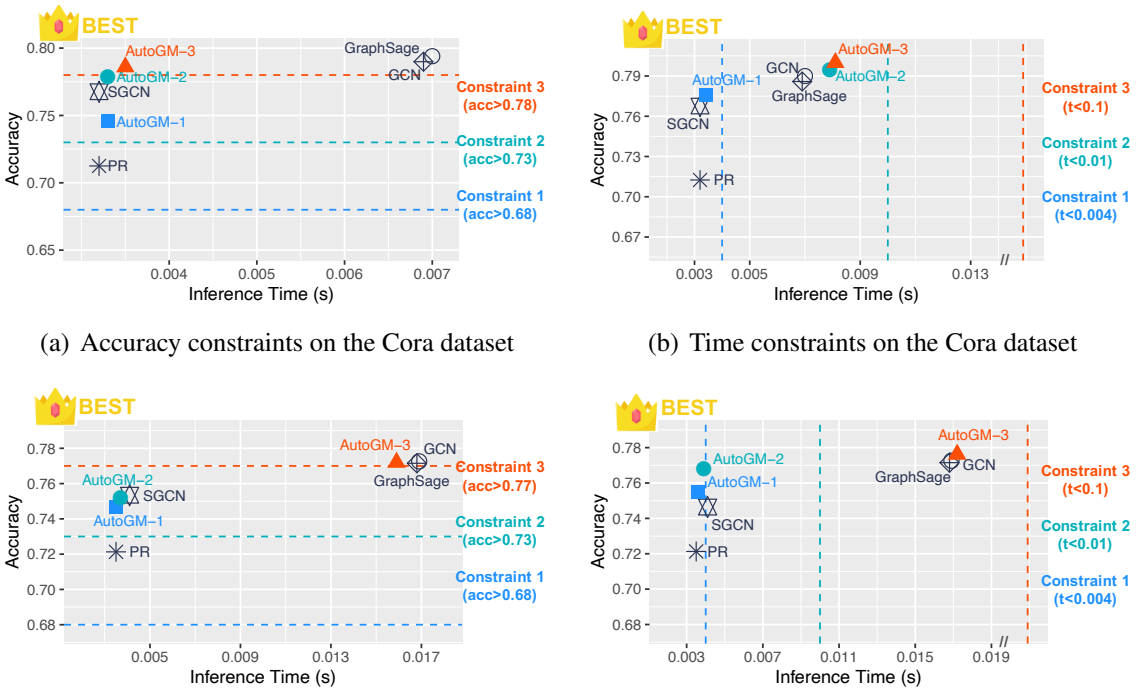
3.5.1 Experimental Setting

We evaluate the performance of graph mining algorithms on two semi-supervised tasks, node classification and link prediction. All experiments were conducted on identical machines using the Amazon EC2 service (p2.xlarge with 4 vCPUs, 1 GPU and 61 GB RAM).

Dataset: We use the three citation networks (Cora, Citeseer, and Pubmed) [126], two Amazon co-purchase graphs (Amazon Computers and Amazon Photo) [127], and two co-authorship graphs (MS CoauthorCS and MS CoauthorPhysics) [127]. We report their statistics in Table 3.5.

Baseline: Our baselines are PageRank [110], GCN [75], GraphSage [55], and SGCN [162]. We generate each algorithm under UNIFIEDGM by setting the five parameters as follows:

- PageRank: $d = 1, k = 30, w = -1, l =$ False, $a =$ NA
- GCN: $d = 64, k = 2, w = -1, l =$ True, $a =$ SS



(a) Accuracy constraints on the Cora dataset

(b) Time constraints on the Cora dataset

(c) Accuracy constraints on the Pubmed dataset

(d) Time constraints on the Pubmed dataset

Figure 3.5: **AUTOGM finds the algorithms with the best accuracy/inference time trade-off on the node classification task.** Given three different accuracy/inference time constraints $1, 2, 3$, AUTOGM generates three novel graph algorithms, AUTOGM-$1, 2, 3$, respectively.

- GraphSAGE: $d = 64, k = 2, w = 25, l = \text{True}, a = \text{SA}$
- SGCN: $d = 64, k = 2, w = -1, l = \text{False}, a = \text{SS}$

When w is larger than the number of neighbors, we sample neighbors with replacement. For PageRank, the original algorithm outputs the sum of intermediate scores that each node receives ($\sum X_i$), but we use only the final scores X_k in our experiments. The goal of our experiments is to compare PageRank with other algorithms in terms of its main feature in UNIFIEDGM, low dimension ($d = 1$).

Bayesian optimization: We use an open-sourced Bayesian optimization package[2]. For the parameters d, k, and w which take integer values, we round the real-valued parameters chosen by BO to integer values. For the parameter l and a, which take boolean and categorical values, we bound the search space ($0 < l < 1$and $0 < a < 6$), round the real-valued parameters chosen by BO to the closest integer values, and map (0: False, 1: True, 0: NN, 1: NS, 2: NA, 3: SN, 4: SS, 5: SA). We set the BO search budget (total number of evaluations) as 20 for all datasets. The resulting search time of each dataset is reported in Table 3.7. For the penalty coefficient λ, the smaller λ brings the tighter budget constraints. To make our budget constraints strict, we set λ as 10^{-19}.

We use the Adam optimizer [73] and tune each baseline with a grid search on each dataset. Most

[2]https://github.com/fmfn/BayesianOptimization

baselines perform best on most datasets with a learning rate of 0.01, weight decay of 5×10^{-4}, and dropout probability of 0.5. We fix these parameters in our autonomous graph mining algorithm search through Bayesian Optimization. We report the average performance across 10 runs for each experiment.

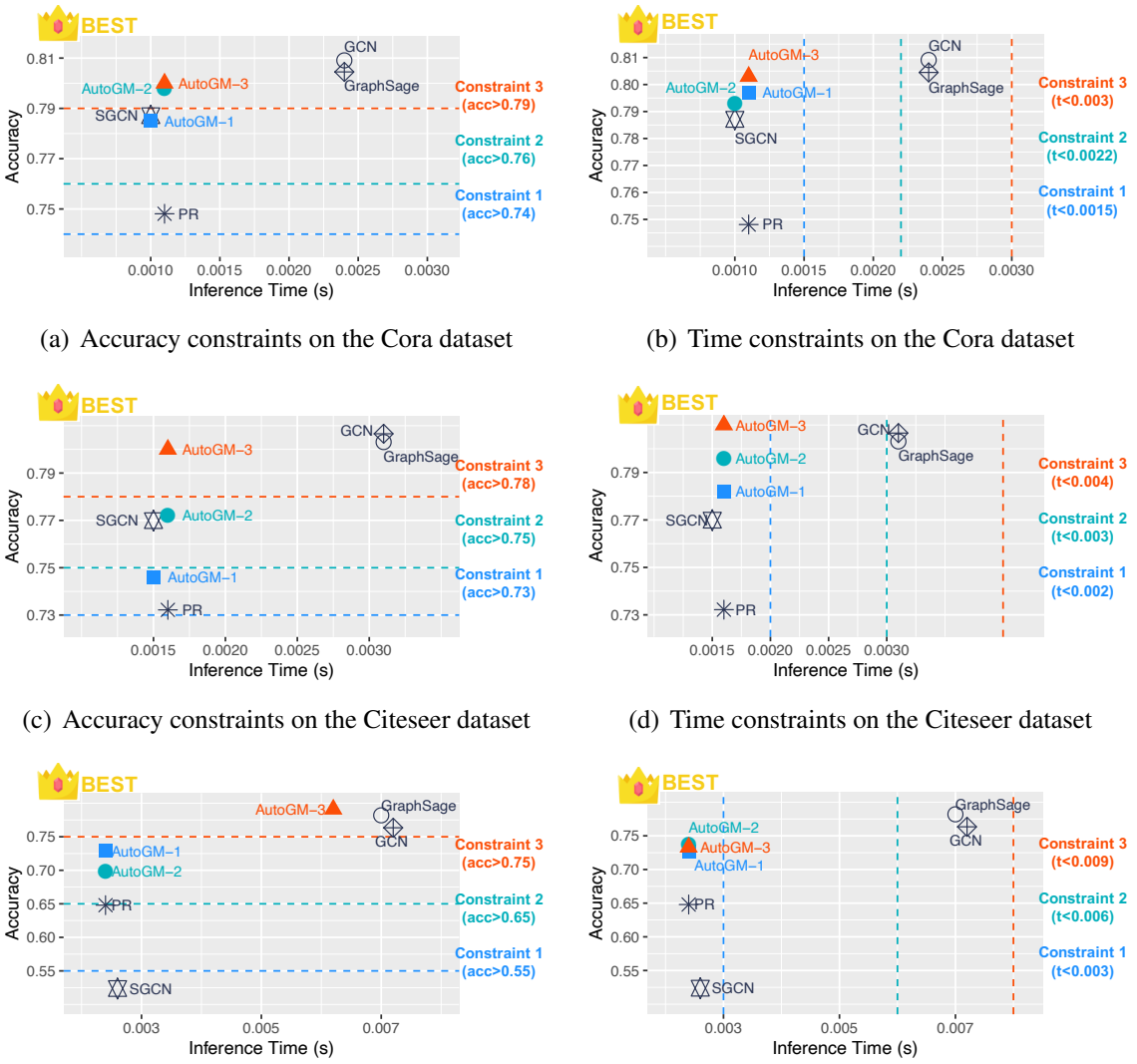
3.5.2 Effectiveness of AUTOGM

In this section, we demonstrate how AUTOGM trades off accuracy and inference time on real-world graphs with two different tasks, node classification and link prediction. We compare the best algorithms found by AUTOGM with baselines in terms of accuracy and inference time. For each dataset, we run AUTOGM with three different accuracy lower bounds and three inference time upper bounds, as illustrated in Figures 3.1 and 3.5. For each constraint, AUTOGM generates a novel graph algorithm corresponding to a set of five parameters of UNIFIEDGM. For space efficiency, we show the result on the Cora, Citeseer, and Pubmed datasets.

Node Classification In the node classification task, each graph mining algorithm predicts the label of a given node. Among algorithms satisfying an accuracy lower bound, the algorithms generated by AUTOGM show the best trade-off between accuracy and inference time. For instance, in the Citeseer dataset in Figure 3.1(a), AUTOGM-2 has the fastest inference time above accuracy constraint 2 among PageRank (PR), GCN, SGCN, and GraphSage. Given the highest or tightest accuracy constraint 3, only AUTOGM-3 satisfies it. Conversely, among algorithms satisfying inference time upper bounds, the algorithms generated by AUTOGM have the highest accuracy. For instance, in the Pubmed dataset in Figure 3.5(d), AUTOGM-1 has the highest accuracy below time constraint 1 among PR and SGCN. Given the most generous time constraint 3, AUTOGM-3 achieves the highest accuracy among all algorithms.

The empirical performance of our baselines is consistent with our guidelines for how to choose the parameters (d, k, w, l, a) in Section 3.2.5. PageRank achieves fast inference time with a low dimension of messages ($d = 1$) and no nonlinearities ($l =$ False), but sacrifice accuracy. GCN and GraphSage achieve high accuracy with a high dimension of messages ($d = 64$) and nonlinearities ($l =$ True) at the cost of a high inference time. SGCN removes nonlinearities ($l =$ False) to decrease the inference time while maintaining high accuracy.

Table 3.6 shows the parameter set of UNIFIEDGM that corresponds to the algorithms found by AUTOGM on the Citeseer dataset. When encouraged to find higher accuracy algorithms (through a larger time upper bound or higher accuracy lower bound), AUTOGM is likely to use high values of d and w and nonlinearities ($l =$ True). For instance, AUTOGM chooses higher values $d = 255, w = 45$ for the larger time upper bound $time < 0.01$ than the values $d = 70, w = 25$ for the bound $time < 0.004$. With the largest upper bound $time < 0.1$, AUTOGM chooses $l = True$ to use nonlinearities. This result is consistent with our intuition over the parameter selection in Section 3.2.5. Vastly different parameter sets for each algorithm in Table 3.6 show that AUTOGM searches the parameter space beyond human intuition, which underlines the value of autonomous graph mining algorithm development.



(a) Accuracy constraints on the Cora dataset

(b) Time constraints on the Cora dataset

(c) Accuracy constraints on the Citeseer dataset

(d) Time constraints on the Citeseer dataset

(e) Accuracy constraints on the Pubmed dataset

(f) Time constraints on the Pubmed dataset

Figure 3.6: **AUTOGM finds the algorithms with the best accuracy/inference time trade-off on the link prediction task.** Given three different accuracy/inference time constraints $1, 2, 3$, AUTOGM generates three novel graph algorithms, AUTOGM-$1, 2, 3$, respectively.

Link Prediction In the link prediction task, the algorithm predicts whether there exists an edge between two given nodes. To build our training set, we randomly hide 30% of edges in the original graph, use the remaining edges as positive ground-truth labels, and sample an equal number of disconnected node pairs as negative ground-truth labels. Our test set consists of the hidden 30% edges as positive ground-truth labels and an equal number of random disconnected node pairs as negative ground-truth labels. After we get the node embeddings for a graph algorithm, we dot-product each pair of node embeddings to predict the probability of edge existence for the given node pair.

Among algorithms satisfying accuracy lower bounds, the algorithms generated by AUTOGM have the fastest inference time. For instance, in the Pubmed dataset in Figure 3.6(e), AUTOGM-3 has the fastest inference time above accuracy constraint 3 among GCN and GraphSage. Conversely, among algorithms satisfying inference time upper bounds, the algorithms generated by AUTOGM show the best trade-off between accuracy and inference time. For instance, in the Citeseer dataset in Figure 3.6(d), AUTOGM-1 has the highest accuracy below time constraint 1 among PR and SGCN. A noteworthy phenomenon in the link prediction task is that algorithms generated from AUTOGM have similar inference times but diverse accuracies. This shows that it's easier to manipulate accuracy by designing graph algorithms, while the dataset largely determines the inference time.

3.5.3 Search efficiency of AUTOGM

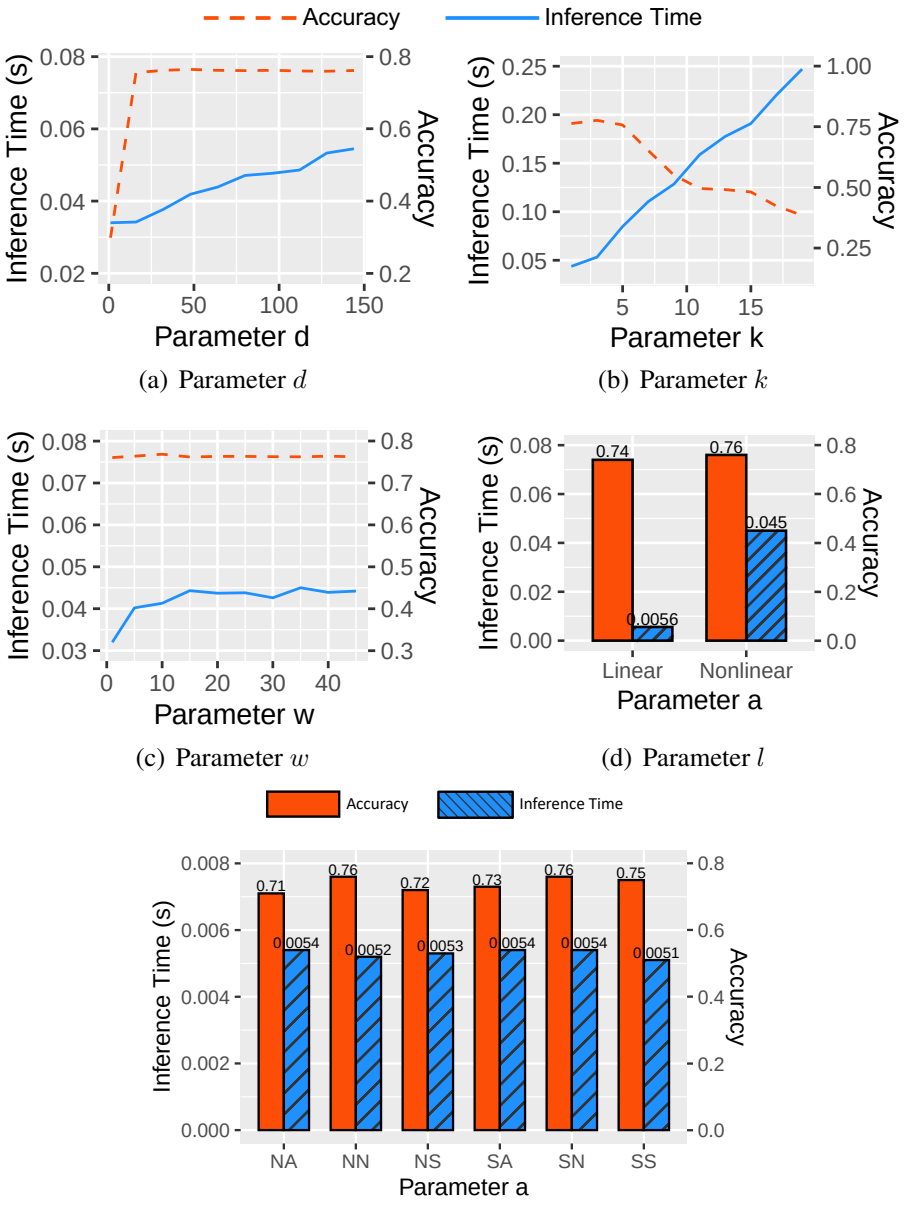
AUTOGM searches for the optimal graph algorithm in a five-dimensional space (d, k, w, l, a) defined by UNIFIEDGM. To show the search efficiency of AUTOGM, we give the same maximum search time and budget constraints to AUTOGM and RandomSearch, then compare the performance of the best graph algorithms each method finds. RandomSearch samples each parameter (d, k, w, l, a) randomly and defines a graph algorithm based on the sampled parameters. We set the maximum search time proportional to the size of the dataset. The budget constraints are chosen based on the best performance among the baseline methods (PageRank, GCN, GraphSage, SGCN). We select the tightest constraints (i.e., fastest inference time and highest accuracy among the baselines) to examine the search efficiency.

Table 3.7 shows the inference time and accuracy of the optimal graph algorithms AUTOGM and RandomSearch find. RandomSearch fails to find any algorithm satisfying the given accuracy constraints on the Cora, Pubmed, and CoauthorP datasets. It also fails to find any algorithm satisfying the inference time constraints on the Citeseer, AmazonC, and AmazonP datasets. When RandomSearch finds graph algorithms satisfying the given constraints, their performance is still lower than the algorithms found by AUTOGM. For instance, given the inference time upper bound ($t < 0.02$) on the CoauthorC dataset, AUTOGM finds an algorithm with accuracy 0.83 while RandomSearch finds an algorithm with accuracy 0.75.

Table 3.7 presents how much accuracy/inference time is used under the given budgets to find the optimal graph algorithms (column $6, 7$ and $11, 12$). AUTOGM generates algorithms whose accuracy (time) is as close as possible to the given accuracy (time) budgets. For instance, AUTOGM finds the fastest graph algorithm with an accuracy of 0.8 when the accuracy lower bound is given as 0.8 on the CoauthorC dataset. By exhausting the budget, AUTOGM improves the target metric time (accuracy) and brings the best trade-off between computation time and accuracy.

3.5.4 Effect of UNIFIEDGM parameters

In this section, we investigate the effects of parameters of UNIFIEDGM on the performance of a graph mining algorithm. Given a set of parameters ($d = 64, k = 2, w = -1, l =$ True, $a =$ SS), we vary one parameter while fixing the others and measure the performance of the generated algorithm. For the experiment where we vary the aggregation parameter a, we use a different set of parameters



(a) Parameter d (b) Parameter k

(c) Parameter w (d) Parameter l

(e) Parameter a

Figure 3.7: **Effects of the five parameters (d, k, w, l, a) of UNIFIEDGM on the performance of graph algorithms.**

($d = 16, k = 2, w = 10, l = False$) to better illustrate changes in accuracy and inference time. For brevity, we show the result on the Pubmed dataset.

- **Dimension d:** Figure 3.7(a) shows that inference time increases linearly with d, while accuracy increases only until $d > 20$. For the Pubmed dataset, 20-dimensional messages are expressive enough that the accuracy stops increasing. Larger datasets would likely benefit from higher dimensional messages.
- **Length k:** In Figure 3.7(b), when k increases, inference time increases linearly, but accuracy decreases for $k > 3$. The decrease in accuracy is due to oversmoothing: repeated graph aggregations eventually make node embeddings indistinguishable.
- **Width w:** In Figure 3.7(c), when w increases, inference time increases until $w > 15$, but accuracy does not change noticeably. The plateau in accuracy is due to most nodes having few neighbors and nearby nodes sharing similar feature information, which makes a single sampled node be a representative of a node's whole neighborhood. The plateau in inference time indicates that nodes have fewer than 15 neighbors on average on the Pubmed dataset.
- **Nonlinearity l:** Figure 3.7(d) shows that adding nonlinearities ($l =$ True) increases accuracy due to richer expressiveness, but also inference time.
- **Aggregation strategy a:** Figure 3.7(e) shows that the choice of aggregation strategy a has a considerable effect on the accuracy of a graph mining algorithm. Still, we cannot conclude that any aggregation strategy is always superior to others.

Figure 3.7 shows the general tendency in the effects of the parameters. Different datasets have slightly different results (e.g., which w stops increasing accuracy or which k starts bringing oversmoothing). This shows the need for AUTOGM, which chooses the best parameter set automatically for the dataset we employ.

3.5.5 Discussion

In this section, we discuss few interesting observations we find during the experiments.

- **Linear model is fast:** As shown in Section 3.5.2, linear models including PageRank and SGCN are faster than nonlinear models. The fast speed of linear models does not merely come from the absence of a nonlinear operation at each layer. Without the nonlinear operation, multi-layers of linear transformation operations could be compressed to one layer as follows:

$$\begin{aligned} X_3 &= \phi(A\phi(AX_0W_0)W_1) \\ &= A(AX_0W_0)W_1 = A^2X_0W_* \end{aligned}$$

  where $\phi(x) = x$ is a linear operation, X_i denotes hidden embeddings at the i-th layer, A is the adjacency matrix, W_i denotes the transformation matrix at the i-th layer, and $W_* = W_0W_1$ is the compressed matrix. Then, in the linear model, we can precompute A^2X_0 in advance and multiply it only with W_* during training. On the other hand, in the nonlinear model, we need to execute matrix multiplication (AX_iW_i) in every layer. This explains the fast speed of linear models compared to nonlinear models.

- **Winning strategy:** It is hard to define a single winning strategy for graph mining algorithm development that is generalizable to various graphs and applications. However, we observe a few tendencies. High dimensions of messages generally bring high accuracy with a negligible increase in computation time. For instance, SGCN which has high dimension ($d = 64$) shows higher accuracy than PageRank which has low dimension ($d = 1$) across different tasks (Figure 3.5 and 3.6). Second, nonlinear operations are not necessarily required for high accuracy. For example, SGCN shows comparably high accuracy with GCN and GraphSage in both node classification and link prediction tasks. While maintaining similar accuracy, SGCN is faster than GCN and GraphSage due to the absence of nonlinear operations.
- **Performance is affected by input graphs:** In Figures 3.6(b), 3.6(d), 3.6(f), the graph algorithms AUTOGM-1,2,3 that are generated by AUTOGM with different time constraints show similar inference times (sometimes even similar accuracies). The performance of graph algorithms is not only decided by the algorithms but also by input graphs. When graphs are sparse and have simple structures, the graph algorithms will have short inference times regardless of how long inference time constraints we give to AUTOGM. Likewise, when graphs are well-clustered, and features are well-aligned with labels, the tasks become easy, and any graph algorithms would easily get high accuracy. In Figures 3.6(b) and 3.6(d), the algorithms generated with longer time constraints show higher accuracies while having similar inference times. Longer time constraints allow AUTOGM to explore broader scope in the search space and find better algorithms with higher accuracies, while all algorithms end up showing similar inference times thanks to simple input graph structures.

3.6 Related work

3.6.1 AutoML

AutoML is the closest line of related work and the main inspiration for this paper. AutoML algorithms are developed to automate the process of algorithm selection and hyperparameter optimization in the machine learning community. The most closely related to our work in AutoML is Neural Architecture Search (NAS), which focuses on the problem of searching for the deep neural network architecture with the best performance. The search space includes the number of layers, the number of neurons, and the type of activation functions, among other design decisions. NAS broadly falls into three categories: evolutionary algorithms (EA), reinforcement learning (RL), and Bayesian optimization (BO).

EA-based NAS [43, 77, 94] explores the space of architectures by making a sequence of changes (inspired by evolutionary mutations) to networks that have already been evaluated. In RL-based NAS [198, 201], a recurrent neural network iteratively decides if and how to extend a neural architecture; the non-differentiable cost function is optimized with stochastic gradient techniques borrowed from the RL literature. Finally, BO-based NAS [72] models the cost function probabilistically and carefully determines future evaluations to minimize the total number of evaluated architectures. Since EA and RL-based NAS need to evaluate a vast number of architectures to find the optimum, these approaches are not ideally suited for neural architecture search [72].

On the other hand, BO emphasizes being cautious in selecting which architecture to try next to minimize the number of evaluations. As we discuss later, this makes BO suitable for our problem. In the following section, we give a brief description of Bayesian Optimization.

3.6.2 Bayesian Optimization

Given a black-box objective function f with domain $\mathcal{X}$, BO sequentially updates a Gaussian Process prior over f. At time t, it incorporates results of previous evaluations $1, ..., t-1$ into a posterior $P(f|\mathcal{D}_{1:t-1})$ where $\mathcal{D}_{1:t-1} = \{x_{1:t-1}, f(x_{1:t-1})\}$. BO uses this posterior to construct an acquisition function $\phi_t(x)$ that is an approximate measure of evaluating $f(x)$ at time t. BO evaluates f at the maximizer of the acquisition function $x_t = \arg\max_{x \in \mathcal{X}} \phi_t(x)$. The evaluation $f(x_t)$ is then incorporated into the posterior $P(f|\mathcal{D}_{1:t})$, and the process is iterated.

The evaluation point x_t chosen by the acquisition function is an approximation of the maximizer of f. After T iterations, BO returns the parameter set of the maximum f among $x_{1:T}$. When choosing the point x_t to evaluate, the acquisition function $\phi_t(x)$ trades off exploration (sampling from areas of high uncertainty) with exploitation (sampling areas likely to offer an improvement over the current best observation). This cautious trade-off helps to minimize the number of evaluations of f. More details about BO can be found in [15].

These AutoML techniques cannot be directly applied to graph mining, as they require first formalizing autonomous algorithm selection as an optimization problem in a hyperparameter search space. Before UNIFIEDGM, the hyperparameter search space for graph mining was not even defined due to the lack of unity among algorithms. Hence, our proposed UNIFIEDGM allows the graph mining field to exploit state-of-the-art techniques developed in AutoML.

3.6.3 Graph Neural Architecture Search

Various discussions [195] on graph neural architecture search have been initiated. [48] adopts the RL-based neural architecture search approach. [48] uses a recurrent neural network to generate variable-length strings that describe the architectures of graph neural networks, and trains the recurrent network with policy gradient to maximize the expected accuracy of the generated architectures on a validation data set. [185] focuses on designing general space for graph neural networks, that includes three crucial aspects of graph neural architecture design: intra-layer design, inter-layer design, and learning configuration. Based on this design space, [185] develops a controlled random search evaluation procedure to understand the trade-offs of each design dimension. [53, 150] and [186] focus on how to make the graph neural architecture search process more scalable. While most previous works focus on graph neural networks, we broaden the scope to embrace conventional graph mining algorithms such as PageRank [110], Pixie [35], and K-core [143]. We analyze why conventional graph mining algorithms and recent graph neural networks look unrelated at first glance and describe how they could be unified under one framework (Section 3.2.4).

3.7 Summary

Graph mining is generally application-driven. The development of a new mining algorithm is usually motivated by solving a specific real-world problem. Given how general graphs are as an abstraction, the resulting algorithm is usually customized to a dataset, application, and domain. Sometimes, to squeeze out the best performance, graph mining uses various heuristics specialized to certain scenarios — 0.85 for the decaying coefficient in PageRank for web recommendation [110], 2-layered GCNs for citation networks [55], 3-layer GCNs for open academic graphs [65]. These heuristics make graph mining algorithms less generalizable. Practitioners cannot simply apply existing graph algorithms to their problems but must do trial-and-errors until they find optimal (sometimes suboptimal) algorithms for their scenarios. This widens a gap in which state-of-the-art techniques developed in academic settings fail to be optimally deployed in real-world applications.

This paper shows graph mining has enough room to be further generalized. Various message-passing-based graph algorithms stem from the same intuition, homophily, applied in different ways. Based on this shared intuition, UNIFIEDGM unifies graph algorithms using five parameters of the message-passing mechanism: the dimension of the communicated messages, the number of neighbors to communicate with, the number of steps to communicate for, the nonlinearity of the communication, and the message aggregation strategy. UNIFIEDGM-EXT extends UNIFIEDGM with attention and sampling methodologies and unifies a broader scope of graph algorithms under one framework. This unification helps users understand which aspect of algorithms leads to different accuracy/computation time/memory efficiency and which part of algorithms they should tune to achieve their goals. Furthermore, we automate graph mining algorithm development under this unified framework to prevent users from running trial-and-error and reaching suboptimal algorithms. Our main contributions are:

- **Unification:** UNIFIEDGM and UNIFIEDGM-EXT allow conventional graph mining and graph neural network algorithms to be unified under the same framework for the first time, helping practitioners to understand the first principles in message-passing-based algorithms.
- **Design space for graph mining algorithms:** UNIFIEDGM provides the parameter search space necessary to automate graph mining algorithm development.
- **Automation:** Based on the search space defined by UNIFIEDGM, AUTOGM finds the optimal graph algorithm using Bayesian optimization.
- **Budget awareness:** AUTOGM maximizes the performance of an algorithm under a given time/accuracy budget.
- **Effectiveness:** AUTOGM finds novel graph algorithms with the best speed/accuracy trade-off on real-world datasets.

We hope this paper will spark further research in this direction and empower practitioners without much expertise in graph mining to deploy graph algorithms tailored to their scenarios. In this era of big data, new graphs and tasks are generated every day. We believe automated graph mining will bring even more impact on a wider range of users across academia and industry in the future.

Table 3.3: **Example graph mining algorithms under UNIFIEDGM.** Graph mining algorithms can be fully reproduced under UNIFIEDGM with the respective initial node statistics and parameters (d, k, w, l, a). Notation: n is the number of nodes, A denotes an $(n \times n)$ binary adjacency matrix, D denotes an $(n \times n)$ diagonal matrix where $D_{ii} = \sum_j A_{ij}$, I_n denotes an identity matrix of size n, s is the number of seeds, $\mathrm{N}(u)$ denotes the set of sampled neighbors of node u, and $0 < c < 1$ is a decay coefficient. For PageRank, see the formulation given in [179].

Algorithm	Original message passing equation	Initial node statistics	d	k	w	l	a
PageRank	$X_k = c(D^{-1}A)X_{k-1} + \frac{1-c}{n}$	$\frac{1}{n}$ for all nodes	1	∞	-1	False	NA
Pixie	$X_k = c(D^{-1}A)X_{k-1} + \frac{1-c}{s}X_0$	1 for s seeds, 0 others	1	sample	$\frac{2000}{k}$	False	NA
GCN	$X_k = ReLU\left((D^{-\frac{1}{2}}(A+I_n)D^{-\frac{1}{2}})X_{k-1}W_k\right)$	feature vectors	64	2	-1	True	SS
GraphSAGE	$X_k(u) = ReLU\left(\frac{1}{\lvert\mathrm{N}(u)\rvert+1}\sum_{v\in\mathrm{N}(u)\cup u} X_{k-1}(v)W_k\right)$	feature vectors	64	2	25	True	SA
SGCN	$X_k = D^{-\frac{1}{2}}(A+I_n)D^{-\frac{1}{2}}X_{k-1}W_k$	feature vectors	64	2	-1	False	SS

Table 3.4: **Sampling strategies in UNIFIEDGM-EXT.** UNIFIEDGM-EXT defines a sampling strategy based on 1) where the sampling probabilities are learnable, 2) how the sampling probabilities are designed, and 3) when the sampling is executed.

Learnability	**Sampling probability model form**	**Sampling timing**
Heuristic	Uniform distribution	Static
		Dynamic
	Proportional to degree	Static
		Dynamic
Learnable	Concatenation-based attention model	Static
		Dynamic
	Dot-product-based attention model	Static
		Dynamic
	Low-pass filter attention model	Static
		Dynamic

Table 3.5: **Dataset statistics.** AmazonC and AmazonP denote the Amazon Computer and Amazon Photo datasets, respectively. CoauthorC and CoauthorP denote the MS Coauthor CS and Physics, repectively.

Dataset	**Node**	**Edge**	**Feature**	**Label**	**Train/Val/Test**
Cora	2,485	5,069	1,433	7	140/500/1,000
Citeseer	2,110	3,668	3,703	6	120/500/1,000
Pubmed	19,717	44,324	500	3	60/500/1,000
AmazonC	13,381	245,778	767	10	410/1,380/12,000
AmazonP	7,487	119,043	745	8	230/760/6,650
CoauthorC	18,333	81,894	6,805	15	550/1,830/15,950
CoauthorP	34,493	247,962	8,415	5	1,030/3,450/30,010

Table 3.6: **Parameters corresponding to algorithms found by AUTOGM in Figures 3.1.** The Budget column denotes the constraint input to AUTOGM to generate an algorithm.

Dataset	**Budget**	d	k	w	l	a	**Time**	**Acc**
Citeseer	t<0.004	70	4	25	F	SA	0.0039	0.674
	t<0.01	255	4	45	F	SS	0.004	0.683
	t<0.1	68	1	47	T	SS	0.0134	0.686
	a>0.58	138	1	36	F	SA	0.0039	0.622
	a>0.63	25	4	54	F	NA	0.0039	0.665
	a>0.68	39	1	10	T	SS	0.0121	0.69

Table 3.7: **Search efficiency of AUTOGM.** Given the same search time (column 2) and accuracy lower bounds (column 3), AUTOGM finds faster algorithms than RandomSearch across all datasets; similarly, given the same search time (column 2) and inference time upper bounds (column 8), AUTOGM finds more accurate algorithms than RandomSearch across all datasets.

			Fastest Inference (s)		**Accuracy**			**Highest Accuracy**		**Inference (s)**	
Dataset	**Search(s)**	**Min.Acc.**	**AutoGM**	**Rand**	**AutoGM**	**Rand**	**Max.Time(s)**	**AutoGM**	**Rand**	**AutoGM**	**Rand**
Cora	450	0.78	**0.0034**	-	0.79	-	0.004	**0.77**	**0.77**	0.0036	0.0033
Citeseer	800	0.67	**0.0039**	**0.0039**	0.67	0.67	0.004	**0.67**	-	0.0039	-
Pubmed	1,800	0.75	**0.021**	-	0.77	-	0.004	**0.76**	0.71	0.0036	0.0039
AmazonC	5,700	0.85	**0.032**	0.033	0.89	0.87	0.04	**0.85**	-	0.032	-
AmazonP	18,000	0.93	**0.047**	0.065	0.94	0.93	0.05	**0.94**	-	0.048	-
CoauthorC	2,500	0.8	**0.015**	0.016	0.8	0.82	0.02	**0.83**	0.75	0.015	0.02
CoauthorP	1,500	0.9	**0.01**	-	0.91	-	0.01	**0.92**	0.86	0.01	0.01

Chapter 4

Scalability

The main challenge of adapting Graph convolutional networks (GCNs) to large-scale graphs is the scalability issue due to the uncontrollable neighborhood expansion in the aggregation stage. Several sampling algorithms have been proposed to limit the neighborhood expansion. However, these algorithms focus on minimizing the variance in sampling to approximate the original aggregation. This leads to two critical problems: 1) low accuracy because the sampling policy is agnostic to the performance of the target task, and 2) vulnerability to noise or adversarial attacks on the graph.

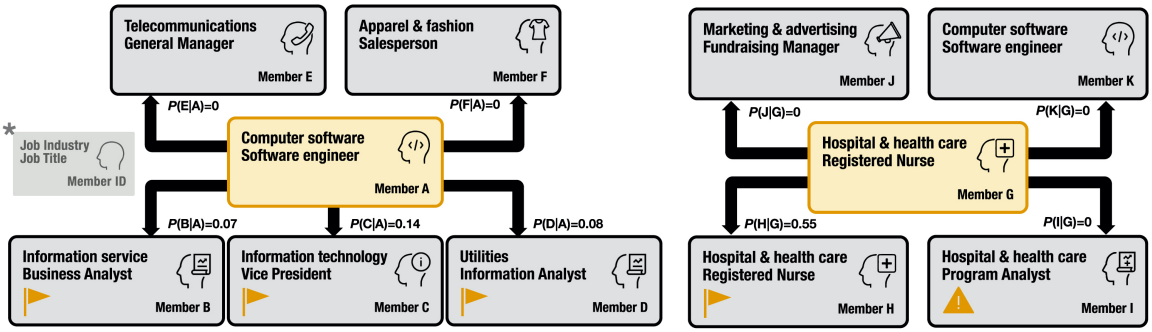
In this paper, we propose a performance-adaptive sampling strategy PASS that samples neighbors informative for a target task. PASS optimizes directly towards task performance, as opposed to variance reduction. PASS trains a sampling policy by propagating gradients of the task performance loss through GCNs and the non-differentiable sampling operation. We dissect the back-propagation process and analyze how PASS learns from the gradients which neighbors are informative and assigned high sampling probabilities. In our extensive experiments, PASS outperforms state-of-the-art sampling methods by up to 10% accuracy on public benchmarks and up to 53% accuracy in the presence of adversarial attacks.

4.1 Motivation

Graph convolutional networks (GCN) [75] have garnered considerable attention as a powerful deep learning tool for representation learning of graph data [7, 129]. For instance, GCNs demonstrate state-of-the-art performance on node classification [27], link prediction [125, 163], and graph property prediction tasks [44]. Motivated by convolutional neural networks, GCNs aggregate information from a node's neighbors analogously to how convolution filters process text or image data [60, 82].

The main challenge of adapting GCNs to large-scale graphs is that GCNs expand neighbors recursively in the aggregation operations, leading to high computation and memory footprints. For instance, given a graph whose average degree is d, L-layer GCNs access d^L neighbors per node on average. If the graph is dense or has many high degree nodes, GCNs need to aggregate a huge number of neighbors for most of the training/test examples. The only way to alleviate this neighbor explosion problem is to sample a fixed number of neighbors in the aggregation operation, thereby regulating the computation time and memory usage [55].

Most samplers minimize the variance in sampling to approximate the original aggregation of the full neighborhood [23, 68, 97, 202]. These sampling policies learn neighbors helpful for



(a) Neighborhood for a software engineer member

(b) Neighborhood for a registered nurse member

Figure 4.1: **PASS learns which neighbors are informative for the job industry classification task on the LinkedIn member-to-member network.** (a) Given Member A from the "Computer software" industry, PASS learns high sampling probabilities for Members B, C, and D from similar industries but low probabilities for Members E and F from different industries. (b) Given Member G from the "Hospital & health care" industry, PASS assigns a low sampling probability to Member I, who has an unrelated career as a "Program Analyst" although he works in the same industry. This shows PASS is able to determine that the attributes of Member I are different from Member G's and thus not informative. For space efficiency, we show part of neighbors; thus, the sum of sampling probabilities does not sum to 1. See Section 4.5 for details.

variance reduction, not neighbors informative for the target task's performance. Thus, those variance reduction-oriented samplers suffer from two critical problems: 1) low accuracy because the sampling policy is agnostic to the performance, and 2) vulnerability to noise or adversarial attacks on the graph because the sampling policy cannot distinguish relevant neighbors from irrelevant ones or true neighbors from adversarially added fake neighbors.

Then what is the optimal sampling policy for GCNs? To answer this question, we come back to the motivation of the aggregation operation. In GCNs, each node aggregates its neighbors' embeddings assuming that neighbors are informative for the target task. We extend this motivation to the sampling policy and sample neighbors informative for the target task. In other words, we aim for a sampler that maximizes the target task's performance instead of minimizing sampling variance.

Here we propose PASS, a performance-adaptive sampling strategy that optimizes a sampling policy directly for task performance. PASS trains the sampling policy based on gradients of the performance loss passed through the GCN. To receive the gradients from the GCN, we need to pass them through the sampling operations, which is non-differentiable. To address this, PASS borrows the log derivative trick commonly used in the reinforcement learning community to train stochastic policies [106, 142]. PASS optimizes the sampling policy jointly with the GCN to minimize the task performance loss, resulting in a considerable performance improvement.

Graph attention networks (GATs) [155] share the same objective of learning the importance of neighbors. They select neighbors through an attention mechanism trained by back-propagating gradients of the performance loss. This mechanism was originally designed as a continuous approximation of the non-differentiable hard selection (i.e., sampling) operation [5, 155]. However, GATs suffer from the same scalability issues as GCNs. Since sampling is inevitable in large scale graphs, we embed the informative neighbor selection directly in the sampler, instead of

Table 4.1: **Commonly used notation in PASS.**

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	input graph with nodes $v_i \in \mathcal{V}$ and edges $(v_i, v_j) \in \mathcal{E}$
L	number of layers in GCN model
$D^{(l)}$	dimension of the l-th hidden layer where $l = 0, 1, \cdots, L$
$H^{(0)}$	$N \times D^{(0)}$ input node feature matrix
$H^{(l)}$	$N \times D^{(l)}$ hidden embeddings at the l-th layer
$W^{(l)}$	$D^{(l)} \times D^{(l+1)}$ transformation matrix at the l-th layer where $l = 0, \cdots, L-1$
$\alpha(\cdot)$	nonlinear activation function
$\alpha_{W^{(l)}}(\cdot)$	abbreviation of $\alpha(W^{(l)} \cdot \cdot)$
$p(j\|i)$	probability of sampling node v_j given node v_i
$q(j\|i)$	approximation of $p(j\|i)$

approximating it downstream with an attention mechanism. In our experiments, we show how PASS not only alleviates scalability issues of GATs but also shows higher performance.

Another advantage of PASS compared to previous sampling-based methods is that we provide theoretical foundations on how sampling policy is updated to optimize the task performance. While other samplers present the back-propagation algorithm to learn the sampling policy as a black box, PASS cracks it open. We present a transparent reasoning process on how PASS learns whether a neighbor is informative from the back-propagated gradients and why it assigns a certain sampling probability to a neighbor.

Through extensive experiments on seven public benchmarks and one LinkedIn production dataset, we demonstrate the superior performance of PASS over existing sampling algorithms. We also present various case studies examining the effectiveness of PASS on real-world datasets (Figure 4.1). Our main contributions are:

- **Performance-adaptiveness:** PASS learns a sampling policy that samples neighbors informative for the task performance.
- **Effectiveness:** PASS outperforms state-of-the-art samplers, being up to 10.4% more accurate.
- **Robustness:** PASS shows up to 53.1% higher accuracy than the baselines in the presence of adversarial attacks.
- **Theoretical foundation:** PASS presents a transparent reasoning process on how it learns whether a neighbor is informative.

4.2 Preliminaries

In this section, we briefly review graph convolutional networks (GCNs) then describe how sampling operations operate and solve the scalability issue in GCNs.

Notations. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a graph with N nodes $v_i \in \mathcal{V}$ and edges $(v_i, v_j) \in \mathcal{E}$. Denote an adjacency matrix $A = (a(v_i, v_j)) \in \mathbb{R}^{N \times N}$ and a feature matrix $H^{(0)} \in \mathbb{R}^{N \times D^{(0)}}$ where $h_i^{(0)}$ denotes the $D^{(0)}$-dimensional feature vector of node v_i. Table 4.1 gives a list of symbols and definitions.

GCN. The GCN models stack layers of first-order spectral filters followed by a nonlinear activation functions to learn node embeddings. When $h_i^{(l)}$ denotes the hidden embeddings of node v_i in the l-th layer, the simple and general form of GCNs is as follows [23]:

$$h_i^{(l+1)} = \alpha(\frac{1}{N(i)} \sum_{j=1}^{N} a(v_i, v_j) h_j^{(l)} W^{(l)}), \quad l = 0, \ldots, L-1 \tag{4.1}$$

where $a(v_i, v_j)$ is set to 1 when there is an edge from v_i to v_j, otherwise 0. $N(i) = \sum_{j=1}^{N} a(v_i, v_j)$ is the degree of node v_i; $\alpha(\cdot)$ is a nonlinear function; $W^{(l)} \in \mathbb{R}^{D^{(l)} \times D^{(l+1)}}$ is the learnable transformation matrix in the l-th layer with $D^{(l)}$ denoting the hidden dimension at the l-th layer.

Sampling operation in GCN. GCNs require the full expansion of neighborhoods across layers, leading to high computation and memory costs. To circumvent this issue, sampling operations are added to GCNs to regulate the size of neighborhood. We first recast Equation 4.1 as follows:

$$h_i^{(l+1)} = \alpha_{W^{(l)}}(\mathbb{E}_{j \sim p(j|i)}[h_j^{(l)}]), \quad l = 0, \ldots, L-1 \tag{4.2}$$

where we combine the transformation matrix $W^{(l)}$ into the activation function $\alpha_{W^{(l)}}(\cdot)$ for concision; $p(j|i) = \frac{a(v_i, v_j)}{N(i)}$ defines the probability of sampling v_j given v_i. Then we approximate the expectation by Monte-Carlo sampling as follows:

$$h_i^{(l+1)} = \alpha_{W^{(l)}}(\frac{1}{k} \sum_{j \sim p(j|i)}^{k} h_j^{(l)}), \quad l = 0, \ldots, L-1 \tag{4.3}$$

where k is the number of sampled neighbors for each node. Now, we regulate the size of neighborhood using k.

Scalability solution. Equations 4.1 and 4.3 describe the computation at the l-th layer in the original GCN and GCN with sampling, respectively. In Equation 4.1, the numbers of nodes that participate in the l-th and $(l+1)$-th layers are both up to $O(N)$, resulting in a time complexity of $O(|\mathcal{E}| D^{(l)} D^{(l+1)})$ where $|\mathcal{E}|$ denotes the number of edges. On the other hand, we can regulate the number of nodes engaged at each layer in the GCN with sampling. When we set the number of nodes sampled for the l-th and $(l+1)$-th layers to k, the number of edges engaged in Equation 4.3 is up to $O(k^2)$, leading to a time complexity of $O(k^2 D^{(l)} D^{(l+1)})$. With $k \ll N$, sampling solves the scalability issue in the GCN successfully [68, 97].

Table 4.2: **PASS out-features competitors**. Comparison of our proposed PASS and existing sampling methods for GCNs.

Method / Property	GraphSage [55]	FastGCN [23]	LADIES [202]	AS-GCN [68]	GCN-BS [97]	**PASS**
Importance Sampling		✓	✓	✓	✓	✓
Learnability				✓	✓	✓
Performance adaptiveness						✓

4.3 Proposed Method

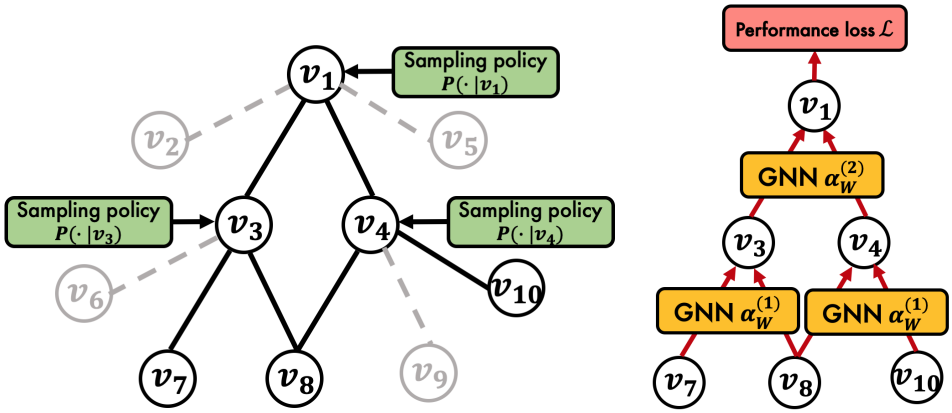
What is the optimal sampling policy for GCNs? To answer this question, we come back to the motivation of the aggregation operation in GCNs. The aggregation operation intends to complement node embeddings with neighbors' embeddings on the assumption that neighbors are informative for the target task. We extend this motivation to the sampling policy and sample neighbors informative for the target task. In other words, we train a sampler that directly maximizes the GCN performance.

The key idea behind our approach is that we learn a sampling policy by propagating gradients of the GCN performance loss through the non-differentiable sampling operation. We first describe a learnable sampling policy function and how it operates in the GCN (i.e., forward propagation) in Section 4.3.1. We then describe how to learn the parameters of the sampling policy by back-propagating gradients through the sampling operation in Section 4.3.2. Finally, we present the overall algorithm and discuss implementation considerations in Section 4.3.3.

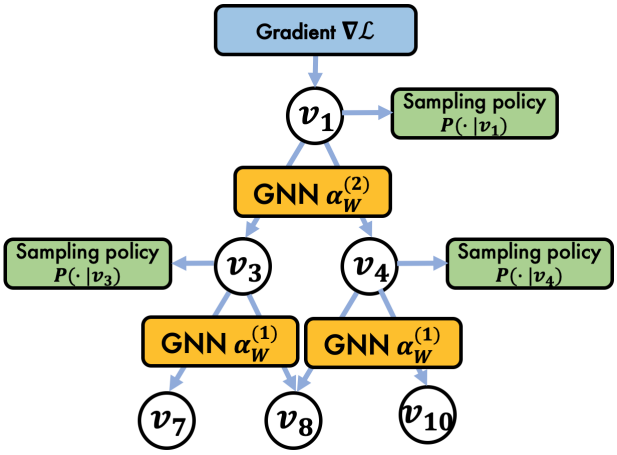
4.3.1 Sampling Policy

Fig. 4.2 shows an overview of PASS. In the forward pass, PASS samples neighbors with its sampling policy (Fig. 4.2(a)), then propagates their embeddings through the GCN (Fig. 4.2(b)). In this section, we introduce our parameterized sampling policy $q^{(l)}(j|i)$ that estimates the probability of sampling node v_j given node v_i at the l-th layer.

The policy $q^{(l)}(j|i)$ is composed of two methodologies, importance $q^{(l)}_{imp}(j|i)$ and random



(a) Sampling process (b) Forward propagation



(c) Backward propagation

Figure 4.2: **PASS is composed of three steps: 1) sampling, 2) feedforward propagation, and 3) backpropagation.** In the backpropagation process, the GCN and the sampling policy are optimized jointly to minimize the GCN performance loss.

sampling $q_{rand}^{(l)}(j|i)$ as follows:

$$q_{imp}^{(l)}(j|i) = (\mathbf{W}_s \cdot h_i^{(l)}) \cdot (\mathbf{W}_s \cdot h_j^{(l)}) \tag{4.4}$$

$$q_{rand}^{(l)}(j|i) = \frac{1}{N(i)} \tag{4.5}$$

$$\tilde{q}^{(l)}(j|i) = a_s \cdot [q_{imp}^{(l)}(j|i), \quad q_{rand}^{(l)}(j|i)] \tag{4.6}$$

$$q^{(l)}(j|i) = \tilde{q}^{(l)}(j|i) / \sum_{k=1}^{N(i)} \tilde{q}^{(l)}(k|i) \tag{4.7}$$

where $\mathbf{W}_s \in \mathbb{R}^{D^{(s)} \times D^{(l)}}$ is a transformation matrix with $D^{(s)}$ denoting the hidden dimension in

the sampling policy and $D^{(l)}$ denoting the hidden dimension of the l-th layer; $h_i^{(l)}$ is the hidden embedding of node v_i at the l-th layer; $N(i)$ is the degree of node v_i; $a_s \in \mathbb{R}^{1\times 2}$ is an attention vector; and $q^{(l)}(\cdot|i)$ is normalized to sum to 1. $\mathbf{W}_s$ and a_s are learnable parameters of our sampling policy, which will be updated toward performance improvement.

We describe each component in the sampling policy.

Importance Sampling. The first term $q_{imp}^{(l)}(j|i)$ computes the intermediate score of sampling v_j given node v_i in the l-th layer, corresponding to importance sampling. We first map the hidden embeddings $h_i^{(l)}$ and $h_j^{(l)}$ into the $D^{(s)}$-dimension through the transformation matrix $\mathbf{W}_s$, then compute the similarity between these two embeddings by dot product. We describe the intuition behind this dot product-based importance sampling in Section 4.4.

Random Sampling. The second term $q_{rand}^{(l)}(j|i)$ assigns the same sampling probability to each node the neighborhood. When a graph is well-clustered, nodes are connected with all informative neighbors. Then random sampling becomes effective since its randomness helps aggregate diverse neighbors, thus preventing the GCN from overfitting. By capitalizing on both importance and random samplings, our sampling policy better generalizes across various graphs. We show how random sampling complements importance sampling experimentally in Section 4.5.

Attention of Sampling. The attention a_s regulates the tradeoff between importance sampling $q_{imp}^{(l)}(j|i)$ and random sampling $q_{rand}^{(l)}(j|i)$. a_s learns which sampling methodology is more effective on a given task. We initialize a_s with higher attention to the random sampling than the importance sampling and allow the model to examine a broad scope of neighbors at first.

While our sampling policy $q^{(l)}(j|i)$ assigns a distinct sampling probability to each edge at each layer, it shares the parameters $(\mathbf{W}_s, a_s)$ across all edges and all layers. This parameter sharing helps our model generalize and prevents the sampling policy from overfitting to the training set.

4.3.2 Training the Sampling Policy

As shown in Fig. 4.2(c), after a forward pass with sampling, the GCN computes the performance loss (e.g., cross-entropy for node classification) then back-propagates gradients of the loss. Next, we describe how the gradients of the loss pass through the non-differentiable sampling operation to update our sampling policy.

When θ denotes parameters $(\mathbf{W}_s, a_s)$ in our sampling policy $q_\theta^{(l)}$, we can write the sampling operation with $q_\theta^{(l)}(j|i)$ as follows:

$$h_i^{(l+1)} = \alpha_{W^{(l)}}(\mathbb{E}_{j\sim q_\theta^{(l)}(j|i)}[h_j^{(l)}]), \quad l = 0, \ldots, L-1 \tag{4.8}$$

Before being fed as input to the GCN transformation, $\alpha_{W^{(l)}}$, the hidden embeddings go through a non-differentiable expectation under the sampling policy, which is non-differentiable. To pass gradients of the loss through the expectation, we apply the log derivative trick [160], widely used in reinforcement learning to compute gradients of stochastic policies. Then the gradient $\nabla_\theta \mathcal{L}$ of the loss $\mathcal{L}$ w.r.t. the sampling policy $q_\theta^{(l)}(j|i)$ is computed as follows:

Theorem 3. *Given the loss $\mathcal{L}$ and the hidden embedding $h_i^{(l)}$ of node v_i at the l-th layer, the gradient*

of $\mathcal{L}$ w.r.t. the parameter θ of the sampling policy $q_\theta^{(l)}(j|i)$ is computed as follows:

$$\nabla_\theta \mathcal{L} = \frac{d\mathcal{L}}{dh_i^{(l+1)}} \alpha_{W^{(l)}} \mathbb{E}_{j \sim q_\theta^{(l)}(j|i)} [\nabla_\theta \log q_\theta^{(l)}(j|i) h_j^{(l)}]$$

Proof. By the chain rule, $\frac{d\mathcal{L}}{d\theta}$ is decomposed as follows:

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{dh_i^{(l+1)}} \frac{dh_i^{(l+1)}}{d\theta}$$

We compute the gradient of $h_i^{(l+1)}$ w.r.t. θ as follows:

$$\begin{aligned}
\frac{dh_i^{(l+1)}}{d\theta} &= \nabla_\theta \alpha_{W^{(l)}} (\mathbb{E}_{j \sim q_\theta^{(l)}(j|i)} [h_j^{(l)}]) \\
&= \alpha_{W^{(l)}} (\nabla_\theta \sum_{k=0}^{N(i)} q_\theta^{(l)}(u_k|i) h_{u_k}^{(l)}) \\
&= \alpha_{W^{(l)}} (\sum_{k=0}^{N(i)} \nabla_\theta q_\theta^{(l)}(u_k|i) h_{u_k}^{(l)}) \\
&= \alpha_{W^{(l)}} (\sum_{k=0}^{N(i)} q_\theta^{(l)}(u_k|i) \nabla_\theta \log q_\theta^{(l)}(u_k|i) h_{u_k}^{(l)}) \\
&= \alpha_{W^{(l)}} (\mathbb{E}_{j \sim q_\theta^{(l)}(j|i)} [\nabla_\theta \log q_\theta^{(l)}(j|i) h_j^{(l)}])
\end{aligned}$$

where the nodes $\{u_k\}_{k=1}^{N(i)}$ are neighbors of v_i. The log derivative trick leveraging the property of the logarithm $\nabla_\theta \log q_\theta = \nabla_\theta q_\theta / q_\theta$ to tranform the sum into an expectation under q_θ that we can sample is applied in the fourth equation. □

In Theorem 3, we describe the gradient of the loss w.r.t the sampling policy of a single edge (i.e., sampling probability $q_\theta^{(l)}(j|i)$ of node j given node i). In the implementation, we average the gradients $\nabla_\theta \mathcal{L}$ passed through all edges. Also, to show how the gradients w.r.t. the sampling policy is passed through GCN parameters (σ_W) more efficiently, we omit how the gradients pass through the ReLU. Except for the ReLU condition ($x > 0$), there is no difference in the final result.

Based on Theorem 3, we pass the gradients of the GCN performance loss to the sampling policy through the non-differentiable sampling operation and optimize the sampling policy for the GCN performance.

4.3.3 Algorithm

Algorithms 3 and 4 describe how we train graph convolutional networks with our sampling policy. Our algorithm's framework is composed of three steps: 1) sampling, 2) feedforward propagation, and 3) backpropagation.

In the sampling process, we define a computation graph. The computation graph is a L-layer network composed of nodes and edges participating in a minibatch. We generate the computation

Algorithm 3: One minibatch in PASS

Require: a minibatch of labeled nodes: $\{v_i, y_i\}_{i=1}^{b}$, sample number: k, GCN model: $\{W^{(l)}\}_{l=1}^{L-1}$, sampling policy: $q^{(l)}(j|i)$

Ensure: updated GCN model and sampling policy

```
G_comp = Sampler({v_i}_{i=1}^b, q^(l)(j|i), k)
for l from 1 to L − 1 do
    for v_i in G_comp[l + 1] do
        Neighbor(v_i) = neighbor nodes of v_i in G_comp[l]
        h_i^(l+1) = α(Σ_{j∈Neighbor(v_i)} h_j^(l) W^(l))
    end for
end for
L = loss({h_i^(L), y_i}_{i=1}^b)
for l from L − 1 to 1 do
    update W^(l) using ∇_{W^(l)} L
    update q^(l)(j|i) using ∇_{q^(l)(j|i)} L
end for
return {W^(l)}_{l=1}^{L-1} and q^(l)(j|i)
```

Algorithm 4: Sampler

Require: a minibatch of nodes: $\{v_i\}_{i=1}^{b}$, sampling policy: $q^{(l)}(j|i)$, sample number: k

Ensure: computation graph $\mathcal{G}_{comp}$

```
G_comp[L] = {v_i}_{i=0}^b
for l = L − 1 to 1 do
    for v_i in G_comp[l + 1] do
        G_comp[l] = G_comp[l] + {v_{u_j}}_{j=1}^k ∼ q^(l)(u_j|i)
    end for
end for
return G_comp
```

graph using our sampling policy $q^{(l)}(j|i)$ in a top-down manner ($l : L \to 1$). When a minibatch of size b is given, the b nodes are located at the L-th layer; each node samples k neighbors following the sampling policy $q^{(L)}(j|i)$; the sampled kb nodes are located at the $(L-1)$-th layer; each node samples k neighbors following the sampling policy $q^{(L-1)}(j|i)$; the sampled k^2b nodes are located at the $(L-2)$-th layer; repeat until the 1-st layer.

After acquiring the computation graph, we do feedforward propagation in a bottom-up manner ($l : 1 \to L$), i.e., iteratively aggregate neighboring embeddings and pass them through transformations. After computing the loss, we do backpropagation and update parameters using gradients of the loss in a top-down manner ($l : L \to 1$). In the backpropagation phase, we update the parameters of both the GCN and the sampling policy. In practice, we find that the gradients from the 1-st layer are sufficient to successfully update the sampling policy. We repeat the whole process with each minibatch.

Implementation In the backpropagation phase, PASS uses the log derivative trick [160] to pass gradients of the loss from the GCN to the sampling policy through an expectation operation. In reinforcement learning, the log derivative trick is used to compute the gradient of the expectation of a scalar function (e.g., a reward function) [106, 160]. However, our model applies the log derivative trick to compute the gradient of an expectation of vectors (matrices for a batch implementation) located in the middle of the neural network. The implementation of the log derivative trick in this context requires hand-crafted backpropagation. If we were to brute-force the implementation, we would need to compute $d\mathcal{L}/dh_i^{(l)}$, compute $dh_i^{(l)}/d\theta$ using the log derivative trick, multiply them to output $d\mathcal{L}/d\theta$, and finally update θ manually using $d\mathcal{L}/d\theta$.

Here, we introduce an additional SUB-LOSS trick that allows us to leverage the backpropagation mechanisms built in deep learning frameworks (e.g., PyTorch, TensorFlow).

Theorem 4 (SUB-LOSS trick)**.** *Given $\theta \in \mathbb{R}^{D^{(s)}}$, a hidden embedding $h(\theta) \in \mathbb{R}^{D^{(l)}}$, and a loss $\mathcal{L}(h) \in \mathbb{R}$, the gradient of the loss $\mathcal{L}$ w.r.t. θ is presented as follows:*

$$\frac{d\mathcal{L}}{d\theta} = \frac{d}{d\theta}(\frac{d\mathcal{L}}{dh} \cdot h)$$

with an assumption $\frac{d}{d\theta}(\frac{d\mathcal{L}}{dh}) = 0$.

Proof. Proofs are given in Appendix 4.8.1 □

With the SUB-LOSS trick, we compute an auxiliary loss $\mathcal{L}_{aux} = d\mathcal{L}/dh_i^{(l)} \cdot h_i^{(l)}$ and simply call the backpropagation function of our deep learning framework on $\mathcal{L}_{aux}$ to compute the gradient w.r.t. θ. More details are in Appendix 4.8.1.

4.4 Theoretical Foundation

In this section, we dissect the back-propagation process of PASS and analyze how PASS learns whether a neighbor is informative for the target task from gradients of the performance loss (i.e., why it assigns a certain sampling probability to the neighbor).

GCNs train their parameters to move the node embeddings $h_i^{(l)}$ in the direction that minimizes the performance loss $\mathcal{L}$, i.e., the gradient $-d\mathcal{L}/dh_i^{(l)}$. PASS promotes this process by sampling neighbors whose embeddings are aligned with the gradient $-d\mathcal{L}/dh_i^{(l)}$. When $h_i^{(l)}$ is aggregated with the embedding $h_j^{(l)}$ of a sampled neighbor aligned with the gradient, it moves in the direction that reduces the loss $\mathcal{L}$.

In other words, PASS decides a neighbor node v_j is informative when its embedding $h_j^{(l)}$ is aligned with the gradient $-d\mathcal{L}/dh_i^{(l)}$. In Fig.4.3, PASS considers v_3 more informative than v_5 since $h_3^{(l)}$ is better aligned with $-d\mathcal{L}/dh_2^{(l)}$, thereby helping $h_2^{(l)}$ move towards loss reduction. In Theorem 5, we show how PASS measures the alignment between $-d\mathcal{L}/dh_i^{(l)}$ and $h_j^{(l)}$ and how it increases the sampling probability $q^{(l)}(j|i)$ in proportion to this alignment.

Theorem 5. *Given a source node v_i and its neighbor node v_j,* PASS *increases a sampling probability $q^{(l)}(j|i)$ in proportion to the dot product of $-d\mathcal{L}/dh_i^{(l)}$ and $h_j^{(l)}$.*

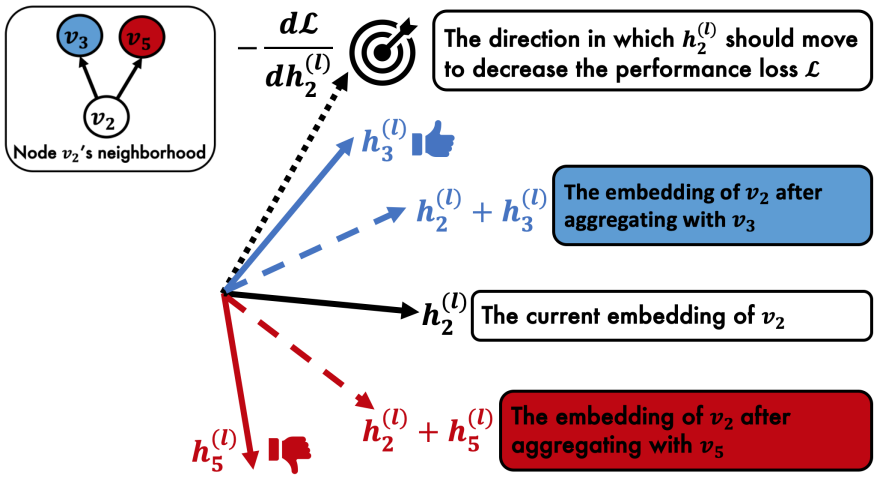


Figure 4.3: **Interpretation of why PASS assigns higher sampling probability to node v_3 than v_5 given source node v_2.** Node v_3's embedding $h_3^{(l)}$ helps v_2's embedding $h_2^{(l)}$ move in the direction $-d\mathcal{L}/dh_2^{(l)}$ that decreases the performance loss $\mathcal{L}$ while aggregating the embedding of node v_5 would move v_2 in the opposite direction.

Proof. Let $z_i^{(l)} = \mathbb{E}_{j\sim q^{(l)}(j|i)}[h_j^{(l)}]$ denote an intermediate hidden embedding of node v_i at the l-th layer after the aggregation operation. By the chain rule, $-d\mathcal{L}/dq^{(l)}(j|i)$ decomposes into $\left(-d\mathcal{L}/dz_i^{(l)}\right)\cdot\left(dz_i^{(l)}/dq^{(l)}(j|i)\right)$. The first component $-d\mathcal{L}/dz_i^{(l)}$ is the direction $z_i^{(l)}$ needs to move towards to decrease the loss $\mathcal{L}$. The second component $dz_i^{(l)}/dq^{(l)}(j|i)$ is computed as follows:

$$\begin{aligned}\frac{dz_i^{(l)}}{dq^{(l)}(j|i)} &= \frac{d}{dq^{(l)}(j|i)}\mathbb{E}_{k\sim q^{(l)}(k|i)}[h_k^{(l)}] \\ &= \frac{d}{dq^{(l)}(j|i)}\sum_{k=0}^{N(i)} q^{(l)}(u_k|i)h_{u_k}^{(l)} \\ &= \frac{d}{dq^{(l)}(j|i)}(q^{(l)}(j|i)h_j^{(l)}) = h_j^{(l)}\end{aligned}$$

where the nodes $\{u_k\}_{k=1}^{N(i)}$ are neighbors of v_i. In the third equation, $\nabla_{q(j|i)}q(u_k|i)$ is zero for nodes $u_k \neq j$. Then $-d\mathcal{L}/dq^{(l)}(j|i)$ is presented as follows:

$$-\frac{d\mathcal{L}}{dq^{(l)}(j|i)} = (-\frac{d\mathcal{L}}{dz_i^{(l)}})\cdot\frac{dz_i^{(l)}}{dq^{(l)}(j|i)} = (-\frac{d\mathcal{L}}{dz_i^{(l)}})\cdot h_j^{(l)} \tag{4.9}$$

Since $d\mathcal{L}/dz_i^{(l)} = d\mathcal{L}/dh_i^{(l)}$, $-d\mathcal{L}/dq^{(l)}(j|i)$ is decided by the dot product of $-d\mathcal{L}/dh_i^{(l)}$ and $h_j^{(l)}$. When $-d\mathcal{L}/dh_i^{(l)}$ and $h_j^{(l)}$ have similar directions, $-d\mathcal{L}/dq^{(l)}(j|i)$ becomes large and the probability $q^{(l)}(j|i)$ is updated to increase by the gradient descent. □

In Theorem 5, PASS estimates the alignment between $-d\mathcal{L}/dh_i^{(l)}$ and $h_j^{(l)}$ from their dot product.

Table 4.3: **Dataset statistics.** LinkedIn dataset on member networks has two labels, member industry and job title.

Dataset	Nodes	Edges	Features	Labels
Cora	2,485	5,069	1,433	7
Citeseer	2,110	3,668	3,703	6
Pubmed	19,717	44,324	500	3
AmazonC	13,381	245,778	767	10
AmazonP	7,487	119,043	745	8
MsCS	18,333	81,894	6,805	15
MsPhysics	34,493	247,962	8,415	5
LinkedIn	39K+	1.7M+	20+	(industry) ∼150 (title) ∼8,000

However, the dot product as a measure of alignment prefers $h_j^{(l)}$ with a large L1 norm. To prevent this issue, we normalize $h_j^{(l)}$ in our experiments.

This reasoning process leads to two important considerations. First, it crystallizes our understanding of the aggregation operation in GCNs. The aggregation operation enables a node's embedding to move towards its neighbors' to reduce the performance loss. Second, this reasoning process shows the benefits of jointly optimizing the GCN and the sampling policy. Optimizing the sampling policy for task performance allows an embedding to choose which neighbors to move towards, leading to the minimum loss more efficiently.

4.4.1 Design of Sampling Policy

In Equation 4.9, $-d\mathcal{L}/dq^{(l)}(j|i)$ measures alignment/similarity between $-d\mathcal{L}/dz_i^{(l)}$ and $h_j^{(l)}$ by a dot product. We choose the same similarity measurer, the dot product, to estimate the importance of neighbors in our sampling policy (Equation 4.4). When we choose another similarity measurer, for instance, a concatenation-based measurer $a(v_i, v_j) = a \cdot [\mathbf{W} \cdot h_i^{(l)} || \mathbf{W} \cdot h_j^{(l)}]$ used in graph attention networks (GAT), we observe up to 28% drop in accuracy (more details in Section 4.5). This shows a careful design of the sampling policy has a large impact on performance.

4.5 Experiments

In this section, we evaluate the performance of PASS compared to state-of-the-art sampling algorithms on GCNs.

Table 4.4: **Effectiveness of PASS.** PASS outperforms all baselines up to 10.4% on the benchmark datasets and up to 10.2% on our production datasets (LnkIndustry, LnkTitle). Results on the benchmark datasets are presented in precision. Results on our production datasets are presented in percentage point (pp) with respect to GraphSage (random sampling). A higher precision/percentage point is better.

Method	**Cora**	**Citeseer**	**Pubmed**	**AmazonC**	**AmazonP**	**MsCS**	**MsPhysics**	**LnkIndustry**	**LnkTitle**
FastGCN	0.582	0.496	0.569	0.480	0.542	0.520	0.638	-4.2pp	-2.0pp
AS-GCN	0.462	0.387	0.502	0.419	0.480	0.403	0.516	-7.1pp	-0.6pp
GraphSage	0.788	0.698	0.792	0.707	0.787	0.766	0.875	0.0pp	0.0pp
GCN-BS	0.788	0.693	0.809	0.736	0.800	0.780	0.887	1.8pp	0.7pp
PASS	**0.821**	**0.715**	**0.858**	**0.757**	**0.855**	**0.884**	**0.934**	**10.2pp**	**1.3pp**

4.5.1 Experimental setting

We compare the performance of PASS and other sampling algorithms on semi-supervised node classification tasks. All experiments were conducted on the same p2.xlarge Amazon EC2 instance.
Datasets. We use seven public datasets — three citation networks (Cora, Citeseer, and Pubmed) [126], two co-purchase graphs (Amazon Computers and Amazon Photo) [127], and two co-authorship graphs (MS CS and MS Physics) [127]. In addition, we also evaluate on a subset of LinkedIn social networks where nodes are alumni from a US university, and edges are connections between them. We use members' latest job [88] title and their industry as labels. We split $50\%/10\%/40\%$ of the datasets into the training/validation/test sets, respectively. We report their statistics in Table 4.3.
Baselines. We compare PASS with four state-of-the-art sampling methods: GraghSage [55], FastGCN [23], AS-GCN [68], and GCN-BS [97]; and one attention method: GAT [155]. For fair comparison, all methods share the same network structure, two-hidden-layer GCN with all hidden dimensions set to 64. Please refer to Appendix 4.8.2 for more details.
Unified time complexity bound. With the batch size set to 64, layer-wise sampling methods (FastGCN, AS-GCN) sample 64 nodes per layer. For a fair comparison, node-wise sampling methods (GraghSage, GCN-BS, and PASS) sample one neighbor per node, thus sampling 64 nodes per layer in total. You can find the results with larger numbers of samples in Appendix 4.8.4.
Evaluation with Sampling. Previous works [23, 68, 97] sample during training but not during testing: they compute node embeddings on the test set by aggregating full neighborhoods. This setting is unrealistic: the prohibitive time and memory costs from the full neighborhood expansion that prompted us to sample during training are also issues during testing. **In this work, we sample both during training and testing.** This results in a significant drop in accuracy for certain baselines, especially layer-wise samplers.

4.5.2 Effectiveness

We measure the accuracy of each sampling algorithm on the node classification tasks. In Table 4.4, our proposed PASS shows the highest accuracy among all baselines across all datasets. Layer-wise methods (FastGCN, AS-GCN) show lower accuracy than node-wise methods (GraphSage, GCN-BS, PASS).

Layer-wise samplers define the probability of sampling node v_j given a set of source nodes $\{v_k\}_{k=i_1}^{i_n}$ as $q(j|i_1, \cdots, i_n)$. Since they sample from a union pool of each source node's neighborhood, there is no guarantee for each source node to fairly sample their neighbors. Moreover, a node's sampling probability in a layer-wise sampler is proportional to its degree, which follows a power-law distribution [39]. If a source node v_i's neighbors all have smaller degrees than neighbors of other source nodes, none of v_i's neighbors are likely to be sampled, and v_i fails to aggregate any neighbor information. This results in sparse connections between layers and poor performance for layer-wise samplers.

Among node-wise sampling methods, PASS outperforms GraphSage and GCN-BS. One interesting result is that GraphSage, which just samples neighbor randomly, still shows good performance among carefully-designed sampling algorithms. The seven public datasets are well-clustered; thus there is not much room to be improved by importance sampling. In the following Section, we show

Table 4.5: **Robustness of PASS** PASS maintains high accuracy in various graph noise scenarios, while the accuracy of all other baselines plummets with noise. PASS is effective not only in sampling informative neighbors but also in removing irrelevant neighbors. *Cite*, *Pub*, *AC*, and *AP* denote *Citeseer*, *Pubmed*, *AmazonC*, and *AmazonP*, respectively.

	Fake connections among existing nodes						**Fake neighbors with random feature vectors**					
Method	**Cora**	**Cite**	**Pub**	**AC**	**AP**	**MsCS**	**Cora**	**Cite**	**Pub**	**AC**	**AP**	**MsCS**
FastGCN	0.293	0.254	0.416	0.300	0.307	0.292	0.597	0.513	0.614	0.502	0.566	0.563
AS-GCN	0.229	0.171	0.334	0.206	0.167	0.176	0.233	0.152	0.379	0.271	0.169	0.252
GraphSage	0.312	0.261	0.439	0.376	0.306	0.262	0.282	0.269	0.459	0.264	0.264	0.248
GCN-BS	0.320	0.265	0.457	0.387	0.305	0.264	0.571	0.493	0.681	0.639	0.686	0.622
PASS	**0.658**	**0.603**	**0.811**	**0.669**	**0.698**	**0.822**	**0.722**	**0.681**	**0.761**	**0.672**	**0.783**	**0.667**

when the graphs have noise (e.g., random connections between different communities), GraphSage plummets in accuracy. GCN-BS shows higher accuracy than other baselines but lower than our method. While PASS learns a *shared sampling policy* across all edges with the *performance loss*, GCN-BS trains *individual sampling probabilities* for each edge with *variance reduction loss*. This result presents the effectiveness of the parameter sharing and the performance loss of PASS.

4.5.3 Robustness

To examine the robustness of sampling algorithms, we inject noise into graphs. We investigate two different noise scenarios: 1) fake connections among existing nodes, and 2) fake neighbors with random feature vectors. These two scenarios are common in real-world graphs. The first "fake connection" scenario simulates connections made by mistake or unfit for purpose (e.g., connections between family members in a job search platform). The second scenario simulates fake accounts with random attributes used for fraudulent activities. For each node, we generate five true neighbors and five fake neighbors for each scenario. We keep the rest of the experimental setting as in Section 4.5.1.

Table 4.5 shows that PASS consistently has high accuracy across all scenarios, while the performance of all other methods plummets. The sparse connection problems of layer-wise sampling methods (FastGCN, AS-GCN) become worse with graph noise. Node-wise sampling methods also show much lower accuracy than on the original graphs (Table 4.4). GraphSage gives the same sampling probability to true neighbors and fake neighbors, resulting in a sharp drop in accuracy. GCN-BS is likely to sample high-degree or dense-feature nodes, which help stabilize the sampling variance, regardless of their relationship with the source node. Thus GCN-BS fails to distinguish fake neighbors from true neighbors. On the other hand, PASS learns which neighbors are informative or fake from gradients of the performance loss (Theorem 5). These results show that the optimization of the sampling policy toward performance brings robustness to graph noise.

4.5.4 Convergence $\&$ Variance

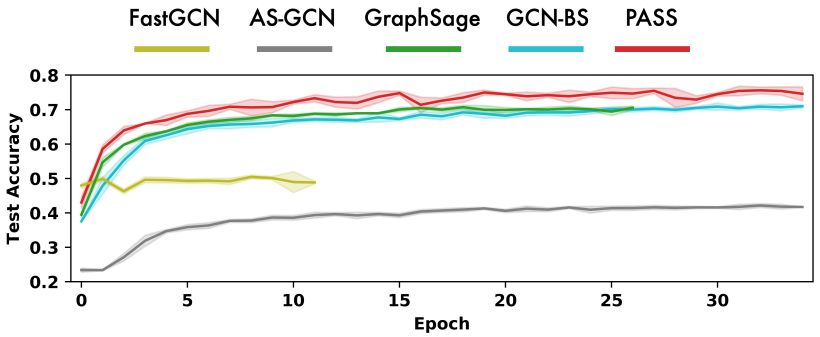
In this section, we analyze the convergence and variance of sampling algorithms across epochs. We train each algorithm 5 times and plot the mean and standard deviation. PASS shows the highest accuracy by a significant margin ($+5.5\%$) with a slightly higher variance ($0.5-1.2\%$) than baselines ($0.1-0.6\%$). Static algorithms, including GraphSage and FastGCN, show low variance since their sampling policy is decided heuristically and fixed. Learnable algorithms, including AS-GCN and GCN-BS, show low variance because their sampling policy is optimized for variance reduction. On the other hand, PASS optimizes for performance improvement. PASS explores neighbors to find the most informative, leading to slightly higher variance and much higher accuracy than baselines that exploit the neighbors that minimize variance.

4.5.5 Comparison with GAT

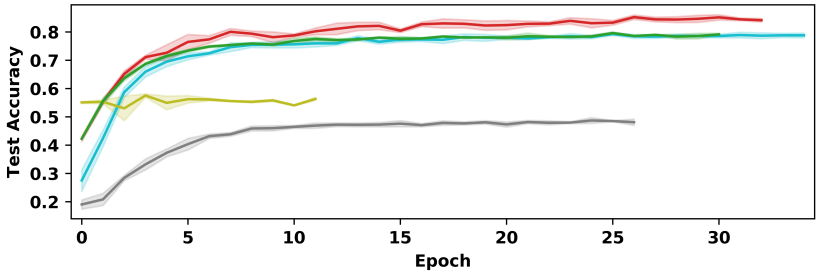
In this section, we compare PASS with GATs. PASS and GATs share the same objective of learning the importance of neighbors, respectively through sampling probabilities and attention scores. While

Table 4.6: **Comparison between PASS and GATs.** PASS is scalable across all datasets while GATs run out of memory on Pubmed, Amazon Computer, MS CS, and MS Physics datasets. We run PASS with both 1 and 5 sampled neighbors, trading-off speed for accuracy. On the few datasets where GATs are applicable, PASS (5) shows comparable or higher accuracy as GATs with considerably shorter training and test time.

	Accuracy			**Training time (s)**			**Test time (s)**		
Dataset	**GATs**	**PASS (1)**	**PASS (5)**	**GATs**	**PASS (1)**	**PASS (5)**	**GATs**	**PASS (1)**	**PASS (5)**
Cora	**0.850**	0.821	**0.847**	189.670	9.459	7.226	0.122	0.022	0.033
Citeseer	**0.744**	0.715	0.735	404.904	13.962	13.225	0.175	0.043	0.069
Pubmed	-	0.858	**0.871**	-	87.660	94.918	-	0.612	1.510
AmazonC	-	0.757	**0.886**	-	52.060	184.522	-	0.256	1.218
AmazonP	0.905	0.855	**0.944**	1869.690	30.060	68.134	0.709	0.094	0.338
MS CS	-	0.884	**0.918**	-	101.840	142.099	-	0.811	3.113
MS Physics	-	0.934	**0.952**	-	439.378	507.816	-	4.162	8.445



(a) Amazon Computer



(b) Amazon Photo

Figure 4.4: **The convergence of PASS on the test set in terms of epochs.**

PASS solves the scalability issues of GCNs with sampling, GATs suffer from high computation and memory footprints. To investigate their scalability, we train GATs and PASS on a GPU with 16GB of memory. We run PASS with both 1 and 5 sampled neighbors (denoted as PASS-1 and PASS-5), trading-off speed for accuracy.

Table 4.6 shows PASS is scalable across all datasets while GAT runs out of memory on the Pubmed, Amazon Computer, MS CS, and MS Physics datasets. On the few datasets where GATs are applicable (Cora, Citeseer, and Amazon Photo), PASS-1 shows up to $\times 60$ shorter training time and $\times 8$ shorter test time than GAT while having 5% lower accuracy. When sampling more neighbors to increase accuracy at the price of speed, PASS-5 shows comparable or higher accuracy as GATs while maintaining shorter training and test times. On the Amazon Photo dataset, where neighbors have 20 neighbors on average, PASS-5 shows 5% higher accuracy than GATs while only sampling 5 neighbors when GATs consider the full neighborhood. This shows PASS is scalable and learns neighbors informative for performance improvement.

4.5.6 Ablation Study

In this section, we examine the effectiveness of importance and random sampling in PASS. We compare the performance of our dot-product-based importance sampling with the importance sampling mechanism introduced in GATs, presented as

$$q^{(l)}_{GAT}(j|i) = a \cdot [\mathbf{W} \cdot h^{(l)}_i || \mathbf{W} \cdot h^{(l)}_j]$$

Table 4.7: **Ablation study of PASS.** Our dot-product-based importance sampling q_{imp} outperforms the GAT-version importance sampling mechanism. Random sampling q_{rand} complements importance sampling q_{imp}.

Dataset	GAT-version	q_{imp}	$q_{imp} + q_{rand}$
Cora	0.574	0.779	0.821
Citeseer	0.456	0.706	0.715
Pubmed	0.606	0.862	0.858
AmazonC	0.482	0.746	0.757
AmazonP	0.575	0.854	0.855
MsCS	0.661	0.883	0.884
MsPhysics	0.683	0.933	0.934

where $\mathbf{W}$ and a are a trainable $(D^{(s)} \times D^{(l)})$ matrix and a $(1 \times 2D^{(s)})$ vector, respectively.

Table 4.7 shows our dot-product-based importance sampling outperforms the GATs version by up to 27.9% accuracy. The addition of random sampling improves accuracy by up to another 4.2% accuracy as the noise helps aggregate diverse neighbors (i.e., exploration), preventing the GCN from overfitting.

4.5.7 Case Study

Figure 4.1 shows a case study where PASS is used to classify the job industry of nodes in the LinkedIn social network. PASS learns which neighbors are informative for the task. Given Member A from the "Computer software" industry, PASS learns high sampling probabilities for Members B, C, and D from similar industries but low probabilities for Members E and F from different industries. Given Member G from the "Hospital $\&$ health care" industry, PASS assigns a low sampling probability to Member I, who has an unrelated career as a "Program Analyst" although he works in the same industry. This shows PASS is able to determine that the attributes of Member I are different from Member G's and thus not informative. These case studies show the effectiveness of PASS at identifying informative neighbors on real-world graphs. Additional case studies on the Cora and Amazon photo datasets are in Appendix 4.8.

4.5.8 Visualization of PASS

In Section 4.4, we saw that PASS decides whether a neighbor v_j is informative based on the alignment between its embedding $h_j^{(l)}$ and the gradient $-d\mathcal{L}/dh_i^{(l)}$ of the loss w.r.t the source node v_i. Figure 4.5 shows the hidden-layer embeddings projected to 2D via t-SNE [17]. Numbers denote the increase/decrease in sampling probabilities. The neighbors in the red area, which are close to the gradient (the red cross), see an increase in their sampling probabilities. Conversely, the neighbors in the blue area, which are far from the gradient, see a decrease in their sampling probabilities. This result shows our theoretical foundation holds on real-world datasets.

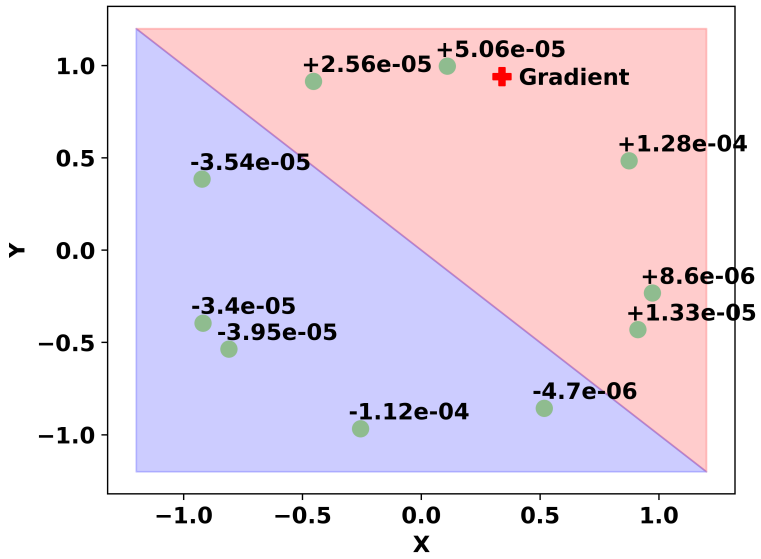


Figure 4.5: **Visualization of PASS.** The hidden-layer embeddings of a neighborhood in the Amazon Computer dataset (visualized by t-SNE [17]). The red cross denotes the gradient of the loss w.r.t the source node and green points denote the embeddings of neighbor nodes. Numbers denote the increase/decrease in sampling probabilities. PASS increases sampling probabilities for neighbors in the red area, close to the gradient, while decreasing probabilities for the neighbors in the blue zone, which are far from the gradients.

4.6 Related Work

The sampling algorithms for GCNs broadly fall into two categories: node-wise sampling and layer-wise sampling.

- **Node-Wise Sampling.** The sampling distribution $q(j|i)$ is defined as a probability of sampling node v_j given a source node v_i. In node-wise sampling, each node samples k neighbors from its sampling distribution, then the total number of nodes in the l-th layer becomes $O(k^l)$. GraphSage [55] is one of the most well-known node-wise sampling method with the uniform sampling distribution $q(j|i) = \frac{1}{N(i)}$. GCN-BS [97] introduces a variance reduced sampler based on multi-armed bandits. GCN-BS defines an individual sampling probability $q(j|i)$ for each edge and trains them toward minimum sampling variance.
- **Layer-Wise Sampling.** To alleviate the exponential neighbor expansion $O(k^l)$ of the node-wise samplers, layer-wise samplers define the sampling distribution $q(j|i_1, \cdots, i_n)$ as a probability of sampling node v_j given a set of nodes $\{v_k\}_{k=i_1}^{i_n}$. Each layer samples k neighbors from their sampling distribution $q(j|i_1, \cdots, i_n)$, then the number of sampled nodes in each layer becomes $O(k)$. FastGCN [23] defines $q(j|i_1, \cdots, i_n)$ proportional to the degree of the target node v_j, thus every layer has independent-identical-distributions. LADIES [202] adopts the same iid as FastGCN but limits the sampling domain to the neighborhood of the sampler layer. AS-GCN [68] parameterizes the sampling distributions $q(j|i_1, i_2, \ldots, i_n)$ with a learnable linear function. While the layer-wise samplers successfully regulate the neighbor expansion, they suffer from sparse connection problems — some nodes fail to sample any neighbors while other nodes sample their

neighbors repeatedly in a given layer.

Learnable Sampling Policy. GraphSage [55], FastGCN [23], and LADIES [202] use the heuristic sampling probability distributions (e.g., proportional to degrees of nodes). GCN-BS [97] and AS-GCN [68] train their sampling distributions towards minimum sampling variance. They compute the optimal sampling probability model with the minimum variance theoretically, then update their sampling models towards the optimal variance.

Our proposed PASS is a learnable node-wise sampler. Table 4.2 compares PASS with existing sampling methods.

4.7 Summary

In this paper, we propose a novel sampling algorithm PASS for graph convolutional networks. Our main contributions are:

- **Performance-adaptive sampler:** PASS samples neighbors informative for the task performance.
- **Effectiveness:** PASS outperforms state-of-the-art samplers, being up to 10.4% more accurate.
- **Robustness:** PASS shows up to 53.1% higher accuracy than the baselines in the presence of adversarial attacks.
- **Theoretical foundation:** PASS explains why a neighbor is considered informative and assigned a high sampling probability.

Future works include learning an edge imputation policy and combining it with our proposed edge sampling policy to improve the overall performance of graph neural networks.

4.8 Appendix

4.8.1 Proof of SUB-LOSS trick

Our model applies the log derivative trick to compute the gradient of an expectation of vectors (matrices for a batch implementation) located in the middle of the neural network. The implementation of the log derivative trick in this context requires hand-crafted backpropagation. Here, we introduce an additional SUB-LOSS trick that allows us to leverage the backpropagation mechanisms built in deep learning frameworks (e.g., PyTorch, TensorFlow).

Theorem 6 (SUB-LOSS trick)**.** *Given $\theta \in \mathbb{R}^{D^{(s)}}$, a hidden embedding $h(\theta) \in \mathbb{R}^{D^{(l)}}$, and a loss $\mathcal{L}(h) \in \mathbb{R}$, the gradient of the loss $\mathcal{L}$ with respect to θ is presented as follows:*

$$\frac{d\mathcal{L}}{d\theta} = \frac{d}{d\theta}(\frac{d\mathcal{L}}{dh} \cdot h)$$

with an assumption $\frac{d}{d\theta}(\frac{d\mathcal{L}}{dh}) = 0$.

Proof. By the chain rule, $\frac{d\mathcal{L}}{d\theta}$ is decomposed as follows:

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{dh} \cdot \frac{dh}{d\theta}$$

where $\frac{d\mathcal{L}}{dh}$ is an $(1 \times D^{(l)})$ matrix and $\frac{dh}{d\theta}$ is a $(D^{(l)} \times D^{(s)})$ matrix. The i-th component of $\frac{d\mathcal{L}}{d\theta}$ is presented as follows:

$$\frac{d\mathcal{L}}{d\theta_i} = \sum_{j=0}^{D^{(l)}} \frac{d\mathcal{L}}{dh_j} \cdot \frac{dh_j}{d\theta_i}$$

The dot product of $\frac{d\mathcal{L}}{dh}$ and h is presented as follows:

$$\frac{d\mathcal{L}}{dh} \cdot h = \frac{d\mathcal{L}}{dh_0} h_0 + \frac{d\mathcal{L}}{dh_1} h_1 + \cdots + \frac{d\mathcal{L}}{dh_{(D^{(l)}-1)}} h_{(D^{(l)}-1)}$$

where $\frac{d\mathcal{L}}{dh} \cdot h$ is a scalar value. With an assumption $\frac{d}{d\theta_i}(\frac{d\mathcal{L}}{dh}) = 0$, the gradient of $\frac{d\mathcal{L}}{dh} \cdot h$ with respect to θ_i is presented as follows:

$$\begin{aligned}
\frac{d}{d\theta_i}(\frac{d\mathcal{L}}{dh} \cdot h) &= \frac{d\mathcal{L}}{dh_0}\frac{dh_0}{d\theta_i} + \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{d\theta_i} + \cdots + \frac{d\mathcal{L}}{dh_{(D^{(l)}-1)}}\frac{dh_{(D^{(l)}-1)}}{d\theta_i} \\
&= \sum_{j=0}^{D^{(l)}} \frac{d\mathcal{L}}{dh_j}\frac{dh_j}{d\theta_i} \\
&= \frac{d\mathcal{L}}{d\theta_i}
\end{aligned}$$

Then $\frac{d\mathcal{L}}{d\theta_i} = \frac{d}{d\theta_i}(\frac{d\mathcal{L}}{dh} \cdot h)$ for every $0 \leq i < D^{(s)}$. This shows $\frac{d\mathcal{L}}{d\theta}$ is equal to $\frac{d}{d\theta}(\frac{d\mathcal{L}}{dh} \cdot h)$. □

With the SUB-LOSS trick, we compute an auxiliary loss $\mathcal{L}_{aux} = d\mathcal{L}/dh_i^{(l)} \cdot h_i^{(l)}$ and simply call the backpropagation function of our deep learning framework on $\mathcal{L}_{aux}$ to compute the gradient w.r.t. θ.

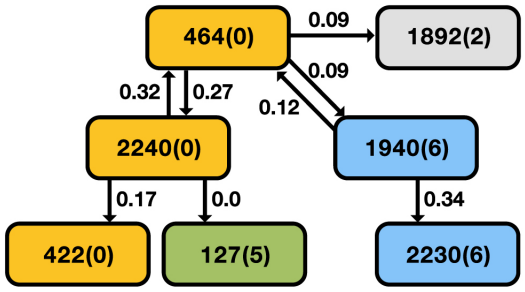
4.8.2 Experimental Setting

Hyper-parameters. We use the Adam optimizer [73] and tune each baseline with a grid search on each dataset. Most baselines perform best on most datasets with a learning rate of 0.01, weight decay of 5×10^{-4}. We report the average performance across 5 runs for each experiment.
Baselines. We refer to the following websites when implementing the baseline models:

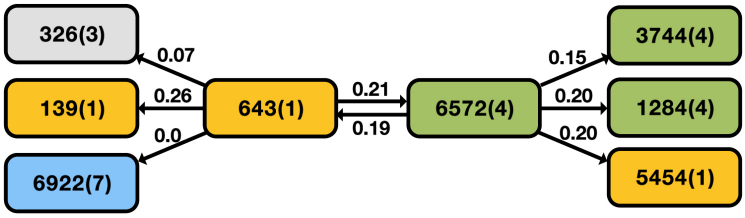
- **FastGCN:** `https://github.com/matenure/FastGCN`
- **AS-GCN:** `https://github.com/huangwb/AS-GCN`
- **GraphSage:** `https://github.com/williamleif/GraphSAGE`
- **GCN-BS:** `https://github.com/xavierzw/ogb-geniepath-bs`
- **GAT:** `https://github.com/PetarV-/GAT`

4.8.3 Case Study

In Figure 4.6(a), we find PASS distinguishes informative neighbors (same labels) from less informative ones (different labels). The node 464 with label 0 has a high sampling probability (0.27)



(a) Nodes and subset of neighbors from the Cora dataset



(b) Nodes and subset of neighbors from the Amazon Photo dataset

Figure 4.6: **PASS learns which neighbors are informative or not.** The numbers in nodes denote node ids and labels. The numbers in edges denote sampling probabilities computed by PASS.

for the node 2240 with label 0 while low probabilities (0.09) for the nodes 1940 and 1892 with different labels. In Figure 4.6(b), the node 643 with label 1 gives a high sampling probability (0.21) to the node 6572 with different label 4. The node 6572 has a high sampling probability (0.20) for the neighbor node 5454 with label 1; thus, the node 6572 contains information of label 1. In the two-layer GCNs, the node 643 aggregates the node 5454 through the node 6572 and supplements its embedding with another label 1 node.

4.8.4 Different sample numbers

In Section 4.5, we sample one neighbor per node for a fair comparison between layer-wise samplers and node-wise samplers. With the batch size set to 64, both the layer-wise and node-wise samplers samples 64 nodes in total for each layer. Under the same time/memory efficiency, the node-wise samplers outperform the layer-wise sampler in accuracy. Here, we compare the performance of the node-wise samplers with larger numbers of samples ($k > 1$).
In Table 4.8, PASS shows higher or similar accuracy with its competitors. The accuracy gap between PASS and its competitors is smaller than when the sampling number is 1. The large sample

Table 4.8: **Comparison between node-wise samplers with large numbers of samples.**

Dataset	#sample	GraphSage	GCN-BS	PASS
Cora	3	0.845	0.840	0.844
Citeseer	3	0.740	0.708	0.735
Pubmed	3	0.839	0.877	0.874
AmazonC	5	0.844	0.880	0.889
AmazonC	10	0.862	0.898	0.885
AmazonP	5	0.900	0.919	0.942
AmazonP	10	0.923	0.937	0.945
MsCS	3	0.862	0.909	0.912

number allows the informative neighbors sampled at some point by GraphSage and GCN-BS. Thus the accuracy of GraphSage and GCN-BS could catch up with our accuracy, not surpass ours. In addition, the accuracy is saturated around 0.88 and 0.94 from the sampling number 5 on the AmazonC and AmazonP datasets, respectively. This shows the number of informative neighbors is under 5. Thus sampling neighbors more than 5 does not bring further increase in accuracy. These results show that our experimental setting with one sample per node is more effective at comparing the performance of the sampling algorithms.

Chapter 5

Privacy I: transfer learning within a heterogeneous graph

Data continuously emitted from industrial ecosystems such as social or e-commerce platforms are commonly represented as heterogeneous graphs (HG) composed of multiple node/edge types. State-of-the-art graph learning methods for HGs known as heterogeneous graph neural networks (HGNNs) are applied to learn deep context-informed node representations. However, many HG datasets from industrial applications suffer from label imbalance between node types. As there is no direct way to learn using labels rooted at different node types, HGNNs have been applied on only a few node types with abundant labels. We propose a zero-shot transfer learning module for HGNNs called a Knowledge Transfer Network (KTN) that transfers knowledge from *label-abundant* node types to *zero-labeled* node types through rich relational information given in the HG. KTN is derived from the theoretical relationship, which we introduce in this work, between distinct *feature extractors* for each node types given in a HGNN model. KTN improves performance of 6 different types of HGNN models by up to 960% for inference on zero-labeled node types and outperforms state-of-the-art transfer learning baselines by up to 73% across 18 different transfer learning tasks on HGs.

5.1 Motivation

Large technology companies commonly maintain large relational datasets, derived from their internal logs, that can be represented as or joined into a massive heterogeneous graph (HG) composed of nodes and edges with multiple types [141]. For instance, in e-commerce networks, there are product, user, and review nodes, all interconnected by many edge types that represent forms of interactions such as spending (user-product), reviewing (user-review), and reviews-of (product-review). To learn powerful features representing the complex multimodal structure of HGs, various heterogeneous graph neural networks (HGNN) have been proposed [65, 125, 158, 191].

A common issue in these industrial applications of HGNNs is the label imbalance among different node types. For instance, publicly available *content* nodes – such as those representing video, text, and image content – are abundantly labelled, whereas labels for other types (such as *user* or *account* nodes) may be much more expensive to collect (or even not available, e.g. due to privacy restrictions). This means that in most standard training settings, HGNN models can only learn to make good inferences for a few label-abundant node types, and can usually not make any

inferences for the remaining node types, given the absence of any labels for them.

If there is a pair of *label-abundant* and *zero-labeled* node types which share an inference task, could we transfer knowledge between them? One body of work has focused on transferring knowledge between nodes of the *same* type from two *different* HGs (i.e., graph-to-graph transfer learning) [67, 171]. However, these approaches are not applicable in many real-world scenarios for three reasons. First, any external large-scale HG that could be used in a graph-to-graph transfer learning setting would almost surely be proprietary. Second, even if practitioners could obtain access to an external industrial HG, it is unlikely the distribution of that (source) graph would match their target graph well enough to apply transfer learning. Finally, node types suffering label scarcity are likely to suffer the same issue on other HGs (e.g. user nodes).

In this paper, we introduce a zero-shot transfer learning approach for a *single* HG (assumed to be fully-owned by the practitioners), transferring knowledge from labelled to unlabelled node types. This setting is distinct from any graph-to-graph transfer learning scenarios, since the source and target domains exist in the same HG dataset, and are assumed to have different node types. Our model utilizes the shared context between source and target node types; for instance, in the e-commerce network, the latent (unknown) labels of user nodes can be strongly correlated with spending/reviewing patterns that are encoded in the cross-edges between user nodes and product/review nodes. We propose a novel zero-shot transfer learning problem for this HG learning setting as follows:

Informal Problem Definition 1. Zero-shot cross-type transfer learning running on a HG: *Given a heterogeneous graph $\mathcal{G}$ with node types $\{\mathbf{s}, \mathbf{t}, \cdots\}$ with abundant labels for source type $\mathbf{s}$ but no labels for target type $\mathbf{t}$, can we train HGNNs to infer the labels of target-type nodes?*

A naïve solution to this problem would be to re-use an HGNN pre-trained on the source nodes for target node inference, given that both domains exist in the same HG. However, as we show in our paper, HGNNs have distinct parameter sets for each node type [65], edge type [125], and meta-path type [46, 158]. These facts cause HGNNs to learn entirely different *feature extractors* for nodes and edges of different types – in other words, the final embeddings for source and target nodes are computed by different sets of parameters in HGNNs. Thus, a classifier pre-trained on source nodes will fail to perform well on inference tasks for target nodes. The field of domain adaptation (DA) targets this setting, seeking to transfer knowledge from a source domain with abundant labels to a target domain which lacks them [47, 99, 100, 128]. However, distinct feature extractors across node types in HGNNs break a standard assumption of DA setting, namely that source and target domains share the same feature extractors (e.g., CNNs for both source and target image domains). As we demonstrate in this paper, in our problem setting, DA approaches fail to achieve the outstanding performance they are known for in computer vision and NLP.

In our work, we first dissect the gradient path of HGNN models to see how feature extractors are designed independently for each node type, and some empirical consequences. Then we theoretically analyze how feature extractors across node types relate to each other and how their output distributions could be represented in terms of each other. We model this theoretical relationship between two feature extractors as a Knowledge Transfer Network (KTN) which can be optimized to transform target embeddings to fit the source domain distribution. We perform an extensive evaluation of our method on 18 different transfer learning tasks on HGs where we

compare against state-of-the-art domain adaptation baselines. Additionally, in order to understand which environments are ideal for transferring knowledge between different node types for HGs, we formulate a synthetic heterogeneous graph generator that allows us to study the sensitivity of these methods.

Our main contributions are:

- **Novel and practical problem definition:** To the best of our knowledge, KTN is the first zero-shot cross-type transfer learning method running on a heterogeneous graph — transfer knowledge across different node types within a heterogeneous graph.
- **Generality:** KTN is a principled approach analytically induced from the architecture of HGNNs, thus applicable to any HGNN models, showing up to 960% performance improvement for zero-labeled node inference across 6 different HGNN models.
- **Effectiveness:** We show that KTN outperforms state-of-the-art domain adaptation methods, being up to 73.3% higher in MRR on 18 different transfer learning tasks on HGs.
- **Sensitivity Analysis:** We provide a HG generator model to analyze how the node attribute and edge distributions of HGs affect the performance of KTN and other methods on the task.

5.2 Preliminaries

In this section we review heterogeneous graphs and heterogeneous graph neural networks (HGNNs).

5.2.1 Heterogeneous graph

Heterogeneous graphs (HGs) are an important abstraction for modeling the relational data of multi-modal systems. Formally, a heterogeneous graph is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$ where the node set $\mathcal{V}$; the edge set $\mathcal{E}$ consisting of ordered tuples $e_{ij} := (i, j)$ with $i, j \in \mathcal{V}$, where $e_{ij} \in \mathcal{E}$ iff an edge exists from i to j; the set of node types $\mathcal{T}$ with associated map $\tau : \mathcal{V} \mapsto \mathcal{T}$; the set of relation types $\mathcal{R}$ with associated map $\phi : \mathcal{E} \mapsto \mathcal{R}$. This flexible formulation allows directed, multi-type edges. We additionally assume the existence of a node attribute vector $x_i \in \mathcal{X}_{\tau(i)}$ for each $i \in \mathcal{V}$, where $\mathcal{X}_t$ is an attribute matrix specific to nodes of type t.

5.2.2 Heterogeneous Graph Neural Networks (HGNN)

Various HGNN models have been proposed [65, 125, 158, 172, 191]. Fully-specified HGNN models have specialized parameters for each node type [65], edge type [125], and meta-path type [46] to most effectively utilize the complex relationships encoded in the HG data structure. In this paper, we use a flavor of HGNN known as a Heterogeneous Message-Passing Neural Network (HMPNN) as our base model on which to demonstrate KTN (though KTN can be implemented in almost any HGNN, as we show in experiments in Section 5.5). The HMPNN merely extends the standard MPNN [49] by specializing all transformation and message matrices in each node/edge type. With its generality, HMPNN is itself a base model for RGCN [125] and HGT [65], and is also widely used as a default HGNN model in popular GNN libraries (e.g., pyG [41], TF-GNN [40], DGL [157]).

In a HMPNN, for any node j, the embedding of node j at the *l-th* layer is obtained with the following generic formulation:

$$h_j^{(l)} = \textbf{Transform}^{(l)}\left(\textbf{Aggregate}^{(l)}(\mathcal{E}(j))\right) \tag{5.1}$$

where $\mathcal{E}(j) = \{(i,j) \in \mathcal{E} : i,j \in \mathcal{V}\}$ denotes all the edges which connect (directionally) to j. The above operations typically involve type-specific parameters to exploit the inherent multiplicity of modalities in heterogeneous graphs. First, we define a linear **Message** function:

$$\textbf{Message}^{(l)}(i,j) = M_{\phi((i,j))}^{(l)} \cdot \left(h_i^{(l-1)} \parallel h_j^{(l-1)}\right) \tag{5.2}$$

where $M_r^{(l)}$ are the specific message passing parameters for each edge type $r \in \mathcal{R}$ and each of L HMPNN layers. Then defining $\mathcal{E}_r(j)$ as the set of edges of type r pointing to node j, the **Aggregate** function mean-pools messages by edge type, and concatenates:

$$\textbf{Aggregate}^{(l)}(\mathcal{E}(j)) = \mathop{\Vert}_{r\in\mathcal{R}} \frac{1}{|\mathcal{E}_r(j)|} \sum_{e\in\mathcal{E}_r(j)} \textbf{Message}^{(l)}(e) \tag{5.3}$$

Finally, the **Transform** function maps the message into a type-specific latent space:

$$\textbf{Transform}^{(l)}(j) = \alpha(W_{\tau(j)}^{(l)} \cdot \textbf{Aggregate}^{(l)}(\mathcal{E}(j))) \tag{5.4}$$

where $W_t^{(l)}$ are the specific transformation parameters for each node type $t \in \mathcal{T}$ and each of L HMPNN layers. By stacking L layers, HMPNN outputs highly contextualized final node representations, and the final node representations can be fed into another model to perform downstream heterogeneous network tasks, such as node classification or link prediction.

5.2.3 Problem definition

Using notations defined above, we formalize our novel transfer learning problem on HGs.

Problem 1. Zero-shot cross-type transfer learning running on a HG:
In a given heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$ with node attributes $\mathcal{X} = \cup_{t\in\mathcal{T}}\mathcal{X}_t$, assume node types $\mathtt{s}$ *and* $\mathtt{t}$ *share a classification task $\{(i, y_i) : i \in \mathcal{V}_s, \mathcal{V}_t\}$. During the training phase, using labels $\{(i, y_i) : i \in \mathcal{V}_s\}$ only for source-type nodes, we train an HGNN model $\boldsymbol{f} : \boldsymbol{f}(\mathcal{G}, \mathcal{X}) = h_i$ to get node embeddings h_i for all nodes $i \in \mathcal{V}$ and a classifier $\boldsymbol{g} : \boldsymbol{g}(h_i) = \hat{y}_i$ to predict labels $\hat{y}_i$ from the node embeddings h_i. During the test phase, our task is to predict labels $\{(j, y_j) : j \in \mathcal{V}_t\}$ of target-type nodes where none of labels of target-type nodes were used for training.*

5.3 Cross-Type Feature Extractor Transformations in HGNNs

We define $f_t : \mathcal{G} \mapsto \mathbb{R}^d$ to be the "feature extractor" of a HGNN, which is composed of parameters participating to map input node attributes of type t into a shared feature space $\mathbb{R}^d$. In this section, we derive a strict transformation between feature extractors within a HMPNN. Specifically, for any

two nodes i, j with types $\tau(i) = s$ and $\tau(j) = t$, we derive an expression for f_s in terms of f_t, and use that expression to inspire a trainable transfer learning module called KTN in the following section. For simplicity, throughout this section we ignore the activation $\alpha(\cdot)$ within the **Transform** function in Equation (5.4).

5.3.1 Feature extractors in HMPNNs

We first reason intuitively about the differences between f_s and f_t when $s \neq t$, using a toy heterogeneous graph shown in Figure 5.1(a). Consider nodes v_1 and v_2, noticing that $\tau(1) \neq \tau(2)$. Using HMPNN's equations (5.2)-(5.4) from Section 5.2.2, for any $l \in \{0, \ldots, L-1\}$ we have

$$h_1^{(l)} = W_s^{(l)} \left[M_{ss}^{(l)} \left(h_3^{(l-1)} \parallel h_1^{(l-1)} \right) \parallel M_{ts}^{(l)} \left(h_2^{(l-1)} \parallel h_1^{(l-1)} \right) \right] \tag{5.5}$$

$$h_2^{(l)} = W_t^{(l)} \left[M_{st}^{(l)} \left(h_1^{(l-1)} \parallel h_2^{(l-1)} \right) \parallel M_{tt}^{(l)} \left(h_4^{(l-1)} \parallel h_2^{(l-1)} \right) \right] \tag{5.6}$$

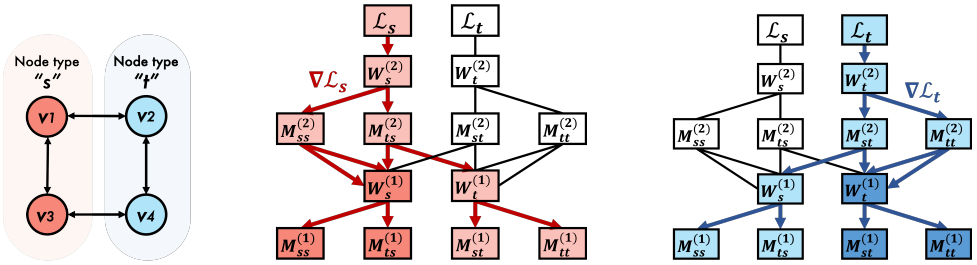
where $h_j^{(0)} = x_j$. From these equations, we see that $h_1^{(l)}$ and $h_2^{(l)}$, which are features of different types, are extracted using *disjoint* sets of model parameters at l-th layer. In a 2-layer HMPNN, this creates unique gradient backpropagation paths between the two node types, as illustrated in Figures 5.1(b)-5.1(c). In other words, even though the same HMPNN is applied to node types s and t, the feature extractors f_s and f_t have different computational paths. Therefore they project node features into different latent spaces, and have different update equations during training.

5.3.2 Empirical gap between f_s and f_t

Here we study the experimental consequences of the above observation via simulation. We first construct a synthetic graph extending the 2-type graph in Figure 5.1(a) to have multiple nodes per-type, and multiple classes. To maximize the effects of having different feature extractors, we sample source and target nodes from the same feature distributions and each classes are well-separated in the both the graph and feature space (more details available in Appendix 5.8.7).

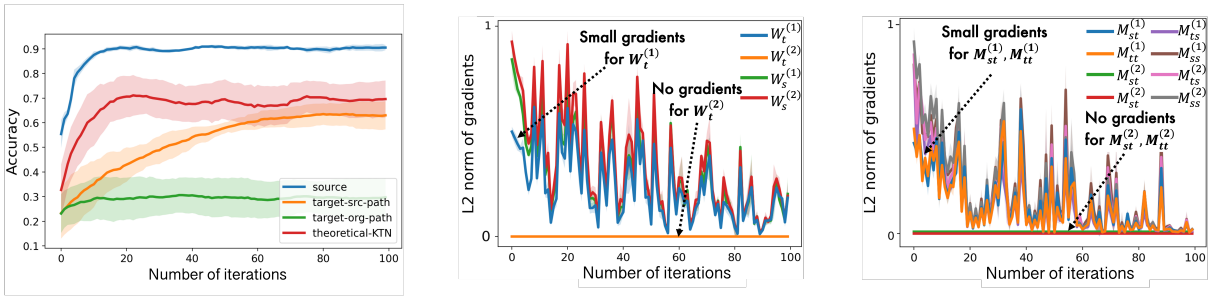
On such a well-aligned heterogeneous graph, without considering the observation in Section 5.3.1, there may seem to be no need for domain adaptation from f_t to f_s. However, when we train the HMPNN model solely on s-type nodes, as shown in Figure 5.2(a) we find the test accuracy for s-type nodes to be high (90%, blue line) and the test accuracy for t-type nodes to be quite low (25%, green line). Now if instead we make the t-type nodes use the source feature extractor f_s, much more transfer learning is possible ($\sim 65\%$, orange line). This shows that the different feature extractors present in the HMPNN model result in the significant performance drop, and simply matching input data distributions can not solve the problem.

To analyze this phenomenon at the level of backpropagation, in Figures 5.2(b)-5.2(c) we show the magnitude of gradients passed to parameters of source and target node types. As illustrated in Figures 5.1(b)-5.1(c), we find that the final-layer **Transform** parameter $W_t^{(2)}$ for type-t nodes have zero gradients (Figure 5.2(b)), and similarly for the final-layer **Message** parameters (Figure 5.2(c)). Additionally, those same parameters in the first-layer for t-type nodes have much smaller gradients than their s-type counterparts: $W_t^{(1)}$ (blue line in Figure 5.2(b)), $M_{st}^{(1)}$ and $M_{tt}^{(1)}$ (blue and orange



(a) Toy graph (b) Gradient path for feature extractor f_s (c) Gradient path for feature extractor f_t

Figure 5.1: **Illustration of a toy heterogeneous graph and the gradient paths for feature extractors f_s and f_t.** Colored arrows in figures (b) and (c) show that the same HGNN nonetheless produces different gradient paths for each feature extractor. Color density of each box in (b) and (c) is proportional to the degree of participation of the corresponding parameter in each feature extractor.



(a) Test accuracy across various feature extractors (b) L2 norms of gradients of $W_{\tau(\cdot)}$ (c) L2 norms of gradients of $M_{\phi(\cdot)}$

Figure 5.2: **HGNNs trained on a source domain underfit a target domain even on a "nice" heterogeneous graph.** (a) Performance on the simulated heterogeneous graph for 4 kinds of feature extractors; (*source*: source extractor f_s on source domain, *target-src-path*: source extractor f_s on target domain, *target-org-path*: target extractor f_t on target domain, and *theoretical-KTN*: target extractor f_t on target domain using KTN.) (b-c) L2 norms of gradients of parameters $W_{\tau(\cdot)}$ and $M_{\phi(\cdot)}$ in HGNN models.

lines in Figure 5.2(c)) appear below than other lines. This is because they contribute to f_s less than f_t

This case study shows that even when an HGNN is trained on a relatively simple, balanced, and class-separated heterogeneous graph, a model trained only on the source domain node type cannot transfer to the target domain node type.

5.3.3 Relationship between feature extractors in HMPNNs

We show that a HMPNN model provides different feature extractors for each node type. However, still, f_s and f_t are built inside one HMPNN model and interchange intermediate feature embeddings with each other. Both $H_s^{(L)}$ and $H_t^{(L)}$ (the output of f_s and f_t) are computed using the previous layer's intermediate embeddings $H_s^{(L-1)}$, $H_t^{(L-1)}$, and any other connected node type embeddings $H_x^{(L-1)}$ at the L-th HMPNN layer. Therefore $H_s^{(L)}$ and $H_t^{(L)}$ can be mathematically presented by

each other using the $(L-1)$-th layer embeddings as connecting points, so do f_s and f_t. Based on this intuition, we derive a strict transformation between f_s and f_t, which will motivate the core domain adaptation component of our proposed KTN model.

Theorem 7. *Given a heterogeneous graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R}\}$. For any layer $l > 0$, define the set of $(l-1)$-th layer HMPNN parameters as*

$$\mathcal{Q}^{(l-1)} = \{M_r^{(l-1)} : r \in \mathcal{R}\} \cup \{W_t^{(l-1)} : t \in \mathcal{T}\}. \tag{5.7}$$

Let A be the total $n \times n$ adjacency matrix. Then for any $s, t \in \mathcal{T}$ there exist matrices $A_{ts}^* = a_{ts}(A)$ and $Q_{ts}^* = q_{ts}(\mathcal{Q}^{(l-1)})$ such that

$$H_s^{(l)} = A_{ts}^* H_t^{(l)} Q_{ts}^* \tag{5.8}$$

where $a_{ts}(\cdot)$ and $q_{ts}(\cdot)$ are matrix functions that depend only on s, t.

The full proof of Theorem 1 can be found in Appendix 5.8.1. Notice that in Equation 5.8, Q_{ts}^* acts as a macro-**Transform** operator that maps $H_t^{(L)}$ into the source domain, then A_{ts}^* aggregates the mapped embeddings into s-type nodes. In other words, Q_{ts}^* acts as a mapping matrix from the target domain to the source domain. To examine the implications, we run the same experiment as described in Section 5.3.2, while this time mapping the target features $H_t^{(L)}$ into the source domain by multiplying with Q_{ts}^* in Equation 5.8 before passing over to a task classifier. We see via the red line in Figure 5.2(a) that, with this mapping, the accuracy in the target domain becomes much closer to the accuracy in the source domain ($\sim 70\%$). Thus, we use this theoretical transformation as a foundation for our trainable HGNN domain adaptation module, introduced in the following section.

5.3.4 Generalized cross-type transformations for HGNNs

In this section we showed that a HMPNN feature extractor on the (label-abundant) source node type can be expressed in terms of the (label-scarce) target node type feature extractor, and this transformation enables cross-type zero-shot learning for the target node type. As most HGNNs have distinct feature extractors for each node types (even single-layer HGNNs, which have specialized parameters for each node/edge attribute projection layer), they will suffer from the under-trained target embeddings phenomena we showed in Section 5.3.2. For instance, in the meta-path based MAGNN model [46], meta-paths directing toward the target node types are generally less engaged in the source node feature computation and thus receive smaller gradients. While we cannot derive the exact cross-type transformation for all possible HGNNs, the core intuition in the HMPNN theory holds, namely that $H_s^{(L)}$ and $H_t^{(L)}$ are both computed using the previous layer's intermediate embeddings (see Section 5.3.3) across all HGNN models. This observation allows us to extend our KTN and apply it to almost any HGNN. We illustrate this by applying KTN to 6 different HGNN models in Section 5.5, where we see greatly increased target-type accuracy.

5.4 KTN: Trainable Cross-Type Transfer Learning for HGNNs

Algorithm 5: Training step on a source domain

Require: heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$, node feature matrices $\mathcal{X}$, source node type s, target node type t, adjacency matrix A_{ts}, source node label matrix $\mathcal{Y}_s$.

Ensure: HGNN $\mathbf{f}$, classifier $\mathbf{g}$, KTN $\mathbf{t}_{\text{KTN}}$

1: $H_s^{(L)}, H_t^{(L)} = \mathbf{f}(\mathcal{G}, H^{(0)} = \mathcal{X})$
2: $H_t^* = \mathbf{t}_{KTN}(H_t^{(L)}) = A_{ts} H_t^{(L)} T_{ts}$
3: $\mathcal{L}_{\text{KTN}} = \left\| H_s^{(L)} - H_t^* \right\|_2$
4: $\mathcal{L} = \mathcal{L}_{\text{CL}}(\mathbf{g}(H_s^{(L)}), \mathcal{Y}_s) + \lambda \mathcal{L}_{\text{KTN}}$
5: Update $\mathbf{f}$, $\mathbf{g}$, $\mathbf{t}$ using $\nabla \mathcal{L}$

Algorithm 6: Test step on a target domain

Require: pretrained HGNN $\mathbf{f}$, classifier $\mathbf{g}$, KTN $\mathbf{t}_{\text{KTN}}$

Ensure: target node label matrix $\mathcal{Y}_t$

1: $H_t^{(L)} = \mathbf{f}(\mathcal{G}, H^{(0)} = \mathcal{X})$
2: $H_t^* = H_t^{(L)} T_{ts}$
3: **return** $\mathbf{g}(H_t^*)$

Inspired by these derivations we introduce our primary contribution, *Knowledge Transfer Networks*. We begin by noting Equation 5.8 in Theorem 7 has a similar form to a single-layer graph convolutional network [75] with a deterministic transformation matrix (Q_{ts}^*) and a combination of adjacency matrices directing from target node type t to source node type s (A_{ts}^*). Instead of hand-computing the mapping function Q_{ts}^* for arbitrary HGs and HGNNs (which would be intractable), we *learn* the mapping function by modelling Equation 5.8 as a trainable graph convolutional network, named the Knowledge Transfer Network, $\mathbf{t}_{\text{KTN}}(\cdot)$. KTN replaces Q_{ts}^* and A_{ts}^* in Equation 5.8 as follows:

$$\mathbf{t}_{\text{KTN}}(H_t^{(L)}) = A_{ts} H_t^{(L)} T_{ts} \tag{5.9}$$

$$\mathcal{L}_{\text{KTN}} = \left\| H_s^{(L)} - \mathbf{t}_{\text{KTN}}(H_t^{(L)}) \right\|_2 \tag{5.10}$$

where A_{ts} is an adjacency matrix from node type t to s, and T_{ts} is a trainable transformation matrix. By minimizing $\mathcal{L}_{\text{KTN}}$, T_{ts} is optimized to a mapping function of the target domain into the source domain.

5.4.1 Algorithm

We minimize a classification loss $\mathcal{L}_{\text{CL}}$ and a transfer loss $\mathcal{L}_{\text{KTN}}$ jointly with regard to a HGNN model $\mathbf{f}$, a classifier $\mathbf{g}$, and a knowledge transfer network $\mathbf{t}_{\text{KTN}}$ as follows:

$$\min_{\mathbf{f},\, \mathbf{g},\, \mathbf{t}_{\text{KTN}}} \mathcal{L}_{\text{CL}}(\mathbf{g}(\mathbf{f}(\mathcal{G}, \mathcal{X})_s), \mathcal{Y}_s) + \lambda \left\| \mathbf{f}(\mathcal{G}, \mathcal{X})_s - \mathbf{t}_{\text{KTN}}(\mathbf{f}(\mathcal{G}, \mathcal{X})_t) \right\|_2$$

where λ is a hyperparameter regulating the effect of $\mathcal{L}_{\text{KTN}}$; and $\mathbf{f}(\mathcal{G}, \mathcal{X})_s$ and $\mathbf{f}(\mathcal{G}, \mathcal{X})_t$ denote $H_s^{(L)}$ and $H_t^{(L)}$, respectively. Algorithm 5 describes a training step on the source domain. After computing the node embeddings $H_s^{(L)}$ and $H_t^{(L)}$, we map $H_t^{(L)}$ to the source domain using $\mathbf{t}_{\text{KTN}}$ and compute $\mathcal{L}_{\text{KTN}}$. Then, we update the models using gradients of $\mathcal{L}_{\text{CL}}$ (computed using only source labels) and $\mathcal{L}_{\text{KTN}}$. Algorithm 6 describes the test phase on the target domain. After we get node embeddings $H_t^{(L)}$ from the trained HGNN model, we map $H_t^{(L)}$ into the source domain using the trained transformation matrix T_{ts}. Finally we pass the transformed target embeddings H_t^* into the classifier which was trained on the source domain.

Indirect Connections We note that in practice, the source and target node types can be indirectly connected in HGs via other node types (i.e., there is no A_{ts}). Appendix 5.8.2 describes how we can easily extend KTN to cover domain adaption scenarios in this case.

5.5 Experiments

5.5.1 Datasets

Open Academic Graph (OAG). A dataset introduced in [192] composed of five types of nodes: papers (P), authors (A), institutions (I), venues (V), fields (F) and their corresponding relationships. Paper and author nodes have text-based attributes, while institution, venue, and field nodes have text- and graph structure-based attributes. Paper, author, and venue nodes are labeled with research fields in two hierarchical levels, L1 and L2. We construct three field-specific subgraphs from OAG: computer science, computer networks, and machine learning academic graphs.

PubMed.[170] A network composed of of four types of nodes: genes (G), diseases (D), chemicals (C), and species (S), and their corresponding relationships. Gene and chemical nodes have graph structure-based attributes, while disease and species nodes have text-based attributes. Each gene and disease is labeled with a set of diseases they belong to or cause.

Synthetic heterogeneous graphs. We generate stochastic block models [1] with multiple node/edge types. We label each node types with the same set of classes. Then we control feature/edge distributions within/between node types by manipulating feature/edge signal-to-noise ratio within/between classes. A complete definition of the generative model is given in Appendix 5.8.7.

5.5.2 Baselines

We compare KTN with two MMD-based DA methods (DAN [98], JAN [100]), three adversarial DA methods (DANN [47], CDAN [99], CDAN-E [99]), one optimal transport-based method (WDGRL [128]), and two traditional graph mining methods (LP and EP [200]). For KTN and DA methods, we use HMPNN as their base HGNN model. More information of each method is in Appendix 5.8.9.

Table 5.1: **KTN on Open Academic Graph on Computer Science field**. The *gain* column shows the relative gain of our method over using no domain adaptation (*Base* column). *o.o.m* denotes *out-of-memory* errors.

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain)
P-A (L1)	**NDCG**	0.399	0.452	0.405	0.292	0.262	0.261	0.260	0.178	0.425	**0.623 (56%)**
	MRR	0.297	0.361	0.314	0.179	0.129	0.111	0.138	0.041	0.363	**0.629 (112%)**
A-P (L1)	**NDCG**	0.401	0.566	0.598	0.294	0.364	0.246	0.195	0.153	0.557	**0.733 (83%)**
	MRR	0.318	0.508	0.544	0.229	0.270	0.090	0.047	0.022	0.507	**0.711 (123%)**
A-V (L1)	**NDCG**	0.459	0.457	0.470	0.382	0.346	0.359	0.403	0.207	0.461	**0.671 (46%)**
	MRR	0.364	0.413	0.458	0.341	0.205	0.253	0.327	0.011	0.389	**0.698 (92%)**
V-A (L1)	**NDCG**	0.283	0.443	0.435	0.242	0.372	0.418	0.272	0.153	0.154	**0.584 (107%)**
	MRR	0.133	0.365	0.345	0.094	0.241	0.444	0.144	0.006	0.006	**0.586 (340%)**
P-A (L2)	**NDCG**	0.229	0.230	o.o.m	0.239	o.o.m	o.o.m	0.168	o.o.m	0.215	**0.282 (23%)**
	MRR	0.121	0.118	o.o.m	0.140	o.o.m	o.o.m	0.020	o.o.m	0.143	**0.2248 (86%)**
A-P (L2)	**NDCG**	0.197	0.162	o.o.m	0.204	0.158	0.161	0.132	o.o.m	0.208	**0.287 (46%)**
	MRR	0.095	0.052	o.o.m	0.106	0.032	0.045	0.017	o.o.m	0.132	**0.242 (155%)**
A-V (L2)	**NDCG**	0.347	0.329	0.295	0.325	0.288	0.273	0.289	o.o.m	0.297	**0.402 (16%)**
	MRR	0.310	0.296	0.198	0.223	0.128	0.097	0.110	o.o.m	0.227	**0.399 (29%)**
V-A (L2)	**NDCG**	0.235	0.249	0.251	0.214	0.197	0.205	0.217	o.o.m	0.119	**0.252 (7%)**
	MRR	0.129	0.157	0.161	0.090	0.044	0.068	0.085	o.o.m	0.000	**0.166 (28%)**

5.5.3 Zero-shot transfer learning

We run 18 different zero-shot transfer learning tasks across three OAG and PubMed graphs. We run each experiment 3 times and report the average value. Due to the space limitation, we report the standard deviations and results on OAG-computer networks and OAG-machine learning in Appendix 5.8.3. Each heterogeneous graph has the same node classification task for both source and target node types. During training, we are given 1) the heterogeneous graph structure information (i.e., adjacency matrices), 2) input node attribute matrices for all node types, and 3) labels on source-type nodes for the classification task. During the test phase, we predict labels on target-type nodes for the same classification task. The performance is evaluated by NDCG and MRR — widely adopted ranking metrics [64, 65].

In Tables 5.1 and 5.2, our proposed method KTN consistently outperforms all baselines on all tasks and graphs by up to 73.3% higher in MRR (P-A(L1) task in OAG-CS, Table 5.1). When we compare with the base accuracy using the model pretrained on the source domain without any domain adaptation (3rd column, *Base*), the results are even more impressive. We see our method KTN provides relative gains of up to 340% higher MRR without using any labels from the target domain. These results show the clear effectiveness of KTN on zero-shot transfer learning tasks on a heterogeneous graph. We mention that venue and author node types are not directly connected in the OAG graphs (Figure 5.5(b) in Appendix), but KTN successfully transfer knowledge by passing intermediate node types.

Baseline Performance. Among baselines, MMD-based models (DAN and JAN) outperform adversarial-based methods (DANN, CDAN, and CDAN-E) and optimal transport-based method (WDGRL), unlike results reported in [99, 128]. These reversed results are a consequence of HGNN's unique feature extractors for each domains. Adversarial- and optimal transport-based methods define separate losses for source and target feature extractors (which are not separated in their shared feature extractor assumption), resulting in divergent gradients between different feature extractors and poor domain adaption performance. This shows again the importance of considering different feature extractors in HGNNs. More analysis can be found in Appendix 5.8.4.

5.5.4 Generality of KTN

Here, we use 6 different HGNN models, R-GCN [125], HAN [158], HGT [65], MAGNN [46], MPNN [49], and HMPNN. MPNN maps all node types to the shared embedding space using projection matrices at the beginning then applies MPNN layers designed for homogeneous graphs. In Table 5.3, KTN improves accuracy on the target nodes across all HGNN models by up to 960%. This shows the strong generality of KTN. More results and analysis can be found in Appendix 5.8.5.

5.5.5 Sensitivity analysis

Using our synthetic heterogeneous graph generator, we generate non-trivial 2-type heterogeneous graphs to examine how the feature and edge distributions affect the performance of KTN and other baselines. We generate a *range* of test-case scenarios by manipulating (1) signal-to-noise ratio σ_e of within-class edge distributions and (2) signal-to-noise ratio σ_f of within-class feature distributions

Table 5.2: **KTN on PubMed graph**. The *gain* column shows the relative gain over using *Base* column.

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain)
D-G	**NDCG**	0.587	0.629	0.615	0.614	0.624	0.646	0.604	0.601	0.571	**0.700** (**19**%)
	MRR	0.372	0.425	0.414	0.397	0.428	0.443	0.388	0.389	0.336	**0.499** (**34**%)
G-D	**NDCG**	0.596	0.599	0.577	0.599	0.581	0.606	0.578	0.576	0.580	**0.662** (**11**%)
	MRR	0.354	0.362	0.332	0.356	0.337	0.362	0.340	0.351	0.353	**0.445** (**26**%)

Table 5.3: **KTN on different HGNN models.** The *Source* column shows accuracy on for source node types. *Base* and *KTN* columns show accuracy for target node types without/with using KTN, respectively. The *Gain* column shows the relative gain of our method over using no domain adaptation.

		P-A (L1)				A-P (L1)			
HGNN type	**Metric**	**Source**	**Base**	**KTN**	**Gain**	**Source**	**Base**	**KTN**	**Gain**
R-GCN	**NDCG**	0.765	0.337	0.577	**71.12%**	0.648	0.388	0.647	**66.82%**
	MRR	0.757	0.236	0.587	**148.73%**	0.623	0.270	0.611	**126.18%**
HAN	**NDCG**	0.476	0.179	0.520	**190.56%**	0.515	0.182	0.512	**181.33%**
	MRR	0.416	0.047	0.497	**960.55%**	0.509	0.055	0.527	**850.90%**
HGT	**NDCG**	0.757	0.294	0.574	**95.07%**	0.670	0.283	0.581	**104.83%**
	MRR	0.749	0.178	0.563	**216.17%**	0.670	0.149	0.565	**279.52%**
MAGNN	**NDCG**	0.657	0.463	0.574	**24.01%**	0.676	0.557	0.622	**11.68%**
	MRR	0.631	0.378	0.556	**47.33%**	0.680	0.509	0.592	**16.14%**
MPNN	**NDCG**	0.602	0.443	0.590	**33.11%**	0.646	0.307	0.621	**101.92%**
	MRR	0.572	0.319	0.575	**80.10%**	0.660	0.145	0.595	**311.42%**
HMPNN	**NDCG**	0.789	0.399	0.623	**56.14%**	0.671	0.401	0.733	**82.88%**
	MRR	0.777	0.297	0.629	**111.86%**	0.661	0.318	0.711	**123.30%**

across all of the (a) source-source ($s \leftrightarrow s$), (b) target-target ($t \leftrightarrow t$), and (c) source-target ($s \leftrightarrow t$) relationships.

For instance, in Figure 5.3, for each edge type ($s \leftrightarrow s$, $t \leftrightarrow t$, and $s \leftrightarrow t$, differentiated by colors), there are two different types of edges, edges within the same class (plain line) and edges across different classes (dotted line). For each edge type, we manipulate σ_e by changing the ratio of within-class and cross-class edges, and σ_f by diverging feature distributions between classes. Thus there will be 6 signal-to-noise ratios in total. A higher signal-to-noise ratio for a particular data dimension (edges or features) across a particular relationship $r \in \{s \leftrightarrow s,\ t \leftrightarrow t,\ s \leftrightarrow t\}$ means that classes are more *separable* in that data dimension, when comparing within r, and hence easier for HGNNs. Note that while tuning one $\sigma_{(\cdot)}$ on the range $[1.0, 10.0]$, the remaining five $\sigma_{(\cdot)}$ are held at 10.0. Additionally, we vary $\sigma_{(\cdot)}$ across two scenarios: (I) "easy": source and target node types have same number of clusters and same feature dimensions, (II) "hard" source and target node types have different number of clusters and feature dimensions. Note that clusters and classes are different concepts in this experiment; several clusters could have the same class label.

Figures 5.4(a) and 5.4(c) show results from changing σ_e across the three relation types. We see that KTN is affected only by σ_e across the $s \leftrightarrow t$ (cross-types) relationship, which accords with our theory, since KTN exploits the between-type adjacency matrix. Surprisingly, as seen in Figures 5.4(b) and 5.4(d), we do not find a similar dependence of KTN on σ_f, which shows that KTN is robust by learning purely from edge homophily in the absence of feature homophily. Regarding the performance of other baselines, EP shows similar tendencies as KTN— only affected by cross-type σ_e — because EP also relies on cross-type propagation along edges. However, its accuracy is bounded above due to the fact that it does not exploit the (unlabelled) target features. DAN and DANN, which do not exploit cross-type edges, are not affected by cross-type σ_e. However,

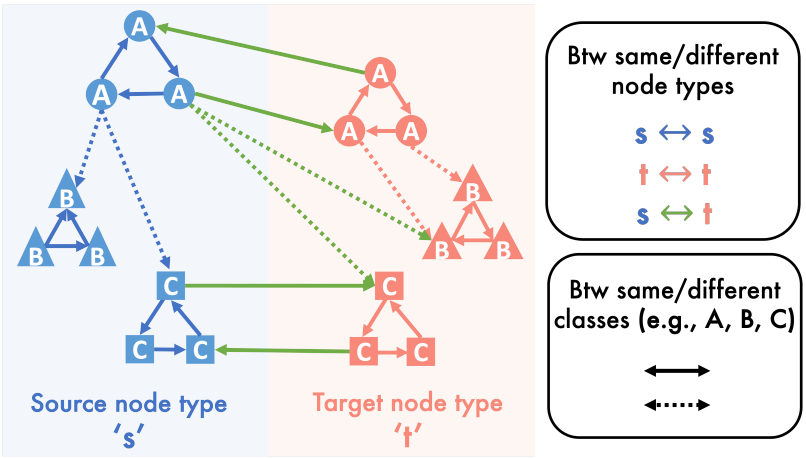


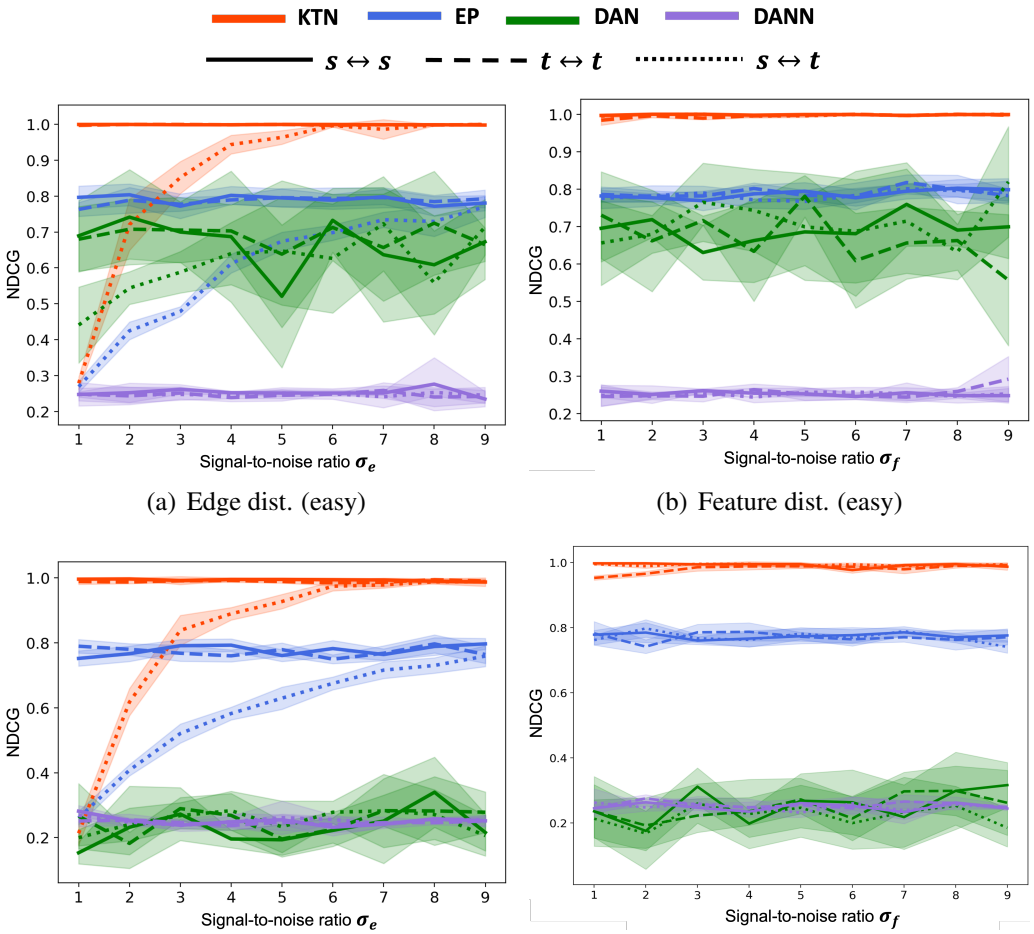
Figure 5.3: **Synthetic HG generator.**

they show either low or unstable performance across different scenarios. DAN shows especially poor performance in the "hard" scenarios (Figure 5.4(c) and 5.4(d)), failing to deal with different feature spaces for source and target domains.

5.6 Related Work

Various transfer learning problems have been defined on the graph domain. [101, 103, 165, 184] construct synthetic graphs from unstructured data and transfer knowledge over the graphs using GNNs. On the other hand, [62, 64, 118, 164] focus on extracting knowledge from the existing graph structures. They pretrain a GNN model on a source graph and re-use the model on a target graph. While these methods focus on homogeneous graphs, [67, 171] transfer HGNNs across different HGs. However, none of them can be directly applied to our cross-type transfer learning problem running on a single HG. Here we cover two classes of learning approaches that are related to our problem. As HGNNs are the models to which our method can be applied, we cover them extensively in Section 5.2.

Zero-shot domain adaptation (DA) transfers knowledge from a source domain with abundant labels to a target domain which lacks them. Zero-shot DA can be categorized into three groups — MMD-based methods, adversarial methods, and optimal-transport-based methods. MMD-based methods [98, 100, 140] minimize the maximum mean discrepancy (MMD) [51] between the mean embeddings of two distributions in reproducing kernel Hilbert space. Adversarial methods [47, 99] are motivated by theory in [8, 9] suggesting that a good cross-domain representation contains no discriminative information about the origin of the input. They learn domain-invariant features by a min-max game between the domain classifier and the feature extractor. Optimal transport-based methods [128] estimate the empirical Wasserstein distance [122] between two domains and minimizes the distance in an adversarial manner. All three categories rely on two networks — a



(a) Edge dist. (easy)

(b) Feature dist. (easy)

(c) Edge dist. (hard)

(d) Feature dist. (hard)

Figure 5.4: **Effects of edge and feature distributions across classes and types in heterogeneous graphs.**

feature extractor network and a task classifier network. Adversarial and OT-based methods use an additional domain classifier network. Due to the assumption that source and target domains have the same modality [1], the standard DA setting assumes identical feature extractors across domains. More descriptions can be found in Appendix 5.8.9.

Label propagation (LP) approaches (e.g., [200]) use message-passing to pass each node's label to its neighbors according to normalized edge weights. LP relies on only a graph's edges, and is therefore easily applied to a heterogeneous graph – labels are simply propagated across edges, regardless of type. In this paper we also evaluate a similarly-simple baseline, embedding propagation (EP). Similar to LP, EP recursively propagates source embeddings (computed using source labels) until they reach the target domain. EP exploits both node attribute information and the node

[1] In our problem, source and target node types could have either (1) different distributions on the same attribute space or (2) entirely different attribute spaces

adjacencies, but only uses the source node embeddings.

5.7 Summary

In this work, we proposed the first cross-type zero-shot transfer learning method for heterogeneous graphs. Our method, Knowledge Transfer Networks (KTN) for Heterogeneous Graph Neural Networks, transfers knowledge from *label-abundant* node types to *label-scarce* node types. We illustrate KTN handily improves HGNN performances up to 960% for zero-labeled node types across 6 different HGNN models and outperforms many challenging baselines up to 73% higher in MRR. Future work in the area includes filtering noisy edges between source and target domains and making KTN more robust and less dependent on structure of given noisy heterogeneous graphs.

Limitation Statement Our transfer learning method is limited to node types sharing the same task (i.e., the same classifier). We plan to extend our work to transfer knowledge between different tasks running on different node types on a heterogeneous graph.

Societal Impact Statement KTN allows organizations to learn better from their own graph data, leveraging its structure without requiring external information. We believe this has a number of positive applications (preserving model quality without needing extra datasets). However like all modeling improvements, its true impact depends on what modeling tasks the technique is applied to.

5.8 Appendix

5.8.1 Proof of Theorem 7

In this proof, we adopt a simplified version of our message-passing function that ignores the skip-connection:

$$\textbf{Message}^{(l)}(i,j) = M^{(l)}_{\phi(i,j)} h_i^{(j)}. \tag{5.11}$$

The HGNN trained in the experimental results shown in Figure 5.2 also does not use skip-connections and hence represents a theoretically-exact KTN component. In the real experiments, we use (1) skip-connections, exploiting their usual benefits [55], and (2) the trainable version of KTN.

Proof. Without loss of generality, we prove the result for the case where $\mathcal{R} = \{(s,t) : s,t \in \mathcal{T}\}$, meaning the type of an edge is identified with the (ordered) types of the neighbor nodes. In other words, there is only one edge modality possible, such as a social networks with multiple node types (e.g. “users", “groups") but only one edge modality (“friendship"). In the case of multiple edge modalities (e.g. “friendship" and “message"), the result is extended trivially (through with more algebraically-dense forms of a_{ts} and q_{ts}).

Throughout this proof, we use the following notation for the set of all j-adjacent edges of relation type r:

$$\mathcal{E}_r(j) := \{(i,j) : i \in \mathcal{V}, (i,j) = r\}. \tag{5.12}$$

We write $A_{x_1x_2}$ to denote the sub-matrix of the total $n_{x_1} \times n_{x_2}$ adjacency matrix A corresponding to node types $x_1, x_2 \in \mathcal{T}$, and $\bar{A}_{x_1x_2}$ to denote the same matrix divided by its column sum. $H_x^{(l)}$ is the (row-wise) $n_x \times d_l$ embedding matrix of x-type nodes at layer l.

We first compute the *l-th* output $g_j^{(l)}$ of the **Aggregate** step defined for HGNNs in Equation 5.3, for any node $j \in \mathcal{V}$ such that $\tau(j) = s$. The output of **Aggregate** is a concatenation of edge-type-specific aggregations (see Equation 5.3). Note that at most $T = |\mathcal{T}|$ elements of this concatenation are non-zero, since the node j only participates in T out of T^2 relation types in $\mathcal{R}$. Thus we can write $g_j^{(l)}$ as

$$\begin{aligned}
g_j^{(l)} &= \Big\|_{r\in\mathcal{R}} \tfrac{1}{|\mathcal{E}_r(j)|} \sum_{e\in\mathcal{E}_r(j)} \textbf{Message}^{(l)}(e) \\
&= \Big\|_{x\in\mathcal{T}} \tfrac{1}{|\mathcal{E}_{xs}(j)|} \sum_{e\in\mathcal{E}_{xs}(j)} \textbf{Message}^{(l)}(e) \\
&= \Big\|_{x\in\mathcal{T}} \tfrac{1}{|\mathcal{E}_{xs}(j)|} \sum_{(i,j)\in\mathcal{E}_{xs}(j)} M_{xs}^{(l)} h_i^{(l-1)} \\
&= \Big\|_{x\in\mathcal{T}} \tfrac{1}{|\mathcal{E}_{xs}(j)|} M_{xs}^{(l)} \sum_{(i,j)\in\mathcal{E}_{xs}(j)} h_i^{(l-1)} \\
&= \Big\|_{x\in\mathcal{T}} M_{xs}^{(l)} \left(H_x^{(l-1)}\right)' \bar{A}_{xs}^{(j)},
\end{aligned}$$

where $\bar{A}_{xs}^{(j)}$ denotes the *j-th* column of $\bar{A}_{xs}$. Notice that

$$h_j^{(l)} = \textbf{Transform}^{(l)}(j) = W_s^{(l)} g_j^{(l)}, \tag{5.13}$$

and (again) at most T elements of the concatenation $g_j^{(l)}$ are non-zero. Therefore let $W_{xs}^{(l)}$ be the columns of $W_s^{(l)}$ that select the concatenated element of $g_j^{(l)}$ corresponding to node type x. Then we can write

$$h_j^{(l)} = \sum_{x\in\mathcal{T}} W_{xs}^{(l)} M_{xs}^{(l)} \left(H_x^{(l-1)}\right)' \bar{A}_{xs}^{(j)}. \tag{5.14}$$

Defining the operator $Q_{xs}^{(l)} := \left(W_{xs}^{(l)} M_{xs}^{(l)}\right)'$, this implies that

$$\begin{aligned}
H_s^{(l)} &= \sum_{x\in\mathcal{T}} \bar{A}_{xs} H_x^{(l-1)} Q_{xs}^{(l-1)} \\
&= [\bar{A}_{x_1s}, \ldots, \bar{A}_{x_Ts}] \begin{bmatrix} H_{x_1}^{(l-1)} & 0 & 0 \\ 0 & \ldots & 0 \\ 0 & 0 & H_{x_T}^{(l-1)} \end{bmatrix} \begin{bmatrix} Q_{x_1s}^{(l-1)} \\ \ldots \\ Q_{x_Ts}^{(l-1)} \end{bmatrix} \\
&= \bar{A}_{\cdot s} H_{\cdot}^{(l-1)} Q_{\cdot s}^{(l-1)}
\end{aligned}$$

Algorithm 7: Training step for one minibatch (indirect version)

Require: heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}, \mathcal{R})$, node feature matrices X, adjacency matrices A_{xy} where $\forall (x, y) \in \mathcal{R}$, source node type s, target node type t, source node label matrix Y_s.
Ensure: HGNN **f**, classifier **g**, KTN $\mathbf{t}_{\text{KTN}}$

$H_s^{(L)}, H_t^{(L)} = \mathbf{f}(H^{(0)} = X, \mathcal{G})$, $H_t^* = \mathbf{0}$
for each meta-path $p = t \to s$ **do**
  $x = t$, $Z = H_t^{(L)}$
  for each node type $y \in p$ **do**
    $Z = A_{xy} Z T_{xy}$
    $x = y$
  end for
  $H_t^* = H_t^* + Z$
end for
$\mathcal{L}_{\text{KTN}} = \left\| H_s^{(L)} - H_t^* \right\|_2$
$\mathcal{L} = \mathcal{L}_{\text{CL}}(\mathbf{g}(H_s^{(L)}), Y_s) + \lambda \mathcal{L}_{\text{KTN}}$
Update **f**, **g**, $\mathbf{t}_{\text{KTN}}$ using $\nabla \mathcal{L}$

Algorithm 8: Test step for a target domain (indirect version)

Require: pretrained HGNN **f**, classifier **g**, KTN $\mathbf{t}_{\text{KTN}}$
Ensure: target node label matrix Y_t

$H_t^{(L)} = \mathbf{f}(H^{(0)} = X, \mathcal{G})$, $H_t^* = \mathbf{0}$
for each meta-path $p = t \to s$ **do**
  $x = t$, $Z = H_t^{(L)}$
  for each node type $y \in p$ **do**
    $X = Z T_{xy}$
    $x = y$
  end for
  $H_t^* = H_t^* + Z$
end for
return $\mathbf{g}(H_t^*)$

Similarly we have $H_t^{(l)} = \bar{A}_{\cdot t} H^{(l-1)}_{\cdot} Q_{\cdot t}^{(l-1)}$. Since $H_s^{(l)}$ and $H_t^{(l)}$ share the term $H^{(l-1)}_{\cdot}$, we can write

$$H_s^{(l)} = \bar{A}_{\cdot s} \bar{A}_{\cdot t}^{-1} H_t^{(l)} (Q_{\cdot t}^{(l-1)})^{-1} Q_{\cdot s}^{(l-1)}, \tag{5.15}$$

where X^{-1} denotes the pseudo-inverse. □

5.8.2 Indirectly Connected Source and Target Node Types

When source and target node types are indirectly connected by another node type x, we can simply extend $\mathbf{t}_{\text{KTN}}(H_t^{(L)})$ to $(A_{xs}(A_{tx} H_t^{(L)} T_{tx}) T_{xs})$ where $T_{tx} T_{xs}$ becomes a mapping function from target to source domains. Algorithms 7 and 8 show how to extend KTN. For every step $(x \to y)$ in

Table 5.4: **KTN on Open Academic Graph on Computer Science field**. The *gain* column shows the relative gain of our method over using no domain adaptation (*Base* column). *o.o.m* denotes *out-of-memory* errors.

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain%)
P-A (L1)	NDCG	0.399	0.452	0.405	0.292	0.262	0.261	0.26	0.178	0.425	**0.623 (56)**
	std	0.010	0.012	0.032	0.009	0.021	0.014	0.021	0.000	0.006	0.004
	MRR	0.297	0.361	0.314	0.179	0.129	0.111	0.138	0.041	0.363	**0.629 (112)**
	std	0.024	0.006	0.041	0.011	0.032	0.031	0.033	0.000	0.005	0.004
A-P (L1)	NDCG	0.401	0.566	0.598	0.294	0.364	0.246	0.195	0.153	0.557	**0.733 (83)**
	std	0.003	0.012	0.014	0.034	0.049	0.046	0.025	0.000	0.002	0.007
	MRR	0.318	0.508	0.544	0.229	0.27	0.09	0.047	0.022	0.507	**0.711 (123)**
	std	0.001	0.029	0.028	0.093	0.117	0.037	0.029	0.000	0.003	0.009
A-V (L1)	NDCG	0.459	0.457	0.47	0.382	0.346	0.359	0.403	0.207	0.461	**0.671 (46)**
	std	0.030	0.033	0.036	0.015	0.029	0.109	0.024	0.000	0.002	0.004
	MRR	0.364	0.413	0.458	0.341	0.205	0.253	0.327	0.011	0.389	**0.698 (92)**
	std	0.079	0.08	0.093	0.05	0.098	0.143	0.044	0.000	0.004	0.003
V-A (L1)	NDCG	0.283	0.443	0.435	0.242	0.372	0.418	0.272	0.153	0.154	**0.584 (107)**
	std	0.045	0.012	0.007	0.004	0.048	0.039	0.004	0.000	0.006	0.005
	MRR	0.133	0.365	0.345	0.094	0.241	0.444	0.144	0.006	0.006	**0.586 (340)**
	std	0.074	0.027	0.017	0.011	0.103	0.115	0.018		0.007	0.010
P-A (L2)	NDCG	0.229	0.23	o.o.m	0.239	o.o.m	o.o.m	0.168	o.o.m	0.215	**0.282 (23)**
	std	0.005	0.003	-	0.006	-	-	0.007	-	0.004	0.002
	MRR	0.121	0.118	o.o.m	0.14	o.o.m	o.o.m	0.02	o.o.m	0.143	**0.2248 (86)**
	std	0.019	0.004	-	0.01	-	-	0.006	-	0.003	0.003
A-P (L2)	NDCG	0.197	0.162	o.o.m	0.204	0.158	0.161	0.132	o.o.m	0.208	**0.287 (46)**
	std	0.006	0.009	-	0.006	0.019	0.022	0.012	-	0.004	0.001
	MRR	0.095	0.052	o.o.m	0.106	0.032	0.045	0.017	o.o.m	0.132	**0.242 (155)**
	std	0.009	0.022	-	0.016	0.018	0.027	0.008	-	0.005	0.002
A-V (L2)	NDCG	0.347	0.329	0.295	0.325	0.288	0.273	0.289	o.o.m	0.297	**0.402 (16)**
	std	0.003	0.034	0.014	0.013	0.011	0.058	0.011	-	0.002	0.003
	MRR	0.310	0.296	0.198	0.223	0.128	0.097	0.11	o.o.m	0.227	**0.399 (29)**
	std	0.004	0.109	0.047	0.065	0.003	0.096	0.034	-	0.001	0.015
V-A (L2)	NDCG	0.235	0.249	0.251	0.214	0.197	0.205	0.217	o.o.m	0.119	**0.252 (7)**
	std	0.002	0.002	0.006	0.004	0.008	0.004	0.002	-	0.001	0.007
	MRR	0.130	0.157	0.161	0.09	0.044	0.068	0.085	o.o.m	0.000	**0.166 (28)**
	std	0.010	0.011	0.009	0.015	0.007	0.009	0.005	-	0.000	0.012

a meta-path ($t \rightarrow \cdots \rightarrow s$) connecting target node type t to source node type s, we define a transformation matrix T_{xy}, run a convolution operation with an adjacency matrix A_{xy}, then map the transformed embedding to the source domain. We run the same process for all meta-paths connecting from target node type t to source node type s, and sum up them to match with the source embeddings. In the test phase, we run the same process to get the transformed target embeddings, but this time, without adjacency matrices. We run Algorithm 7 and 8 for transfer learning tasks between author and venue nodes which are indirectly connected by paper nodes in OAG graphs (Figure 5.5(b)). As shown in Tables 5.4, 5.6, and 5.7, we successfully transfer HGNN models between author and venue nodes (A-V and V-A) for both L1 and L2 tasks.

Will lengths of meta-paths affect the performance? We examine the performance of KTN varying the length of meta-paths between source and target node types. In Table 5.8, accuracy decreases with longer meta-paths. When we add additional meta-paths than the minimum path, it also brings noise in every edge types. Note that author and venue nodes are indirectly connected by paper nodes; thus the minimum length of meta-paths in the A-V (L1) task is 2. The accuracy in the A-V (L1) task with a meta-path of length 1 is low because KTN fails to transfer anything with a meta-path shorter than the minimum. Using the minimum length of meta-paths is enough for KTN.

5.8.3 More results for Zero-shot Transfer Learning in Section 5.5.3

We show the zero-shot transfer learning results with error bars on OAG-computer science and Pubmed in Tables 5.4 and 5.5. We also present the results with error bars on OAG-computer networks and OAG-machine learning in Tables 5.6 and 5.7, respectively. Across all tasks and graphs, our proposed method KTN consistently outperforms all baselines.

5.8.4 Analysis for Baselines in Section 5.5.3

Among baselines, MMD-based models (DAN and JAN) outperform adversarial based methods (DANN, CDAN, and CDAN-E) and optimal transport-based method (WDGRL), unlike results reported in [99, 128]. These reversed results are a consequence of HGNN's unique feature extractors for source and target domains. When f_s and f_t denote feature extractors for source and target domains, respectively, DANN and CDAN define their adversarial losses as a cross entropy loss ($\mathbb{E}[\log f_s] - \mathbb{E}[\log f_t]$) where gradients of the subloss $\mathbb{E}[\log f_s]$ are passed only back to f_s, while gradients of the subloss $\mathbb{E}[\log f_t]$ are passed only back to f_t. Importantly, source and target feature extractors do not share any gradient information, resulting in divergence. This did not occur in their original test environments where source and target domains share a single feature extractor. Similarly, WDGRL measures the first-order Wasserstein distance as an adversarial loss, which also brings the same effect as the cross-entropy loss we described above, leading to divergent gradients between source and target feature extractors. On the other hand, DAN and JAN define a loss in terms of higher-order MMD between source and target features. Then the gradients of the loss passed to each feature extractor contain both source and target feature information, resulting in a more stable gradient estimation. This shows again the importance of considering different feature extractors in HGNNs.

JAN, CDAN, and CDAN-E often show out of memory issues in Tables 5.4, 5.6, and 5.7. These baselines consider the classifier prediction whose dimension is equal to the number of classes in a given task. That is why JAN, CDAN, and CDAN-E fail at the L2 field prediction tasks in OAG graphs where the number of classes is $17,729$.

LP performs worst among the baselines, showing the limitation of relying only on graph structures. LP maintains a label vector with the length equal to the number of classes for each node, thus shows out-of-memory issues on tasks with large number of classes on large-size graphs (L2 tasks with $17,729$ labels on the OAG-CS graph). EP performs moderately well similar to other DA methods, but lower than KTN up to 60% absolute points of MRR, showing the limitation of not using target node attributes.

5.8.5 More results for Generality of KTN in Section 5.5.4

We show KTN performance on 6 different types of HGNN models across 4 different zero-shot domain adaptation tasks on the OAG-computer science dataset in Table 5.9. Descriptions of each HGNN model can be found in Appendix 5.8.10. While KTN consistently improves all HGNN models' performance on zero-labeled node types using labels rooted at other node types, the magnitude of improvements varies. While HAN sees up to 4958% (V-A (L1) task in Table 5.9),

Table 5.5: **KTN on PubMed**

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain%)
D-G	NDCG	0.587	0.629	0.615	0.614	0.624	0.646	0.604	0.601	0.571	**0.700 (19)**
	std	0.004	0.013	0.028	0.008	0.078	0.015	0.022	0.000	0.004	0.005
	MRR	0.372	0.425	0.414	0.397	0.428	0.443	0.388	0.389	0.336	**0.499 (34)**
	std	0.003	0.007	0.054	0.013	0.066	0.027	0.035	0.000	0.003	0.006
G-D	NDCG	0.596	0.599	0.577	0.599	0.581	0.606	0.578	0.576	0.580	**0.662 (11)**
	std	0.007	0.020	0.032	0.011	0.054	0.019	0.019	0.000	0.011	0.003
	MRR	0.354	0.362	0.332	0.356	0.337	0.362	0.340	0.351	0.353	**0.445 (26)**
	std	0.005	0.015	0.019	0.008	0.023	0.031	0.015	0.000	0.008	0.002

Table 5.6: **KTN on Open Academic Graph on Computer Network field**

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain%)
P-A (L2)	NDCG	0.331	0.344	o.o.m	0.335	o.o.m	o.o.m	0.287	0.221	0.270	**0.382 (16)**
	std	0.004	0.005	-	0.004	-	-	0.012	0.000	0.003	0.004
	MRR	0.250	0.277	o.o.m	0.280	o.o.m	o.o.m	0.199	0.130	0.270	**0.360 (44)**
	std	0.024	0.012	-	0.007	-	-	0.004	0.000	0.003	0.010
A-P (L2)	NDCG	0.313	0.290	o.o.m	0.250	0.234	0.168	0.266	0.114	0.319	**0.364 (17)**
	std	0.002	0.023	-	0.021	0.041	0.025	0.030	0.000	0.004	0.003
	MRR	0.250	0.233	o.o.m	0.130	0.116	0.051	0.212	0.038	0.296	**0.368 (47)**
	std	0.015	0.039	-	0.051	0.069	0.037	0.061	0.000	0.005	0.004
A-V (L2)	NDCG	0.539	0.521	0.519	0.510	0.467	0.362	0.471	0.232	0.443	**0.567 (5)**
	std	0.012	0.031	0.008	0.022	0.008	0.045	0.024	0.000	0.002	0.008
	MRR	0.584	0.528	0.461	0.510	0.293	0.294	0.365	0.000	0.406	**0.628 (8)**
	std	0.042	0.015	0.011	0.054	0.013	0.088	0.019	0.000	0.004	0.016
V-A (L2)	NDCG	0.256	0.343	0.345	0.265	0.328	0.316	0.263	0.133	0.119	**0.341 (33)**
	std	0.006	0.012	0.005	0.005	0.005	0.003	0.003	0.000	0.001	0.005
	MRR	0.117	0.296	0.286	0.151	0.285	0.275	0.147	0.000	0.000	**0.281 (141)**
	std	0.020	0.009	0.004	0.009	0.006	0.008	0.009	0.000	0.000	0.014

MAGNN is improved by up to 47% (P-A(L1) task) or sees no improvement (A-V(L1) task). This gap stems from how many parameters each HGNN model shares across node types. HAN does not share any parameters during message-passing operations (every parameters are specialized to each meta-path), while MAGNN shares the transformation matrices across all node types at every layer. By sharing more parameters with other node types, the gradients are more likely passed to target node type-specific parameters, leaving less room for improvement by KTN. However, KTN is still necessary for any HGNN models. MPNN who shares all parameters except the projection matrices that map different input attributes into the same embedding space at the beginning still sees improvements by up to 311%. Again, these experimental results show the impact of having different feature extractors for each node type in HGNN models.

5.8.6 Effect of trade-off coefficient λ

We examine the effect of λ on transfer learning performance. In Table 5.10, as λ decreases, target accuracy decreases as expected. Source accuracy also sees small drops since $\mathcal{L}_{\text{KTN}}$ functions as a regularizer; by removing the regularization effect, source accuracy decreases. When λ becomes large, both source and target accuracy drop significantly. Source accuracy drops since the effect of $\mathcal{L}_{\text{KTN}}$ becomes larger than the classification loss $\mathcal{L}_{\text{CL}}$. Even the effect of transfer learning become larger by having larger λ, since the source accuracy which will be transferred to the target domain is low, the target accuracy is also low. Thus we set λ to 1 throughout the experiments.

Table 5.7: **KTN on Open Academic Graph on Machine Learning field**

Task	Metric	Base	DAN	JAN	DANN	CDAN	CDAN-E	WDGRL	LP	EP	KTN (gain%)
P-A (L2)	**NDCG**	0.268	0.290	o.o.m	0.291	o.o.m	0.249	0.232	0.272	0.215	**0.318 (19)**
	std	0.002	0.009	-	0.004	-	0.005	0.004	0.000	0.002	0.004
	MRR	0.134	0.220	o.o.m	0.222	o.o.m	0.095	0.098	0.195	0.143	**0.269 (102)**
	std	0.006	0.020	-	0.026	-	0.003	0.037	0.000	0.003	0.006
A-P (L2)	**NDCG**	0.261	0.225	o.o.m	0.234	0.228	0.241	0.241	0.119	0.267	**0.319 (22)**
	std	0.002	0.009	-	0.004	0.005	0.011	0.002	0.000	0.001	0.005
	MRR	0.207	0.127	o.o.m	0.155	0.152	0.095	0.182	0.035	0.214	**0.287 (39)**
	std	0.018	0.042	-	0.008	0.009	0.003	0.017	0.000	0.012	0.011
A-V (L2)	**NDCG**	0.465	0.493	0.463	0.477	0.408	0.422	0.393	0.224	0.424	**0.538 (16)**
	std	0.006	0.004	0.003	0.003	0.006	0.013	0.005	0.000	0.005	0.004
	MRR	0.469	0.542	0.537	0.519	0.412	0.240	0.213	0.001	0.391	**0.632 (35)**
	std	0.039	0.008	0.005	0.003	0.015	0.008	0.009	0.000	0.021	0.006
V-A (L2)	**NDCG**	0.252	0.293	0.292	0.237	0.242	0.255	0.250	0.137	0.119	**0.302 (20)**
	std	0.006	0.011	0.009	0.004	0.003	0.002	0.004	0.000	0.003	0.007
	MRR	0.131	0.212	0.199	0.086	0.085	0.129	0.118	0.000	0.000	**0.227 (73)**
	std	0.016	0.023	0.013	0.005	0.021	0.007	0.012	0.000	0.000	0.015

Table 5.8: **Meta-path length in KTN:** increasing the meta-path longer than the minimum does not bring significant improvement to KTN. Note that the minimum length of meta-paths in the A-V (L1) task is 2.

Task	P-A (L1)		A-V (L1)	
Meta-path length	**NDCG**	**MRR**	**NDCG**	**MRR**
1	0.623	0.621	0.208	0.010
2	0.627	0.628	0.673	0.696
3	0.608	0.611	0.627	0.648
4	0.61	0.623	0.653	0.671

5.8.7 Synthetic Heterogeneous Graph Generator

Our synthetic heterogeneous graph generator is based on attributed Stochastic Block Models (SBM) [148, 149], using blocks (clusters) as the node classes. In the attributed SBM, graphs exhibit *within-type* cluster homophily at the *edge-level* (nodes most-frequently connect to other nodes in the same cluster), and at the *feature-level* (nodes are closest in feature space to other nodes in the same cluster). To produce heterogeneous graphs, we additionally introduce *between-type* cluster homophily, which allows us to model real-world heterogeneous graphs in which knowledge can be shared across node types.

The first step in generating a heterogeneous SBM is to decide how many clusters will partition each node type. Assume *within-type* cluster counts $k_1, \ldots, k_T$. We allow for *between-type* cluster homophily with a $K_T := \min_t\{k_t\}$-partition of clusters such that each cluster group has at least one corresponding cluster from other node types.

Secondly, edge-level homophily is controlled by signal-to-noise ratios $\sigma_e = p/q$ where nodes *within-cluster* are connected with probability p and nodes *between-cluster* are connected with probability q. Additionally, edges *within one cluster group across different types* (see previous paragraph) is controlled together with edges *between different cluster groups across different types* using some σ_e. In Section 5.5.5, we describe the manipulation of multiple σ_e parameters *within-and-between* types.

Table 5.9: **KTN on different HGNN models**: The *Source* column shows accuracy on source node types. *Base* and KTN columns show accuracy on target node types without/with using KTN, respectively. The *Gain* column shows the relative gain of our method over using no transfer learning.

		P-A (L1)				A-P (L1)			
HGNN type	**Metric**	**Source**	**Base**	**KTN**	**Gain%**	**Source**	**Base**	**KTN**	**Gain%**
R-GCN	**NDCG**	0.765	0.337	0.577	**71.12**	0.648	0.388	0.647	**66.82**
	std	0.004	0.005	0.002		0.006	0.007	0.004	
	MRR	0.757	0.236	0.587	**148.73**	0.623	0.270	0.611	**126.18**
	std	0.002	0.003	0.001		0.005	0.008	0.004	
HAN	**NDCG**	0.476	0.179	0.520	**190.56**	0.515	0.182	0.512	**181.33**
	std	0.004	0.006	0.003		0.004	0.009	0.011	
	MRR	0.416	0.047	0.497	**960.55**	0.509	0.055	0.527	**850.90**
	std	0.001	0.002	0.002		0.005	0.004	0.005	
HGT	**NDCG**	0.757	0.294	0.574	**95.07**	0.670	0.283	0.581	**104.83**
	std	0.002	0.003	0.004		0.001	0.003	0.009	
	MRR	0.749	0.178	0.563	**216.17**	0.670	0.149	0.565	**279.52**
	std	0.005	0.007	0.001		0.002	0.007	0.006	
MAGNN	**NDCG**	0.657	0.463	0.574	**24.01**	0.676	0.557	0.622	**11.68**
	std	0.003	0.001	0.003		0.001	0.001	0.003	
	MRR	0.631	0.378	0.556	**47.33**	0.680	0.509	0.592	**16.14**
	std	0.003	0.002	0.004		0.001	0.002	0.005	
MPNN	**NDCG**	0.602	0.443	0.590	**33.11**	0.646	0.307	0.621	**101.92**
	std	0.002	0.002	0.001		0.005	0.013	0.004	
	MRR	0.572	0.319	0.575	**80.10**	0.660	0.145	0.595	**311.42**
	std	0.001	0.003	0.005		0.002	0.024	0.003	
H-MPNN	**NDCG**	0.789	0.399	0.623	**56.14**	0.671	0.401	0.733	**82.88**
	std	0.001	0.005	0.001		0.003	0.005	0.009	
	MRR	0.777	0.297	0.629	**111.86**	0.661	0.318	0.711	**123.30**
	std	0.003	0.001	0.002		0.007	0.004	0.008	

		V-A (L1)				A-V (L1)			
HGNN type	**Metric**	**Source**	**Base**	**KTN**	**Gain%**	**Source**	**Base**	**KTN**	**Gain%**
R-GCN	**NDCG**	0.664	0.426	0.530	**24.36**	0.660	0.599	0.744	**24.26**
	std	0.003	0.006	0.002		0.001	0.008	0.004	
	MRR	0.683	0.325	0.514	**58.39**	0.656	0.524	0.785	**49.87**
	std	0.003	0.008	0.004		0.011	0.009	0.005	
HAN	**NDCG**	0.618	0.153	0.510	**232.35**	0.515	0.546	0.689	**26.21**
	std	0.005	0.007	0.003		0.008	0.003	0.005	
	MRR	0.634	0.010	0.516	**4958.82**	0.508	0.511	0.758	**48.28**
	std	0.002	0.005	0.002		0.001	0.008	0.007	
HGT	**NDCG**	0.615	0.234	0.536	**128.95**	0.694	0.367	0.735	**100.22**
	std	0.002	0.005	0.002		0.006	0.007	0.009	
	MRR	0.638	0.095	0.537	**464.88**	0.699	0.267	0.772	**189.21**
	std	0.006	0.002	0.005		0.002	0.005	0.012	
MAGNN	**NDCG**	0.536	0.513	0.513	**0.00**	0.684	0.676	0.692	**2.37**
	std	0.005	0.001	0.001		0.001	0.002	0.001	
	MRR	0.586	0.522	0.522	**0.00**	0.686	0.751	0.752	**0.13**
	std	0.004	0.001	0.002		0.002	0.001	0.004	
MPNN	**NDCG**	0.578	0.380	0.532	**40.03**	0.639	0.578	0.794	**37.19**
	std	0.008	0.008	0.004		0.007	0.007	0.005	
	MRR	0.603	0.253	0.505	**100.12**	0.652	0.584	0.847	**44.96**
	std	0.001	0.003	0.007		0.006	0.001	0.006	
H-MPNN	**NDCG**	0.670	0.283	0.584	**106.50**	0.676	0.459	0.671	**46.22**
	std	0.002	0.002	0.006		0.005	0.004	0.003	
	MRR	0.689	0.133	0.586	**339.76**	0.677	0.364	0.698	**91.92**
	std	0.003	0.003	0.005		0.01	0.005	0.002	

Table 5.10: **Effect of λ**

	P-A (L1)				A-V (L1)			
Metric	**NDCG**		**MRR**		**NDCG**		**MRR**	
λ	**Source**	**Target**	**Source**	**Target**	**Source**	**Target**	**Source**	**Target**
10^{-5}	0.780	0.587	0.772	0.595	0.689	0.626	0.690	0.642
10^{-3}	0.788	0.58	0.779	0.576	0.687	0.654	0.689	0.677
10^{0}	0.792	0.621	0.788	0.633	0.689	0.670	0.692	0.696
10^{2}	0.750	0.617	0.757	0.623	0.654	0.644	0.659	0.668
10^{4}	0.143	0.177	0.007	0.031	0.411	0.432	0.373	0.421

Table 5.11: **Statistics of Open Academic Graph**

Domain	#papers	#authors	#fields	#venues	#institues	
Computer Science	544,244	510,189	45,717	6,934	9,097	
Computer Network	75,015	82,724	12,014	2,115	4,193	
Machine Learning	90,012	109,423	19,028	3,226	5,455	
Domain	**#P-A**	**#P-F**	**#P-V**	**#A-I**	**#P-P**	**#F-F**
Computer Science	1,091,560	3,709,711	544,245	612,873	11,592,709	525,053
Computer Network	155,147	562,144	75,016	111,180	1,154,347	110,869
Machine Learning	166,119	585,339	90,013	156,440	1,209,443	163,837

Finally, node attributes are generated by a multivariate Normal mixture model, using the cluster partition as the mixture groups. Thus feature-level homophily is controlled by increasing the variance of the cluster centers σ_f, while keeping the within-cluster variance fixed. Cross-type feature homophily is controlled by setting a center of cluster centers *within-type* with linear combinations of centers (of cluster centers) of other types. Note that features of different types are allowed to have different dimensions, as we generate different mixture-model cluster centers for each cluster *within each type*.

Toy Heterogeneous Graph in Section 5.3.2 Using the synthetic graph procedure described above, we used the following hyperparameters to simulate the toy heterogeneous graph shown in Figure 5.2. We generate the graph with 2 node types and 4 edge types as described in Figure 5.1(a), then we divide each node type into 4 classes of 400 nodes. To generate an easy-to-transfer scenario, signal-to-noise ratio σ_f between means of feature distributions are all set to 10. The ratio σ_e of the number of intra-class edges to the number of inter-class edges is set to 10 among the same node types and across different node types. The dimension of features is set to 24 for both node types.

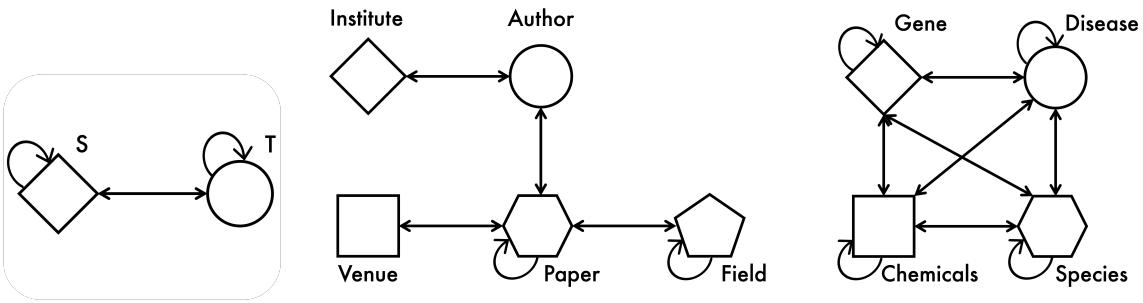
Sensitivity test in Section 5.5.5 Figure 5.5(a) shows the structures of graphs we used in Section 5.5.5. The dimension of features are set to 24 for both node types for the "easy" scenario, and 32 and 48 for types s and t, respectively, for the "hard" scenario. Additionally, for the "hard" scenario, we divide the t nodes into 8 clusters instead of 4. The other hyperparameters σ_e and σ_f are described in Section 5.5.5. For each unique value of $\sigma_{(\cdot)}$ across the six $(\sigma_{(\cdot)}, r)$ pairs, we generate 5 heterogeneous graphs.

5.8.8 Real-world Dataset

Open Academic Graph (OAG) [134, 145, 192] is the largest publicly available heterogeneous graph. It is composed of five types of nodes: papers, authors, institutions, venues, fields and their corresponding relationships. Papers and authors have text-based attributes, while institutions, venues, and fields have text- and graph structure-based attributes. To test the generalization of the

Table 5.12: **Statistics of PubMed Graph**

#gene	**#disease**	**#chemicals**	**#species**	
13,561	20,163	26,522	2,863	
#G-G	**#G-D**	**#D-D**	**#C-G**	**#C-D**
32,211	25,963	68,219	31,278	51,324
#C-C	**#C-S**	**#S-G**	**#S-D**	**#S-S**
124,375	6,298	3,156	5,246	1,597



(a) Synthetic graph (b) OAG (c) PubMed

Figure 5.5: **Schema of synthetic and real-world heterogeneous graphs.**

proposed model, we construct three field-specific subgraphs from OAG: the Computer Science (OAG-CS), Computer Networks (OAG-CN), and Machine Learning (OAG-ML) academic graphs.

Papers, authors, and venues are labeled with research fields in two hierarchical levels, L1 and L2. OAG-CS has both L1 and L2 labels, while OAG-CN and OAG-ML have only L2 labels (their L1 labels are all "computer science"). Transfer learning is performed on the L1 and L2 field prediction tasks between papers, authors, and venues for each of the aforementioned subgraphs. Note that paper-author (P-A) and paper-venue (P-V) are directly connected, while author-venue (A-V) are indirectly connected via papers.

The number of classes in the L1 task is 275, while the number of classes in the L2 task is $17,729$. The graph statistics are listed in Table 5.11, in which P–A, P–F, P–V, A–I, P–P, and F-F denote the edges between paper and author, paper and field, paper and venue, author and institute, the citation links between two papers, and the hierarchical links between two fields. The graph structure is described in Figure 5.5(b).

For paper nodes, features are generated from each paper's title using a pre-trained XLNet [161]. For author nodes, features are averaged over features of papers they published. Feature dimension of paper and author nodes is 769. For venue, institution, and field node types, features of dimension 400 are generated from their heterogeneous graph structures using metapath2vec [30].

PubMed [170] is a novel biomedical network constructed through text mining and manual processing on biomedical literature. PubMed is composed of genes, diseases, chemicals, and species. Each gene or disease is labeled with a set of diseases (e.g., cardiovascular disease) they belong to or cause. Transfer learning is performed on a disease prediction task between genes and disease node types.

The number of classes in the disease prediction task is 8. The graph statistics are listed in Table 5.12, in which G, D, C, and S denote genes, diseases, chemicals, and species node types. The graph structure is described in Figure 5.5(c).

For gene and chemical nodes, features of dimension 200 are generated from related PubMed papers using word2vec [105]. For diseases and species nodes, features of dimension 50 are generated based on their graph structures using TransE [13].

5.8.9 Baselines

Zero-shot domain adaptation can be categorized into three groups — MMD-based methods, adversarial methods, and optimal-transport-based methods. MMD-based methods [98, 100, 140] minimize the maximum mean discrepancy (MMD) [51] between the mean embeddings of two distributions in reproducing kernel Hilbert space. DAN [98] enhances the feature transferability by minimizing multi-kernel MMD in several task-specific layers. JAN [100] aligns the joint distributions of multiple domain-specific layers based on a joint maximum mean discrepancy (JMMD) criterion.

Adversarial methods [47, 99] are motivated by theory in [8, 9] suggesting that a good cross-domain representation contains no discriminative information about the origin of the input. They learn domain invariant features by a min-max game between the domain classifier and the feature extractor. DANN [47] learns domain invariant features by a min-max game between the domain classifier and the feature extractor. CDAN [99] exploits discriminative information conveyed in the classifier predictions to assist adversarial adaptation. CDAN-E [99] extends CDAN to condition the domain discriminator on the uncertainty of classifier predictions, prioritizing the discriminator on easy-to-transfer examples.

Optimal transport-based methods [128] estimate the empirical Wasserstein distance [122] between two domains and minimizes the distance in an adversarial manner. Optimal transport-based method are based on a theoretical analysis [122] that Wasserstein distance can guarantee generalization for domain adaptation. WDGRL [128] estimates the empirical Wasserstein distance between two domains and minimizes the distance in an adversarial manner.

5.8.10 HGNN models

All heterogeneous graph neural networks (HGNN) models we used in the experiments have layer-wise parameters. As the HGNN models have parameters specialized in either node/edge/meta-path types, they all have distinct feature extractors for each node types, thus, they will suffer from the under-trained target node phenomena we showed in Section 5.3.2. Also, because the core intuition in KTN — namely that embeddings of any node types at the last layer are computed using the same set of the previous layer's intermediate embeddings (see Section 5.3.3) — holds across all

HGNN models, KTN can be applied to any HGNN models and show greatly increased target-type accuracy.

5.8.11 Experimental Settings

All experiments were conducted on the same p2.xlarge Amazon EC2 instance. Here, we describe the structure of HGNNs used in each heterogeneous graph.

Open Academic Graph: We use a 4-layered HGNN with transformation and message parameters of dimension 128 for KTN and other baselines. Learning rate is set to 10^{-4}.

PubMed: We use a single-layered HGNN with transformation and message parameters of dimension 10 for KTN and other baselines. Learning rate is set to 5×10^{-5}.

Synthetic Heterogeneous Graphs: We use a 2-layered HGNN with transformation and message parameters of dimension 128 for KTN and other baselines. Learning rate is set to 10^{-4}.

We implement LP, EP and KTN using Pytorch. For the domain adaptation baselines (DAN, JAN, DANN, CDAN, CDAN-E, and WDGRL), we use a public domain adaptation library ADA [2]. We match the numbers of layers and dimensions of hidden embeddings across all HGNN models. We implement MPNN and HMPNN using Pytorch. For other HGNN models (R-GCN, HAN, HGT, and MAGNN), we use an open-source toolkit for Heterogeneous Graph Neural Network (OpenHGNN) [3]. Our code is publicly available [4].

[2] https://github.com/criteo-research/pytorch-ada
[3] https://github.com/BUPT-GAMMA/OpenHGNN
[4] https://github.com/minjiyoon/KTN

Chapter 6

Privacy II: privacy-enhanced graph generative model

As the field of Graph Neural Networks (GNN) continues to grow, it experiences a corresponding increase in the need for large, real-world datasets to train and test new GNN models on challenging, realistic problems. Unfortunately, such graph datasets are often generated from online, highly privacy-restricted ecosystems, which makes research and development on these datasets hard, if not impossible. This greatly reduces the amount of benchmark graphs available to researchers, causing the field to rely only on a handful of publicly-available datasets. To address this problem, we introduce a novel graph generative model, Computation Graph Transformer (CGT) that learns and reproduces the distribution of real-world graphs in a privacy-controlled way. More specifically, CGT (1) generates effective benchmark graphs on which GNNs show similar task performance as on the source graphs, (2) scales to process large-scale graphs, (3) incorporates off-the-shelf privacy modules to guarantee end-user privacy of the generated graph. Extensive experiments across a vast body of graph generative models show that only our model can successfully generate privacy-controlled, synthetic substitutes of large-scale real-world graphs that can be effectively used to benchmark GNN models.

6.1 Motivation

Graph Neural Networks (GNNs) [20, 75] are machine learning models that learn the dependences in graphs via message passing between nodes. Various GNN models have been widely applied on a variety of industrial domains such as misinformation detection [10], financial fraud detection [156], traffic prediction [197], and social recommendation [176]. However, datasets from these industrial tasks are overwhelmingly proprietary and privacy-restricted and thus almost always unavailable for researchers to study or evaluate new GNN architectures. This state-of-affairs means that in many cases, GNN models cannot be trained or evaluated on graphs that are appropriate for the actual tasks that they need to execute.

In this paper, we propose a novel graph generation problem to overcome the limited access to real-world graph datasets. Given a graph, our goal is to generate synthetic graphs that follow its distribution in terms of graph structure, node attributes, and labels, making them usable as substitutes for the original graph for GNN research. Any observations or results from experiments on the original graph should be near-reproduced on the synthetic graphs. Additionally, the graph generation

process should be scalable and privacy-controlled to consume large-scale and privacy-restricted real-world graphs. Formally, our new graph generation problem is stated as follow:

Problem 2. *Let $\mathcal{A}$, $\mathcal{X}$, and $\mathcal{Y}$ denote adjacency, node attribute, and node label matrices; given an original graph $\mathcal{G} = (\mathcal{A}, \mathcal{X}, \mathcal{Y})$, generate a synthetic graph dataset $\mathcal{G}'$ satisfying:*

- ***Benchmark effectiveness:*** *performance rankings among m GNN models on $\mathcal{G}'$ should be similar to the rankings among the same m GNN models on $\mathcal{G}$.*
- ***Scalability:*** *computation complexity of graph generation should be linearly proportional to the size of the original graph $O(|\mathcal{G}|)$ (i.e., number of nodes or edges).*
- ***Privacy guarantee:*** *any syntactic privacy notions are given to end users (e.g., k-anonymity).*

While there is already a vast body of work on graph generation, we found that no study has fully addressed the problem setting above. [85, 111] generate random graphs using a few known graph patterns, while [92, 183] learn only graph structures without considering node attribute/label information, and [38] learn the structures with boolean node attributes. Recent graph generative models [102, 130] are mostly specialized to small-scale molecule graph generation.

In this work, we introduce a novel graph generative model, Computation Graph Transformer (CGT) that addresses the three requirements above for the benchmark graph generation problem. First, we reframe the graph generation problem into a discrete-value sequence generation problem. Motivated by GNN models that avoid scalability issues by operating on egonets sampled around each node, called *computation graphs* [55], we learn the distribution of *computation graphs* rather than the whole graph. In other words, our generated graph dataset $\mathcal{G}'$ will have a form of *a set of computation graphs* where GNN models can run immediately without preceded egonet sampling process. In addition to the scalability benefit, learning distributions of computation graphs which are the direct input to GNN models may also help to get better benchmark effectiveness. Then, instead of learning the joint distribution of graph structures and node attributes, we devise a novel *duplicate encoding* scheme for computation graphs that transforms an adjacency and feature matrix pair into a single, dense feature matrix that is isomorphic to the original pair. Finally, we quantize the feature matrix into a discrete value sequence that will be consumed by a Transformer architecture [152] adapted to our graph generation setting. After the quantization, our model can be easily extended to provide k-anonymity or differential privacy guarantees on node attributes and edge distributions by incorporating off-the-shelf privacy modules.

Extensive experiments on real-world graphs with a diverse set of GNN models demonstrate CGT provides significant improvement over existing generative models in terms of benchmark effectiveness (up to 1.03 higher Spearman correlations, up to 33% lower MSE between original and reproduced GNN accuracies), scalability (up to 35k nodes and 8k node attributes), and privacy guarantees (k-anonymity and differential privacy for node attributes). CGT also preserves graph statistics on computation graphs by up to 11.01 smaller Wasserstein distance than previous approaches.

In sum, our contributions are: 1) a novel graph generation problem featuring three requirements of modern graph learning; 2) reframing of the graph generation problem into a discrete-valued sequence generation problem; 3) a novel Transformer architecture able to encode the original computation graph structure in sequence learning; and finally 4) comprehensive experiments that evaluate the effectiveness of graph generative models to benchmark GNN models.

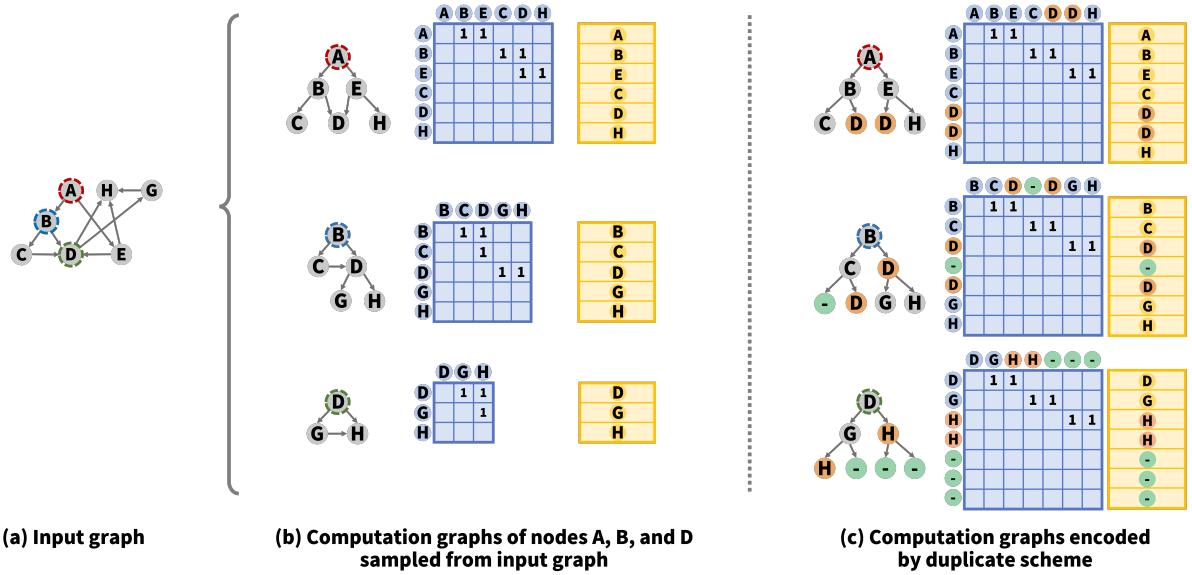


Figure 6.1: **Computation graphs with $s = 2$ neighbor samples and $L = 2$ depth.** (a) input graph; (b) original computation graphs have differently-shaped adjacency (blue) and attribute (yellow) matrices; (c) duplicate encoding scheme outputs the *same adjacency matrix* and *identically-shaped* attribute matrices.

6.2 From Graph Generation to Sequence Generation

In this section, we illustrate how to convert the whole-graph generation problem into a discrete-valued sequence generation problem. An input graph $\mathcal{G}$ is given as a triad of adjacency matrix $\mathcal{A} \in \mathbb{R}^{n \times n}$, node attribute matrix $\mathcal{X} \in \mathbb{R}^{n \times d}$, and node label matrix $\mathcal{Y} \in \mathbb{R}^{n}$ with n nodes and d-dimensional node attribute vectors.

6.2.1 Computation graph sampling in GNN training

Given large-scale real-world graphs, instead of operating on the whole graph, GNNs extract each node v's egonet $\mathcal{G}_v$, namely a *computation graph*, then compute embeddings of node v on $\mathcal{G}_v$. This means that in order to benchmark GNN models, we are not necessarily required to learn the distribution of the whole graph; instead, we can learn the distribution of computation graphs which are the direct input to GNN models. As with the global graph, a computation graph $\mathcal{G}_v$ is composed of a sub-adjacency matrix $\mathcal{A}_v \in \mathbb{R}^{n_v \times n_v}$, a sub-feature matrix $\mathcal{X}_v \in \mathbb{R}^{n_v \times d}$, and node v's label $\mathcal{Y}_v \in \mathbb{R}$, where each of n_v rows correspond to nodes sampled into the computation graph. Our problem then reduces to: *given a set of computation graphs* $\{\mathcal{G}_v = (\mathcal{A}_v, \mathcal{X}_v, \mathcal{Y}_v) : v \in \mathcal{G}\}$ *sampled from an original graph, we generate a set of computation graphs* $\{\mathcal{G}'_v = (\mathcal{A}'_v, \mathcal{X}'_v, \mathcal{Y}'_v)\}$. This reframing allows the graph generation process to scale to large-scale graphs.

6.2.2 Duplicate encoding scheme for computation graphs

Various sampling methods have been proposed to decide which neighboring nodes to add to a computation graph $\mathcal{G}_v$ given a target node v [23, 55, 68, 181]. Two common rules across these sampling methods are 1) the number of neighbors sampled for each node is limited to keep computation graphs small and 2) the maximum distance (i.e., maximum number of hops) from the target node v to sampled nodes is decided by the depth of GNN models. Details on how to sample computation graphs can be found in Appendix 6.7.3. This maximum number of neighbors is called the neighbor sampling number s and the maximum number of hops from the target node is called the depth of computation graphs L. Figure 6.1(b) shows computation graphs of nodes A, B, and D sampled with sampling number $s = 2$ and depth $L = 2$. Note that the shapes of computation graphs are variable.

Here we introduce a *duplicate encoding* scheme for computation graphs that is conceptually simple but brings a significant consequence: it *fixes the structure of all computation graphs* to the L-layered s-nary tree structure, allowing us to model all adjacency matrices as a constant. Starting from the target node v as a root node, we sample s neighbors iteratively L times from the computation graph. When a node has fewer neighbors than s, the duplicate encoding scheme defines a null node with zero attribute vector (node '$-$' in node B and D's computation graphs in Figure 6.1(c)) and samples it as a padding neighbor. When a node has a neighbor also sampled by another node, the duplicate encoding scheme copies the shared neighbor and provides each copy to parent nodes (node D in node A's computation graph is copied in Figure 6.1(c)). Each node attribute vector is also copied and added to the feature matrix. As shown in Figure 6.1(c), the duplicate encoding scheme ensures that all computation graphs have an identical adjacency matrix (presenting a balanced s-nary tree) and an identical shape of feature matrices. Under the duplicate encoding scheme, the graph structure information is fully encoded into feature matrices, which we will explain in details in Section 6.4.3. Note that in order to fix the adjacency matrix, we need to fix the order of nodes in adjacency and attribute matrices (e.g., breadth-first ordering in Figure 6.1(c)).

Now our problem reduces to learning the distribution of (duplicate-encoded) feature matrices of computation graphs: *given a set of feature matrix-label pairs* $\{(\tilde{\mathcal{X}}_v, \mathcal{Y}_v) : v \in \mathcal{G}\}$ *of duplicate-encoded computation graphs, we generate a set of feature matrix-label pairs* $\{(\tilde{\mathcal{X}}'_v, \mathcal{Y}'_v)\}$.

6.2.3 Quantization

To learn the distribution of feature matrices of computation graphs, we quantize feature vectors into discrete bins; specifically, we cluster feature vectors in the original graph using k-means and map each feature vector to its cluster id. Quantization is motivated by 1) privacy benefits and 2) ease of modeling. By mapping different feature vectors (which are clustered together) into the same cluster id, we can guarantee k-anonymity among them (more details in Section 6.3.2). Ultimately, quantization further reduces our problem to *learning the distribution of sequences of discrete values*, namely the sequences of cluster ids of feature vectors in each computation graph. Such a problem is naturally addressed by Transformers, state-of-the-art sequence generative models [152]. In Section 6.3, we introduce the Computational Graph Transformer (CGT), a novel architecture which learns the distribution of computation graph structures encoded in the sequences effectively.

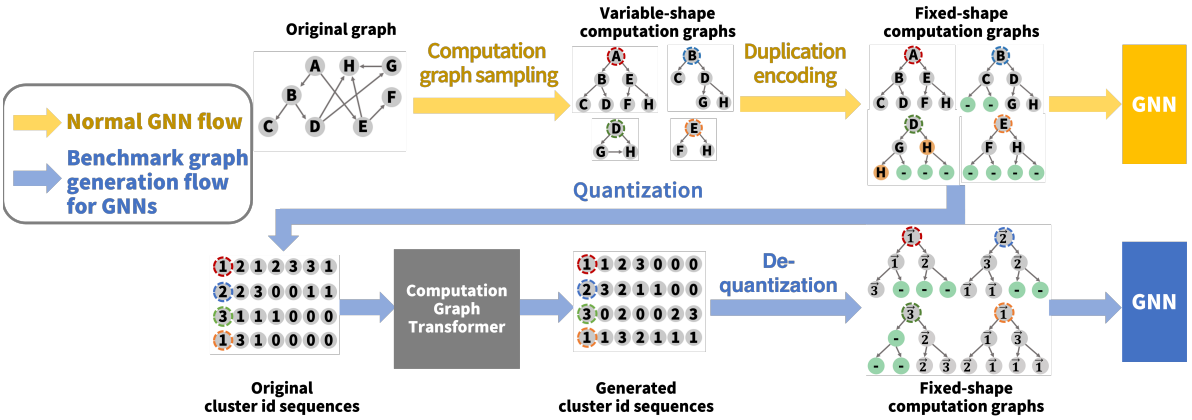


Figure 6.2: **Overview of our benchmark graph generation framework.** (1) We sample a set of computation graphs of variable shapes from the original graph, then (2) duplicate-encode them to fix adjacency matrices to a constant. (3) Duplicate-encoded feature matrices are quantized into cluster id sequences and fed into our Computation Graph Transformer. (4) Generated cluster id sequences are de-quantized back into duplicate-encoded feature matrices and fed into GNN models with the constant adjacency matrix.

6.2.4 End-to-end framework for a benchmark graph generation problem

Figure 6.2 summarizes the entire process of mapping a graph generation problem into a discrete sequence generation problem. In the training phase, we 1) sample a set of computation graphs from the input graph, 2) encode each computation graph using the duplicate encoding scheme to fix adjacency matrices, 3) quantize feature vectors to cluster ids they belong to, and finally 4) hand over a set of *(sequence of cluster ids, node label)* pairs to our new Transformer architecture to learn their distribution. In the generation phase, we follow the same process in the opposite direction: 1) the trained Transformer outputs a set of *(sequence of cluster ids, node label)* pairs, 2) we de-quantize cluster ids back into the feature vector space by replacing them with the mean feature vector of the cluster, 3) we regenerate a computation graph from each sequence of feature vectors with the adjacency matrix fixed by the duplicate encoding scheme, and finally 4) we feed the set of generated computation graphs into the GNN model we want to train or evaluate.

6.3 Proposed Work

We present the Computation Graph Transformer that encodes the computation graph structure into sequence generation process with minimal modification to the Transformer architecture. Then we check our model satisfies the privacy and scalability requirements from Problem Definition 2.

6.3.1 Computation Graph Transformer (CGT)

In this work, we extend a two-stream self-attention mechanism, XLNet [173], which modifies the Transformer architecture [152] with a causal self-attention mask to enable auto-regressive generation. Given a sequence $\mathbf{s} = [s_1, \cdots, s_T]$, the M-layered Transformer maximizes the likelihood under the

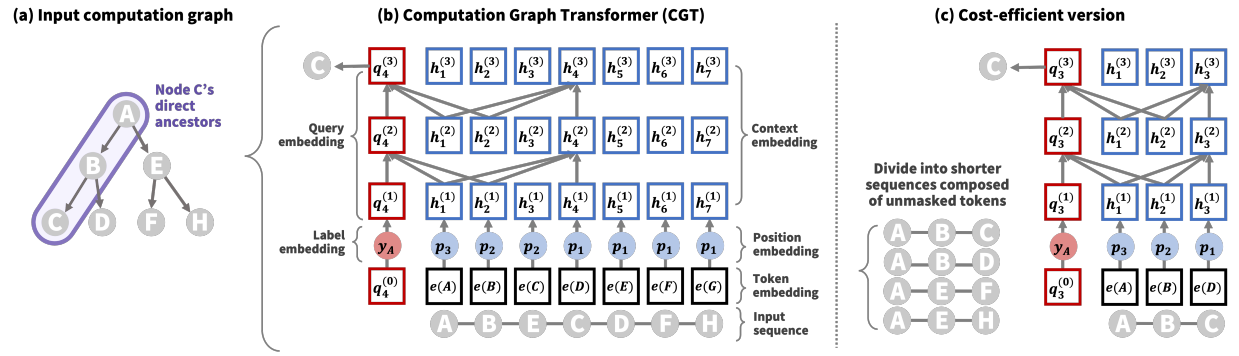


Figure 6.3: **Computation Graph Transformer (CGT).** (a,b) Given a sequence flattened from the input computation graph, CGT generates context in the forward direction. $e(s_t)$, $q_t^{(l)}$, and $h_t^{(l)}$ denote the token, query, and context embedding of t-th token at the l-th layer; $p_{l(t)}$ and y_{s_1} denote the position embeddings of t-th token and label embedding of the whole sequence, respectively. (c) The cost-efficient version of CGT divides the input sequence into shorter ones composed only of direct ancestor nodes.

forward auto-regressive factorization as follows:

$$\max_{\theta} \log p_\theta(\mathbf{s}) = \sum_{t=1}^{T} \log p_\theta(s_t|\mathbf{s}_{<t})$$
$$= \sum_{t=1}^{T} \log \frac{exp(q_\theta^{(L)}(\mathbf{s}_{1:t-1})^\top e(s_t))}{\sum_{s' \neq s_t} exp(q_\theta^{(L)}(\mathbf{s}_{1:t-1})^\top e(s'))}$$

where token embedding $e(s_t)$ maps discrete input id s_t to a randomly initialized trainable vector, and query embedding $q_\theta^{(L)}(\mathbf{s}_{1:t-1})$ encodes information until $(t-1)$-th token in the sequence. More details on the XLNet architecture can be found in the Appendix 6.7.12. Here we describe how we modify XLNet to encode computation graphs effectively.

Position embeddings: In the original Transformer architecture, each token receives a position embedding encoding its position in the sequence. In our model, sequences are flattened computation graphs (the input computation graph in Figure 6.3(a) is flattened into input sequence in Figure 6.3(b)). To encode the original computation graph structure, we provide different position embeddings to different layers in the computation graph, while nodes at the same layer share the same position embedding. When $l(t)$ denotes the layer number where t-th node is located at the original computation graph, position embedding $p_{l(t)}$ indexed by the layer number is assigned to t-th node. In Figure 6.3(b), node C, D, F and H located at the 1-st layer in the computation graph have the same position embedding p_1.

Attention masks: In the original architecture, query and context embeddings, $q_t^{(l)}$ and $h_t^{(l)}$, attend to all context embeddings $\mathbf{h}_{1:t-1}^{(l-1)}$ before t. In the computation graph, each node is sampled based on its parent node (which is sampled based on its own parent nodes) and is not directly affected by its

sibling nodes. To encode this relationship more effectively, we mask all nodes except direct ancestor nodes in the computation graph, i.e., the root node and any nodes between the root node and the leaf node. In Figure 6.3(b), node C's context/query embeddings attend only to direct ancestors, nodes A and B. Note that the number of unmasked tokens are fixed to L in our architecture because there are always $L-1$ direct ancestors in L-layered computation graphs. Based on this observation, we design a cost-efficient version of CGT that has shorter sequence length and preserves XLNet's auto-regressive masking as shown in Figure 6.3(c).

Label conditioning: Distributions of neighboring nodes are not only affected by each node's feature information but also by its label. It is well-known that GNNs improve over MLP performance by adding convolution operations that augment each node's features with neighboring node features. This improvement is commonly attributed to nodes whose feature vectors are noisy (outliers among nodes with the same label) but that are connected with "good" neighbors (whose features are well-aligned with the label). In this case, without label information, we cannot learn whether a node has feature-wise homogeneous neighbors or feature-wise heterogeneous neighbors but with the same label. In our model, query embeddings $q_t^{(0)}$ are initialized with label embeddings y_{s_1} that encode the label of the root node s_1.

6.3.2 Theoretical analysis

Our framework provides k-anonymity for node attributes and edge distributions by using k-means clustering with the minimum cluster size k [14] during the quantization phase. Note that we define edge distributions as neighboring node distributions of each node. The full proofs for the following claims can be found in Appendix 6.7.4.

Claim 1 (k-anonymity for node attributes and edge distributions)**.** *In the generated computation graphs, each node's attributes and edge distribution appear at least k times.*

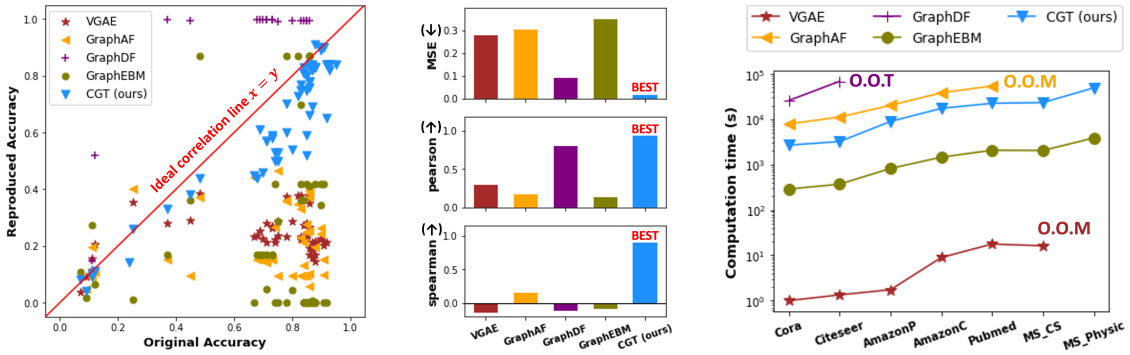
We can also provide differential privacy (DP) for node attributes and edge distributions by exploiting DP k-means clustering [21] during the quantization phase and DP stochastic gradient descent (DP-SGD) [135] to train the Transformer. Unfortunately, however, DP-SGD for Transformer networks doesn't yet work reliably in practice. Thus we cannot guarantee *strict* DP for edge distributions in practice (experimental results in Section 6.4.2 and more analysis in Appendix 6.7.4). Thus, here, we claim DP only for node attributes.

Claim 2 ((ϵ, δ)-Differential Privacy for node attributes)**.** *With probability at least $1-\delta$, our generative model A gives ϵ-differential privacy for any graph $\mathcal{G}$, any neighboring graph $\mathcal{G}_{-v}$ without any node $v \in \mathcal{G}$, and any new computation graph $\mathcal{G}_{cg}$ generated from our model as follows:*

$$e^{-\epsilon} \leq \frac{Pr[A(\mathcal{G}) = \mathcal{G}_{cg}]}{Pr[A(\mathcal{G}_{-v}) = \mathcal{G}_{cg}]} \leq e^{\epsilon}$$

Finally, we show that CGT satisfies the scalability requirement in Problem Definition 2:

Claim 3 (Scalability)**.** *To generate L-layered computation graphs with neighbor sampling number s on a graph with n nodes, computational complexity of* CGT *training is $O(s^{2L}n)$, and the cost-efficient version is $O(L^2 s^L n)$.*



(a) Reproduced GNN accuracy (b) Benchmark effectiveness (c) Scalability

Figure 6.4: **Benchmark effectiveness and scalability in graph generation.** (a) We evaluate graph generative models by how well they reproduce GNN performance from the original graph (X-axis: original accuracy) on synthetic graphs (Y-axis: reproduced accuracy). Our method is closest to $x = y$, which is ideal. (b) We measure Mean Square Error (MSE) and Pearson/Spearman correlations from results in (a). Our method shows the lowest MSE and highest correlations. (c) We measure the computation time (training + evaluation) of each graph generative model. Only our method is scalable across all datasets while showing the best performance. O.O.T denotes out-of-time (> 20 hrs) and O.O.M denotes out-of-memory errors.

6.4 Experiments

6.4.1 Experimental setting

Baselines: We choose 5 state-of-the-art graph generative models that learn graph structures with node attribute information: two VAE-based general graph generative models, VGAE [76] and GraphVAE [132] and three molecule graph generative models, GraphAF [130], GraphDF [102], and GraphEBM [139]. While VGAE encodes the large-scale whole graph at once, the other 4 graph generative models are designed to process a set of small-sized graphs. Thus we provide the original whole graph to GVAE and a set of sampled computation graphs to the other baselines, respectively.

Datasets: We evaluate on 7 public datasets — 3 citation networks (Cora, Citeseer, and Pubmed) [126], 2 co-purchase graphs (AmazonC and AmazonP) [127], and 2 co-authorship graph (MS CS and MS Physic) [127]. Note that these datasets are the largest ones the baselines have been applied on. Data statistics can be found in Appendix 6.7.15.

GNN models: We choose 9 of the most popular GNN models for benchmarking: 4 GNN models with different aggregators, GCN [75], GIN [168], SGC [162], and GAT [155], 4 GNN models with different sampling strategies, GraphSage [55], FastGCN [23], AS-GCN [68], and PASS [181], and one GNN model with PageRank operations, PPNP [79]. Descriptions of each GNN model can be found in the Appendix 6.7.11.

Table 6.1: **Privacy-Performance trade-off in graph generation.**

	Original	**No privacy**	**K-anonymity**			**DP kmean** ($\delta = 0.01$)			**DP SGD** ($\delta = 0.1$)	
			$k = 100$	$k = 500$	$k = 1000$	$\epsilon = 1$	$\epsilon = 10$	$\epsilon = 25$	$\epsilon = 10^6$	$\epsilon = 10^9$
Pearson ($\uparrow$)	1.000	0.934	0.916	0.862	0.030	0.874	0.844	0.804	0.112	0.890
Spearman ($\uparrow$)	1.000	0.935	0.947	0.812	0.018	0.869	0.805	0.807	0.116	0.959

6.4.2 Main results

In this experiment, each graph generative model learns the distributions of 7 graph datasets and generates synthetic graphs. Then we train and evaluate 9 GNN models on each pair of original and synthetic graphs, and measure Mean Square Error (MSE) and Pearson/Spearman correlations [107] between the GNN performance on each pair of graphs. As shown in Figure 6.4(a), each graph generative model compares up to 63 pairs of original and reproduced GNN performances. Unless additionally specified, K-anonymity is set to $K = 30$ across all experiments.

Benchmark effectiveness. In Figure 6.4(b), our proposed CGT shows up to 33% lower MSE, 0.80 higher Pearson and 1.03 higher Spearman correlations than all baselines. GraphVAE fails to converge, thus omitted in Figure 6.4. This results clearly show the graph generative models specialized to molecules cannot be generalized to the large-scale graphs with a high-dimensional feature space. The predicted distributions by baselines sometimes collapse to generating the the same node feature/labels across all nodes (e.g., 0% or 100% accuracy for all GNN models in Figure 6.4(a)), which is obviously not the most effective benchmark.

Scalability. Figure 6.4(c) shows scalability of each graph generative model. VGAE and GraphAF meet out-of-memory errors on MS Physic and MS CS, respectively. GraphDF takes more than 20 hours on the third smallest dataset, AmazonP. As GraphDF does not generate any meaningful graph structures even on the Cora and Citeseer datasets, we stop running GraphDF and declare an out-of-time error. These results are not surprising, given they are originally designed for small-size molecule graphs, thus having many un-parallelizable operations. Only CGT and GraphEBM scale to all graphs successfully. However, note that GraphEBM fails to learn any meaningful distributions from the original graphs as shown in Figures 6.4(a) and 6.4(b). In Appendix 6.7.5, we show our proposed CGT scales to ogbn-arxiv ($170K$ nodes and $1.2M$ edges) and ogbn-products ($2.4M$ nodes and $61.8M$ edges) successfully.

Privacy. As none of our baseline generative models provides privacy guarantees, we examine the performance-privacy trade-off across different privacy guarantees on the Cora dataset only using our method. For k-anonymity, we use the k-means clustering algorithm [14] varying the minimum cluster size k. For Differential Privacy (DP) for node attributes, we use DP k-means [21] varying the privacy cost ϵ while setting $\delta = 0.01$. In Table 6.1, higher k and smaller ϵ (i.e., stronger privacy) hinder the generative model's ability to learn the exact distributions of the original graphs; thus, the GNN performance gaps between original and generated graphs increase (lower Pearson and Spearman correlations). To provide DP for edge distributions, we use DP stochastic gradient descent [135] to train the transformer, varying the privacy cost ϵ while setting $\delta = 0.1$. In Table 6.1, even with astronomically low privacy cost ($\epsilon = 10^6$), the performance of our generative model degrades significantly. When we set $\epsilon = 10^9$ (which is impractical), we can finally see a reasonable performance. This shows the limited performance of DP SGD on the transformer architecture. Detailed GNN accuracies could be found in Appendix 6.7.7.

Table 6.2: **Comparison with simple privacy baselines that add noisy nodes and edges to the original graph.** *Node/Edge re-ident.* columns show node/edge re-identification probabilities of each privacy method. **-** denotes no privacy trick has applied.

Node attributes	**Edge distribution**	**Node re-ident.** (↓)	**Edge re-ident.** (↓)	**GCN**	**SGC**	**GIN**	**GAT**	**MSE** (↓)
-	Edge addition ($\times 2$)	100%	50%	0.82	0.82	0.80	0.55	0.021
	Edge addition ($\times 10$)	100%	10%	0.39	0.40	0.37	0.70	0.168
	Edge deletion (50%)	100%	50%	0.83	0.83	0.82	0.84	0.001
	Edge deletion (100%)	100%	0%	0.73	0.73	0.73	0.72	0.014
Noise addition ($\times 5$)	-	20%	100%	0.82	0.82	0.82	0.18	0.106
	Edge addition ($\times 2$)	20%	50%	0.67	0.67	0.68	0.07	0.169
	Edge addition ($\times 10$)	20%	10%	0.07	0.30	0.31	0.07	0.449
	Edge deletion (50%)	20%	50%	0.78	0.77	0.77	0.15	0.120
	Edge deletion (100%)	20%	0%	0.39	0.40	0.38	0.11	0.291
K-anonymity (5)	K-anonymity (5)	20%	20%	0.83	0.82	0.83	0.83	0.001
K-anonymity (100)	K-anonymity (100)	1%	1%	0.75	0.74	0.76	0.74	0.010
K-anonymity (500)	K-anonymity (500)	0.2%	0.2%	0.52	0.49	0.51	0.52	0.114
K-anonymity (1000)	K-anonymity (1000)	0.1%	0.1%	0.12	0.12	0.11	0.08	0.548
Original graph		100%	100%	0.86	0.85	0.85	0.83	0.000

To verify the effectiveness of K-anonymity in terms of re-identification attacks, we compare it with simple privacy baselines that add noise on nodes/edges as follow:

- **Edge addition:** We add x times more random edges than the original number of edges. Given a corrupted graph, an original edge can be re-identified with a probability of $1/x$.
- **Edge deletion:** We delete $x\%$ of edges from the original graph. Given a corrupted graph, an original edge can be re-identified with a probability of $(100 - x)/100\%$.
- **Noise addition to node attributes:** Given a binary node attribute vector, when s elements in the vector are '1', we randomly flip '0' to '1' for xs times. Given a corrupted graph, an original attribute can be re-identified with a probability of $1/x$.
- **K-anonymity:** As described in the paper, given a corrupted graph, a node attribute vector and an edge distribution of a node can be re-identified with a probability of $1/K$ (Claim 1 in the original paper).

We run four GNN models (GCN, SGC, GIN, GAT) with different privacy approaches on the Cora dataset and computed MSE between GNN performance on the original and synthetic (corrupted) graphs. As presented in the table, K-anonymity (K=5) shows the smallest MSE (0.001) while providing stronger privacy guarantees (20% re-identification for both node and edge distribution) than the baselines of adding noise. For instance, the edge deletion (50%, 3rd row) also shows the smallest MSE (0.001), but this approach does not guarantee any privacy for node attributes and provides a 50% chance of successful edge re-identification. Note that K-anonymity ($K = 100$), which provides a 1% re-identification ratio, shows lower MSE (0.010) than most of the other baselines.

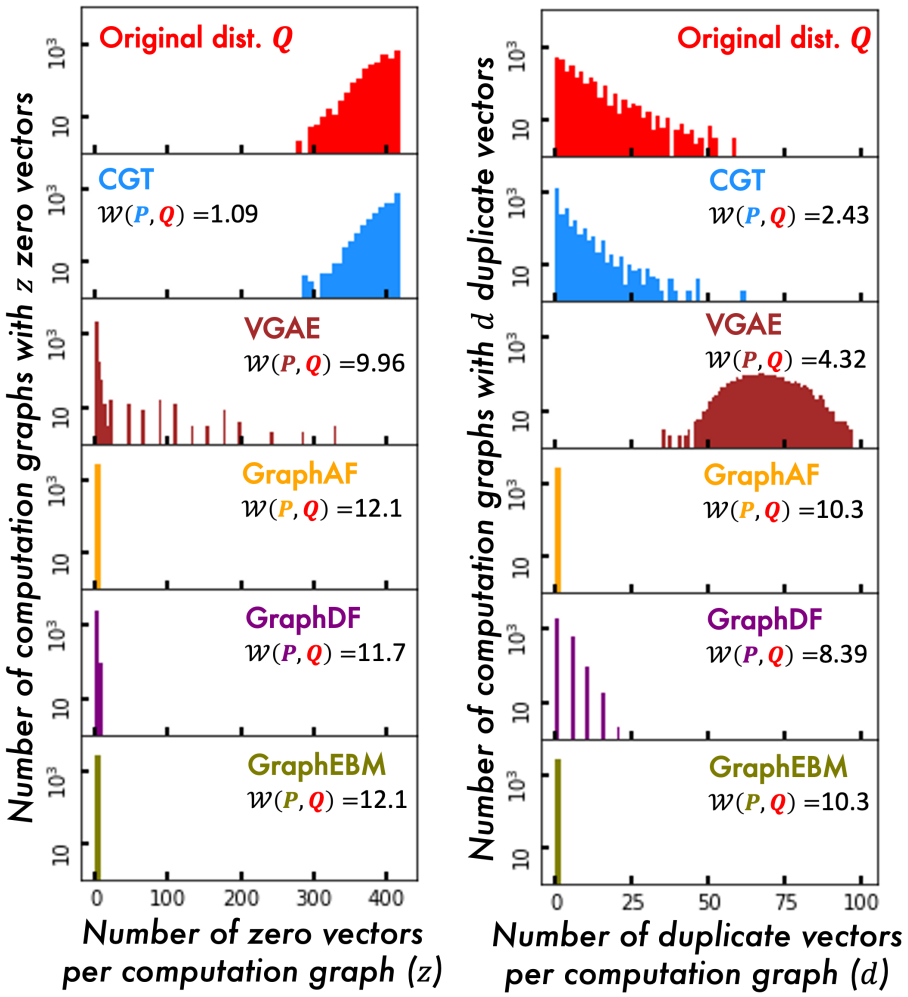
These results are not surprising, according to a recent work [36] that analyzes noise required for privacy guarantees on graph data. [36] shows that the noise addition approach does not work well for low-degree nodes and requires many mutations to provide strong privacy guarantees. However, as we stated in the limitations of this work (Appendix 6.7.2), we need stronger privacy guarantees than K-anonymity to use the generator in practice. We believe that by formally defining the benchmark graph generation problem and providing an end-to-end framework where we can easily adapt off-the-shelf state-of-the-art privacy modules (e.g., differential privacy), we can promote more research in this direction.

6.4.3 Graph statistics.

Given a source graph, our method generates a set of computation graphs without any node ids. In other words, attackers cannot merge the generated computation graphs to restore the original graph and re-identify node information. Thus, instead of traditional graph statistics such as orbit counts or clustering coefficients that rely on the global view of graphs, we define new graph statistics for computation graphs that are encoded by the duplicate scheme.

Duplicate scheme fixes adjacency matrices across all computation graphs by infusing structural information (originally encoded in adjacency matrices) into feature matrices.

- **Number of zero vectors:** In duplicate-encoded feature matrices, zero vectors correspond to null nodes that are padded when a node has fewer neighbors than a sampling neighbor number. This metric is inversely proportional to *node degree distributions* of the underlying graph.



(a) (b)

Figure 6.5: **CGT preserves distributions of graph statistics in generated graphs.** Duplicate encoding encodes graph structure into feature matrices of computation graphs. In each computation graph, # zero vectors is inversely proportional to node degree, while # redundant vectors is proportional to edge density. We measure Wasserstein distance $\mathcal{W}(P, Q)$ between the original distribution Q and the distribution P generated by each baseline.

- **Number of duplicate feature vectors:** Feature vectors are duplicated when nodes share neighbors. This metric is proportional to number of cycles in a computation graph, indicating the *edge density* of the underlying graph.

For fair comparison, we provide the same set of duplicate-encoded computation graphs to each baseline as CGT, then compute the two proxy graph statistics we described above in each generated computation graph. In Figure 6.5, we plot the distributions of this two statistics generated by each baseline. Only our method successfully preserves the distributions of the graph statistics on the generated computation graphs with up to 11.01 smaller Wasserstein distance than other baselines.

In Figure 6.5(a), the competing baselines have basically no zero vectors in the computation graphs. In the set of duplicate-encoded computation graphs given to each baseline, the input graph structures are fixed with variable feature matrices. GraphAF, GraphDF, and GraphEBM all fail to learn the distributions of feature vectors (i.e., the number of zero vectors in each computation graph) and generate highly dense feature matrices for almost all computation graphs. This shows that the existing graph generative models cannot jointly learn the distribution of node features with graph structures.

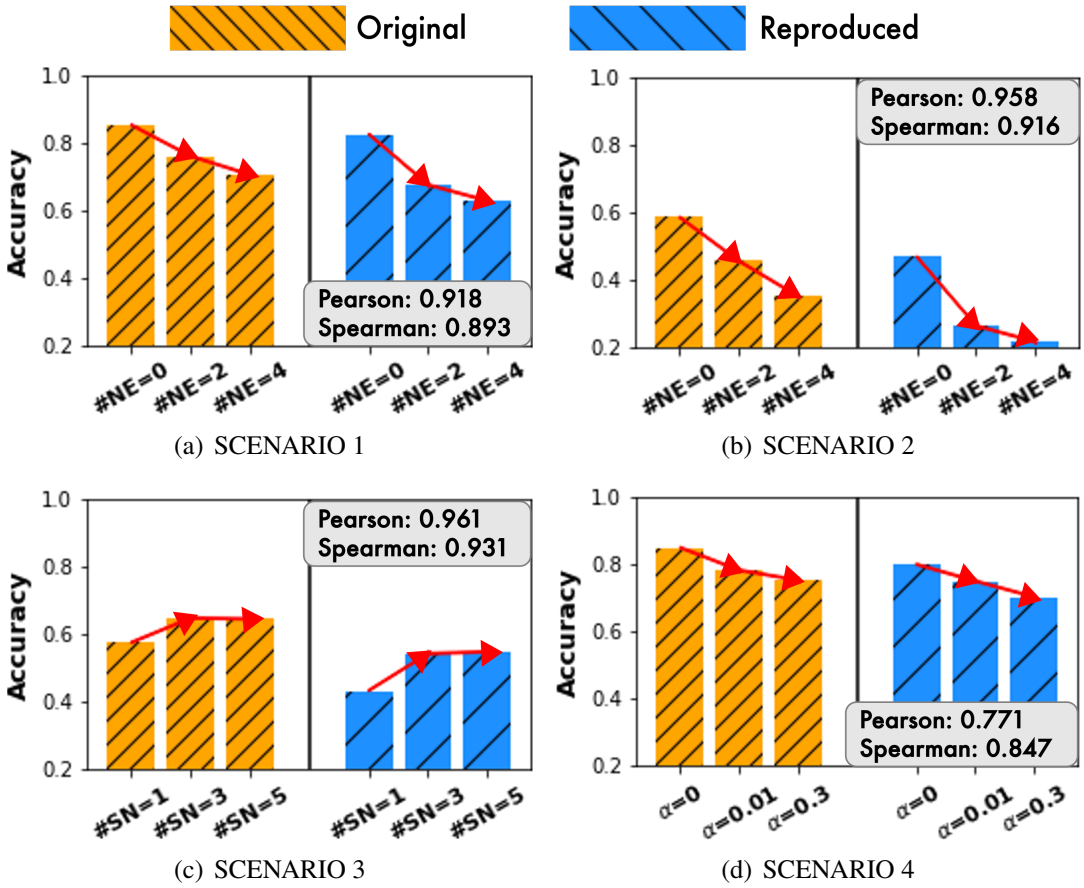
6.4.4 Various scenarios to evaluate benchmark effectiveness

To study the benchmark effectiveness of our generative model in depth, we design 4 different scenarios where GNN performance varies widely. In each scenario, we make 3 variations of an original graph and evaluate whether our graph generative model can reproduce these variations. In Figure 6.6, we report average performance of 4 GNN models on each variation. *We expect the performance trends across variations of the original graph to be reproduced across variations of synthetic graphs.* Due to the space limitation, we present results on the AmazonP dataset in Figure 6.6. Other datasets with detailed GNN accuracies can be found in Appendix 6.7.9.

SCENARIO 1: noisy edges on aggregation strategies. We choose 4 GNN models with different aggregation strategies: GCN with mean aggregator, GIN with sum aggregator, SGC with linear aggregator, and GAT with attention aggregator. We make 3 variations of the original graph by adding different numbers of noisy edges ($\#NE$) to each node. In Figure 6.6(a), when more noisy edges are added, the GNN accuracy drops in the original graph. These trends are exactly reproduced on the generated graph with 0.918 Pearson correlation, showing our method successfully reproduces different amount of noisy edges in the original graphs.

SCENARIO 2: noisy edges on neighbor sampling. We choose 4 GNN models with different neighbor sampling strategies: GraphSage with random sampling, FastGCN with heuristic layer-wise sampling, AS-GCN with trainable layer-wise sampling, and PASS with trainable node-wise sampling. We make 3 variations of the original graph by adding noisy edges ($\#NE$) as in SCENARIO 1. In Figure 6.6(b), when more noisy edges are added, the sampling accuracy drops in the original graph. This trend is reproduced in the generated graph, showing 0.958 Pearson correlation.

SCENARIO 3: different sampling numbers on neighbor sampling. We choose the same 4 GNN models with different neighbor sampling strategies as in SCENARIO 2. We make 3 variations of the original graph by changing the number of sampled neighbor nodes ($\#SN$). As shown in Figure 6.6(c), trends among original graphs — GNN performance increases sharply from $\#SN = 1$ to $\#SN = 3$, then slowly from $\#SN = 3$ to $\#SN = 5$— are successfully captured in the generated graphs with up to 0.961 Pearson correlation. This shows CGT reproduces the neighbor distributions successfully.



(a) SCENARIO 1 (b) SCENARIO 2

(c) SCENARIO 3 (d) SCENARIO 4

Figure 6.6: **CGT reproduces GNN performance changes with different number of noisy edges ($\#NE$), sampled neighbors ($\#SN$), and different amount of distribution shifts (α) successfully.**

SCENARIO 4: distribution shift. [199] proposed a biased training set sampler to examine each GNN model's robustness to distribution shift between the training/test time. The biased sampler picks a few seed nodes and finds nearby nodes using the Personalized PageRank vectors [110] $\pi_{\text{ppr}} = (I - (1-\alpha)\tilde{A})^{-1}$ with decaying coefficient α, then uses them to compose a biased training set. The higher α is, the larger the distribution is shifted between training/test sets. We make 3 variations of the original graph by varying α and check how 4 different GNN models, GCN, SGC, GAT, and PPNP, deal with the biased training set. In Figure 6.6(d), the performance of GNN models drops as α increases on the original graphs. This trend is reproduced on generated graphs, showing that CGT can capture train/test distribution shifts successfully.

6.4.5 Ablation study

To show the importance of each component in our proposed model, we run four ablation studies: CGT without 1) label conditioning, 2) position embedding trick, 3) masked attention trick, and 4) all three modules (i.e., original Transformer) We run 9 GNN models on 3 datasets (Cora, Citeseer,

Table 6.3: **Ablation study**

Model	MSE (↓)	Pearson (↑)	Spearman (↑)
w/o Label	0.067	0.592	0.591
w/o Position	0.072	0.411	0.413
w/o Attention	0.085	0.329	0.286
w/o All	0.034	0.739	0.574
CGT (Ours)	0.017	0.943	0.914

Pubmed) and compare the 9×3 pairs of GNN accuracies on original and generated graphs. When we remove the position embedding trick, we provide the different position embeddings to all nodes in a computation graph, following the original transformer architecture. When we remove attention masks from our model, the transformer attends all other nodes in the computation graphs to compute the context embeddings. As shown in Table 6.3, removing any component negatively impacts the model performance.

6.5 Related Work

Traditional graph generative models extract common patterns among real-world graphs (e.g. nodes/edge/triangle counts, degree distribution, graph diameter, clustering coefficient) [19] and generate synthetic graphs following a few heuristic rules [3, 37, 84, 85]. However, they cannot generate unseen patterns on synthetic graphs [183]. More importantly, most of them generate only graph structures with boolean node attributes [38]. **General-purpose deep graph generative models** exploit GAN [50], VAE [74], and RNN [188] to learn graph distributions [54]. Most of them focus on learning graph structures [52, 92, 132, 183], thus their evaluation metrics are graph statistics such as orbit counts, degree coefficients, and clustering coefficients which do not consider quality of generated node attributes and labels. **Molecule graph generative models** are actively studied for generating promising candidate molecules using VAE [71], GAN [28], RNN [115], and recently invertible flow models [102, 130]. However, most of their architectures are specialized to small-scaled molecule graphs (e.g., 38 nodes per graph in the ZINC datasets) with low-dimensional attribute space (e.g., 9 node attributes indicating atom types) and distinct molecule-related information (e.g., SMILES representation or chemical structures such as bonds and rings) [139].

6.6 Summary

We propose a new graph generative model CGT that (1) generates effective benchmark graphs on which GNNs show similar performance as on the source graphs, (2) scales to process large-scale graphs, and (3) incorporates off-the-shelf privacy modules to guarantee end-user privacy of the generated graph. We hope our work sparks further research to address the limited access to

(highly proprietary) real-world graphs, enabling the community to develop new GNN models on challenging, realistic problems.

6.7 Appendix

6.7.1 Reproducibility

Our code is publicly available [1]. Dataset information can be found in Appendix 6.7.15 and can be downloaded from the open data source [2]. Open source libraries for DP K-means and DP-SGD we used are listed in Appendix 6.7.13. Baseline graph generative models and their open source libraries are described in Appendix 6.7.15. GNN models we benchmark during experiments and their open source libraries are described in Appendix 6.7.11.

6.7.2 Limitation of the study

This paper shows that clustering-based solutions can achieve k-anonymity privacy guarantees. We stress, however, that implementing a real-world system with strong privacy guarantees will need to consider many other aspects beyond the scope of this paper. We leave as future work the study of whether we can combine stronger privacy guarantees with those of k-anonymity to enhance privacy protection

6.7.3 Computation graph sampling in GNN training

The main challenge of adapting GNNs to large-scale graphs is that GNNs expand neighbors recursively in the aggregation operations, leading to high computation and memory footprints. For instance, if the graph is dense or has many high degree nodes, GNNs need to aggregate a huge number of neighbors for most of the training/test examples. To alleviate this neighbor explosion problem, GraphSage [55] proposed to sample a fixed number of neighbors in the aggregation operation, thereby regulating the computation time and memory usage.

To train a L-layered GNN model with a user-specified neighbor sampling number s, a computation graph is generated for each node in a top-down manner ($l : L \rightarrow 1$): A target node v is located at the L-th layer; the target node samples s neighbors, and the sampled s nodes are located at the $(L-1)$-th layer; each node samples s neighbors, and the sampled s^2 nodes are located at the $(L-2)$-th layer; repeat until the 1-st layer. When the neighborhood is smaller than s, we sample all existing neighbors of the node. Which nodes to sample varies across different sampling algorithms. The sampling algorithms for GNNs broadly fall into two categories: node-wise sampling and layer-wise sampling.

- **Node-Wise Sampling.** The sampling distribution $q(j|i)$ is defined as a probability of sampling node v_j given a source node v_i. In node-wise sampling, each node samples k neighbors

[1] https://github.com/minjiyoon/CGT
[2] https://github.com/shchur/gnn-benchmark

from its sampling distribution, then the total number of nodes in the l-th layer becomes $O(k^l)$. GraphSage [55] is one of the most well-known node-wise sampling method with the uniform sampling distribution $q(j|i) = \frac{1}{N(i)}$. GCN-BS [97] introduces a variance reduced sampler based on multi-armed bandits, and PASS [181] proposes a performance-adaptive node-wise sampler.

- **Layer-Wise Sampling.** To alleviate the exponential neighbor expansion $O(k^l)$ of the node-wise samplers, layer-wise samplers define the sampling distribution $q(j|i_1, \cdots, i_n)$ as a probability of sampling node v_j given a set of nodes $\{v_k\}_{k=i_1}^{i_n}$ in the previous layer. Each layer samples k neighbors from their sampling distribution $q(j|i_1, \cdots, i_n)$, then the number of sampled nodes in each layer becomes $O(k)$. FastGCN [23] defines $q(j|i_1, \cdots, i_n)$ proportional to the degree of the target node v_j, thus every layer has independent-identical-distributions. LADIES [202] adopts the same iid as FastGCN but limits the sampling domain to the neighborhood of the sampler layer. AS-GCN [68] parameterizes the sampling distributions $q(j|i_1, i_2, \ldots, i_n)$ with a learnable linear function. While the layer-wise samplers successfully regulate the neighbor expansion, they suffer from sparse connection problems — some nodes fail to sample any neighbors while other nodes sample their neighbors repeatedly in a given layer.

Note that the layer-wise samplers also define a maximum number of neighbors to sample (but per each layer) and the depth of computation graphs as the depth of the GNN model. All sampling methods we describe above can be applied to our computation graph sampling module described in Section 6.2.2. As the depth of computation graph L is decided by the depth of GNN models, oversmoothing [87] or oversquashing [4] could happen with the deep GNN models. To handle this issue, [190] proposes to disentangle the depth of computation graphs and the depth of GNN models, then limit the computation graph sizes to small to avoid oversmoothing/oversquashing.

There are many different clustering or subgraph sampling methodologies other than what we described above. Note that, even after we get subgraphs using any clustering/subgraph sampling methods, to do message-passing under GCN models, each node eventually has a tree-structure-shaped computation graph that is composed of nodes engaged in the node's embedding computation. In other words, CGT receives subgraphs sampled by ClusterGCN [25] and GraphSAINT [189] and extracts a computation graph for each node (in this case, we can set the sampling number as the maximum degree in the subgraph not to lose any further neighbors by sampling). GNNAutoScale [42] and IGLU [108] are recently proposed frameworks for scaling arbitrary message-passing GNNs to large graphs, as an alternative paradigm to neighbor sampling. As our method adopts neighbor sampling — the most common way to deal with the scalability issue of GNNs so far — we cannot directly apply our graph benchmark generation method to these methods. This is an interesting avenue for future work.

6.7.4 Proof of privacy and scalability claims

Claim 1 (k-Anonymity for node attributes and edge distributions)**.** *In the generated computation graphs, each node attribute and edge distribution appear at least k times, respectively.*

Proof. In the quantization phase, we use the k-means clustering algorithm [14] with a minimum cluster size k. Then each node id is replaced with the id of the cluster it belongs to, reducing the original $(n \times n)$ graph into a $(m \times m)$ hypergraph where $m = n/k$ is the number of clusters. Then

Computation Graph Transformer learns edge distributions among m hyper nodes (i.e., clusters) and generates a new $(m \times m)$ hypergraph. In the hypergraph, there are at most m different node attributes and m different edge distributions. During the de-quantization phase, a $(m \times m)$ hypergraph is mapped back to a $(n \times n)$ graph by letting k nodes in each cluster follow their cluster's node attributes/edge distributions as follows: k nodes in the same cluster will have the same feature vector that is the average feature vector of original nodes belonging to the cluster. When s denotes the number of sampled neighbor nodes, each node samples s clusters (with replacement) following its cluster's edge distributions among m clusters. When a node samples cluster i, it will be connected to one of nodes in the cluster i randomly. At the end, each node will have s neighbor nodes randomly sampled from s clusters the node samples with the cluster's edge distribution, respectively. Likewise, all k nodes belonging to the same cluster will sample neighbors following the same edge distributions. Thus each node attribute and edge distribution appear at least k times in a generated graph. □

Claim 2 ((ϵ, δ)-Differential Privacy for node attributes)**.** *With probability at least $1 - \delta$, our generative model A gives ϵ-differential privacy for any graph $\mathcal{G}$, any neighboring graph $\mathcal{G}_{-v}$ without any node $v \in \mathcal{G}$, and any new computation graph $\mathcal{G}_{cg}$ generated from our model as follows:*

$$e^{-\epsilon} \leq \frac{Pr[A(\mathcal{G}) = \mathcal{G}_{cg}]}{Pr[A(\mathcal{G}_{-v}) = \mathcal{G}_{cg}]} \leq e^{\epsilon}$$

Proof. $\mathcal{G}_{-v}$ denotes neighboring graphs to the original one $\mathcal{G}$, but without a specific node v. During the quantization phase, we use (ϵ, δ)-differential private k-means clustering algorithm on node features [21]. Then clustering results are differentially private with regard to each node features. In the generated graphs, each node feature is decided by the clustering results (i.e., the average feature vector of nodes belonging to the same cluster). Then, by looking at the generated node features, one cannot tell whether any individual node feature was included in the original dataset or not. □

Remark 1 ((ϵ, δ)-Differential Privacy for edge distributions)**.** In our model, individual nodes' edge distributions are learned and generated by the transformer. When we use (ϵ, δ)-differential private stochastic gradient descent (DP-SGD) [135] to train the transformer, the transformer becomes differentially private in the sense that by looking at the output (generated edge distributions), one cannot tell whether any individual node's edge distribution (input to the transformer) was included in the original dataset or not. If we have DP-SGD that can train transformers successfully with reasonably small ϵ and δ, we can guarantee (ϵ, δ)-differential privacy for edge distribution of any graph generated by our generative model. However, as we show in Section 6.4.2, current DP-SGD is not stable yet for transformer training, leading to very coarse or impractical privacy guarantees.

Claim 3 (Scalability)**.** *When we aim to generate L-layered computation graphs with neighbor sampling number s on a graph with n nodes, computational complexity of* CGT *training is $O(s^{2L}n)$, and that of the cost-efficient version is $O(L^2 s^L n)$.*

Proof. During k-means, we randomly sample n_k node features to compute the cluster centers. Then we map each feature vector to the closest cluster center. By sampling n_k nodes, we limit the k-mean computation cost to $O(n_k^2)$. The sequence flattened from each computation graph is

Table 6.4: **CGT on ogbn-arxiv and ogbn-products:** *Training time (hr)* column denotes the total training/generation time of CGT.

Dataset	Node num	Edge num	Noise num	Model	Original acc.	Generated acc/	MSE	Training time (hr)	Pearson
ogbn-arxiv	169,343	1,166,243	0	GCN	0.69	0.7	0.00032	1.1	0.989
				SGC	0.68	0.7			
				GIN	0.69	0.71			
				GAT	0.69	0.71			
			2	GCN	0.58	0.6	0.00015	1.7	
				SGC	0.57	0.58			
				GIN	0.61	0.62			
				GAT	0.62	0.62			
			4	GCN	0.53	0.55	0.00015	2.8	
				SGC	0.54	0.53			
				GIN	0.56	0.56			
				GAT	0.57	0.58			
ogbn-products	2,449,029	61,859,140	0	GCN	0.87	0.89	0.00258	14.7	
				SGC	0.75	0.84			
				GIN	0.86	0.89			
				GAT	0.87	0.9			

Table 6.5: **CGT as training/test set generators.** We replace the original training/test sets of the target dataset (Cora) with irrelevant graphs (Citeseer or Pubmed) and synthetic Cora generated by our proposed CGT.

Train set	Test set	Accuracy
Cora	Cora	0.86
Citeseer	Cora	0.14
Pubmed	Cora	0.09
Synthetic Cora (CGT)	Cora	0.77
Cora	Synthetic Cora (CGT)	0.74
Synthetic Cora (CGT)	Synthetic Cora (CGT)	0.76

$O(1 + s + \cdots + s^L)$ and the number of sequences (computation graphs) is $O(n)$. Then the training time of the transformer is proportional to $O(s^{2L}n)$. In total, the complexity is $O(s^{2L}n + n_k^2)$. As $s^{2L}n >> n_k^2$, the final computation complexity becomes $O(s^{2L}n)$. In the cost-efficient version, the length of sequences (composed only of direct ancestor nodes) is reduced to L. However, the number of sequences increases to $s^L n$ because each nodes has one computation graph composed of s^L shortened sequences. Then the final computation complexity become $O(L^2 s^L n)$. □

6.7.5 CGT on ogbn-arxiv and ogbn-products

To examine its scalability, we run CGT on two large-scale datasets, ogbn-arxiv and ogbn-products [63]. We run CGT on 4 NVIDIA TITAN X GPUs with 12 GB memory size with sampling number 5 and $K = 30$ for K-anonymity. In Table 6.4, CGT takes 1.1 hours for ogbn-arxiv with $170K$ nodes and $1.2M$ edges, while taking 14.7 hours for ogbn-products with $2.4M$ nodes and $61.8M$ edges. This shows CGT's strong scalability. In terms of benchmark effectiveness, CGT shows low MSE (up to 1.5×10^{-4}) and high Pearson correlation (0.989). Note that we could not compare with other baselines as they all fail to scale even on MS Physic dataset with with $35K$ nodes and $248K$ edges (Figure 6.4(c)).

6.7.6 CGT as training/test set generators

In this experiment, we train GNNs on synthetic graphs generated by CGT and test them on real graphs, and vice versa. For comparison, we train GCN on the two independent graphs (Citeseer and Pubmed) and test on the target graph (Cora). Since the feature dimensions of Citeseer and Pubmed differ from those of Cora, we mapped the original node feature vectors to Cora's feature dimension using PCA. The results in the Table 6.5 demonstrate that our CGT generates synthetic graphs that follow Cora's distribution and preserve high accuracy, whereas GCN models trained on Citeseer and Pubmed show low accuracy on Cora. The accuracy drop induced by CGT is mainly due to privacy, as we provided 30-Anonymity in this experiment. We conducted a similar experiment in Section 6.4.4 SCENARIO 4, where we prepared different distributions for the training and test sets of GNNs. As the distribution shift becomes larger, the performance of GNNs drops. Our proposed

CGT successfully reproduces this distribution shift, and thus, it also reproduces the performance drop in the generated graphs.

6.7.7 Detailed GNN performance in the privacy experiment in Section 6.4.2

Table 6.6 shows detailed privacy-GNN performance trade-off on the Cora dataset. In k-anonymity, higher k (i.e., more nodes in the same clusters, thus stronger privacy) hinders the generative model's ability to learn the exact distributions of the original graphs, and the GNN performance gaps between original and generated graphs increase, showing lower Pearson and Spearman coefficients. DP kmeans shows higher Pearson and Spearman coefficients with smaller ϵ values (i.e., stronger privacy). However, when we examine the detailed GNN performance, we observe that GNN accuracy is significantly lower with smaller ϵ values. For your convenience, we compare their MSE from the original accuracy as well as the correlation coefficients in Table 6.6: MSE is descreasing from $0.134(\epsilon = 1)$ to $0.093(\epsilon = 10)$ and $0.063(\epsilon = 25)$. Stronger privacy can lead to higher correlations as DP k-means can remove noise in graphs (while hiding outliers for privacy) and capture representative distributions from the original graph more effectively. While DP kmeans is capable of providing reasonable privacy to node attribute distributions, DP-SGD is impractical, showing low GNN performance even with astronomically low privacy cost ($\epsilon = 10^6$) as explained in Section 6.3.2. Note that reasonable ϵ values typically range between 0.1 and 5.

6.7.8 Additional experiments on graph statistics

Figure 6.7 shows distributions of graph statistics on computation graphs sampled from the original/quantized/generated graphs. Quantized graphs are graphs after the quantization process: each feature vector is replaced by the mean feature vector of a cluster it belongs to, and adjacency matrices are a constant encoded by the duplicate encoding scheme. Quantized graphs are input to CGT, and generated graphs are output from CGT as presented in Figure 6.2. While converting from original graphs to quantized graphs, CGT trades off some of the graph statistics information for k-anonymity privacy benefits. In Figure 6.7, we can see distributions of graphs statistics have changed slightly from original graphs to quantized graphs. Then CGT learns distributions of graph statistics on the quantized graphs and generates synthetic graphs. The variations given by CGT are presented as differences in distributions between quantized and generated graphs in Figure 6.7.

6.7.9 Detailed GNN performance in the benchmark effectiveness experiment in Section 6.4.4

As nodes are the minimum unit in graphs that compose edges or subgraphs, we can generate subgraphs for edges by merging computation graphs of their component nodes. Here we show link prediction results on original graphs are also preserved successfully on our generated graphs. We run GCN, SGC, GIN, and GAT on graphs, followed by Dot product or MLP to predict link probabilities. Table 6.7 shows Pearson and Spearman correlations across 8 different combinations of link prediction models (4 GNN models $\times$ 2 link predictors) on each dataset and across the whole

Table 6.6: **Privacy-Performance trade-off in graph generation on the Cora dataset.**

#NE	model	Original	No privacy	K-anonymity			DP kmean ($\delta = 0.01$)			DP SGD ($\delta = 0.1$)	
				$k = 100$	$k = 500$	$k = 1000$	$\epsilon = 1$	$\epsilon = 10$	$\epsilon = 25$	$\epsilon = 10^6$	$\epsilon = 10^9$
0	**GCN**	0.860	0.760	0.750	0.520	0.120	0.530	0.570	0.650	0.130	0.640
	SGC	0.850	0.750	0.740	0.490	0.120	0.510	0.590	0.620	0.150	0.620
	GIN	0.850	0.750	0.760	0.510	0.110	0.520	0.570	0.650	0.140	0.640
	GAT	0.830	0.750	0.740	0.520	0.080	0.440	0.560	0.640	0.140	0.610
2	**GCN**	0.770	0.680	0.570	0.380	0.110	0.500	0.400	0.450	0.110	0.580
	SGC	0.770	0.680	0.580	0.360	0.080	0.350	0.410	0.450	0.140	0.570
	GIN	0.780	0.670	0.590	0.390	0.140	0.390	0.410	0.470	0.140	0.580
	GAT	0.680	0.660	0.560	0.380	0.110	0.350	0.390	0.430	0.120	0.530
4	**GCN**	0.720	0.610	0.510	0.280	0.090	0.280	0.390	0.430	0.100	0.410
	SGC	0.720	0.600	0.500	0.280	0.110	0.300	0.410	0.450	0.140	0.410
	GIN	0.660	0.590	0.480	0.300	0.160	0.320	0.410	0.460	0.150	0.400
	GAT	0.600	0.570	0.470	0.290	0.080	0.250	0.370	0.450	0.140	0.380
Pearson		1.000	0.934	0.916	0.862	0.030	0.874	0.844	0.804	0.112	0.890
Spearman		1.000	0.935	0.947	0.812	0.018	0.869	0.805	0.807	0.116	0.959
MSE		0.000	0.008	0.026	0.136	0.427	0.134	0.093	0.063	0.396	0.053

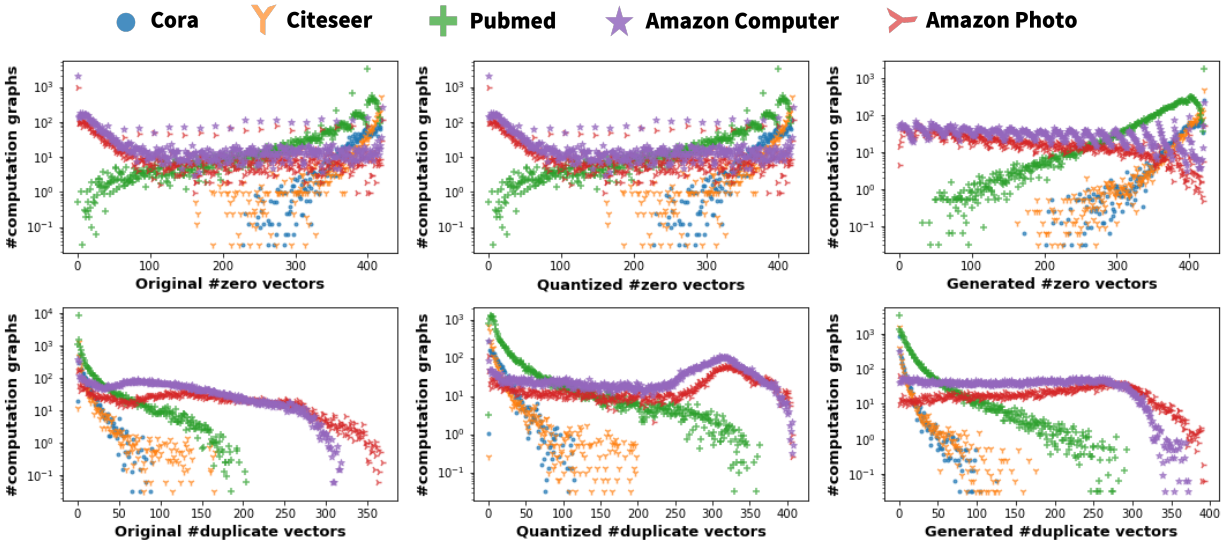


Figure 6.7: **CGT preserves distributions of graph statistics in generated graphs for each dataset:** While converting from original graphs to quantized graphs, CGT loses some of graph statistics information for k-anonymity privacy benefit. The variations given by CGT are presented as differences in distributions between quantized and generated graphs. X-axis denotes the number of zero vectors (z) and the number of duplicate vectors (d) per computation graph, respectively. Y-axis denotes the number of computation graphs with z zero vectors and d duplicate vectors, respectively.

datasets. Our model generates graphs that substitute original graphs successfully, preserving the ranking of GNN link prediction performance with 0.754 Spearman correlation across the datasets.

6.7.10 Detailed GNN performance in the ablation study in Section 6.4.5

Table 6.8 shows CGT without label conditioning (conditioning on the label of the root node of the computation graph), positional embedding trick (giving the same positional embedding to nodes at the same layers on the computation graph), masked attention trick (attended only on direct ancestor nodes on the computation graph), and all modules (pure Transformer) respectively. Note that this experiment is done on the original version of CGT (not the cost-efficient version in Figure 6.3(c)). When we remove the positional embedding trick, we provide the different positional embeddings to all nodes in a computation graph, following the original transformer architecture. When we remove attention masks from our model, the transformer attends all other nodes in the computation graphs to compute the context embeddings.

6.7.11 GNN models used in the benchmark effectiveness experiment

We choose four different GNN models with different aggregation strategies to examine the effect of noisy edges on the aggregation strategies: GCN [75] with mean aggregator, GIN [168] with sum aggregator, SGC [162] with linear aggregator, and GAT [155] with attention aggregator. We choose four different GNN models with different neighbor sampling strategies to examine the effect of

Table 6.7: **GNN performance on link prediction.**

Dataset	predictor	model	Original	std	Cluster	std	Generated	std	pearson	spearman
Cora	**Dot**	**GCN**	0.720	0.010	0.770	0.009	0.680	0.012	0.781	0.741
		SGC	0.710	0.025	0.760	0.005	0.660	0.016		
		GIN	0.820	0.015	0.760	0.016	0.650	0.022		
		GAT	0.810	0.002	0.810	0.007	0.730	0.015		
	MLP	**GCN**	0.540	0.005	0.620	0.012	0.510	0.01		
		SGC	0.530	0.016	0.590	0.042	0.510	0.006		
		GIN	0.530	0.012	0.690	0.016	0.630	0.017		
		GAT	0.550	0.003	0.660	0.013	0.610	0.034		
Dataset	**predictor**	**model**	**Original**	**std**	**Cluster**	**std**	**Generated**	**std**	**pearson**	**spearman**
Citeseer	**Dot**	**GCN**	0.690	0.007	0.740	0.009	0.650	0.026	0.808	0.824
		SGC	0.700	0.003	0.730	0.013	0.670	0.022		
		GIN	0.830	0.008	0.720	0.003	0.650	0.01		
		GAT	0.750	0.005	0.780	0.012	0.680	0.021		
	MLP	**GCN**	0.580	0.005	0.650	0.012	0.590	0.01		
		SGC	0.580	0.008	0.640	0.025	0.590	0.023		
		GIN	0.570	0.011	0.720	0.012	0.610	0.024		
		GAT	0.610	0.005	0.680	0.001	0.620	0.009		
Dataset	**predictor**	**model**	**Original**	**std**	**Cluster**	**std**	**Generated**	**std**	**pearson**	**spearman**
Pubmed	**Dot**	**GCN**	0.800	0.018	0.810	0.005	0.670	0.019	0.725	0.420
		SGC	0.790	0.002	0.780	0.006	0.660	0.004		
		GIN	0.800	0.008	0.760	0.008	0.650	0.009		
		GAT	0.860	0.003	0.850	0.007	0.720	0.008		
	MLP	**GCN**	0.760	0.003	0.770	0.012	0.640	0.017		
		SGC	0.770	0.006	0.770	0.006	0.610	0.008		
		GIN	0.750	0.004	0.790	0.014	0.660	0.004		
		GAT	0.750	0.004	0.850	0.019	0.660	0.011		
Dataset	**predictor**	**model**	**Original**	**std**	**Cluster**	**std**	**Generated**	**std**	**pearson**	**spearman**
Amazon Computer	**Dot**	**GCN**	0.790	0.010	0.850	0.026	0.810	0.008	0.652	0.559
		SGC	0.760	0.005	0.770	0.030	0.730	0.025		
		GIN	0.800	0.013	0.880	0.004	0.830	0.005		
		GAT	0.750	0.057	0.840	0.014	0.560	0.08		
	MLP	**GCN**	0.810	0.005	0.890	0.005	0.830	0.012		
		SGC	0.800	0.000	0.850	0.020	0.730	0.021		
		GIN	0.800	0.003	0.890	0.010	0.810	0.01		
		GAT	0.860	0.005	0.910	0.005	0.800	0.005		
Dataset	**predictor**	**model**	**Original**	**std**	**Cluster**	**std**	**Generated**	**std**	**pearson**	**spearman**
Amazon Photo	**Dot**	**GCN**	0.890	0.011	0.920	0.005	0.860	0.016	0.887	0.443
		SGC	0.810	0.014	0.840	0.015	0.780	0.011		
		GIN	0.810	0.007	0.910	0.006	0.880	0.002		
		GAT	0.530	0.023	0.740	0.151	0.660	0.134		
	MLP	**GCN**	0.870	0.006	0.930	0.006	0.890	0.001		
		SGC	0.840	0.010	0.900	0.012	0.810	0.015		
		GIN	0.850	0.006	0.930	0.002	0.870	0.004		
		GAT	0.910	0.007	0.930	0.004	0.850	0.007		

noisy edges and number of sampled neighbor numbers on GNN performance: GraphSage [55] with random sampling, FastGCN [23] with heuristic layer-wise sampling, AS-GCN [68] with trainable layer-wise sampling, and PASS [181] with trainable node-wise sampling. Finally, we choose four different GNN models to check their robustness to distribution shifts in training/test time, as the authors of the original paper [199] chose for their baselines: GCN [75], SGC [162], GAT [155], and PPNP [79].

We implement GCN, SGC, GIN, and GAT from scratch for the SCENARIO 1: noisy edges on aggregation strategies. For SCENARIOS 2 and 3: noisy edges and different sampling numbers on neighbor sampling, we use open source implementations of each GNN model, ASGCN [3], FastGCN [4], and PASS [5], uploaded by the original authors. Finally, for SCENARIO 4: distribution shift, we use GCN, SGC, GAT, and PPNP implemented by [199] using DGL library [6].

6.7.12 Architecture of Computation Graph Transformer

Given a sequence $\mathbf{s} = [s_1, \cdots, s_T]$, the M-layered transformer maximizes the likelihood under the forward auto-regressive factorization as follow:

$$\begin{aligned}\max_{\theta} \log p_\theta(\mathbf{s}) &= \sum_{t=1}^{T} \log p_\theta(s_t|\mathbf{s}_{<t}) \\ &= \sum_{t=1}^{T} \log \frac{exp(q_\theta^{(L)}(\mathbf{s}_{1:t-1})^\top e(s_t))}{\sum_{s' \neq s_t} exp(q_\theta^{(L)}(\mathbf{s}_{1:t-1})^\top e(s'))}\end{aligned}$$

where node embedding $e(s_t)$ maps discrete input id s_t to a randomly initialized trainable vector, and query embedding $q_\theta^{(L)}(\mathbf{s}_{1:t-1})$ encodes information until $(t-1)$-th token in the sequence. Query embedding $q_t^{(l)}$ is computed with context embeddings $\mathbf{h}_{1:t-1}^{(l-1)}$ of previous $t-1$ tokens and query embedding $q_t^{(l-1)}$ from the previous layer. Context embedding $h_t^{(l)}$ is computed from $\mathbf{h}_{1:t}^{(l-1)}$, context embeddings of previous $t-1$ tokens and t-th token from the previous layer. Note that, while the query embeddings have access only to the previous context embeddings $\mathbf{h}_{1:t-1}^{(l)}$, the context embeddings attend to all tokens $\mathbf{h}_{1:t}^{(l)}$. The context embedding $h_t^{(0)}$ is initially encoded by node embeddings $e(s_t)$ and position embedding $p_{l(t)}$ that encodes the location of each token in the sequence. The query embedding is initialized with a trainable vector and label embeddings y_{s_1} as shown in Figure 6.3. This two streams (query and context) of self-attention layers are stacked M time and predict the next tokens auto-regressively.

6.7.13 Differentially Private k-means and SGD algorithms

Given a set of data points, k-means clustering identifies k points, called cluster centers, by minimize the sum of distances of the data points from their closest cluster center. However, releasing the

[3] https://github.com/huangwb/AS-GCN
[4] https://github.com/matenure/FastGCN
[5] https://github.com/linkedin/PASS-GNN
[6] https://github.com/GentleZhu/Shift-Robust-GNNs

set of cluster centers could potentially leak information about particular users. For instance, if a particular data point is significantly far from the rest of the points, so the k-means clustering algorithm returns this single point as a cluster center. Then sensitive information about this single point could be revealed. To address this, DP k-means clustering algorithm [21] is designed within the framework of differential privacy. To generate the private core-set, DP k-means partitions the points into buckets of similar points then replaces each bucket by a single weighted point, while adding noise to both the counts and averages of points within a bucket.

Training a model is done through access to its parameter gradients, i.e., the gradients of the loss with respect to each parameter of the model. If this access preserves differential privacy of the training data, so does the resulting model, per the post-processing property of differential privacy. To achieve this goal, DP stochastic gradient descent (DP-SGD) [135] modifies the minibatch stochastic optimization process to make it differentially private.

We use the open source implementation of DP k-means provided by Google's differential privacy libraries [7]. We extend implementations of DP SGD provided by a public differential library Opacus [8].

6.7.14 Privacy-enhanced graph synthesis

Various privacy-enhanced graph synthesis [45, 116, 117, 124, 167, 169] has been proposed to ensure differentially-private (DP) [34] graph sharing. However, most of them are limited to small-scaled graphs using a few heuristic rules, while all of them do not consider node attributes and labels in their graph generation process [124, 167]. Some GNN models have been proposed with DP guarantees [109, 123], but this line of work concerns the *models* and not the *graphs*, and is therefore outside of our scope.

6.7.15 Experimental settings

All experiments were conducted on the same p3.2xlarge Amazon EC2 instance. We run each experiment three times and report the mean and standard deviation.

Dataset: We evaluate on seven public datasets — three citation networks (Cora, Citeseer, and Pubmed) [126], two co-purchase graphs (Amazon Computer and Amazon Photo) [127], and two co-authorship graph (MS CS and MS Physic) [127]. We use all nodes when training CGT. For GNN training, we split $50\%/10\%/40\%$ of each dataset into the training/validation/test sets, respectively. We report their statistics in Table 6.9. AmazonC and AmazonP denote Amazon COmputer and Amazon Photo datasets, respectively.

Baselines: For the molecule graph generative models, GraphAF, GraphDF, and GraphEBM, we extend implementations in a public domain adaptation library DIG [96]. We extend implementations of VGAE [9], GraphVAE [10] from codes uploaded by the authors of [76, 183].

[7] https://github.com/google/differential-privacy/tree/main/python/dp_accounting

[8] https://github.com/pytorch/opacus

[9] https://github.com/tkipf/gae

[10] https://github.com/JiaxuanYou/graph-generation

Model architecture: For our Computation Graph Transformer model, we use 3-layered transformers for Cora, Citeseer, Pubmed, and Amazon Computer, 4-layered transformers for Amazon Photo and MS CS, and 5-layered transformers for MS Physic, considering each graph size. For all experiments to examine the benchmark effectiveness of our model in Section 6.4.4, we sample $s = 5$ neighbors per node. For graph statistics shown in Section 6.4.3, we sample $s = 20$ neighbors per node.

Table 6.8: **Ablation study**

Dataset	model	Original	Label	Position	Attention	All gone	Ours
	GCN	0.860	0.510	0.710	0.580	0.570	0.760
	SGC	0.850	0.520	0.700	0.580	0.570	0.750
	GIN	0.850	0.510	0.620	0.600	0.570	0.750
	GAT	0.830	0.520	0.450	0.350	0.560	0.750
Cora	**GraphSage**	0.750	0.210	0.590	0.320	0.600	0.500
	AS-GCN	0.120	0.170	0.240	0.070	0.140	0.110
	FastGCN	0.450	0.570	0.830	0.560	0.630	0.380
	PASS	0.800	0.470	0.750	0.410	0.600	0.540
	PPNP	0.840	0.555	0.850	0.743	0.584	0.810
Dataset	**model**	**Original**	**Label**	**Position**	**Attention**	**All gone**	**Ours**
	GCN	0.730	0.450	0.670	0.530	0.520	0.590
	SGC	0.730	0.460	0.640	0.530	0.530	0.580
	GIN	0.710	0.450	0.520	0.530	0.510	0.570
	GAT	0.710	0.460	0.210	0.590	0.530	0.570
Citeseer	**GraphSage**	0.680	0.280	0.580	0.370	0.550	0.440
	AS-GCN	0.110	0.200	0.280	0.220	0.160	0.100
	FastGCN	0.370	0.530	0.860	0.610	0.610	0.330
	PASS	0.700	0.480	0.550	0.450	0.550	0.460
	PPNP	0.690	0.540	0.760	0.393	0.547	0.610
Dataset	**model**	**Original**	**Label**	**Position**	**Attention**	**All gone**	**Ours**
	GCN	0.860	0.680	0.970	0.670	0.740	0.780
	SGC	0.860	0.680	0.970	0.580	0.740	0.780
	GIN	0.830	0.670	0.990	0.670	0.740	0.770
	GAT	0.860	0.690	0.940	0.120	0.740	0.780
Pubmed	GraphSage	0.780	0.460	0.360	0.920	0.740	0.600
	AS-GCN	0.250	0.320	0.200	0.770	0.360	0.260
	FastGCN	0.480	0.670	0.560	0.650	0.740	0.440
	PASS	0.860	0.690	0.330	1.000	0.740	0.660
	PPNP	0.820	0.687	0.190	0.997	0.736	0.730

Table 6.9: **Dataset statistics.**

Dataset	Nodes	Edges	Features	Labels
Cora	2,485	5,069	1,433	7
Citeseer	2,110	3,668	3,703	6
Pubmed	19,717	44,324	500	3
AmazonC	13,381	245,778	767	10
AmazonP	7,487	119,043	745	8
MS CS	18,333	81,894	6,805	15
MS Physic	34,493	247,962	8,415	5

Chapter 7

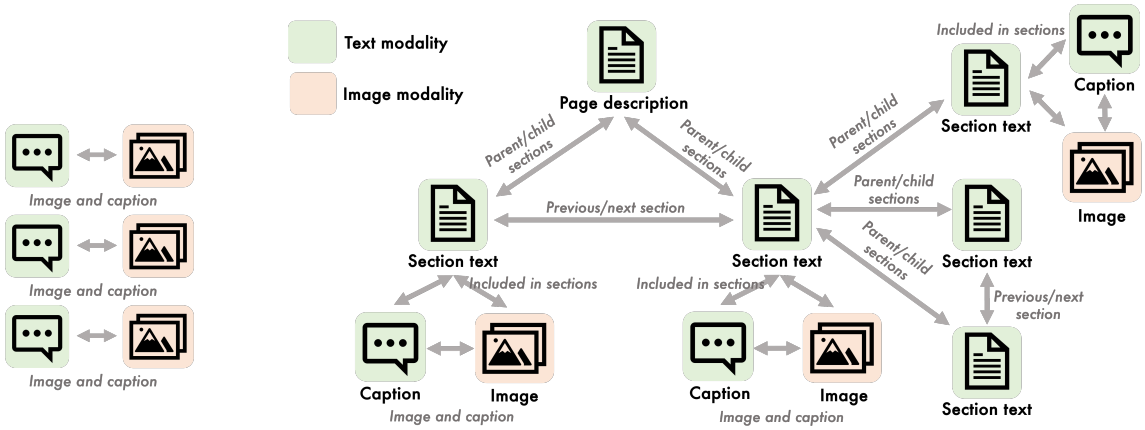
Multimodality

Multimodal learning combines multiple data modalities, broadening the types and complexity of data our models can utilize: for example, from plain text to image-caption pairs. Most multimodal learning algorithms focus on modeling simple one-to-one pairs of data from two modalities, such as image-caption pairs, or audio-text pairs. However, in most real-world settings, entities of different modalities interact with each other in more complex and multifaceted ways, going beyond one-to-one mappings. We propose to represent these complex relationships as graphs, allowing us to capture data with any number of modalities, and with complex relationships between modalities that can flexibly vary from one sample to another. Toward this goal, we propose Multimodal Graph Learning (MMGL), a general and systematic framework for capturing information from multiple multimodal neighbors with relational structures among them. In particular, we focus on MMGL for *generative* tasks, building upon pretrained Language Models (LMs), aiming to augment their text generation with multimodal neighbor contexts. We study three research questions raised by MMGL: (1) how can we infuse multiple neighbor information into the pretrained LMs, while avoiding scalability issues? (2) how can we infuse the graph structure information among multimodal neighbors into the LMs? and (3) how can we finetune the pretrained LMs to learn from the neighbor context in a parameter-efficient manner? We conduct extensive experiments to answer these three questions on MMGL and analyze the empirical results to pave the way for future MMGL research.

7.1 Motivation

There are diverse data modalities in real-world applications, from commonly observed texts, images, and videos to time series data or domain-specific modalities like protein sequences. These various modalities are not collected individually but together with multifaceted relations among them. Wikipedia [18] is one of the most popular sources of multimodal web content, providing multimodal data such as texts, images, and captions. TimeBuilder [144], recently released by Meta, builds personal timelines using each user's multimodal data, including their photos, maps, shopping, and music history. In addition to these examples, important industrial and medical decisions are also made by considering diverse multimodal data such as images, tables, or audio [66, 133]. These multimodal data have complicated $many$-to-$many$ relations among their multimodal entities — which can be represented as graphs — providing open research space on how to understand them holistically.

With the rise of multimodal datasets, various ground-breaking research has been done in



(a) 1-to-1 Multimodal Learning (b) $many$-to-$many$ Multimodal Graph Learning

Figure 7.1: **Multimodal datasets extracted from Wikipedia.** (a) Most multimodal models target multimodal datasets with clear 1-to-1 mappings between modalities. (b) Multimodal Graph Learning (MMGL) handles multimodal datasets with complicated relations among multiple multimodal neighbors.

multimodal learning. Previously, multimodal learning focused on novel architectures, extending transformers [58, 91, 146] or graph neural networks [65, 125], and training them from scratch using large-scaled multimodal datasets. Fueled by the strong generative power of pretrained Language Models (LMs), recent multimodal approaches [2, 80, 86] are built upon pretrained LMs and focus on the generation of multimodal content. For instance, [80] generates images/text grounded on given text/images using pretrained image encoders and LMs. However, all existing models assume that a pair of modalities with a clear 1-to-1 mapping is provided as input (e.g., image-caption pairs in Figure 7.1(a)). As a result, they cannot be directly applied on multimodal datasets with more general $many$-to-$many$ mappings among modalities (e.g., multimodal Wikipedia webpage in Figure 7.1(b)).

Here, we expand the scope of multimodal learning beyond 1-to-1 mappings into multimodal graph learning (MMGL) while preserving generative abilities by integrating them into pretrained LMs. We introduce a systematic framework on how MMGL processes multimodal neighbor information with graph structures among them and generate free-form texts using pretrained LMs (Figure 7.2). Our MMGL framework extracts *neighbor encodings*, combines them with *graph structure information*, and optimizes the model using *parameter-efficient fine-tuning*. Accordingly, we define three design spaces to study three research questions for MMGL as follows:

- **Research Question 1.** How can we provide multiple multimodal neighbor information to LMs while avoiding scalability issues?
- **Research Question 2.** How can we infuse graph structure information among multimodal neighbors into LMs?
- **Research Question 3.** How can we finetune pretrained LMs to learn through multimodal neighbor information in parameter-efficient ways?

In conventional multimodal learning with the 1-to-1 mapping assumption, typically only one neighbor is provided (e.g., an image for a text caption) [2, 80, 86]. On the contrary, MMGL requires

the processing of several neighbors with various data sizes (e.g., image resolution and text sequences of various lengths), which leads to the scalability issue. For *Research Question 1*, we study three neighbor encoding models: (1) *Self-Attention with Text + Embeddings* (SA-Text+Embeddings) precomputes image embeddings using frozen encoders, then concatenates them to the input text sequences with any raw text from neighbors (originally proposed from [147]), (2) *Self-Attention with Embeddings* (SA-Embeddings) precomputes embeddings for both text and image modalities using frozen encoders and concatenates to the input text, and (3) *Cross-Attention with Embeddings* (CA-Embeddings) feeds precomputed text or image embeddings into cross-attention layers of LMs.

In *Research Question 2*, we study how to infuse graph structure information among multimodal neighbors into LMs (e.g., section hierarchy and image orders in Figure 7.1(b)). We compare the sequential position encoding with two graph position encodings widely used in graph transformers [121, 175]: *Laplacian eigenvector position encoding* (LPE) [33] and *graph neural networks encoding* (GNN) [75] that runs GNNs on precomputed neighbor embeddings using graphs structures before feeding them into LMs.

Research Question 3 seeks to improve the cost and memory efficiency compared to full fine-tuning of LMs. In this work, we explore three parameter-efficient fine-tuning (PEFT) methods [59]: *Prefix tuning* [89], *LoRA* [61], and *Flamingo tuning* [2]. Which PEFT methods to use depends on the neighbor encoding model: when neighbor information is concatenated into the input sequences (*SA-Text+Embeddings* or *SA-Embeddings* neighbor encodings), we can apply *Prefix tuning* or *LoRA* for fine-tuning. When neighbor information is fed into cross-attention layers (*CA-Embeddings* neighbor encoding), we apply *Flamingo tuning* that finetunes only cross-attention layers with gating modules for stable finetuning [2].

Based on our MMGL framework, we run extensive experiments on the recently released multimodal dataset, WikiWeb2M [18]. WikiWeb2M unifies each Wikipedia webpage content to include all text, images, and their structures in a single example. This makes it useful for studying multimodal content understanding with many-to-many text and image relationships, in the context of generative tasks. Here, we focus on the section summarization task that aims to generate a sentence that captures information about the contents of one section by understanding the multimodal content on each Wikipedia page. Through rigorous testing on WikiWeb2M, we provide intuitive empirical answers to research questions raised in MMGL.

In summary, our contributions are:

- **Multimodal Graph Learning (MMGL):** We introduce a systematic MMGL framework for processing multimodal neighbor information with graph structures among them, and generating free-form texts using pretrained LMs.
- **Principled Research Questions:** We introduce three research problems MMGL is required to answer: (1) how to provide multiple neighbor information to the pretrained LMs, (2) how to infuse graph structure information into LMs, and (3) how to fine-tune the LMs parameter-efficiently. This paves research directions for future MMGL research.
- **Extensive Empirical Results:** We show empirically that (1) neighbor context improves generation performance, (2) *SA-Text+Embeddings* neighbor encoding shows the highest performance while sacrificing the scalability, (3) *GNN* embeddings are the most effective graph position encodings, and (4) *SA-Text+Embeddings* neighbor encoding with *LoRA* and *CA-Embeddings* neighbor

encoding with *Flamingo tuning* show the highest performance among different PEFT models. Our code is publicly available at [1].

7.2 Proposed work

Given multimodal graphs with text or images on each node, we aim to generate text conditioned on each node and its neighbor nodes. More specifically, given text input on a target node, pretrained LMs generate free-form text conditioned on the input text and the multimodal context around the target node. In our multimodal graph learning (MMGL) framework, we first encode each neighbor's information individually using frozen encoders (Figure 7.2(b)). The frozen encoders could be pretrained ViT [31] or ResNeT [56] for images that map pixels to embeddings, and pretrained LMs [119] for texts that map texts to embeddings (similarly for other modalities). Then, we encode the graph structure around the target node using graph position encodings (Figure 7.2(c)). Finally, the encoded neighbor information with graph position encodings is fed into the pretrained LMs with the input text to generate text conditioned on the multimodal input content (Figure 7.2(d)).

The framework leaves us with three design spaces: (1) how can we feed neighbor information to the LMs? (2) how can we infuse graph structure information among multimodal neighbors into LMs? (3) how can we finetune the pretrained LMs to learn from the neighbor context parameter-efficiently? In this section, we investigate each problem and discuss possible methodologies we can apply.

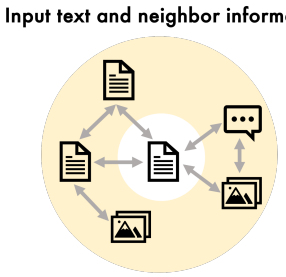
7.2.1 Research Question 1: Neighbor Encoding

Unlike existing multimodal learning, which assumes a single image (corresponding to the input text) as input, multimodal graph learning considers an arbitrary number of neighbor images/texts as input; thus, scalability is the first problem to solve to learn from multiple multimodal neighbors. In vision-text models, a standard recipe is to first process images with an image encoder (e.g., ViT, ResNet) into image embeddings, then map the embeddings into the text-only LM space, and finally feed them into the LMs. Two popular ways to feed image embeddings into LMs are with full self-attention over modalities concatenated across the sequence dimension [147] or with cross-modal attention layers [146].

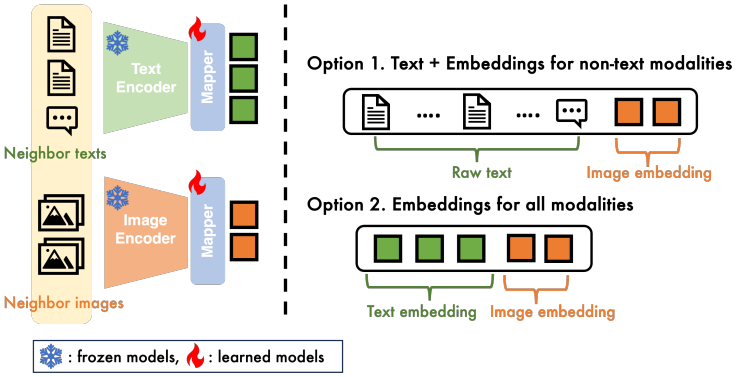
Motivated by these two approaches, we propose three neighbor encoding methods as follows:

- **Self-Attention with Text + Embeddings (SA-Text+Embeddings)**: Text neighbors are concatenated as raw texts, while other modalities are first processed by frozen encoders (e.g., ViT for images), and then their embeddings are concatenated to the input sequence. We add a linear mapper that aligns precomputed embeddings into the text space of LLMs.
- **Self-Attention with Embeddings (SA-Embeddings)**: Same as *SA-Text+Embeddings* except text neighbors are also processed by separate frozen encoders, and their embeddings are concatenated to the input sequence. Text encoders could be the same or different from the base LLM model.

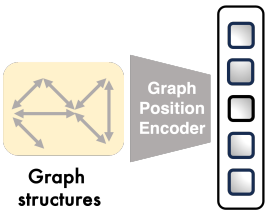
[1] `https://github.com/minjiyoon/MMGL`



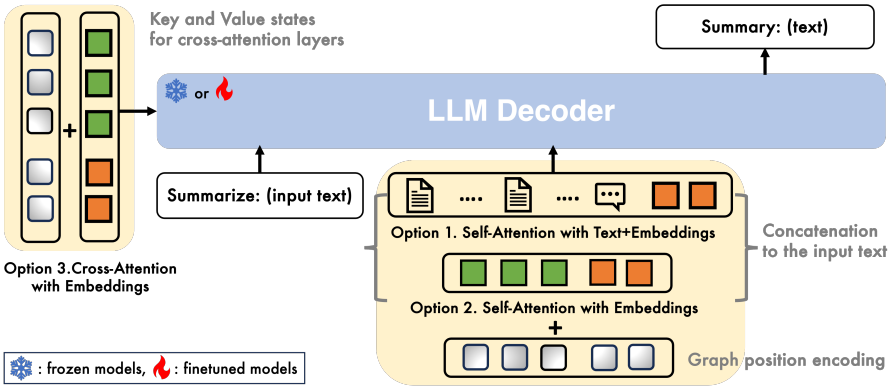
(a) Multimodal neighbors with graph structures



(b) Multimodal neighbor encoding using frozen vision/text encoders



(c) Graph position encodings



(d) Text generation using frozen/finetuned LMs and encoded neighbor information

Figure 7.2: **Multimodal Graph Learning (MMGL) framework.** (a) Multiple multimodal neighbors are given with the input text. (b) Multimodal neighbors are first encoded using frozen vision/text encoders and then aligned to the text-only LM space using 1-layer MLP mappers. The mappers are trained during LM fine-tuning. Based on the neighbor encoding scheme, texts could be used without any preprocessing (*Self-Attention with Text+Embeddings*) or encoded into embeddings (*Self-Attention with Embeddings* or *Cross-Attention with Embeddings*). Images are always encoded into embeddings to align to the text-only LM space. (c) Graph structures among neighbors are encoded as graph position encodings. (d) Encoded neighbor information could be infused either by concatenating to the input sequences (*Self-Attention with Text+Embeddings* or *Self-Attention with Embeddings*) or feeding into cross-attention layers (*Cross-Attention with Embeddings*). The graph position encodings are added to the input token/text/image embeddings.

- **Cross-Attention with Embeddings (CA-Embeddings)**: All neighbors are processed by separate frozen encoders, mapped into the text space by linear mappers, and then fed into cross-attention layers.

In general, when we provide text embeddings instead of raw text, the amount of information the LLMs are able to exploit is bottlenecked by the precomputed embeddings. However, raw texts

introduce scalability issues as the attention mechanism of LMs uses the $O(T^2)$ compute with the sequence length T. Thus, there is a trade-off between computation cost and scalability. For *SA-Text+Embeddings* and *SA-Embeddings*, we have additional parameters only for mappers that are located outside of the LMs, while *CA-Embeddings* inserts additional cross-attention layers into pretrained LMs and trains them from scratch. This means *CA-Embeddings* could result in an unstable initial state as the pretrained LLM layers are affected by randomly initialized cross-attention layers. In Section 7.3.4, we explore these three approaches and discuss their empirical results.

7.2.2 Research Question 2: Graph Structure Encoding

Given neighbor information, we can simply concatenate neighbor information either as raw texts or embeddings and treat them as a sequence. But the neighbors have structures among them. For instance, sections have hierarchical structures, and images are included in certain sections in WikiWeb2M (Figure 7.1(b)). To encode this graph structure among the neighbor information, we borrow two popular graph position encodings from graph transformers and compare them with sequential position encoding.

- **Laplacian Position Encoding (LPE)**: We exploit Laplacian eigenvectors of neighbors computed from their graph structure as their position encodings.
- **Graph Neural Networks (GNN)**: We first compute neighbor embeddings from frozen encoders and run GNN over the embeddings using the graph structure. Then, we use the output GNN embeddings, which encode graph structure information as position encodings.

LPE has an additional 1-layer MLP mapper to map the Laplacian eigenvectors to the text space of LMs. Parameters used for graph structure encoding (e.g., mappers for *LPE* or *GNN* parameters) are trained with LMs in an end-to-end manner during LM fine-tuning. In Section 7.3.5, we explore how well these different position encodings bring additional graph structure information among neighbors into LMs and improve performance.

7.2.3 Research Question 3: Parameter-Efficiency

While we need to fine-tune the pretrained LM model for the specific task and newly added neighbor information, full fine-tuning requires high computation costs and also brings inconvenience in sharing MMGL modules when users decide to use neighbor information. Recently, various parameter-efficient fine-tuning (PEFT) methods have been proposed to fine-tune only a small amount of parameters while preserving the full fine-tuning performance. We choose three different PEFT models proper for the three neighbor encoding approaches we described above.

- **Prefix tuning**: When we choose *SA-Text+Embeddings* or *SA-Embeddings* for neighbor encoding, we do not have any newly added parameters but self-attention layers; thus, we can easily apply Prefix tuning [89], which keeps language model parameters frozen and instead optimizes a sequence of continuous task-specific vectors prepended to the original activation vectors across all layers.
- **LoRA**: Like *Prefix tuning*, low-rank adaptation (LoRA) [61] is suitable for *SA-Text+Embeddings* or *SA-Embeddings* neighbor encodings. LoRA injects trainable rank decomposition matrices into

each layer while freezing the original parameters.

- **Flamingo**: For *CA-Embeddings* neighbor encoding, we can directly apply *Flamingo* [2], which fine-tunes only newly added cross-attention layers with *tanh* gating to keep the pretrained LM intact at initialization for improved stability and performance.

In Section 7.3.6, we explore how well PEFT models preserve the full fine-tuning performance by tuning a small number of parameters.

7.3 Experiments

7.3.1 WikiWeb2M dataset

WikiWeb2M dataset [18] is built for the general study of multimodal content understanding with many-to-many text and image relationships. Built upon the WIT dataset [136] which contains only image-caption pairs, WikiWeb2M includes the page title, section titles, section text, images and their captions, and indices for each section, their parent section, their children sections, and many more.

In this work, we focus on the section summarization task to generate a single sentence that highlights a particular section's content. The summary is generated given all images and (non-summary) text present in the target and context sections. We sample 600k Wikipedia pages randomly from WikiWeb2M for the section summarization task. In total, the training/validation/test set sizes for the section summarization task are 680k/170k/170k, respectively.

7.3.2 Experimental Settings

From WikiWeb2M, we can get four types of information for section summarization: (1) section text, (2) section images, (3) text from page description and other sections, and (4) images from page description and other sections. We provide information incrementally to LMs to study the effectiveness of multimodal neighbor information: (1) *section text*, 2) *section all* (text + image), 3) *page text* (all text from a Wikipedia page the input section belongs to), and 4) *page all* (all text and images from the Wikipedia page).

We use Open Pre-trained Transformer (OPT-125m) [194] for the base LM to read the input section text and generate a summary. For text and image encoders for neighbor information, we use text/image encoders from CLIP [119]. Following [120], we finetune OPT for 10000 steps of 125 batch size with learning rate 10^{-4}. The text/image encoders are frozen across all experiments. We measure BLEU-4 [113], ROUGE-L [93], and CIDEr [154] scores on the validation set. All experiments are run on 4 Nvidia-RTX 3090 GPUs with 24GB memory.

7.3.3 Effectiveness of Neighbor Information

We first examine the effectiveness of multimodal neighbor information. As described in Section 7.3.2, we provide more information incrementally to the base LM: (1) *section text*, (2) *section all* (text + image), 3) *page text*, and 4) *page all* (all texts and images). Here, we use *Self-Attention*

Table 7.1: **Effectiveness of neighbor information.** As more neighbor information is fed to LMs together with input texts (*section text, section all* => *page text, page all*), generation performance is improved. We increase the input sequence length to 1024 to encode *page text* and *page all* as more information is required to be encoded. The best results are colored in red, while the second-best results are colored in blue.

Input type	Input length	BLEU-4	ROUGE-L	CIDEr
Section text	512	8.31	40.85	79.68
Section all	512	8.03	40.41	77.45
Page text	1024	9.81	42.94	92.71
Page all	1024	9.96	43.32	96.62

Table 7.2: **Neighbor encodings in MMGL.** We encode multiple multimodal neighbor information using three different neighbor encodings, *Self-Attention with Text+Embeddings* (SA-TE), *Self-Attention with Embeddings* (SA-E), and *Cross-Attenion with Embeddings* (CA-E). While SA-TE shows the best performance, SA-TE requires a longer input length (1024) to encode texts from neighbors in addition to the original text input, leading to scalability issues. The best results are colored in red.

	BLEU-4			ROUGE-L			CIDEr		
Input type	**SA-TE**	**SA-E**	**CA-E**	**SA-TE**	**SA-E**	**CA-E**	**SA-TE**	**SA-E**	**CA-E**
Section all	8.03	7.56	8.35	40.41	39.89	39.98	77.45	74.33	75.12
Page text	9.81	8.37	8.47	42.94	40.92	41.00	92.71	80.14	80.72
Page all	9.96	8.58	8.51	43.32	41.01	41.55	96.01	82.28	80.31
Max input length	1024	512	512	1024	512	512	1024	512	512

with Text+Embeddings (SA-Text+Embeddings) neighbor encoding across different input types. For images, we first compute the image embeddings from the frozen CLIP image encoder and concatenate them right after the text of a section each image belongs to preserve the structure. The results in Table 7.1 indicate that *more multimodal neighbor information is helpful:* performance significantly improves when going from *section* to *page* content, and further when adding *page all* content, based on their BLEU-4, ROUGE-L, and CIDEr scores.

Discussion: Missing Modalities. Performance of *section all* decreased slightly from *section text*, despite the addition of section images. In Wikipedia, not every section has corresponding images. Thus, in the *section all* case, input to the LMs is inconsistent with some samples having text and images, while other samples only have text. This points to an important unaddressed *missing modality issue* that is common in the real world, which is not typically encountered in the conventional 1-to-1 multimodal setting, emphasizing the importance of developing MMGL approaches that are robust to the presence of missing modalities.

Table 7.3: **Graph structure encoding in MMGL.** We encode graph structures among multimodal neighbors using sequential position encodings (*Sequence*), Graph Neural Network embeddings (*GNN*), and Laplacian position encodings (*LPE*). Computed position encodings are added to input token/text/image embeddings and fed into LMs. We use *Self-Attention with Embeddings (SA-E)* neighbor encoding and *Prefix tuning* in this experiment. The best results are colored in red.

Metric	PEFT	Sequence	GNN	LPE
BLEU-4	**Prefix tuning**	6.91	6.98	6.80
	LoRA	7.12	7.30	7.13
ROUGE-L	**Prefix tuning**	38.98	39.13	39.10
	LoRA	39.05	39.48	39.35
CIDEr	**Prefix tuning**	68.20	69.29	68.15
	LoRA	68.86	70.86	69.34

7.3.4 Neighbor Encoding

We encode multiple multimodal neighbor information using three different neighbor encodings, *Self-Attention with Text+Embeddings* (SA-TE), *Self-Attention with Embeddings* (SA-E), and *Cross-Attenion with Embeddings* (CA-E). While SA-E and CA-E encode all modalities, including text, into embeddings using frozen encoders, SA-TE encodes text neighbors as they are by concatenating them to the input text sequence. Thus SA-TE requires longer input sequence lengths (1024) to encode additional texts, leading to potential scalability issues. On the other hand, SA-E and CA-E require one token length to encode one text neighbor, improving scalability with shorter input lengths (512). The results in Table 7.2 indicate that *scalability is traded off with performance:* SA-TE consistently performs better than SA-E and CA-E on different input types at the cost of longer input lengths.

Discussion: Information Loss. In conventional multimodal learning with 1-to-1 mappings, SA-TE is commonly used to infuse text input as it is, and image inputs as embeddings are precomputed by frozen encoders [2, 80, 86]. These methods successfully generate texts grounded on the input images, showing image embeddings' effectiveness as input to the pretrained LMs. However, the performance gap between SA-TE and SA-E in Table 7.2 indicates that text embeddings likely lead to *information loss* in the LMs. This could be either because the 1-layer MLP mapper that aligns precomputed text embeddings into the text space of the LMs is not expressive enough, or because longer input texts compared to short texts used in the conventional multimodal learning (e.g., one-sentence captions) makes LMs hard to learn from precomputed text embeddings. From a practical angle, our results illuminate the trade-off between scalability and performance. Meanwhile, our results emphasize the need for more MMGL research to address the challenging issue of information loss when using embeddings to capture text information.

Table 7.4: **Parameter-efficient finetuning in MMGL.** We apply *Prefix tuning* and *LoRA* for *Self-Attention with Text+Embeddings (SA-TE)* and *Self-Attention with Embeddings (SA-E)* neighbor encodings. For *Cross-Attention with Embeddings (CA-E)* neighbor encoding, we apply *Flamingo*-style finetuning that finetunes only newly added cross-attention layers with gating modules. Note that *SA-E* and *CA-E* neighbor encodings have more parameters than *SA-TE* because (frozen) text encoders are added to encode text neighbors. The best results are colored in red, while the second-best results are colored in blue.

Neighbor encoding (max length)		**SA-TE (1024)**		**SA-E (512)**		**CA-E (512)**
Metric	**Input type**	**Prefix tuning**	**LoRA**	**Prefix tuning**	**LoRA**	**Flamingo**
BLEU-4	**Section all**	6.70	6.65	6.80	7.07	6.96
	Page text	7.84	7.94	6.88	7.09	7.81
	Page all	8.21	8.18	6.91	7.12	8.12
ROUGE-L	**Section all**	38.67	38.84	38.97	39.30	39.43
	Page text	40.61	40.98	38.38	39.69	40.29
	Page all	41.08	41.25	38.98	39.05	40.95
CIDEr	**Section all**	65.84	65.00	67.24	68.61	69.31
	Page text	78.12	78.60	66.55	69.26	76.20
	Page all	81.07	80.75	68.20	68.86	82.37
# Finetuned parameters		20M	82M	20M	84M	90M
# Total parameters		230M	250M	300M	320M	363M
% Finetuned parameters		9%	33%	7%	26%	25%

7.3.5 Graph Structure Encoding

In addition to each modality on neighbors, multimodal graphs contain graph structure information among neighbors. We encode the graph structures among multimodal neighbors using sequential position encodings (*Sequence*), Graph Neural Network embeddings (*GNN*), and Laplacian position encodings (*LPE*). Computed position encodings are first mapped to the text space of LMs by 1-layer MLP, added to input token/text/image embeddings, and fed into LMs. In Table 7.3, *GNN* embeddings show the best performance. Especially, the improvement over *Sequence* position encoding shows the *importance of graph-aware structure encoding methods* in MMGL.

7.3.6 Parameter-Efficient Fine-Tuning

Full fine-tuning of pretrained LMs requires high computational costs. For parameter-efficient fine-tuning for MMGL, we study *Prefix tuning* and *LoRA* for *Self-Attention with Text+Embeddings (SA-TE)* and *Self-Attention with Embeddings (SA-E)* neighbor encodings. For *Cross-Attention with Embeddings (CA-E)* neighbor encoding, we apply *Flamingo*-style finetuning that finetunes only newly added cross-attention layers with gating modules.

The results in Table 7.4 show that *LoRA performs better than Prefix tuning* for *SA-TE* and *SA-E* neighbor encodings with more fine-tuned parameters ($7-9\%$ for *Prefix tuning* and $26-33\%$

for *LoRA*). However, *Prefix tuning* still shows comparable performance with *LoRA* using nearly 4 times fewer parameters with *SA-TE* neighbor encoding. *Flamingo* with *CA-E* neighbor encoding shows comparable performance with *LoRA* with *SA-TE* neighbor encoding employing the similar numbers of fine-tuned parameters ($82M$ for *LoRA* and $90M$ for *Flamingo*). Note that *SA-E* and *CA-E* neighbor encodings have more parameters than *SA-TE*, attributed to the inclusion of (frozen) text encoders for text neighbor processing.

In Table 7.2 (without PEFT), it is evident that *CA-E* neighbor encoding lags in performance compared to *SA-TE* neighbor encoding. However, when infused with Flamingo, gating modules in Flamingo effectively ensure that the pre-trained LMs remain unaffected by randomly set cross-attention layers at initialization, thereby enhancing the performance of *CA-E*, as shown in Table 7.4 (with PEFT). This underscores the pivotal role of strategic initialization when introducing supplementary modules for neighbor encoding in MMGL and when integrating them into the pre-trained LMs.

7.4 Related Work

End-to-End Multimodal Learning: While many discriminative multimodal models [69, 119] have also been developed, we primarily consider related work on generative multimodal models, as this is most closely related with our approach. Several recent approaches tackle multimodal learning by building upon the Transformer [152] architecture. Multimodal extensions typically use either full self-attention over modalities concatenated across the sequence dimension [24, 138] or a cross-modal attention layer [146]. Self-supervised multimodal pretraining methods train these architectures from large-scale unlabeled multimodal data before transferring them to downstream multimodal tasks via fine-tuning [58, 91]. These methods perform end-to-end pre-training, incurring extremely high computation costs, especially as model parameters increase [86]. Moreover, this framework is relatively inflexible for end-to-end pre-trained models to leverage readily available unimodal pre-trained models, such as text-only LMs or pretrained vision models.

Multimodal Learning with Frozen Image Encoders and Large Language Models: Recently, various vision-language models have been proposed to leverage off-the-shelf pre-trained models and keep them frozen during pretrainig [2, 80, 86]. To input visual information directly to a frozen text-only LLM, a key challenge is to align visual features to the text space. Motivated by Frozen [147], which finetunes a visual encoder to map images into the hidden space of a text-only LLM, Blip-2 [86] and GILL [80] finetune separate image mapping networks whose inputs are precomputed by frozen image encoders and outputs are directly used as soft prompts to LLMs. On the other hand, Flamingo [2] inserts new cross-attention layers into the LLM to inject visual features and pre-trains the new layers on image-text pairs. Note that all these methods primarily focus on processing *interleaved image and text inputs* to generate text outputs.

7.5 Summary

In this work, we extend the conventional multimodal learning with 1-to-1 mappings between a pair of modalities into multimodal graph learning (MMGL) with $many$-to-$many$ relations among multiple modalities. Our MMGL framework is systematically structured around three critical components: (1) neighbor encodings, (2) graph structure encodings, and (3) parameter-efficient fine-tuning. Through rigorous testing on the WikiWeb2M dataset, we explored different options for each component: (1) three variations of neighbor encodings, *Self-Attention with Text+Embeddings*, *Self-Attention with Embeddings*, and *Cross-Attention with Embeddings*, highlighting the balance between scalability and performance, (2) three different graph position encodings, *sequence*, *LPE*, and *GNN*, and (3) three PEFT models, *prefix tuning*, *LoRA*, and *Flamingo*, and their trade-off between parameter-efficiency and performance. Our in-depth analyses and findings aim to lay the groundwork for future MMGL research, igniting further exploration in this field.

Chapter 8

Conclusion

In this thesis, we address factors limiting the adoption of DLG and its downstream impact on real-world applications. First, DLG often requires tedious work for hyperparameter tuning as hyperparameters for optimal algorithms vary across applications. Second, problem formulations often do not consider real-world constraints like scalability and privacy. Finally, there has been a growing demand for leveraging unimodal foundation models in multimodal graph learning to manage multimodal graphs effectively. To cope with these challenges, we redefine conventional problem formulations and develop novel algorithms for **1) automated, 2) scalable, 3) privacy-enhanced**, and **4) multimodal DLG**. As the data collected by humanity increases in scale and diversity, the relationships among individual elements increase quadratically in scale and complexity. By making DLG more practical, we hope to enable better processing of these relationships and positively impact a wide array of domains. For reproducibility and the benefit of the community, we make most of the algorithms and datasets used throughout this thesis available at `www.minjiyoon.xyz`.

Bibliography

[1] Emmanuel Abbe. Community detection and stochastic block models: recent developments. *The Journal of Machine Learning Research*, 18(1):6446–6531, 2017. 5.5.1

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022. 7.1, 7.1, 7.2.3, 7.3.4, 7.4

[3] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002. 6.5

[4] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020. 6.7.3

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. 4.1

[6] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 4(3), 2010. 3.1, 3.2.3

[7] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018. 4.1

[8] Shai Ben-David, John Blitzer, Koby Crammer, Fernando Pereira, et al. Analysis of representations for domain adaptation. *Advances in neural information processing systems*, 19:137, 2007. 5.6, 5.8.9

[9] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine learning*, 79 (1):151–175, 2010. 5.6, 5.8.9

[10] Adrien Benamira, Benjamin Devillers, Etienne Lesot, Ayush K Ray, Manal Saadi, and Fragkiskos D Malliaros. Semi-supervised learning and graph neural networks for fake news detection. In *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 568–569. IEEE, 2019. 1, 6.1

[11] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, 2007. 3.1

[12] Siddharth Bhatia, Bryan Hooi, Minji Yoon, Kijung Shin, and Christos Faloutsos. Midas: Microcluster-based detector of anomalies in edge streams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3242–3249, 2020. 2.5

[13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26, 2013. 5.8.8

[14] Paul S Bradley, Kristin P Bennett, and Ayhan Demiriz. Constrained k-means clustering. *Microsoft Research, Redmond*, 20(0):0, 2000. 6.3.2, 6.4.2, 1

[15] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010. 3.4.2, 3.6.2

[16] Sylvain Brohee and Jacques Van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC bioinformatics*, 7(1):488, 2006. 3.1

[17] Andreas Buja, Dianne Cook, and Deborah F Swayne. Interactive high-dimensional data visualization. *JCGS*, 5(1):78–99, 1996. (document), 4.5.8, 4.5

[18] Andrea Burns, Krishna Srinivasan, Joshua Ainslie, Geoff Brown, Bryan A. Plummer, Kate Saenko, Jianmo Ni, and Mandy Guo. A suite of generative tasks for multi-level multimodal webpage understanding, 2023. 7.1, 7.1, 7.3.1

[19] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)*, 38(1):2–es, 2006. 6.5

[20] 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(89):1–64, 2022. 1, 6.1

[21] Alisa Chang, Badih Ghazi, Ravi Kumar, and Pasin Manurangsi. Locally private k-means in one round. In *International Conference on Machine Learning*, pages 1441–1451. PMLR, 2021. 6.3.2, 6.4.2, 2, 6.7.13

[22] Serina Chang, Emma Pierson, Pang Wei Koh, Jaline Gerardin, Beth Redbird, David Grusky, and Jure Leskovec. Mobility network models of covid-19 explain inequities and inform reopening. *Nature*, 589(7840):82–87, 2021. 1

[23] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018. 4.1, 4.2, 4.2, 4.5.1, 4.5.1, 4.6, 6.2.2, 6.4.1, 6.7.3, 6.7.11

[24] Yen-Chun Chen, Linjie Li, Licheng Yu, Ahmed El Kholy, Faisal Ahmed, Zhe Gan, Yu Cheng, and Jingjing Liu. Uniter: Universal image-text representation learning. In *European conference on computer vision*, pages 104–120. Springer, 2020. 7.4

[25] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019. 6.7.3

[26] Eli Chien, Wei-Cheng Chang, Cho-Jui Hsieh, Hsiang-Fu Yu, Jiong Zhang, Olgica Milenkovic, and Inderjit S Dhillon. Node feature extraction by self-supervised multi-scale neighborhood prediction. *arXiv preprint arXiv:2111.00064*, 2021. 2.4

[27] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, 2016. 4.1

[28] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018. 6.5

[29] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3767–3776, 2021. 1

[30] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 135–144, 2017. 5.8.8

[31] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 7.2

[32] Rafał Dreżewski, Jan Sepielak, and Wojciech Filipkowski. The application of social network analysis algorithms in a system supporting money laundering detection. *Information Sciences*, 295:18–32, 2015. 3.1

[33] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020. 7.1

[34] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008. 6.7.14

[35] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 world wide web conference*, 2018. 3.2.3, 3.6.3

[36] Alessandro Epasto, Vahab Mirrokni, Bryan Perozzi, Anton Tsitsulin, and Peilin Zhong. Differentially private graph learning via sensitivity-bounded personalized pagerank. *arXiv preprint arXiv:2207.06944*, 2022. 6.4.2

[37] Paul Erdős, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960. 6.5

[38] Dhivya Eswaran, Reihaneh Rabbany, Artur W Dubrawski, and Christos Faloutsos. Social-affiliation networks: Patterns and the soar model. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 105–121. Springer, 2018. 6.1, 6.5

[39] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, 29(4):251–262, 1999. 4.5.2

[40] Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, et al.

Tf-gnn: Graph neural networks in tensorflow. *arXiv preprint arXiv:2207.03522*, 2022. 5.2.2

[41] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019. 5.2.2

[42] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*, pages 3294–3304. PMLR, 2021. 6.7.3

[43] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1), 2008. 3.6.1

[44] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In *NIPS*, pages 6530–6539, 2017. 4.1

[45] Arik Friedman and Assaf Schuster. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–502, 2010. 6.7.14

[46] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding. In *Proceedings of The Web Conference 2020*, pages 2331–2341, 2020. 2.3, 5.1, 5.2.2, 5.3.4, 5.5.4

[47] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The journal of machine learning research*, 17(1):2096–2030, 2016. 5.1, 5.5.2, 5.6, 5.8.9

[48] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. Graph neural architecture search. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 1403–1409, 2021. 3.6.3

[49] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017. 2.3, 5.2.2, 5.5.4

[50] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014. 6.5

[51] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *The Journal of Machine Learning Research*, 13(1):723–773, 2012. 5.6, 5.8.9

[52] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. In *International conference on machine learning*, pages 2434–2444. PMLR, 2019. 6.5

[53] Mengying Guo, Tao Yi, Yuqing Zhu, and Yungang Bao. Jitune: Just-in-time hyperparameter tuning for network embedding algorithms. *arXiv preprint arXiv:2101.06427*, 2021. 3.6.3

[54] Xiaojie Guo and Liang Zhao. A systematic survey on deep generative models for graph generation. *arXiv preprint arXiv:2007.06686*, 2020. 6.5

[55] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017. 2.2, 3.2.3, 3.5.1, 3.7, 4.1, 4.2, 4.5.1, 4.6, 5.8.1, 6.1, 6.2.2, 6.4.1, 6.7.3, 6.7.11

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 7.2

[57] Xiaoxin He, Xavier Bresson, Thomas Laurent, and Bryan Hooi. Explanations as features: Llm-based features for text-attributed graphs. *arXiv preprint arXiv:2305.19523*, 2023. 2.4

[58] Lisa Anne Hendricks, John Mellor, Rosalia Schneider, Jean-Baptiste Alayrac, and Aida Nematzadeh. Decoupling the role of data, attention, and losses in multimodal transformers. *Transactions of the Association for Computational Linguistics*, 9:570–585, 2021. 7.1, 7.4

[59] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019. 7.1

[60] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *NIPS*, pages 2042–2050, 2014. 4.1

[61] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 7.1, 7.2.3

[62] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019. 5.6

[63] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020. 6.7.5

[64] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. Gpt-gnn: Generative pre-training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1857–1867, 2020. 5.5.3, 5.6

[65] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pages 2704–2710, 2020. 2.3, 2.4, 3.7, 5.1, 5.1, 5.2.2, 5.5.3, 5.5.4, 7.1

[66] Bing Huang, Feng Yang, Mengxiao Yin, Xiaoying Mo, Cheng Zhong, et al. A review of multimodal medical image fusion techniques. *Computational and mathematical methods in medicine*, 2020, 2020. 7.1

[67] Tiancheng Huang, Ke Xu, and Donglin Wang. Da-hgt: Domain adaptive heterogeneous graph transformer. *arXiv preprint arXiv:2012.05688*, 2020. 5.1, 5.6

[68] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards

fast graph representation learning. In *NIPS*, pages 4558–4567, 2018. 4.1, 4.2, 4.2, 4.5.1, 4.5.1, 4.6, 6.2.2, 6.4.1, 6.7.3, 6.7.11

[69] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision. In *International conference on machine learning*, pages 4904–4916. PMLR, 2021. 7.4

[70] Dejun Jiang, Zhenxing Wu, Chang-Yu Hsieh, Guangyong Chen, Ben Liao, Zhe Wang, Chao Shen, Dongsheng Cao, Jian Wu, and Tingjun Hou. Could graph neural networks learn better molecular representation for drug discovery? a comparison study of descriptor-based and graph-based models. *Journal of cheminformatics*, 13(1):1–23, 2021. 1

[71] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International conference on machine learning*, pages 2323–2332. PMLR, 2018. 6.5

[72] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018. 3.1, 3.6.1

[73] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3.5.1, 4.8.2

[74] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 6.5

[75] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 1, 2.2, 3.2.3, 3.5.1, 4.1, 5.4, 6.1, 6.4.1, 6.7.11, 7.1

[76] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016. 6.4.1, 6.7.15

[77] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4), 1990. 3.6.1

[78] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5), 1999. 3.1

[79] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997*, 2018. 6.4.1, 6.7.11

[80] Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Generating images with multimodal language models. *arXiv preprint arXiv:2305.17216*, 2023. 7.1, 7.1, 7.3.4, 7.4

[81] Danai Koutra, Tai-You Ke, U Kang, Duen Horng Chau, Hsing-Kuo Kenneth Pao, and Christos Faloutsos. Unifying guilt-by-association approaches: Theorems and fast algorithms. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part II 22*, pages 245–260. Springer, 2011. 3.2.3

[82] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012. 4.1

[83] Carl Lemaire, Andrew Achkar, and Pierre-Marc Jodoin. Structured pruning of neural networks with budget-aware regularization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9108–9116, 2019. 3.1

[84] Jure Leskovec and Christos Faloutsos. Scalable modeling of real graphs using kronecker multiplication. In *Proceedings of the 24th international conference on Machine learning*, pages 497–504, 2007. 6.5

[85] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research*, 11(2), 2010. 6.1, 6.5

[86] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023. 7.1, 7.1, 7.3.4, 7.4, 7.4

[87] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence*, 2018. 6.7.3

[88] Shan Li, Baoxu Shi, Jaewon Yang, Ji Yan, Shuai Wang, Fei Chen, and Qi He. Deep job understanding at linkedin. In *SIGIR*, pages 2145–2148, 2020. 1, 4.5.1

[89] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021. 7.1, 7.2.3

[90] Xin Li, Yiming Zhou, Zheng Pan, and Jiashi Feng. Partial order pruning: for best speed/accuracy trade-off in neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9145–9153, 2019. 3.1

[91] Paul Pu Liang, Yiwei Lyu, Xiang Fan, Jeffrey Tsaw, Yudong Liu, Shentong Mo, Dani Yogatama, Louis-Philippe Morency, and Russ Salakhutdinov. High-modality multimodal transformer: Quantifying modality & interaction heterogeneity for high-modality representation learning. *Transactions on Machine Learning Research*, 2022. 7.1, 7.4

[92] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Will Hamilton, David K Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. *Advances in Neural Information Processing Systems*, 32, 2019. 6.1, 6.5

[93] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/W04-1013`. 7.3.2

[94] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017. 3.6.1

[95] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search.

arXiv preprint arXiv:1806.09055, 2018. 3.1

[96] Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Haiyang Yu, Zhao Xu, Jingtun Zhang, Yi Liu, et al. Dig: a turnkey library for diving into graph deep learning research. *Journal of Machine Learning Research*, 22(240):1–9, 2021. 6.7.15

[97] Ziqi Liu, Zhengwei Wu, Zhiqiang Zhang, Jun Zhou, Shuang Yang, Le Song, and Yuan Qi. Bandit samplers for training graph neural networks. *arXiv preprint arXiv:2006.05806*, 2020. 4.1, 4.2, 4.2, 4.5.1, 4.5.1, 4.6, 6.7.3

[98] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael Jordan. Learning transferable features with deep adaptation networks. In *International conference on machine learning*, pages 97–105. PMLR, 2015. 5.5.2, 5.6, 5.8.9

[99] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and Michael I Jordan. Conditional adversarial domain adaptation. *arXiv preprint arXiv:1705.10667*, 2017. 5.1, 5.5.2, 5.5.3, 5.6, 5.8.4, 5.8.9

[100] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I Jordan. Deep transfer learning with joint adaptation networks. In *International conference on machine learning*, pages 2208–2217. PMLR, 2017. 5.1, 5.5.2, 5.6, 5.8.9

[101] Yadan Luo, Zijian Wang, Zi Huang, and Mahsa Baktashmotlagh. Progressive graph learning for open-set domain adaptation. In *International Conference on Machine Learning*, pages 6468–6478. PMLR, 2020. 5.6

[102] Youzhi Luo, Keqiang Yan, and Shuiwang Ji. Graphdf: A discrete flow model for molecular graph generation. In *International Conference on Machine Learning*, pages 7192–7203. PMLR, 2021. 6.1, 6.4.1, 6.5

[103] Xinhong Ma, Tianzhu Zhang, and Changsheng Xu. Gcan: Graph convolutional adversarial network for unsupervised domain adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8266–8276, 2019. 5.6

[104] Krzysztof Michalak and Jerzy Korczak. Graph mining approach to suspicious transaction detection. In *2011 Federated conference on computer science and information systems (FedCSIS)*. IEEE, 2011. 3.1

[105] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 5.8.8

[106] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *arXiv preprint arXiv:1906.10652*, 2019. 4.1, 4.3.3

[107] Jerome L Myers, Arnold D Well, and Robert F Lorch Jr. *Research design and statistical analysis*. Routledge, 2013. 6.4.2

[108] S Deepak Narayanan, Aditya Sinha, Prateek Jain, Purushottam Kar, and Sundararajan Sellamanickam. Iglu: Efficient gcn training via lazy updates. *arXiv preprint arXiv:2109.13995*, 2021. 6.7.3

[109] Iyiola E Olatunji, Thorben Funke, and Megha Khosla. Releasing graph neural networks with

differential privacy guarantees. *arXiv preprint arXiv:2109.08907*, 2021. 6.7.14

[110] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. 3.1, 3.1, 3.2.3, 3.5.1, 3.6.3, 3.7, 6.4.4

[111] John Palowitch, Anton Tsitsulin, Brandon Mayer, and Bryan Perozzi. Graphworld: Fake graphs bring real insights for gnns. *arXiv preprint arXiv:2203.00112*, 2022. 6.1

[112] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. Transfer graph neural networks for pandemic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 4838–4845, 2021. 1

[113] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002. 7.3.2

[114] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *Probabilistic and Causal Inference: The Works of Judea Pearl*, pages 129–138. 2022. 3.2.3

[115] Mariya Popova, Mykhailo Shvets, Junier Oliva, and Olexandr Isayev. Molecularrnn: Generating realistic molecular graphs with optimized properties. *arXiv preprint arXiv:1905.13372*, 2019. 6.5

[116] Davide Proserpio, Sharon Goldberg, and Frank McSherry. A workflow for differentially-private graph synthesis. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 13–18, 2012. 6.7.14

[117] Zhan Qin, Ting Yu, Yin Yang, Issa Khalil, Xiaokui Xiao, and Kui Ren. Generating synthetic decentralized social graphs with local differential privacy. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 425–438, 2017. 6.7.14

[118] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. Gcc: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1150–1160, 2020. 5.6

[119] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 7.2, 7.3.2, 7.4

[120] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 7.3.2

[121] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35:14501–14515, 2022. 7.1

[122] Ievgen Redko, Amaury Habrard, and Marc Sebban. Theoretical analysis of domain adaptation with optimal transport. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 737–753. Springer, 2017. 5.6, 5.8.9

[123] Sina Sajadmanesh and Daniel Gatica-Perez. Locally private graph neural networks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2130–2145, 2021. 6.7.14

[124] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y Zhao. Sharing graphs using differentially private graph models. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 81–98, 2011. 6.7.14

[125] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018. 2.3, 2.4, 4.1, 5.1, 5.1, 5.2.2, 5.5.4, 7.1

[126] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008. 3.5.1, 4.5.1, 6.4.1, 6.7.15

[127] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018. 3.5.1, 4.5.1, 6.4.1, 6.7.15

[128] Jian Shen, Yanru Qu, Weinan Zhang, and Yong Yu. Wasserstein distance guided representation learning for domain adaptation. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 5.1, 5.5.2, 5.5.3, 5.6, 5.8.4, 5.8.9

[129] Baoxu Shi, Jaewon Yang, Tim Weninger, Jing How, and Qi He. Representation learning in heterogeneous professional social networks with ambiguous social connections. In *IEEE BigData*, 2019. 4.1

[130] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, and Jian Tang. Graphaf: a flow-based autoregressive model for molecular graph generation. *arXiv preprint arXiv:2001.09382*, 2020. 6.1, 6.4.1, 6.5

[131] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 469–478. IEEE, 2016. 3.2.3

[132] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *International conference on artificial neural networks*, pages 412–422. Springer, 2018. 6.4.1, 6.5

[133] Prabhjot Singh, Yanyan Wu, Robert Kaucic, Jiaqin Chen, and Francis Little. Multimodal industrial inspection and analysis. 2007. 7.1

[134] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June 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. 5.8.8

[135] Shuang Song, Kamalika Chaudhuri, and Anand D Sarwate. Stochastic gradient descent with differentially private updates. In *2013 IEEE Global Conference on Signal and Information Processing*, pages 245–248. IEEE, 2013. 6.3.2, 6.4.2, 6.7.4, 6.7.13

[136] Krishna Srinivasan, Karthik Raman, Jiecao Chen, Michael Bendersky, and Marc Najork. Wit: Wikipedia-based image text dataset for multimodal multilingual machine learning. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2443–2449, 2021. 7.3.1

[137] Alexander Strehl and Joydeep Ghosh. A scalable approach to balanced, high-dimensional clustering of market-baskets. In *International Conference on High-Performance Computing*, pages 525–536. Springer, 2000. 3.1

[138] Weijie Su, Xizhou Zhu, Yue Cao, Bin Li, Lewei Lu, Furu Wei, and Jifeng Dai. Vl-bert: Pre-training of generic visual-linguistic representations. *arXiv preprint arXiv:1908.08530*, 2019. 7.4

[139] Mohammed Suhail, Abhay Mittal, Behjat Siddiquie, Chris Broaddus, Jayan Eledath, Gerard Medioni, and Leonid Sigal. Energy-based learning for scene graph generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13936–13945, 2021. 6.4.1, 6.5

[140] Baochen Sun, Jiashi Feng, and Kate Saenko. Return of frustratingly easy domain adaptation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. 5.6, 5.8.9

[141] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: principles and methodologies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 3(2):1–159, 2012. 5.1

[142] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 4.1

[143] George Szekeres and Herbert S Wilf. An inequality for the chromatic number of a graph. *Journal of Combinatorial Theory*, 4(1):1–3, 1968. 3.2.3, 3.6.3

[144] Wang-Chiew Tan, Jane Dwivedi-Yu, Yuliang Li, Lambert Mathias, Marzieh Saeidi, Jing Nathan Yan, and Alon Y Halevy. Timelineqa: A benchmark for question answering over timelines. *arXiv preprint arXiv:2306.01069*, 2023. 7.1

[145] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 990–998, 2008. 5.8.8

[146] Yao-Hung Hubert Tsai, Shaojie Bai, Paul Pu Liang, J Zico Kolter, Louis-Philippe Morency, and Ruslan Salakhutdinov. Multimodal transformer for unaligned multimodal language sequences. In *Proceedings of the conference. Association for Computational Linguistics. Meeting*, volume 2019, page 6558. NIH Public Access, 2019. 7.1, 7.2.1, 7.4

[147] Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal few-shot learning with frozen language models. *Advances in Neural Information*

Processing Systems, 34:200–212, 2021. 7.1, 7.2.1, 7.4

[148] Anton Tsitsulin, John Palowitch, Bryan Perozzi, and Emmanuel Müller. Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904*, 2020. 5.8.7

[149] Anton Tsitsulin, Benedek Rozemberczki, John Palowitch, and Bryan Perozzi. Synthetic graph generation to benchmark graph learning. *WWW'21, Workshop on Graph Learning Benchmarks*, 2021. 5.8.7

[150] Ke Tu, Jianxin Ma, Peng Cui, Jian Pei, and Wenwu Zhu. Autone: Hyperparameter optimization for massive network embedding. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 216–225, 2019. 3.6.3

[151] Robert J Vanderbei. Linear programming foundations and extensions, 2014. 3.4.1, 3.4.1

[152] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 6.1, 6.2.3, 6.3.1, 7.4

[153] Alexei Vazquez, Alessandro Flammini, Amos Maritan, and Alessandro Vespignani. Global protein function prediction from protein-protein interaction networks. *Nature biotechnology*, 21(6):697–700, 2003. 3.1

[154] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575, 2015. 7.3.2

[155] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017. 3.3.1, 4.1, 4.5.1, 6.4.1, 6.7.11

[156] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. A semi-supervised graph attentive network for financial fraud detection. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 598–607. IEEE, 2019. 1, 6.1

[157] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. 5.2.2

[158] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, pages 2022–2032, 2019. 2.3, 5.1, 5.1, 5.2.2, 5.5.4

[159] Yiwei Wang, Shenghua Liu, Minji Yoon, Hemank Lamba, Wei Wang, Christos Faloutsos, and Bryan Hooi. Provably robust node classification via low-pass message passing. In *2020 IEEE International Conference on Data Mining (ICDM)*, pages 621–630. IEEE, 2020. 2.5, 3.3.1

[160] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 4.3.2, 4.3.3

[161] Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony

Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020. 5.8.8

[162] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr, Christopher Fifty, Tao Yu, and Kilian Q Weinberger. Simplifying graph convolutional networks. *arXiv preprint arXiv:1902.07153*, 2019. 3.2.3, 3.5.1, 6.4.1, 6.7.11

[163] Liwei Wu, Hsiang-Fu Yu, Nikhil Rao, James Sharpnack, and Cho-Jui Hsieh. Graph dna: Deep neighborhood aware graph encoding for collaborative filtering. In *AISTAT*, pages 776–787. PMLR, 2020. 4.1

[164] Man Wu, Shirui Pan, Chuan Zhou, Xiaojun Chang, and Xingquan Zhu. Unsupervised domain adaptive graph convolutional networks. In *Proceedings of The Web Conference 2020*, pages 1457–1467, 2020. 5.6

[165] Qitian Wu, Chenxiao Yang, and Junchi Yan. Towards open-world feature extrapolation: An inductive graph learning approach. *Advances in Neural Information Processing Systems*, 34: 19435–19447, 2021. 5.6

[166] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019. 3.1

[167] Qian Xiao, Rui Chen, and Kian-Lee Tan. Differentially private network data release via structural inference. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 911–920, 2014. 6.7.14

[168] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018. 6.4.1, 6.7.11

[169] Carl Yang, Haonan Wang, Ke Zhang, Liang Chen, and Lichao Sun. Secure deep graph generation with link differential privacy. *arXiv preprint arXiv:2005.00455*, 2020. 6.7.14

[170] Carl Yang, Yuxin Xiao, Yu Zhang, Yizhou Sun, and Jiawei Han. Heterogeneous network representation learning: A unified framework with survey and benchmark. *IEEE Transactions on Knowledge and Data Engineering*, 2020. 5.5.1, 5.8.8

[171] Shuwen Yang, Guojie Song, Yilun Jin, and Lun Du. Domain adaptive classification on heterogeneous information networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 1410–1416, 2021. 5.1, 5.6

[172] Yaming Yang, Ziyu Guan, Jianxin Li, Wei Zhao, Jiangtao Cui, and Quan Wang. Interpretable and efficient heterogeneous graph convolutional network. *IEEE Transactions on Knowledge and Data Engineering*, 2021. 5.2.2

[173] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019. 6.3.1

[174] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 2019. 3.1

[175] 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? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021. 7.1

[176] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018. 1, 6.1

[177] Minji Yoon. Graph fraud detection based on accessibility score distributions. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 483–498. Springer, 2021. 2.5

[178] Minji Yoon, Woojeong Jin, and U Kang. Fast and accurate random walk with restart on dynamic graphs with guarantees. In *Proceedings of the 2018 World Wide Web Conference*, 2018. 3.2.3

[179] Minji Yoon, Jinhong Jung, and U Kang. Tpa: Fast, scalable, and accurate method for approximate random walk with restart on billion scale graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018. (document), 3.1, 3.2.3, 3.3

[180] Minji Yoon, Bryan Hooi, Kijung Shin, and Christos Faloutsos. Fast and accurate anomaly detection in dynamic graphs with a two-pronged approach. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 647–657, 2019. 2.5, 3.1

[181] Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. Performance-adaptive sampling strategy towards fast and accurate graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2046–2056, 2021. 6.2.2, 6.4.1, 6.7.3, 6.7.11

[182] Minji Yoon, John Palowitch, Dustin Zelle, Ziniu Hu, Ruslan Salakhutdinov, and Bryan Perozzi. Zero-shot domain adaptation of heterogeneous graphs via knowledge transfer networks. *arXiv preprint arXiv:2203.02018*, 2022. 2.4

[183] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR, 2018. 6.1, 6.5, 6.7.15

[184] Jiaxuan You, Xiaobai Ma, Yi Ding, Mykel J Kochenderfer, and Jure Leskovec. Handling missing data with graph representation learning. *Advances in Neural Information Processing Systems*, 33:19075–19087, 2020. 5.6

[185] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020. 3.6.3

[186] Yingfang Yuan, Wenjun Wang, George M Coghill, and Wei Pang. A novel genetic algo-

rithm with hierarchical evaluation strategy for hyperparameter optimisation of graph neural networks. *arXiv preprint arXiv:2101.09300*, 2021. 3.6.3

[187] Oren Zamir and Oren Etzioni. Web document clustering: A feasibility demonstration. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 46–54, 1998. 3.1

[188] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. 6.5

[189] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019. 6.7.3

[190] 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. *Advances in Neural Information Processing Systems*, 34:19665–19679, 2021. 6.7.3

[191] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 793–803, 2019. 5.1, 5.2.2

[192] Fanjin Zhang, Xiao Liu, Jie Tang, Yuxiao Dong, Peiran Yao, Jie Zhang, Xiaotao Gu, Yan Wang, Bin Shao, Rui Li, et al. Oag: Toward linking large-scale heterogeneous entity graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2585–2595, 2019. 5.5.1, 5.8.8

[193] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019. 2.4

[194] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. 7.3.2

[195] Ziwei Zhang, Xin Wang, and Wenwu Zhu. Automated machine learning on graphs: A survey. *arXiv preprint arXiv:2103.00742*, 2021. 3.6.3

[196] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. Learning on large-scale text-attributed graphs via variational inference. *arXiv preprint arXiv:2210.14709*, 2022. 2.4

[197] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9):3848–3858, 2019. 6.1

[198] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 6, 2017. 3.6.1

[199] Qi Zhu, Natalia Ponomareva, Jiawei Han, and Bryan Perozzi. Shift-robust gnns: Overcoming the limitations of localized graph training data. *Advances in Neural Information Processing Systems*, 34, 2021. 6.4.4, 6.7.11

[200] Xiaojin Zhu. *Semi-supervised learning with graphs*. Carnegie Mellon University, 2005. 5.5.2, 5.6

[201] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 3.6.1

[202] 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 *NIPS*, pages 11249–11259, 2019. 4.1, 4.2, 4.6, 6.7.3