Efficient Learning-based Graph Generation and Beyond

by

Sheng Xiang

A THESIS SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Australian Artificial Intelligence Institute (AAII)
The School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney

April 2025

Certificate of Original Authorship

I, Sheng Xiang, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signed: Production Note:
Signature removed prior to publication.

Date: 4-Apr-2025

Acknowledgements

The past four years at the University of Technology Sydney (UTS) have been unforgettable and invaluable. When I began my Ph.D. in 2021, my English proficiency was far from fluent—I narrowly passed UTS's English test—and I had little idea about what generative models entailed. It is remarkable how, over time, I not only improved my English writing and speaking abilities but also dedicated myself to researching learning-based graph generation models. Simultaneously, deep generative models, often referred to as generative AI, were taking off and leading us into a new era of artificial intelligence. I witnessed the explosive growth of generative AI, moving from laboratory experiments to widespread applications, and felt both excited and a bit overwhelmed to be part of this transformative phase.

I would not have been able to complete this journey without the support and guidance of many individuals, and I am deeply grateful for their contributions.

First and foremost, I extend my heartfelt thanks to my advisor, Ying Zhang, for his invaluable support throughout my Ph.D. career. His hands-on mentoring was instrumental in shaping my research trajectory. He taught me how to conduct cohesive and progressive studies, write clear and readable papers, and refine my writing with precise and appropriate language choices. Ying possesses a unique ability to identify the finer details that needed improvement while maintaining a comprehensive understanding of our research problems. Moreover, he is an incredibly supportive and compassionate mentor who has taught me almost everything I know as a researcher. Despite my lack of confidence at times, Ying has always believed in my capabilities. Having him as my advisor feels like having a slightly older friend by my side, and I am endlessly thankful for his guidance.

I would also like to express my gratitude to Lu Qin and Dawei Cheng professors, who provided me with invaluable assistance and insights during my Ph.D. journey. Lu is a friendly and patient individual whose support has been a constant source of comfort. Dawei, often seen as a "superman", amazes me with his ability to juggle multiple responsibilities simultaneously. I am honored to have received their feedback and guidance.

Additionally, I extend my thanks to Yi Zhang and Hanchen Wang, who served on my candidate assessment committee. Their expertise and willingness to engage with my research have been greatly appreciated.

The impact of teaching assistants cannot be overlooked. During my role as a teaching assistant for the UTS course 42913, I received valuable support from Rong Hu and Hanchen Wang, which significantly contributed to enhancing my understanding of teaching methodologies. Similarly, my experience at UNSW, where I collaborated with Junhua Zhang and other students, helped me improve my English teaching skills.

Collaboration has been a significant lesson and a delightful aspect of my Ph.D. journey. I am particularly grateful to my research partners: Fangzhou Yang, Jin Liu, Yibo Yan, Peng Zhu, Mingzhi Zhu, Chenhao Xu, Guibin Zhang, Yujia Ye, Ruihui Zhao, Yi Ouyang, Yao Zou, Qijun Miao, Jin Ouyang, Jiacheng Ma, and Yihan Dai. Their contributions have enriched both my academic work and personal growth.

I would also like to acknowledge the unconditional support of family members who have shaped me into who I am today. My deepest gratitude goes to my grandparents, parents, and especially Yunting, my fiancée. Our journey together has been filled with shared experiences, from our time at Shanghai Jiao Tong University to successfully pursuing Ph.D. degrees at UTS. Her unwavering support has been a constant source of strength.

To all those who have supported and guided me during these four years, I extend my heartfelt thanks. Your encouragement and mentorship have been instrumental in making this journey possible.

Efficient Learning-based Graph Generation and Beyond

by

Sheng Xiang

Abstract

Graphs serve as powerful models for representing complex relationships among data in various fields, such as social networks, biological systems, and transportation infrastructures. However, generating realistic, large-scale graphs for analysis and simulation purposes remains a significant challenge, particularly in balancing computational efficiency and representational accuracy. To address this challenge, this thesis introduces efficient learning-based methodologies designed to enable scalable and accurate graph generation, aiming to bridge existing research gaps and provide practical solutions.

The first chapter of this thesis lays the groundwork by proposing a comprehensive benchmark for general graph generation models. This benchmark not only consolidates progress in the literature but also provides a rigorous comparison of the performance of existing general graph generators. Through this analysis, we identify key research gaps, particularly the limited focus on capturing the complex properties of real-world networks and the challenge of achieving an optimal trade-off between efficiency and quality in graph generation. After introducing the graph generation benchmark, the remaining chapters of the thesis are organized into two parts: foundations and applications.

The first part focuses on foundational methodologies and spans three chapters. Chapter 2 delves into the development of an efficient learning-based graph generation framework that successfully balances computational efficiency and generation quality. Building on this foundation, Chapter 3 introduces a novel solution for learning the underlying distribution of input graphs, enabling the generation of realistic synthetic counterparts with well-defined community structures. Chapter 4 extends this approach further to address the simulation of temporal graphs, presenting techniques to capture the dynamic nature of temporal relationships effectively.

The second part transits to practical applications of the proposed methodologies, encompassing two chapters. Chapter 5 demonstrates the use of generated temporal graphs in promoting the accuracy of detecting fraudulent financial transactions, showcasing the real-world applicability and potential of the proposed graph generators in addressing critical financial fraud challenges. Chapter 6 explores another application area, employing generated temporal graphs and advanced temporal-heterogeneous graph neural networks for financial time series prediction, further highlighting the versatility of the proposed techniques.

Finally, this thesis concludes by summarizing the contributions and insights gained, while also discussing potential future applications of graph generation. Through its dual focus on foundational methodologies and practical applications, this work aims to advance the state of the art in scalable and efficient graph generation and foster broader adoption across disciplines.

Publications

Journal Articles

1. **Xiang S.**, Wen D., Cheng D., Zhang Y., Qin L., Qian Z., Lin X., "General graph generators: experiments, analyses, and improvements", *The VLDB Journal*, 2021. (**Chapter 1 & 2**)

2. **Xiang S.**, Zhang G., Cheng D., Zhang Y., "Enhancing Attribute-driven Fraud Detection with Risk-aware Graph Representation", *Under minor revision to IEEE Transactions on Knowledge and Data Engineering*, 2024.

3. **Xiang S.**, Xu C., Cheng D., Zhang Y., "Scalable Learning-based Community-Preserving Graph Generation", *Under major revision to IEEE Transactions on Big Data*, 2024

Conference Papers

1. **Xiang S.**, Cheng D., Zhang J, Ma Z., Wang X., Zhang Y., "Efficient Learning-based Community-Preserving Graph Generation", *IEEE 38th International Conference on Data Engineering* 2022. (**Chapter 3**)

2. **Xiang S.**, Xu C., Cheng D., Wang X., Zhang Y., "Efficient Learning-based Graph Simulation for Temporal Graphs", *Accepted by IEEE 41st International Conference on Data Engineering* 2025. (**Chapter 4**)

3. **Xiang S.**, Zhu M., Cheng D., Li E., Zhao R., Ouyang Y., Chen L., Zheng Y., "Semi-supervised Credit Card Fraud Detection via Attribute-Driven Graph Representation", *AAAI 37th Conference on Artificial Intelligence,*, 2023. (**Chapter 5**)

4. **Xiang S.**, Cheng D., Shang C., Zhang Y., Liang Y., "Temporal and Heterogeneous Graph Neural Network for Financial Time Series Prediction", *ACM 31st International Conference on Information and Knowledge Management*, 2022. (**Chapter 6**)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the graph's strong expressive power, a host of researchers in fields such as e-commerce, cybersecurity, social networks, military, public health, and many more, are turning to graph modeling to support real-world data analysis [1–6]. For instance, the graph can be used to model the interactions between compounds and proteins in bioinformatics for drug discovery where each node represents the compound or a protein, and the interactions between them are captured by the edges [7–10]. In a social network, a node can represent a user and an edge can represent the relationship (e.g., friendship) between two users [11–13].

In the graph processing and analytics, a key step is the collection or generation of the graph data. In some applications, it is important to use graph generators (i.e., graph generative models) to generate simulated graphs based on the real-life graph data for two reasons: (1) the inaccessibility of the whole real-life graphs; and (2) a better understanding of the distribution of graph structures and other features. For instance, as highlighted in [14], data acquisition is key step in responsible data management, and it is essential to collect or generate *representative* data. In some scenarios, users are only able to obtain a small sample of the real-life graph due to various limits such as incomplete observability, privacy concern, and company/government policy. It is necessary to use representative graphs with similar size and distribution to the real-life graphs for the training or performance evaluation purpose. For instance, it is a common practice to start the system development and data collection at the same time, especially when the data collection is cost-consuming (e.g., in the counter-terrorist applications) or the two tasks are managed by two separate

teams. To better tune or validate the efficiency and scalability of the algorithms during system development, it is desirable to use representative large-scale simulated graphs before the real large-scale graph is readily available. Another example is the collaboration between the graph computing team at Alibaba Group and the finance data-analysis team at Ant Group for graph pattern-based fraud detection. Specifically, the graph computing team aims to develop efficient and scalable graph pattern-detection algorithms to find abnormal graph patterns in the finance network such that the finance data-analysis team can quickly identify some potential threats. As the finance network data is sensitive and cannot be directly released, the graph simulator has been deployed to generate a large-scale finance network for the graph computing team. Moreover, graph generators can provide a large number of simulated graphs for the training of the graph-based learning models [9, 10]. By learning the distribution of the real-life graphs, the graph generators can also help to better understand the real-life graphs. For instance, graph generators can be used to generate source code [15] and formulas [16], which help to understand insights of the graph data. Graph generators can be used to obtain node representations of large networks [17] and to extract multiple relation semantics from knowledge graphs [18]. Molecular graph generators extract the distribution of compounds and design new and reasonable drugs[8, 9]. Some researchers use graph generators to create neural network structures for the model architecture search [19, 20].

1.1 Motivation

Due to the graph generators' importance in directly related applications, there is a long history of the study of graph generators in many domains such as database, data mining and, machine learning. Readers can refer to a recent survey [1] for a comprehensive overview of this line of research. In this thesis, we focus on *general graph generators* which aim to reproduce structural properties of observed graphs regardless of the domains, something that is fundamental in the study of graph generative models. Despite the existence of many outstanding achievements, we note that there still exist several unsolved important problems. These form the underlying motivation of this thesis.

(1) No comprehensive overview on emerging deep learning-based general graph generators. With the advance of deep learning techniques, advanced generative models such as Autoencoder

and Generative Adversarial Network (GAN) have been widely used for data generation in many fields (e.g., image and audio), significantly enhancing the performance of traditional approaches. Not surprisingly, these techniques have also been applied recently to graph generators (e.g., [21–25]). However, to our best knowledge, there is no comprehensive overview of these emerging deep generative graph models in the literature. For instance, in their recent survey paper [1] on graph generators, the authors of [1] only briefly mention several machine learning-based general graph generators in the sections devoted to challenges and open problems.

(2) No systematic and comprehensive performance comparison of general graph generators. By their nature, graphs are complex, making it difficult to capture explicitly the distribution of observed graphs. For instance, a graph with n nodes can be represented by up to $n!$ equivalent adjacency matrices, each corresponding to a different, arbitrary node ordering/numbering. Moreover, we need to learn distributions over possible graph structures without assuming a fixed set of nodes (e.g., to generate candidate molecules of varying sizes). This being the case, traditional graph similarity metrics (e.g., graph edit distance [26] and maximum common subgraph [27]) cannot be applied to determine whether two graphs are from the same distribution. Therefore, unlike other data distributions where the (dis)similarity of two sets of objects (e.g., point sets in Euclidean space) can be measured directly by a numeric value (e.g., KL divergence [28] and Earth Mover Distance [29]), we have to resort to the distributions of various graphs' properties (e.g., degree distribution); that is, if two graphs are from the same distribution, the corresponding distribution of particular properties (e.g., degree distribution) should be very similar. This creates a large number of metrics for evaluating the likelihood of two graphs from various perspectives, such as Maximum Mean Discrepancy [21] (MMD) between two node-degree distributions. In addition to the value of graph simulation, there are also many evaluation metrics for general graph generators that are critical for decision-making by researchers and practitioners under different application scenarios, such as training time, inference time, scalability, permutation invariance, and tuning difficulty. To the best of our knowledge, existing studies usually consider only a few metrics in their performance evaluations, and many worthy of attention are overlooked. Moreover, the number of competitors and graphs deployed in experiments is rather limited. We also note that there are discrepancies in experimental results reported in certain papers. For instance, it is reported in [22] that the simulation quality of the GRAN [22] outperformed the GraphRNN-S [21] on the protein dataset, while our experiments have different observations.

(3) No algorithm that can achieve a good trade-off between graph simulating quality and efficiency (scalability). Traditional graph generators usually rely on well-defined graph models, and their corresponding graph simulation algorithms are typically efficient and scale well to large graphs. However, these techniques are hand-engineered to model a particular family of graphs and lack the capacity to learn the generative model directly from observed real-life graphs. For instance, the B-A model [30] is carefully designed to capture the scale-free nature of empirical degree distributions, but fails to capture many other aspects of real-world graphs such as community structures. On the contrary, the emerging deep generative models can achieve far superior graph simulating quality, but suffer from low efficiency and scalability when the deep learning techniques are applied to general graph generators. For instance, the graph generators based on RNN (e.g., GraphRNN [21]) need to store long node-ordering to infer the adjacency matrix of the whole graph, which consumes considerable time and space.Although some research efforts aim to enhance the efficiency of deep neural network-based approaches, the results are not very promising in terms of the trade-off between simulation quality and efficiency (scalability). For instance, GRAN [22] accelerates graph simulation by generating a block of nodes per step, but it still needs to infer the whole graph, whose adjacency matrix requires $O(n^2)$. Due to the limit of floating-point operations per second (FLOPS) and GPU's memory, it cannot generate a graph with more than 10^5 nodes in our experimental environment.

(4) No handy toolkit for the users of general graph generators. To increase the impact of a specific type of technique, support from handy software libraries or toolkits is critical in enabling users to apply these techniques easily to their applications. One well-known example is the development of the LIBSVM library for the support vector machine (SVM) technique [31] in Machine Learning. Though source codes of many existing general graph generators are publicly available, from our research, there appears to be no handy software toolkit for users to apply existing general graph generators easily to their applications. Moreover, there is no end-to-end platform such that users can easily integrate their newly developed general graph generators for a comprehensive performance evaluation with existing approaches.

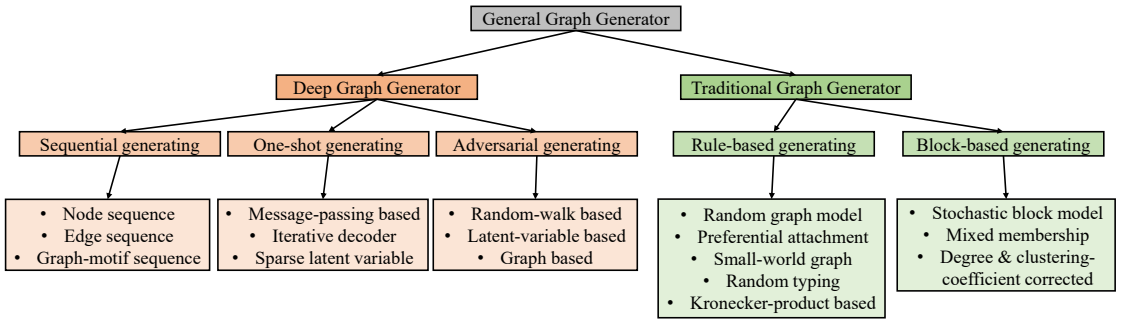


FIGURE 1.1: A classification of graph generators.

1.2 Taxonomy of Graph Generation

For a better understanding the techniques of the representative graph generators, in this subsection, we provide a deep taxonomy as illustrated in Figure 1.1, which consists of 5 categories and 17 sub-categories.

1.2.1 Sequential Generating

Sequential generating means evolutionarily modeling a graph G; that is, relying on an existing incomplete graph to generate new elements. A graph G is represented by a sequence of elements, i.e., $S_N = \{s_1, ...s_N\}$. Then the new element is generated with $s_{N+1} = f(S_N)$, where f denotes the generative model (e.g., RNN, MLP). Following are three sub-categories:

1. **Node sequence.** GraphRNN [21] and [8, 32, 33] generate a graph through generate nodes and associated edges one-by-one. We select GraphRNN as representative of this category because it can generate graphs with more than 1000 nodes (compared with [8, 32]) and has better performance and influence in generating general graphs (compared with [33]).

2. **Edge sequence.** BiGG [23] and [34, 35] model a graph as a sequence of edges. We select BiGG as representative of this category because it achieve better performance and scalability in generating general graphs (compared with [34, 35]).

3. **Motif sequence.** GRAN [22] and [10, 36] generate graph motifs (e.g., a block of nodes and associated edges) sequentially to generate a complete graph. We select GRAN as representative of this category because it is more scalable than [10, 36].

1.2.2 One-shot Generating

One-shot generating means modeling an entire graph directly, that is, the elements on the graph are generated with no sequential dependencies. Corresponding generators can be further classified into 3 sub-categories:

1. **Message-passing based encoder.** VGAE [17] and [37] are proposed to generate one graph in one shot through message-passing based encoder. We choose VGAE as representative because it first proposed graph convolutional technology to generate graphs.

2. **Iterative decoder.** Graphite [38] generates graphs through reserve message-passing decoder. Graphite is selected as representative because its influence and contribution on new technology, i.e., iterative decoder.

3. **Sparse latent variables.** SBMGNN [39] generates graphs through modeling sparse latent variables. SBMGNN is selected as representative because its contribution on preserving community structure through sparse latent variables.

1.2.3 Adversarial Generating

Adversarial generating means training graph generator through a game between generator and discriminator. Corresponding models can be divided into the following 3 sub-categories:

1. **Random-Walk based.** NetGAN [24] and [40] generate random walks through adversarial training. We choose NetGAN as a representative because it has better performance in generating general graphs.

2. **Latent-variable based.** ARVGA [41] is selected because of its contribution on adversarial training for latent variables.

3. **Graph based.** CondGEN [25] is selected because of its contribution on adversarial training for generated graphs.

1.2.4 Rule-based Generating

Rule-based generating means modeling a graph through explicit operations and a small number of parameters. This category's graph generators model graphs through selecting samples from pre-defined graph families [11, 30, 42]. They can be further classified into the following 5 sub-categories:

1. **Random graph model.** E-R [43] generate each edge by sampling from a Bernoulli distribution parameterized with a fixed value. We choose E-R, B-A, and W-S as representatives because of their influence.
2. **Preferential attachment graph model.** B-A [30] randomly add edges for new node to generate scale-free graphs.
3. **Small world graph model.** W-S [44] randomly rewrite edges from cycle graph to generate a small-world graph.
4. **Random typing graph model.** RTG [45] generates edge-list through random-typing process. We select RTG because of its influence and technical contribution.
5. **Kronecker-product based.** R-MAT [46], Kronecker [42] and [47] model a graph through a regressive dropping edges mechanism similar to kronecker-product. We select R-MAT and Kronecker as representatives because of their influence.

1.2.5 Block-based Generating

Block-based generating means modeling a graph through modeling "blocks", i.e., a subset of nodes. Corresponding models can be classified into the following 3 sub-categories:

1. **Vanilla stochastic blockmodel** SBM [48] was first prposed to model social networks.
2. **Mixed-membership.** MMSB [49] models a graph with mixed-membership of blockmodel. MMSB provides a generalization of SBM to have better quality of simulating graphs.

3. **Degree & clustering-coefficient corrected.** DCSBM [50] was proposed to correct the blockmodel with obsedved degree distributions. BTER [51] was proposed to correct the average clustering coefficient in each block, and correct the degree distribution through a two-level edge sampling process.

1.3 Background of Graph Generator

1.3.1 Problem Definition and Notations

We define a graph $G = (V, E)$. V denotes a set of n nodes (vertices), and a set of m edges $E \subseteq V \times V$, where a tuple $e = (u, v) \in E$ represents an edge between two vertices u and v in V. The graph G can also be represented by an adjacency matrix $\mathbf{A} \in \{1, 0\}^{n \times n}$. As reflected in the literature, we assume G is an undirected graph, and hence the adjacency matrix of the graph is symmetric. Additionally, we denote the (optional) node-feature matrix associated with the graph as $\mathbf{X} \in \mathbb{R}^{n \times d}$ where n denotes the number of nodes and d denotes the dimension of the node feature. We denote the initiator matrix of the graph G by $\mathbf{M}_I$.

Problem Statement. Given a set of observed graphs $\{G\}$, a general graph generator aims to learn a generative model to capture the structural distribution of the graphs, such that a set of new graphs $\{G'\}$ with similar structural distribution can be generated.

Ideally, a general graph generator should be able to generate new graph which has exactly the same distribution as the observed graphs. Nevertheless, it is notoriously difficult to tell if two graphs are from the same distribution due to the complex nature of graph structure. In practice, we have to resort to representative evaluating metrics in the literature in our experiments, each of which aims to quantitatively capture the likelihood of two graphs (graph distributions) from one perspective (e.g., degree distribution). Hopefully, a good graph generator should be able to generate new graphs with the same distributions regarding all the above metrics. Moreover, the generating algorithm should be efficient and scalable such that the users can efficiently handle large-scale graph in real-life applications.

1.3.2 Scope

As shown in the recent survey [1], existing graph generators can be classified into two categories: general graph generators (e.g., [17, 21, 24, 45]) and domain-specific graph generators (e.g., [8, 13, 40, 52, 53]). Notably, general graph generators aim to mimic the structures of the observed graphs so that the generated graph can reproduce such properties of the preserved graphs as degree distribution and the path length distribution regardless of the domains. Domain-specific graph generators consider particular domains such as semantic web (e.g., [52]), graph database (e.g., [53]), temporal graphs (e.g., [40]), and social networks (e.g., [13]).

To make a comprehensive yet focused comparison of graph generators, here we mainly consider *general graph generators*. We classify existing general graph generators into four categories based on their foundation technique. For each category, the key features and main characteristics, as well as a set of representative approaches, are introduced in detail in Section 1.4. To make a comprehensive performance evaluation, we include a large sample of representative graph datasets and evaluation metrics as observed in the literature covering experiments with general graph generators.

1.4 General Graph Generators

In the section 1.2, we discuss the taxonomy of graph generation. To make a comprehensive comparison, it is necessary to classify them in terms of algorithm. In this section, we describe the key features of general graph generators in each category and summarize their main characteristics. For each category, we present the details of its representative generators.

1.4.1 Simple Model-based Generator

Simple model-based generators rely on some well-known simple graph models (families), such as Binomial graphs [43], randomized small-world graphs [44] and preferential attachment graph model [30]. Usually, each family of graph models can capture one or a few properties of the real-life graphs well. For instance, the B-A model can generate free-scale networks, and the W-S model can capture the key characteristics of the small-world graphs. Below are details of 8 representative techniques in this category.

E-R. Binomial graph model (E-R) was first proposed and studied by Erdős et al [43]. Each edge of the binomial graph is generated independently with the probability $P(\mathbf{A}_{i,j} = 1) = p$, where the constant p is controlled by the user. Given an undirected and no self-loop graph with n vertices and m edges, the parameter p is immediately available with $p = \frac{2m}{n(n-1)}$. The naive implementation of E-R based graph generator generates an edge with probability p for each pair of vertices, with time complexity $O(n^2)$ and space complexity $O(m+n)$. As shown in [54], an efficient algorithm with time complexity $O(m+n)$ is used in practice.

W-S. Small-world (W-S) graph model was first proposed by Watts and Strogatz [44]. It can simulate the random connection of real graphs by re-sampling edges, and make the distribution of the generated graph between the E-R graph and completely regular graph by adjusting the sampling probability and the number of connected edges. The implementation of W-S-based graph generator randomly resets the target nodes of each source node from a regular circle graph, having a time complexity $O(k \times n)$ and space complexity $O(m+n)$, where k is the number of edges for each source node.

B-A. The preferential-attachment [30] (B-A) graph model needs to generate graph nodes sequentially and add a fixed number of edges to new nodes to generate a scale-free graph with the power-law distribution. The implementation of a B-A based graph generator has the time complexity of $O(k \times n)$ and the space complexity of $O(m+n)$, where k is the number of edges a new node will attach.

RTG. The Random-Typing Generator [45] (RTG) creates a two-dimension (2d) keyboard to type words, that are represented as edges of a graph. The graph generation process is modeled as the process of typing words. There are four parameters (K, W, q, β) to control the generative distribution. K is the number of possible characters in one word, and W is the number of words. q is the probability of typing space. β controls the probability of randomly typing diagonal characters and non-diagonal characters on the 2d keyboard. These parameters help generate graphs with power-law degree distribution and communities. Moreover, RTG can also generate bipartite graphs. The implementation of the RTG-based graph generator requires $O(m \log n)$ time complexity and $O(m+n+K^2)$ space complexity.

BTER. The Block Two-level Erdős-Rényi model [51] (BTER) was proposed to simulate both the realistic graph's degree distribution and its clustering coefficient by degree. These two elements are extracted directedly from observed graphs. BTER creates affinity blocks first, to assign degrees to nodes. Then BTER assigns edges in and between affinity blocks to match the degree distribution. The clustering coefficient per degree is assigned when creating links between nodes in the same affinity blocks. The implementation of the BTER-based graph generator has a time complexity $O(m+n)$ and space complexity $O(m+n)$.

SBM & DCSBM. The Stochastic Block Model [48] (SBM) generates a random graph based on the probability matrix of the block $\mathbf{B}$ and the number of nodes of each block. The SBM-generated graph is considered as a connected set of E-R graphs with community structure. The Degree-corrected Stochastic Block Model [50] (DCSBM) sets the degree of each node following the Stochastic Block Model, the process thereby tuning the degree distribution of the generated graph. To obtain the block probability matrix $\mathbf{B}$, we choose the best block partition by maximizing the modularity [55] of the observed graphs, with linear time complexity of m. After the number of blocks and the probability of generating edges between blocks are determined, the time complexity for generating one graph of SBM and DCSBM is $O(n^2)$ in their naive implementations. As shown in [50], an efficient implementation of the (DC) SBM-based graph generator has a $O(m+n)$ time complexity and $O(m+n)$ space complexity.

R-MAT. The recursive graph model [46] (R-MAT) divides the adjacency matrix recursively and determines a graph by dropping edges into one quadrant of four recursively. As for the parameter optimization of the R-MAT model, we use the empirical parameters mentioned by the author as the initial matrix $\mathbf{M}_I$. R-MAT generates one graph with n nodes and m edges by randomly sampling edges with time complexity $O(m\log(n))$ and space complexity $O(m+n)$. R-MAT also inspired Dai et al. [23] to compress one row of adjacency matrix into a binary tree.

1.4.2 Complex Model-based Generator

With more parameters and complex model architecture, we know statistics learning and parameterized models can simulate and generate the adjacency matrix of one graph numerically [42, 49]. Although there are $n!$ permutations of adjacency matrices to represent the same graph, nevertheless they successfully achieve better simulation quality than simple model-based generators. In

addition to being represented as an adjacency matrix, the graph structure can also be modeled as a decision-making process. In this process, nodes and edges are generated sequentially, which is easy to represent and learn for regular graphs. Below we introduce 5 representative complex model-based general graph generators.

MMSB. The Mixed Membership Stochastic Blockmodel [49] uses a probabilistic model to build a graph. First, the mixed-membership vector π_i of the node i is sampled from the Dirichlet distribution with $\pi_i \sim \text{Dirichlet}(\alpha)$, where α denotes the parameters. Then the indicator of the edge is obtained by sampling from a multinomial distribution with $z_{i,j} \sim \text{Multinomial}(\pi_i)$. Finally, the edge is sampled from the Bernoulli distribution, the probability of which is obtained by a bilinear function of edge indicators with $\mathbf{A}_{i,j} \sim \text{Bernoulli}(z_{i,j}^T \mathbf{B} z_{i,j})$, where $\mathbf{B} \in \mathbb{R}^{K \times K}$ represents the probability of generating one edge between two blocks.

The MMSB maximizes a-posteriori to approximate its parameters, but its hyper-parameter K, the number of blocks, is difficult to select when simulating large graphs. We use a *Louvain* community detection algorithm [55] to select the number of blocks K. Because the MMSB aims to approximate the observed adjacency matrices, it has a time complexity of $O(n^2)$ and space complexity of $O(n^2)$ to update its parameters. Note that the inference algorithm of MMSB can easily be paralleled.

Kronecker. The Kronecker-graph model uses the Kronecker product to build a graph, where the initiator graph is self-connected and has a binary matrix to represent the edges of the graph. The initiator matrix of the stochastic Kronecker graph is not binary, which denotes the probability of generating one edge. Different from R-MAT, the sum of the initiator is not 1 and the probability of each edge can be calculated independently as follows:

$$P(\mathbf{A}_{i,j} = 1) = \prod_{k=0}^{|\log n|-1} \mathbf{M}_I[\lfloor \frac{i-1}{2^k} \rfloor \pmod 2 + 1, \lfloor \frac{j-1}{2^k} \rfloor \pmod 2 + 1]. \tag{1.1}$$

In the optimization stage, the Kronecker graph model looks for the best node permutations and uses the maximum likelihood principle to estimate the parameters of the initiator matrix. To generate

graphs faster, the Kronecker model imitates R-MAT to throw edges recursively into the adjacency matrix, which accelerates the generation process of the stochastic Kronecker graph, with the time complexity $O(mlog(n))$ and space complexity $O(m+n)$.

GraphRNN. Two recurrent neural networks (RNN) are deployed by GraphRNN. The first, called *graph-level* RNN, is used to store generated nodes and to generate new nodes. The second, called *edge-level* RNN, is used to store generated edges on the newly generated node and infer new edges. Each edge is sampled from the Bernoulli distribution, which is parameterized by the output of the *edge-level* RNN. At this point, the graph generation is modeled as a decision sequence. Assuming that h_0 and $h_{i,0}$ represent the initial graph state and node i's hidden state, respectively, GraphRNN's generation process can be formulated as follows:

$$\begin{aligned} h_i &= \text{RNN}_1(h_{i-1}, \mathbf{A}_{i-1}), \\ \theta_{i,j} &= \text{RNN}_2(h_{i,j-1}, \mathbf{A}_{i,j-1}), \quad (h_{i,0} = h_i) \\ \mathbf{A}_{i,j} &\sim \text{Bernoulli}(\theta_{i,j}), \quad (j < i) \end{aligned} \tag{1.2}$$

where $\mathbf{A}_{i-1}$ and $\mathbf{A}_i$ denote the adjacency vectors of the last node and next generated node, respectively. In the GraphRNN, Gated Recurrent Unit [56] (GRU) is used to encode the graph state and infer node-adjacency vectors.

For graph generation with no edge dependence, authors propose a variant named GraphRNN-S which replaces the second RNN with a multi-layer perceptron (MLP). Then the adjacency vector $\mathbf{A}_i$ will be sampled from a multivariate Bernoulli distribution parameterized by θ_i. The generation process of GraphRNN-S can be formulated as follows:

$$\begin{aligned} h_i &= \text{RNN}(h_{i-1}, \mathbf{A}_{i-1}), \\ \theta_{i,j} &= \text{MLP}(h_i), \\ \mathbf{A}_i &\sim \text{Bernoulli}(\theta_i) \end{aligned} \tag{1.3}$$

The variant performs well in generating protein graphs and other real-world graphs, which show less edge dependence than grid and community graphs. GraphRNN-S also promoted subsequent works [22, 23] to scale the complex model-based graph generator up to larger graphs.

After modeling the graph as a sequence of decisions, the most important problem is how to prevent the order of nodes from affecting the generalization performance of the model. GraphRNN utilizes Breadth-first Search (BFS) ordering to reduce the number of permutations of the observed graphs, then maximizes the likelihood of the edge dependence and node permutations. It generates the nodes and edges of the graph step-by-step and requires $O(n^2)$ time complexity in each epoch. Thus, the complexity for training and inference is $O(e \times n^2)$ and $O(n^2)$, respectively, where e is the number of epochs used in training. Due to the necessary dependence on nodes and edges, the generation process of complete GraphRNN can not be carried out in parallel. Therefore, the follow-up work is devoted mainly to improving the scalability of GraphRNN.

GRAN. Graph Recurrent Attention Networks [22] (GRAN) aims to improve the performance of GraphRNN by addressing the following issues: (1) low generalization due to strong dependence on node orderings and edges; (2) expressive capability of only one canonical node ordering; and (3) poor parallelization compared with a graph autoencoder. To reduce its dependence while retaining the expressiveness of the graph auto-regressive model (e.g., GraphRNN), GRAN leverages graph attention networks [57] (GAT) to infer the parameters of Bernoulli distributions when generating the whole graph. GRAN uses t steps to generate a graph, and each step generates B nodes, called a block. If $B > 1$, the generation process can accelerate by commencing the next block in the S-th row of the last generated block, called stride ($1 \leq S \leq B$). At t-th generation step, GRAN reduces the embedding size of previous nodes to generate large graphs by a linear mapping:

$$\begin{aligned} L_i &= [L_{b_i,1}...L_{b_i,B}], \\ h_i &= \mathbf{W}L_i + b, \quad \forall i < t \end{aligned} \tag{1.4}$$

where $[\cdot]$ denotes a concatenation operation of vectors and $L_i \in \mathbb{R}^{Bn}$ is a vector concatenated by the output vectors of a block of nodes. The initial node representations are updated regressively with $h_i = \text{GRU}(h_i, \text{GAT}(h_i))$, as with [58]. After updating node representations, to express the edge dependences in one block, GRAN models the probabilities of generating edges through a

mixture of Bernoulli distributions:

$$\begin{aligned} p(L_{b_t}|L_{b_1},...,L_{b_{t-1}}) &= \sum_{k=1}^{K} \alpha_k \prod_{i \in b_t} \prod_{j \leq i} \theta_{k,i,j}, \\ \alpha_1,...,\alpha_K &= \text{Softmax}(\sum_{i \in b_t, j \leq i} \text{MLP}_\alpha(h_i - h_j)), \\ \theta_{1,i,j},...,\theta_{K,i,j} &= \text{Sigmoid}(\text{MLP}_\theta(h_i - h_j)) \end{aligned} \tag{1.5}$$

where K is the number of mixture components. When $K > 1$, the edges generated in parallel are no longer independent because of the latent mixture components, which maintains the edge dependence without loss of parallelization.

To learn the graph generative model under more than one canonical node orderings, GRAN proposes a new objective to maximize a lower bound of log-likelihood as follows:

$$\log p(G) = \log \sum_{\pi} p(G, \pi) \geq \log \sum_{\pi \in \Omega} p(G, \pi) \tag{1.6}$$

where Ω denotes the selected canonical orderings of the graph node. The greater the quantity of canonical orderings picked, the tighter the bound will be.

Inspired by the parallelism of Graph Neural Networks (GNN), GRAN provides a flexible trade-off between computational cost and generative performance through adjusting the block size and stride length S, so that it requires a time complexity of $O(\frac{n^2}{S})$ in each epoch. Its efficiency and scalability are significantly better than GraphRNN when generating large graphs, e.g., graphs with more than 500 nodes under our experiment setting.

BiGG. Inspired by the recursive graph model [46] (R-MAT), BiGG is a graph auto-regressive model with a tree structure. Assuming that G is a large sparse graph ($m \ll n^2$), generating only the edges of G is a more efficient choice than generating each node's adjacency vector and can be formulated as follows:

$$p(\mathbf{A}) = p(e_1)p(e_2|e_1)...p(e_m|e1,...,e_{m-1}) \tag{1.7}$$

where each edge $e_i = (u, v)$ includes the indices of two nodes. Therefore, the generation process contains m steps. In previous work, a single edge can be factorized with $p(e_i) = p(u)p(v|u)$ and

$p(v|u)$ is assumed to be simple multinomials over n nodes, which will result in the complexity of $O(n)$. BiGG reduces the number of decisions of specifying v through formulating $p(v|u)$ as follows:

$$p(v|u) = \prod_{i=1}^{\lceil \log_2 n \rceil} p(x_i = x_i^v) \tag{1.8}$$

where $x_i^v \in \{\text{left, right}\}$ denotes the i-th decision in the sequence of node v and $p(x_i = x_i^v)$ denotes the probability of the i-th decision leading to v. Note that $x_i^v =$ left (right) means the left (right) sub-tree is chosen in the i-th decision of node v. BiGG uses $E_u = \{(u, v) \in E\}$ to represent the set of edges connecting node u and $\mathcal{N}_u = \{v|(u, v) \in E_u\}$ to represent the set of neighbor nodes of u. For each of node u's row of adjacency matrix, generating all node u's edges E_u is equivalent to generating a node u's binary tree $\mathcal{T}_u$, where for each $v \in \mathcal{N}_u$ the generation process starts from the root node and ends in a leaf node. Each node t is generated with its left subtree $\text{lch}(t)$ and previously generated nodes as conditioning. The right subtree $\text{rch}(t)$ is generated after generating the left subtree and its dependencies, similar to the in-order traversal of the binary tree. Let $\text{context}_u(t)$ and $\text{context}_u(\text{lch}(t))$ represent the previous context and the summary context of the left subtree of node t, respectively. Then the recursively generated $p(E_u)$ can be formulated as follows:

$$\begin{aligned} p(E_u) =& p(\mathcal{T}_u) \\ =& \prod_{t \in \mathcal{T}_u} p(\text{lch}(t)) p(\text{rch}(t)) \\ =& \prod_{t \in \mathcal{T}_u} p(\text{lch}(t)|\text{context}_u(t)) \\ & p(\text{rch}(t)|\text{context}_u(t), \text{context}_u(\text{lch}(t))). \end{aligned} \tag{1.9}$$

where $p(\text{lch}(t)|\cdot)$ and $p(\text{rch}(t)|\cdot)$ are Bernoulli distributions parameterized by TreeLSTM networks [59]. So far, each row of adjacency matrix $\mathbf{A}$ can be generated recursively through the construction of binary tree and $p(E) = \prod_{u=1}^{n} p(E_u)$ costs $O(m \log n)$ time. The full model is going to generate the adjacency matrix row by row. Similarly, BiGG models the root nodes of n *edge-binary* trees as the summary context of nodes. It also models the summary context into a *row-binary* tree recursively, which costs $O(n \log n)$ time. BiGG generates a graph requiring $O((m + n) \log n)$ time complexity, which is especially efficient when generating large sparse

graphs. Each depth of parameters in the binary tree can be updated in parallel, resulting in $O(\log n)$ steps, which is more efficient than $O(n)$ steps in GRAN and GraphRNN-S.

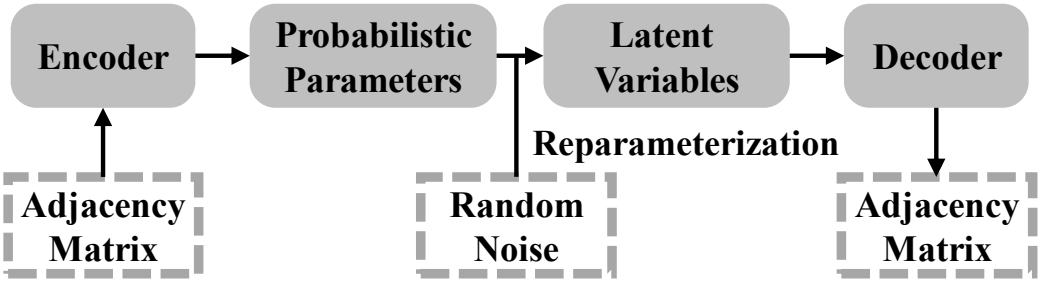


FIGURE 1.2: A summary of Autoencoder-based generators.

1.4.3 Autoencoder-based Generator

The complex model-based graph generators encountered a bottleneck of generative performance until the advent of the graph autoencoders (GAEs). Thanks to the progress of variational autoencoder and deep learning [60, 61], researchers extend autoencoder to the field of graph representation learning and graph generation. Meanwhile, graph neural networks [4, 6, 57, 62] are also proven to be successful on graph representative and generative models.

Fig. 1.2 illustrates the framework of the autoencoder based generators. Specifically, an encoder is used to learn the representation (i.e., encoding) of the observed graphs, and the probabilistic parameters are captured by a deep graph neural network (GNN). By adding some random noise, we can reparameterize the latent variables and reconstruct (i.e., decode) a new graphs with the decoder. The parameters of GNNs will be updated according to the accuracy of the simulation. The above process is repeated till a new graph with high quality (e.g., small reconstruction error) can be generated. Below, we introduce three representative autoencoder based graph generators.

VGAE. The Variational Graph Autoencoder [17] was proposed by Kipf et al. to naturally extract node features and infer the generative distribution of observed graphs. VGAE uses a GNN [62] as the encoder of graph data. The GNN layer parameterized by $\mathbf{W}$ is defined as follows:

$$\text{GNN}_{\mathbf{W}}(\mathbf{X}, \mathbf{A}) = \bar{\mathbf{D}}^{-\frac{1}{2}}\bar{\mathbf{A}}\bar{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W} \tag{1.10}$$

where $\bar{\mathbf{A}} = \mathbf{A} + \mathbf{I_n}$ denotes the adjacency matrix with an added self-loop and $\bar{\mathbf{D}}$ is degree matrix of $\bar{\mathbf{A}}$ with $\bar{\mathbf{D}}_{i,i} = \sum_j \bar{\mathbf{A}}_{i,j}$. The VGAE uses two layers of GNN to infer the parameters of the

stochastic variables $\mathbf{Z} \in \mathbb{R}^{n \times f}$, where f denotes the dimension of latent variables. For the GAE model, the inference model of $q(\mathbf{Z}|\mathbf{X}, \mathbf{A})$ is parameterized as follows:

$$\mathbf{Z} = \text{GNN}_{\mathbf{W_2}}(\sigma(\text{GNN}_{\mathbf{W_1}}(\mathbf{X}, \mathbf{A}))) \tag{1.11}$$

where σ denotes the non-linear activation function. Then the generative model is defined as a bilinear function of nodes' latent variables with $P(\mathbf{A}_{i,j} = 1) = \sigma(\mathbf{Z}_i \mathbf{Z}_j^T)$, where $\mathbf{Z}_i$ denotes the latent variables of node i. VGAE has two optimization objectives, one is to reconstruct the adjacency matrix and the other is to approximate the a priori distribution. That leads to optimizing the variational lower bound as follows:

$$L = E_{q(\mathbf{Z}|\mathbf{X},\mathbf{A})}[\log p(\mathbf{A}|\mathbf{Z})] - KL[q(\mathbf{Z}|\mathbf{X}, \mathbf{A})||p(\mathbf{Z})] \tag{1.12}$$

where $p(\mathbf{A}|\mathbf{Z})$ is the generative distribution with $p(\mathbf{A}|\mathbf{Z}) = \prod_i \prod_j p(\mathbf{A}_{i,j}|\mathbf{Z}_i, \mathbf{Z}_j)$, $KL(\cdot||\cdot)$ denotes the Kullback-Leibler divergence measuring the distance between two distributions, and $p(\mathbf{Z})$ is a Gaussian prior with $p(\mathbf{Z}) = \prod_{i=1}^{n} \mathcal{N}(\mathbf{Z}_i|0, \mathbf{I})$. VGAE needs $O(n^2)$ time to generate a new graph with space $O(m + n)$, and it takes $O(n^2)$ time in each epoch of the training process.

Graphite. Working similarly to VGAE, the iterative generative model of graphs [38] (Graphite) parameterizes the variational autoencoders with graph neural networks. The main contribution of Graphite is that it replaces the inner-product decoder of VGAE with node representations and intermediate graphs. The decoding process is formulated as follows:

$$\mathbf{Z}^* = \text{GNN}_\theta(\hat{\mathbf{A}}, [\mathbf{Z}|\mathbf{X}]), \text{ with } \quad \hat{\mathbf{A}} = \frac{\mathbf{Z}\mathbf{Z}^T}{||\mathbf{Z}||^2} + \{1\}^{n \times n} \tag{1.13}$$

where θ denotes the parameters of GNN and $[\cdot|\cdot]$ means a concatenation operation. $\hat{\mathbf{A}}$ is an intermediate graph, to which is added a constant of 1 to keep the matrix non-negative. The feature matrix $\mathbf{Z}^*$ can be refined gradually until getting the final features to generate an adjacency matrix.

Graphite is consistent with VGAE in encoder and optimization objectives. It still has a time complexity $O(n^2)$ in each epoch and space complexity $O(m+n)$ due to the inner product involved although the implementation of the iterative decoder is accelerated to $O(n \times f^2)$, where f is the dimension of the latent features.

SBMGNN. Nikhil et al. [39] proposed a *sparse* variational autoencoder for graphs by merging the interpretability of SBM and the fast inference of graph neural networks. As in MMSB, SBMGNN uses a stick-breaking construction of the Indian Buffet Process [63] to infer the size of community memberships, which is formulated as follows:

$$\begin{aligned} v_k &\sim \text{Beta}(\alpha, 1), \quad k = 1, ..., K \\ b_{nk} &\sim \text{Bernoulli}(\pi_k), \pi_k = \prod_{j=1}^{k} v_j \end{aligned} \tag{1.14}$$

where K is the number of communities and π_k is the probability of all memberships. Unlike MMSB, SBMGNN infers the variational parameters of these distributions through graph neural networks. SBMGNN also uses the variational graph autoencoder (VGAE) to approximate the dense latent variables r_n. In contrast to VGAE, SBMGNN models the node embeddings as $z_n = b_n \odot r_n$ with remaining other sections consistent. So far, the inference process can be defined as follows:

$$\begin{aligned} q_\phi(v_{nk}) &= \text{Beta}(v_{nk}|\text{GNN}_\alpha(\mathbf{X}, \mathbf{A}), \text{GNN}_\beta(\mathbf{X}, \mathbf{A}) \\ q_\phi(b_{nk}) &= \text{Bernoulli}(b_{nk}|\text{GNN}_\pi(\mathbf{X}, \mathbf{A})) \\ q_\phi(r_n) &= \mathcal{N}(\text{GNN}_{\mu_n}(\mathbf{X}, \mathbf{A}), \text{diag}(\text{GNN}_{\sigma_n^2}(\mathbf{X}, \mathbf{A}))) \end{aligned} \tag{1.15}$$

The graph generation process of SBMGNN is the same as other autoencoder-based graph generators. The overall optimization objective of this inference and generative model can be formulated as the sum of KL divergence of these approximating distributions and the reconstruction loss, which can be extended from the objective of VGAE. SBMGNN has a time complexity of $O(n^2)$, and has realized the model's interpretability with the cost of more model parameters.

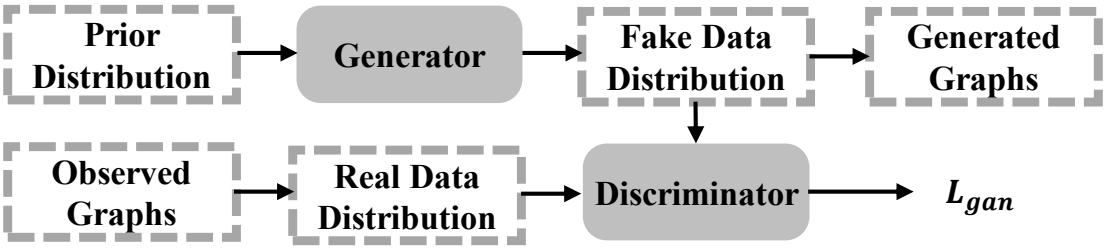


FIGURE 1.3: A summary of Generative Adversarial Network (GAN) based generators.

1.4.4 GAN based Generator

The core of generative adversarial networks [64] (GANs) is to use a discriminator to generate fake data with good quality and robustness. As shown in Fig. 1.3, the key idea of the GAN based graph generators is to use the graph generator to establish the mapping from the random variable to the fake hidden variable of the graph, and put it into the discriminator with the encoded hidden variable of the real graph. The objective of these models is to make the generator generate with stability and produce realistic hidden variables, which can be decoded to simulate the realistic graphs. Below are three representative GAN-based generators.

ARVGA. The adversarial regularized variational graph autoencoder [41] (ARVGA) was proposed to generate embeddings of the graph. Given a graph G, the hidden variable matrix $\mathbf{Z}$ is obtained by using the same method of graph encoding used in VGAE. Then the discriminator will repeatedly update its parameters by optimizing following cross-entropy cost:

$$\mathbb{E}_{p(z)\sim q(\cdot|\mathbf{Z})} \log(1 - D(z)) + \mathbb{E}_{p(a)\sim \mathcal{N}(\cdot|\mathbf{0},\mathbf{I})} \log D(a) \tag{1.16}$$

where z, a are the sample vectors from $\mathbf{Z}$ and real data distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$, respectively, and D is built on a standard multi-layer perceptron (MLP). Before each update of parameters of the graph autoencoder, the parameters of the discriminator are updated for multiple times.

The main difference between ARVGA and VGAE is that the former regularizes the output of the encoder directly into a priori distribution through a discriminator, not just through KL divergence to approximate a priori distribution. The generated robust embeddings are proved to have better performance on link prediction and node clustering than VGAE. The time complexity of ARVGA is $O(n^2)$ in each epoch. Thus, the complexity of training and graph inference are $O(e \times n^2)$ and $O(n^2)$ respectively. Recall that e denotes the number of epochs required in the training process.

NetGAN. NetGAN [24] is the first model to generate graphs through random walks. It leverages long short-term memory [65] (LSTM) to generate random walk sequences. The longer the sequence length of random walks, the more topology information is captured by LSTM. Note that when the sequence length is 2, NetGAN will directly learn the edge probabilities.

As the input of NetGAN's generator component, the initial cell state C_0 and the initial hidden state h_0 of NetGAN are mapped from a multivariate normal distribution as follows:

$$\begin{aligned} z &\sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \\ C_0 = \text{MLP}_{(C)}(z), &\quad h_0 = \text{MLP}_{(h)}(z) \end{aligned} \tag{1.17}$$

where $\text{MLP}_{(\cdot)}$ consists of two linear layers and *tanh* activation. Then LSTM parameterized by θ can start inferring the random walks as follows:

$$\begin{aligned} (p_1, C_1, h_1) &= \text{LSTM}(C_0, h_0, \mathbf{0}), \\ v_1 &\sim \text{Cat}(\text{Softmax}(p_1)), \\ (p_T, C_T, h_T) &= \text{LSTM}(C_{T-1}, h_{T-1}, v_{T-1}), \\ v_T &\sim \text{Cat}(\text{Softmax}(p_T)), \end{aligned} \tag{1.18}$$

where Cat denotes a categorical distribution. So far, the generator $\mathcal{G}$ can sequentially generate random walks $(v_1, ..., v_T)$. However, p_T and v_T have a dimension of n, resulting in a high computation cost in LSTM. Therefore, an up-project matrix $\mathbf{W}_{up} \in \mathbb{R}^{h \times n}$ is used to map the output of LSTM $o_T \in \mathbb{R}^h$ into $\mathbb{R}^n$. A down-project matrix $\mathbf{W}_{down} \in \mathbb{R}^{n \times h}$ is used to map node v_t into a low-dimensional input of LSTM. Generated random walks and real random walks are fed into the discriminator $\mathcal{D}$ parameterized by another LSTM, which outputs a probability of the random walk's being real. The model parameters are trained through Wasserstein GAN [66] (WGAN) framework.

After updating the parameters of the generator $\mathcal{G}$, inferred random walks through $\mathcal{G}$ can be decoded as new graphs through assembling the adjacency matrix. A score matrix $\mathbf{S}$ represents the appearance probability of each edge in generated random walks. Then each edge (i, j) is sampled from the categorical distribution parameterized by $p_{(\cdot,\cdot)}$ with $p_{(i,j)} = \frac{s_{i,j}}{\sum_{u,v} s_{u,v}}$. The edges of the whole graph are generated by selecting the top m entries of the score matrix. Because of the edge-sampling strategy of NetGAN, the graph generation process has a complexity of $O(n^2)$, and the numbers of nodes and edges are fixed to n and m, respectively.

CondGEN. The conditional variational autoencoder with a generative adversarial network (CondGEN) was proposed in [25]. The encoding and generation of the conditional graph structure are

also considered. Permutation-invariance and generating arbitrary size of graphs are the main contributions of CondGEN, and are achieved by modifying the derivation of stochastic latent variable $\mathbf{Z}$ of VGAE below:

$$\begin{aligned} \bar{\mu} &= \frac{1}{n}\sum_{i=1}^{n} g_\mu(\mathbf{X}, \mathbf{A})_i, \\ \bar{\sigma}^2 &= \frac{1}{n^2}\sum_{i=1}^{n} g_\sigma(\mathbf{X}, \mathbf{A})_i^2, \\ q(z_i|\mathbf{X}, \mathbf{A}) &\sim \mathcal{N}(\bar{z}|\bar{\mu}, \text{diag}(\bar{\sigma}^2)) \end{aligned} \tag{1.19}$$

where $g(\mathbf{X}, \mathbf{A}) = \text{GNN}_{\mathbf{W}_2}(\text{ReLU}(\text{GNN}_{\mathbf{W}_1}(\mathbf{X}, \mathbf{A})))$ is a two-layer GNN model. The modeling of $\bar{z}$ is essential for preserving permutation-invariance and can be regarded as the *graph embedding* of the graph G. Samples from $\bar{z}$ can also be decoded into a new graph through an FNN-based decoder. For the adversarial optimizing objectives, different from ARVGA, CondGEN is designed to learn the generative distribution of observed data as follows:

$$\begin{aligned} L_{gan} =& \log(\mathcal{D}(\mathbf{A})) + \log(1 - \mathcal{D}(\mathcal{G}(\mathbf{Z}_p)))+ \\ & \log(1 - \mathcal{D}(\mathcal{G}(\mathbf{Z}_q))) \end{aligned} \tag{1.20}$$

where $\mathcal{D}$ is a two-layer GNN followed by a two-layer FNN and $\mathbf{Z}_p$ and $\mathbf{Z}_q$ are the latent variables sampled from both the Gaussian prior and the latent variable distribution $q(z_i|\mathbf{X}, \mathbf{A})$, respectively. CondGEN is designed to learn the structure distributions of a set of graphs, then generate permutation-invariant graphs, which can be controlled by the corresponding conditions. Due to the spectral embeddings' derivation in CondGEN, it has a training time complexity $O(n^3)$ at each epoch. In our experiment, we leverage CondGEN to generate new graphs without any encoding process. Thus, CondGEN has an inference time complexity $O(n^2)$.

1.5 Contributions

In this chapter, we aim to address the four problems above and our principal contributions are summarized as follows:

- We give a systematic and fair empirical evaluation of existing general graph generators, including recently emerged deep learning-based approaches. We group existing methods into four categories based on their foundation techniques. For each category, we describe the key features of generators, representative approaches and summarize their main characteristics.

- We conduct systematic and comprehensive experiments to compare the performance of general graph generators. Specifically, 20 representative general graph generators in all categories are evaluated; 12 popular graph datasets and 17 representative evaluation metrics are deployed in experiments. We also provide easy-to-follow standard recommendations about how to select the general graph generator under different settings. We believe such a comprehensive experimental evaluation is beneficial to both scientific communities and practitioners.

- The experience and insights we gained throughout the study enable us to engineer a new algorithm, Scalable Graph Autoencoder (SGAE), which can achieve a satisfactory trade-off between the graph simulation quality and efficiency (scalability).

- We implement an end-to-end platform for researchers and practitioners such that not only can they apply a variety of existing general graph generators directly to their applications but can also easily integrate their own-built general graph generators for comprehensive performance comparison and analytics. We believe this will greatly benefit future research and the application of graph generators.

- We develop a novel method to learn the underlying community distribution of input graphs, enabling the generation of synthetic graphs with well-defined community structures while maintaining real-world properties.

- We extend our approach to handle temporal graphs, introducing techniques to effectively capture and simulate the dynamic nature of real-world network evolution over time.

- We demonstrate practical applications of our proposed methods in critical domains such as financial fraud detection and time series prediction, showcasing their versatility and real-world impact.

Part I

Efficient Graph Generation: Foundations

Chapter 2

Efficient Graph Generation Models

2.1 Chapter Overview

This chapter provides a comprehensive foundation for understanding general graph generators, including their properties, limitations, and comparative analysis. This chapter has been published in [67]. Section 2.2 summarizes various graph generators based on time and space complexity, permutation invariance, and community-preserving properties. Section 2.3 introduces a novel scalable graph autoencoder (SGAE), which achieves a trade-off between simulation quality and efficiency. Section 2.4 evaluates the performance of existing and proposed graph generators across multiple datasets and metrics.

2.2 Summary of Existing Graph Generators

In this section, we provide a summary of graph generators in terms of time complexity and space complexity. We also evaluate important properties of graph generators including *permutation invariance* and *community preserving*.

Time Complexity. In Table 2.1, we report the time complexity of each generator for the leaning (training) process and the new graph inference process. Generally, there is no training process for the simple model-based generators and they can easily calculate the desired parameters with

TABLE 2.1: Complexity, scalability, and permutation invariance of general graph generators. n and m are the amount of nodes and edges, respectively. k is the amount of edges attached to the previous node. K is the number of characters in the keyboard. B is the amount of blocks. m is the amount of edges. S is the length of stride.

Graph Generator	Training Time Complexity per Epoch	Inference Time Complexity	Space Complexity	Permutation Invariance	Preserving Community
E-R [43]	–	$O(m+n)$	$O(m+n)$	✓	
W-S [44]	–	$O(nk)$	$O(m+n)$	✓	
B-A [30]	–	$O(nk)$	$O(m+n)$		
RTG [45]	–	$O(m \log n)$	$O(m+n+K^2)$		
BTER [51]	–	$O(m+n)$	$O(m+n)$		
SBM [48]	–	$O(m+n)$	$O(m+n+B^2)$		✓
DCSBM [50]	–	$O(m+n)$	$O(m+n+B^2)$		✓
R-MAT [46]	–	$O(m \log n)$	$O(m+n)$	✓	
MMSB [49]	$O(n^2)$	$O(n^2)$	$O(n^2)$		✓
Kronecker [42]	$O(m \log n)$	$O(m \log n)$	$O(m+n+\log n)$	✓	
GraphRNN [21]	$O(n^2)$	$O(n^2)$	$O(n^2)$		
GRAN [22]	$O(n^2)$	$O(n^2)$	$O(m+n)$		
BiGG [23]	$O(n^2)$	$O(n^2)$	$O(m+n+\log n)$		
VGAE [17]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓	✓
Graphite [38]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓	✓
SBMGNN [39]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓	✓
ARVGA [41]	$O(n^2)$	$O(n^2)$	$O(m+n)$	✓	✓
NetGAN [24]	$O(n^2)$	$O(n^2)$	$O(n^2)$	✓	✓
CondGEN [25]	$O(n^3)$	$O(n^2)$	$O(m+n)$	✓	

time complexity $O(m+n)$. Due to the simplicity of the model, its inference time is very efficient as well. It takes much more learning and inference time for generators from other categories. A dominant cost of many graph generators is the generation of the adjacency matrix, leading to a time complexity $O(n^2)$ in the inference process. For deep neural network-based generators, the learning time is also determined by the structure of the networks and the number of epochs. Particularly, GNN and RNN are two types of deep neural networks used by existing general graph generators, with time complexity $O(m+n)$ and $O(n^2)$, respectively, at each epoch. Due to the spectral embeddings' derivation in CondGEN, it has time complexity $O(n^3)$ at each epoch of the training process. We remark that the total training time also relies on the number of epochs required. In our experiments, GraphRNN takes much more training time compared to CondGEN due to the former's greater number of epochs involved in the training process. Moreover, the practical performance of the graph generators also depends on whether they can be easily paralleled in the system.

Space Complexity. Table 2.1 reports the space complexity of each generator. Generally, several generators are very space-efficient as they only need to keep the observed graphs with space $O(m+n)$. Note that for deep neural network-based generators, we assume the dimensionality of the latent variables (i.e., embeddings) of vertices is a constant (usually 32, 64 or 128 in practice). For graph neural networks (GNNs), we store the adjacency matrix and identity matrix as a sparse matrix, which costs $O(m+n)$ instead of $O(n^2)$. Thus, the corresponding space of the GNN-based

generator is $O(m + n)$. For general recurrent neural networks (RNNs), such as GraphRNN-S and GRAN, we store the long-term memory of a sequence of nodes, with a space consumption dependent on the number of nodes. Thus, including the observed graphs, the corresponding space of the RNN-based generator is $O(m + n)$. It is also shown that MMSB, GraphRNN and NetGAN are the most space-consuming generators because MMSB and GraphRNN need to maintain the probabilistic graph for all n nodes, while NetGAN needs to assemble a score matrix with $O(n^2)$ space complexity.

Permutation Invariance Property. Table 2.1 shows the generators with the permutation invariance property. The permutation invariance property implies that we can set an arbitrary order of graph nodes for the leaning process, and it has no effect on the simulation and generation results of the graph generator. Graph generators with the permutation invariance property can generalize to large graphs without considering the order of the nodes, which may come up with $O(n!)$ possible instances.

Community Preserving Property. Table 2.1 also indicates the generators with the community preserving property. Graph generators need to preserve community structures of the observed graphs well. For instance, simulated graphs with similar community structures can be utilized to enhance community detection.

2.3 Improvement

The experience and insights gained from this study enable us to engineer a new method, namely Scalable Graph Autoencoder (SGAE), which can achieve a good trade-off between graph simulation quality and efficiency (scalability).

Our proposed method follows the framework of VGAE [17], with new techniques in the following three aspects.

Encoder. We first leveraged Graph Neural Networks (GNNs) to encode graph and infer node representations. Here the implementation of GNNs is formulated as follows:

$$\begin{aligned} \bar{\mathbf{A}} &= \bar{\mathbf{D}}^{-\frac{1}{2}}(\tilde{\mathbf{A}})\bar{\mathbf{D}}^{-\frac{1}{2}} \\ \mathbf{X}_{i+1} &= \text{GNN}_i(\mathbf{X}_i, \bar{\mathbf{A}}) = \bar{\mathbf{A}}\mathbf{X}_i\mathbf{W}_i \end{aligned} \tag{2.1}$$

where $\tilde{\mathbf{A}}$ is a self-loop adjacency matrix with $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_n$, $\bar{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$ with $\bar{\mathbf{D}} = \sum_j \tilde{\mathbf{A}}_{i,j}$, $\mathbf{X}_0$ is set default as $\mathbf{I}_n$, and $\mathbf{W}_i$ is the parameters of the i-th layer of GNNs. To handle the over-smoothing problem, we use the PairNorm [68] layer to retain the distance of node representations after each layer of GNNs. The normalization procedure is formulated as follows:

$$\begin{aligned} x_i^c &= x_i - \frac{1}{n}\sum_{i=1}^{n} x_i \\ x_i^p &= s \cdot \frac{x_i^c}{||x_i^c||^2} \end{aligned} \tag{2.2}$$

where x^i is the i-th node's representation, x_i^c is the node-wise centered node representation, x_i^p is the feature-wise normalized node representation, and s is the hyperparameter to adjust the distance between node representations. Our proposed encoder adds the PairNorm layer before each activation layer. Note that the first layer of GNNs requires the maximum usage of memory. For example, $\mathbf{A}$ requires $O(m)$, $\mathbf{X}_0$ requires $O(n)$, and $\mathbf{W}_0 \in \mathbb{R}^{n \times d}$ requires $O(n \times d)$, where d is the dimension of node representations. Therefore, the time and space complexity outcomes of the encoder are $O(n + m)$ and $O(m + n \times d)$, respectively.

Decoder. In this work, we choose to decode one subgraph with n_s nodes per epoch when training our proposed GAE. After obtaining the node representations from the encoder per epoch, we choose n_s nodes as our temporary ground truth subgraph $\mathbf{A}_s$. The corresponding output of the decoder is calculated as follows:

$$\begin{aligned} P(\mathbf{A}_{i,j} = 1) &= \sigma(\mathbf{Z}_i\mathbf{Z}_j^T) \\ P(\hat{\mathbf{A}}) &= \prod_{i,j \in V_s} P(\mathbf{A}_{i,j} = 1) \end{aligned} \tag{2.3}$$

where σ denotes the Sigmoid activation function, $\hat{\mathbf{A}}$ is the estimated adjacency matrix of the subgraph with n_s nodes, and V_s denotes the set of nodes in the subgraph.

As mentioned by Guillaume Salha et al. [69], nodes with high degrees need to be trained more frequently to avoid losing important node information. Thus, we choose subgraphs with a strategy according to node degrees as follows: $P_i = \frac{deg_i}{\sum_{i=1}^{n} deg_i}$, where P_i is the probability to select node i. In every epoch, we randomly sample nodes to assemble one subgraph for the training purpose. Note that n_s is an important hyperparameter to make a trade-off between efficiency and effectiveness. However, generating the whole graph after the entire training process still requires $O(n^2)$ time and space complexity. To address this issue, we follow the implementation of assembling adjacency matrix in NetGAN [24] and improve it as follows: (i) We obtain the latent variables of n nodes; (ii) We decode one row of the adjacency matrix through sampling edges for each node, and then clear the memory of zero-entry; (iii) We repeat the step (ii) until all rows are generated. This procedure requires $O(m+n)$ space complexity, which is affordable in generating large graphs.

Optimization. Since the loss calculated by each epoch is biased from the real loss $\mathcal{L}$, we use an approximate loss function $\mathcal{L}_{n_s}$ to optimize the model parameters. After each sampling, the connection status of the to-be-simulated subgraph has changed. Therefore, we dynamically sample the negative edges based on the subgraph to speed up the training process. Now the approximate loss function is formulated as follows:

$$\begin{aligned}\mathcal{L}_{n_s} = \frac{1}{m_s}[&\sum_{(i,j)\in E_{pos}} (1-\hat{\mathbf{A}}_{i,j}) \log \hat{\mathbf{A}}_{i,j} \\ &+ \sum_{(i,j)\in E_{neg}} \hat{\mathbf{A}}_{i,j} \log(1-\hat{\mathbf{A}}_{i,j})]\end{aligned} \tag{2.4}$$

where m_s denotes the number of edges in the subgraph, $\mathbf{A}_s$, E_{pos} and E_{neg} are the sets of positive and sampled negative edges from $\mathbf{A}_s$, respectively, and $\hat{\mathbf{A}}_{i,j}$ is the estimated probability derived from the decoder.

Overall, our proposed graph generator is a autoencoder-based and we give it the name, Scalable Graph Autoencoder (**SGAE**). SGAE inherits the excellent expression performance of the graph generator based on the neural network, and meanwhile achieves a significant speed up in the training process compare to other deep neural network-based generators. Particularly, the training time complexity is $O(n+m)$ and $O(n_s^2)$ in each epoch for encoder and decoder, respectively,

where n_s is the size of the sampled subgraph during the training process. The inference time complexity remains $O(n^2)$, and the space complexity of SGAE is $O(m + n + n_s^2)$. Like other autoencoder-based generators, SGAE has the permutation-invariance and community preserving properties.

2.4 Evaluation

We integrated all included models and conducted extensive experiments to evaluate the performance of graph generators. The details of our evaluation platform are provided in Section 2.4.1. The experiment setup is provided in Section 2.4.2. Experiments were conducted in graph simulation quality, preserving community structure, parameter sensitivity, and model scalability in Section 2.4.3, Section 2.4.4, Section 2.4.5, and Section 2.4.6, respectively. Section 2.4.6 investigates the efficiency and the scalability of the graph generators. Finally, according to our comprehensive experimental study, a roadmap of recommendations is provided in Section 2.5 for users.

2.4.1 Toolkit Used for Performance Evaluation.

To help researchers and practitioners apply the general graph generators in their applications or make a comprehensive evaluation of their proposed general graph generators, we implement an end-to-end platform that is now publicly available[1]. In the detailed instruction of this toolkit, we show: (i) how to apply an existing general graph generator in user's application; (ii) how to include a new, developed general graph generator; and (iii) the details of the evaluation metrics and how to include them for performance evaluation. Currently, a python interface is provided in our package and other programming languages will be considered in the future.

Below we briefly introduce the characteristics of the platform.

Modularized Pipelines. We specify the data type as the list of Graph objects implemented under the Networkx library [70]. The graph generators and evaluation metrics are implemented with the same type of input and output. The experimental result can be obtained directly by one-line command including a dataset, a graph generator, and a specific evaluation metric.

[1] `https://github.com/xiangsheng1325/GraphGenerator`

TABLE 2.2: Benchmark datasets included in the experiments.

Category	Dataset	#Nodes	#Edges
Citation Networks	Cora [71]	2708	5429
	Citeseer [71]	3327	4732
	Pubmed [71]	19717	44338
	Cora-ML [72]	2810	7981
Biological Networks	Protein [73]	620	1098
	PPI [74]	2361	6646
Social Networks	Deezer [75]	47538	222887
	Facebook [75]	50515	819090
Other Datasets	3D Point Cloud [76]	5037	10886
	Autonomous System [77]	6474	12572
	EU Email [77]	265214	364481
	Google Pages [78]	875713	4322051

Customization and Extension. We allow users to incorporate their own datasets, graph generators, and new evaluation metrics into their local libraries by integrating their own implementations into respective source code scripts. We also welcome other developers to contribute to our platform on Github. Note that all experiments in this chapter are conducted on this platform.

2.4.2 Experiment Setup

We introduce the experimental datasets, metrics and parameter settings in this subsection.

2.4.2.1 Dataset

We have collected several representative graph datasets used by existing general graph generators, which are shown in Table 2.2. Details of each dataset are provided as follows.

- **Citation Networks** are undirected graphs that consist of papers and their citation relationships. The Cora and Cora-ML datasets contain 2708 and 2810 machine learning publications, respectively. The Citeseer and Pubmed datasets contain 3327 and 19717 publications, respectively.

- **Biological Networks** are graph-structure data extracted by real biological information. Protein dataset contains 620 nodes, each node denoting an amino acid. There are edges between amino acids when their distances are less than 6 Angstroms. Protein-protein Interaction (PPI) dataset contains 2361 nodes, each node representing one yeast protein. Edges are generated if there are interactions between two proteins.

- **Social Networks** are graph-stucture data extracted by real social relationships. Deezer dataset contains 47538 nodes and each node signifies a user. The edges designate the friendship among users. The Facebook dataset contains 50515 nodes, each node denoting one page. Edges are generated if there are mutual likes among them.

- **Other Datasets** are graph-structure data from real objects. 3D point cloud dataset contains 5037 nodes, denoting the points of a household object. Edges are generated for k-nearest neighbors which are measured w.r.t Euclidean distance of the points in 3D space. Autonomous system dataset contains 6474 nodes, which represent routers of computer networks. Edges denote the communication among routers.

Note that, for some observed data with isolated nodes or self-loop edges, experiments on Recurrent Neural Network (RNN)-based graph generators (e.g. GraphRNN [21] and BiGG [23]) and NetGAN cannot be conducted successfully. Therefore, the self-loop edges of all datasets are removed. The largest connected component is selected as the input of the graph generative model.

2.4.2.2 Evaluating Metrics

We collected and designed appropriate evaluating metrics to measure the difference between the original graph and the generated graph. The metrics used for graph simulation quality can be categorized into the following four aspects.

- **Node Distributions** are measured by using the maximum mean discrepancy (MMD) over *Degree*, *Clustering Coefficient*, *Spectral Embedding*, *Betweenness Centrality*, and *Closeness Centrality* distributions. The squared MMD between two sets of samples from distributions

TABLE 2.3: Evaluation results on Protein dataset.

Graph Generator	Degree.	Cluster.	Orbit	Spec.	Between.	Close.	Charact. Path Len.	Gini Coeffi.	Power-law Expo.
E-R	$5.01e^{-2}$	1.92	$6.23e^{-2}$	0.142	0.755	$3.42e^{-2}$	17.5	$8.86e^{-2}$	0.12
W-S	0.139	1.96	0.97	0.203	0.115	0.144	5.25	$2.46e^{-2}$	0.842
B-A	$5.54e^{-2}$	1.77	1.18	0.327	0.754	$7.83e^{-2}$	19.4	0.159	1.43
RTG	$9.71e^{-2}$	1.29	0.282	0.351	0.781	$9.40e^{-2}$	17.8	$9.89e^{-2}$	$3.28e^{-2}$
BTER	$1.91e^{-2}$	1.01	0.270	0.263	0.234	$6.12e^{-2}$	10.2	$9.63e^{-2}$	$3.01e^{-2}$
SBM	$5.38e^{-2}$	1.18	0.102	0.175	0.733	$3.20e^{-2}$	12.4	$4.87e^{-2}$	$5.13e^{-2}$
DCSBM	$9.43e^{-2}$	0.89	0.366	0.215	0.749	$4.27e^{-2}$	13.3	0.142	$8.14e^{-2}$
R-MAT	0.258	0.98	1.18	0.323	0.707	0.131	19.7	0.335	0.258
Kronecker	0.101	1.95	1.96	0.268	0.771	$5.74e^{-2}$	18.5	0.132	$3.12e^{-2}$
MMSB	0.116	1.94	0.326	0.199	0.758	$6.25e^{-2}$	17.9	0.173	0.186
VGAE	0.447	1.56	1.87	0.6	0.535	0.351	18.2	0.477	0.126
Graphite	0.498	2	2	0.629	0.591	0.422	20.1	0.473	0.134
SBMGNN	0.513	1.6	1.94	0.65	0.668	0.374	20.5	0.52	0.209
GraphRNN	$2.32e^{-2}$	**0.407**	$6.77e^{-2}$	$5.74e^{-2}$	0.159	$2.18e^{-2}$	4.85	**1.16**e^{-2}	**2.12**e^{-2}
GraphRNN-S	**1.08**e^{-2}	0.443	**2.92**e^{-3}	**5.61**e^{-2}	**3.68**e^{-2}	**1.87**e^{-2}	**1.26**	$2.45e^{-2}$	$4.41e^{-2}$
GRAN	$4.33e^{-2}$	1.2	0.734	0.183	0.722	$3.13e^{-2}$	18.3	$1.71e^{-2}$	0.631
BiGG	$5.31e^{-2}$	1.87	$7.99e^{-2}$	0.119	0.737	$3.21e^{-2}$	17.3	$7.32e^{-2}$	0.176
ARVGA	0.465	1.33	1.28	0.42	0.766	0.263	19.5	0.501	0.105
NetGAN	$3.77e^{-2}$	1.51	0.128	0.136	0.774	$3.30e^{-2}$	16.8	$6.37e^{-2}$	$5.42e^{-2}$
CondGEN	0.312	1.15	1.1	0.442	0.547	0.376	20.8	0.362	0.295
SGAE	$3.54e^{-2}$	1.62	0.23	0.585	0.606	0.348	19.8	0.298	0.469

TABLE 2.4: Evaluation results on Autonomous System dataset.

Graph Generator	Degree.	Cluster.	Orbit	Spec.	Between.	Close.	Charact. Path Len.	Gini Coeffi.	Power-law Expo.
E-R	$6.75e^{-2}$	0.127	2	$6.57e^{-2}$	0.778	0.328	8.03	0.222	**2.44**e^{-2}
W-S	**8.54**e^{-3}	0.127	2	$7.37e^{-2}$	0.697	0.277	97.2	0.344	0.284
B-A	$8.81e^{-2}$	0.127	1.17	$2.33e^{-3}$	0.63	0.142	5.16	0.187	1.22
RTG	$6.89e^{-2}$	$9.45e^{-2}$	2	$9.78e^{-2}$	0.901	0.269	5.05	0.27	1.13
BTER	0.310	$4.78e^{-2}$	2	$7.32e^{-2}$	0.814	0.215	3.76	0.13	0.967
SBM	0.264	0.105	2	$6.11e^{-2}$	0.778	0.387	3.25	0.31	0.651
DCSBM	$7.89e^{-2}$	$3.51e^{-2}$	1.32	**9.49**e^{-5}	0.536	0.184	0.477	$3.89e^{-2}$	0.271
R-MAT	$5.16e^{-2}$	$6.17e^{-2}$	1.21	$3.53e^{-3}$	**0.102**	0.153	**2.83**e^{-2}	$5.46e^{-2}$	0.498
Kronecker	0.126	$7.77e^{-2}$	1.08	$1.44e^{-3}$	0.166	0.175	0.251	**3.75**e^{-2}	0.743
MMSB	$6.26e^{-2}$	$5.40e^{-2}$	1.97	$1.84e^{-4}$	0.361	**5.51**e^{-2}	$6.50e^{-2}$	$8.01e^{-2}$	0.145
VGAE	$8.04e^{-2}$	$2.60e^{-2}$	1.59	$5.22e^{-4}$	0.41	$9.54e^{-2}$	0.453	0.184	$3.37e^{-2}$
Graphite	0.121	**1.92**e^{-2}	2	$7.31e^{-4}$	0.462	0.123	0.746	0.218	$7.33e^{-2}$
GraphRNN	-	-	-	-	-	-	-	-	-
GraphRNN-S	-	-	-	-	-	-	-	-	-
GRAN	$7.23e^{-2}$	$8.58e^{-2}$	**0.99**	$2.67e^{-3}$	0.155	0.13	0.364	$3.83e^{-2}$	0.5
BiGG	0.109	0.119	1.89	$5.33e^{-2}$	0.769	0.343	3.27	0.25	0.461
ARVGA	0.201	$3.31e^{-2}$	1.12	$9.86e^{-4}$	0.412	0.158	1.18	0.218	0.336
NetGAN	$2.72e^{-2}$	$6.89e^{-2}$	1.07	$6.32e^{-4}$	0.157	$7.66e^{-2}$	0.436	$5.97e^{-2}$	0.251
CondGEN	-	-	-	-	-	-	-	-	-
SGAE	$2.52e^{-2}$	$3.47e^{-2}$	1.01	$6.93e^{-2}$	0.296	0.46	0.912	0.273	0.845

p and q can be derived as:

$$\begin{aligned} \text{MMD}^2(p||q) =& \mathbb{E}_{x,y\sim p}[k(x,y)] + \mathbb{E}_{x,y\sim q}[k(x,y)] \\ & -2\mathbb{E}_{x\sim p,y\sim q}[k(x,y)]. \end{aligned} \tag{2.5}$$

where k denotes the associated kernel. We use the earth mover's distance (EMD) as the Gaussian kernels, which is formulated as:

$$EMD(p,q) = \inf_{\gamma \in \prod(p,q)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||] \tag{2.6}$$

where $\prod(p,q)$ denote the set of all distributions whose marginals are p and q, respectively, and γ is a transport plan.

- **Graphlet Distributions** are measured through computing the number of occurrences of all graphlets within 4 nodes and using *Orbit* MMD to formulate the distance between two distributions with a Gaussian kernel of the total variation (TV), which is formulated as:

$$\text{TV}(p,q) = \mathbb{E}_i[||\pi_p(i) - \pi_q(i)||] \tag{2.7}$$

where $\pi_p(i)$ denotes probability of the i-th graphlet in graph distribution p.

- **Graph Statistics** are measured by 3 metrics: *Characteristic path length* (CPL), *Gini Coefficient* (GINI), and *Power-law Exponent* (PLE). CPL denotes the average value of the minimum path length of total node pairs. GINI denotes the inequality in the nodes' degree distribution. PLE is the exponent of the power-law distribution. All metrics reported in the experiments represent the distances between generated graphs and observed graphs.

- **Community Structures** are measured in two steps: modeling community structure and compare the differences between community ownerships of nodes. For one generated/observed graph, we first use the louvain [55] community detection algorithm to obtain its community memberships of nodes. Then we leverage *Normalized Mutual Information* (NMI) and *Adjusted Rand Index* (ARI) to measure the similarity of the community structure between two graphs.

Note that for all these graph simulation quality-related metrics, the smaller value is preferred in the performance evaluation. For all MMD-based evaluation metrics, the standard deviation of *Orbit* is set to 30, and the other standard deviations are set to 1.0. We use the implementation of *Orbit* counting in [21] to calculate 4-node graphlets to improve efficiency. We repeat sampling

200 nodes from the observed graph to estimate the betweenness centrality of each node. Other settings of evaluating metrics are the same as [23] and [24].

Apart from the graph simulation quality-related metrics, we also evaluate the performance of the graph generators from many other perspectives including training time, inference time (i.e., the time used for generating a new graph), scalability, space (i.e., memory consumption), robustness, community preserving, stableness and sensitivity.

2.4.2.3 Parameter Settings

This section introduces the configuration and parameter settings. By default, we use the best parameter setting given by the original authors. The graph generators and evaluating scripts are implemented and compiled through Python-3.6, PyTorch-1.8.1, CUDA-11.1, and GCC-4.8.5 in our experiments. The experiments are operated on a machine with Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 80 GB RAM and NVIDIA RTX 3090 with 24 GB memory. We use one CPU core and one GPU for every algorithm.

The initiator matrix $\mathbf{M}_I$ of R-MAT is $\{0.90.3, 0.30.1\}$ by default. Following [21–23], the *maximum previous number of nodes* in RNN-based graph generators (e.g., GraphRNN, GRAN, and BiGG) is set as the number of nodes in the observed graph. The stride of GRAN is 1. For all autoencoder-based graph generators and ARVGA, the node attributes $\mathbf{X}$ of each graph are configured based on the matrix $\mathbf{I}_n$, which means each node is represented by a one-hot vector. Moreover, in the training process of GAEs and ARVGA, 20% of observed edges are being masked. For assembling a graph in NetGAN, the number of random walks sampled from the trained model is 1000. We leverage spectral embeddings as the input node features of CondGEN.

2.4.3 Graph Simulation Quality

This section evaluates the quality of simulated graphs via a set of evaluation metrics. For each generator, we report the average result value of 10 repeated experiments.

First, we report the experimental results for all graph simulation quality-related metrics. Table 2.3 summarizes all graph generators' performances on the Protein data set. According to the 14th and 15th rows of Table 2.3, we can see that GraphRNN and its variant GraphRNN-S outperform other

TABLE 2.5: Evaluation results of *Degree* MMD on all 12 datasets.

Graph Generator	Cora	Citeseer	Pubmed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	EU Email	Google
E-R	$3.79e^{-3}$	$1.18e^{-2}$	$4.38e^{-2}$	$1.88e^{-2}$	$1.06e^{-2}$	$4.41e^{-2}$	$3.61e^{-2}$	0.11	$9.71e^{-2}$	$1.66e^{-2}$	$6.40e^{-2}$	$6.16e^{-2}$
W-S	$3.62e^{-2}$	$1.81e^{-2}$	$2.44e^{-2}$	$5.25e^{-2}$	$3.71e^{-2}$	$3.26e^{-2}$	0.128	0.14	0.23	**2.09**e^{-3}	$6.27e^{-2}$	$9.28e^{-2}$
B-A	$2.09e^{-2}$	$2.17e^{-2}$	$7.20e^{-2}$	$3.86e^{-2}$	$1.34e^{-2}$	$7.09e^{-2}$	$4.33e^{-2}$	$8.03e^{-2}$	0.151	$2.19e^{-2}$	$8.97e^{-3}$	$6.75e^{-2}$
RTG	$3.65e^{-2}$	$2.82e^{-2}$	$4.16e^{-3}$	$2.73e^{-2}$	0.141	$1.93e^{-2}$	$7.37e^{-2}$	0.101	0.232	$2.93e^{-2}$	$5.37e^{-4}$	-
BTER	$9.44e^{-3}$	$2.02e^{-3}$	$8.02e^{-3}$	$9.31e^{-3}$	$1.37e^{-2}$	$4.16e^{-3}$	$1.37e^{-4}$	$3.28e^{-3}$	$7.03e^{-2}$	$4.76e^{-3}$	$3.33e^{-2}$	$1.58e^{-3}$
SBM	$8.59e^{-3}$	$1.26e^{-2}$	$6.55e^{-2}$	$2.87e^{-2}$	$1.52e^{-2}$	$5.34e^{-2}$	$3.71e^{-2}$	$7.55e^{-2}$	$8.24e^{-2}$	$6.85e^{-2}$	0.113	$5.86e^{-2}$
DCSBM	$6.99e^{-3}$	$8.10e^{-3}$	$1.52e^{-2}$	**3.99**e^{-3}	$2.58e^{-2}$	$7.83e^{-3}$	$8.34e^{-4}$	$9.27e^{-4}$	$8.18e^{-2}$	$1.95e^{-2}$	$6.29e^{-2}$	$2.44e^{-3}$
MMSB	$6.79e^{-3}$	$8.27e^{-3}$	-	$5.58e^{-3}$	$2.85e^{-2}$	$8.11e^{-3}$	-	-	$9.36e^{-2}$	$1.52e^{-2}$	-	-
R-MAT	$2.62e^{-2}$	$1.42e^{-2}$	$2.84e^{-3}$	$1.71e^{-2}$	$6.57e^{-2}$	$2.96e^{-3}$	$4.12e^{-2}$	$2.25e^{-2}$	0.175	$1.31e^{-2}$	$3.85e^{-2}$	$1.51e^{-2}$
Kronecker	$1.32e^{-2}$	$1.24e^{-2}$	$1.17e^{-2}$	$4.49e^{-3}$	$2.38e^{-2}$	$4.00e^{-3}$	$2.93e^{-3}$	$1.34e^{-2}$	0.102	$3.40e^{-2}$	$6.35e^{-2}$	$6.67e^{-3}$
VGAE	$5.11e^{-2}$	$8.65e^{-2}$	0.158	$5.14e^{-2}$	0.115	$5.71e^{-2}$	-	-	0.213	$2.15e^{-2}$	-	-
Graphite	$5.44e^{-2}$	$7.58e^{-2}$	0.176	$5.02e^{-2}$	0.138	0.1	-	-	0.196	$3.11e^{-2}$	-	-
SBMGNN	$8.10e^{-2}$	0.112	0.128	$6.80e^{-2}$	0.144	0.107	-	-	0.161	$3.88e^{-2}$	-	-
GraphRNN	-	-	-	-	$2.42e^{-3}$	-	-	-	-	-	-	-
GraphRNN-S	$7.69e^{-3}$	$2.56e^{-2}$	-	$2.55e^{-2}$	**1.52**e^{-3}	$4.13e^{-2}$	-	-	-	-	-	-
GRAN	$7.06e^{-3}$	$2.06e^{-2}$	-	$1.28e^{-2}$	$1.34e^{-2}$	$3.37e^{-2}$	-	-	0.145	$2.00e^{-2}$	-	-
BiGG	$3.05e^{-3}$	**1.09**e^{-3}	-	$8.58e^{-3}$	$1.58e^{-2}$	$3.01e^{-2}$	-	-	0.105	$2.83e^{-2}$	-	-
ARVGA	$7.92e^{-2}$	0.114	0.17	$2.40e^{-2}$	0.122	$5.12e^{-2}$	-	-	0.209	$5.24e^{-2}$	-	-
NetGAN	**1.28**e^{-3}	$1.61e^{-3}$	-	$4.71e^{-3}$	$9.04e^{-3}$	$1.02e^{-2}$	-	-	$6.89e^{-2}$	$6.43e^{-3}$	-	-
CondGEN	$4.69e^{-2}$	$1.40e^{-2}$	-	$2.94e^{-2}$	$8.61e^{-2}$	$1.22e^{-2}$	-	-	0.175	-	-	-
SGAE	$6.37e^{-3}$	$2.91e^{-3}$	**2.06**e^{-3}	$5.29e^{-3}$	$2.92e^{-3}$	**1.84**e^{-3}	**1.30**e^{-4}	**8.69**e^{-4}	**6.12**e^{-2}	$2.90e^{-3}$	**1.97**e^{-4}	**9.38**e^{-4}

TABLE 2.6: Evaluation results of *Clustering Coefficient* MMD on all 12 datasets.

Graph Generator	Cora	Citeseer	Pubmed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	EU Email	Google
E-R	$7.35e^{-2}$	$3.90e^{-2}$	$1.46e^{-2}$	0.101	$1.72e^{-2}$	$4.01e^{-2}$	0.117	0.146	$5.97e^{-2}$	$3.29e^{-2}$	$2.19e^{-3}$	0.132
W-S	$7.64e^{-2}$	$4.03e^{-2}$	$1.46e^{-2}$	0.104	$1.98e^{-2}$	$4.22e^{-2}$	0.12	0.151	$6.05e^{-2}$	$3.31e^{-2}$	$2.19e^{-3}$	0.131
B-A	$6.22e^{-2}$	$3.36e^{-2}$	$1.37e^{-2}$	$8.18e^{-2}$	$1.38e^{-2}$	$1.96e^{-2}$	0.106	0.124	$5.80e^{-2}$	$3.31e^{-2}$	$2.19e^{-3}$	0.13
RTG	$3.16e^{-2}$	$4.24e^{-2}$	$2.93e^{-2}$	$2.83e^{-2}$	$2.23e^{-2}$	$2.11e^{-2}$	$8.02e^{-2}$	0.105	0.118	$9.64e^{-3}$	$7.20e^{-4}$	-
BTER	**1.37**e^{-3}	$3.15e^{-3}$	$3.89e^{-3}$	$3.43e^{-3}$	$2.51e^{-3}$	$1.63e^{-2}$	$2.84e^{-3}$	$1.99e^{-2}$	**1.18**e^{-2}	$8.40e^{-3}$	$4.24e^{-3}$	$1.99e^{-2}$
SBM	$4.90e^{-2}$	$2.14e^{-2}$	$1.05e^{-2}$	$7.48e^{-2}$	$6.11e^{-3}$	$1.84e^{-2}$	$9.95e^{-2}$	0.13	$4.57e^{-2}$	$2.79e^{-2}$	$1.99e^{-3}$	0.12
DCSBM	$2.82e^{-2}$	$1.10e^{-2}$	$3.65e^{-3}$	$4.02e^{-2}$	$4.26e^{-3}$	$7.79e^{-3}$	$9.34e^{-2}$	$3.98e^{-2}$	$4.66e^{-2}$	$9.43e^{-3}$	$2.73e^{-4}$	$8.92e^{-2}$
MMSB	$6.44e^{-2}$	$3.29e^{-2}$	-	$7.99e^{-2}$	$1.76e^{-2}$	$1.96e^{-2}$	-	-	$5.96e^{-2}$	$1.50e^{-2}$	-	-
R-MAT	$4.12e^{-2}$	$1.78e^{-2}$	$2.02e^{-3}$	$4.25e^{-2}$	$4.33e^{-3}$	$8.53e^{-3}$	$3.19e^{-2}$	$1.89e^{-2}$	$4.58e^{-2}$	$1.58e^{-2}$	$1.52e^{-3}$	0.106
Kronecker	$6.94e^{-2}$	$3.80e^{-2}$	$1.08e^{-2}$	$8.36e^{-2}$	$1.92e^{-2}$	$1.51e^{-2}$	0.113	$9.46e^{-2}$	$6.04e^{-2}$	$2.02e^{-2}$	$2.03e^{-3}$	0.13
VGAE	$3.91e^{-2}$	$2.00e^{-2}$	$1.06e^{-2}$	$5.77e^{-2}$	$4.12e^{-2}$	$1.43e^{-2}$	-	-	$5.43e^{-2}$	$7.07e^{-3}$	-	-
Graphite	$4.14e^{-2}$	$1.92e^{-2}$	$1.01e^{-2}$	$5.83e^{-2}$	$3.96e^{-2}$	$2.05e^{-2}$	-	-	$4.83e^{-2}$	**5.21**e^{-3}	-	-
SBMGNN	$4.36e^{-2}$	$2.05e^{-2}$	$1.00e^{-2}$	$6.03e^{-2}$	$2.90e^{-2}$	$2.09e^{-2}$	-	-	$4.70e^{-2}$	$5.62e^{-3}$	-	-
GraphRNN	-	-	-	-	**1.33**e^{-3}	-	-	-	-	-	-	-
GraphRNN-S	$2.52e^{-2}$	$1.33e^{-2}$	-	$7.52e^{-2}$	$3.20e^{-3}$	$3.12e^{-2}$	-	-	-	-	-	-
GRAN	$6.66e^{-2}$	$1.49e^{-2}$	-	$6.19e^{-2}$	$7.57e^{-3}$	$2.50e^{-2}$	-	-	0.102	$2.25e^{-2}$	-	-
BiGG	$7.15e^{-2}$	$3.79e^{-2}$	-	$9.32e^{-2}$	$1.45e^{-2}$	**3.13**e^{-3}	-	-	$4.96e^{-2}$	$3.14e^{-2}$	-	-
ARVGA	$2.36e^{-2}$	$1.34e^{-2}$	$7.19e^{-3}$	**1.84**e^{-2}	$1.22e^{-2}$	$5.90e^{-3}$	-	-	$4.03e^{-2}$	$8.47e^{-3}$	-	-
NetGAN	$1.08e^{-2}$	**2.76**e^{-3}	-	$5.00e^{-2}$	$6.62e^{-3}$	$1.87e^{-2}$	-	-	$1.71e^{-2}$	$1.88e^{-2}$	-	-
CondGEN	0.106	0.104	-	$9.83e^{-2}$	0.165	$9.92e^{-2}$	-	-	0.196	-	-	-
SGAE	$1.99e^{-2}$	$2.73e^{-2}$	**1.79**e^{-3}	$2.08e^{-2}$	$1.83e^{-2}$	$1.03e^{-2}$	**1.61**e^{-3}	**1.31**e^{-2}	$3.64e^{-2}$	$1.54e^{-2}$	**1.70**e^{-4}	**5.94**e^{-3}

graph generators. Apart from GraphRNN and GraphRNN-S, GRAN and BiGG also perform well for the protein dataset. It can be seen that, the quality of graphs GraphRNN generated is the best. Other graph generators cannot achieve the best performance in all metrics as well. Taking W-S as an example, according to Table 2.4, W-S has achieved the best performance in the first column, i.e., degree metric. However, W-S performs poorly in other evaluating metrics. Comparing all graph generators and all evaluating metrics in Table 2.4 we come to the finding that no graph generator

TABLE 2.7: Evaluation results of *Orbit* MMD on all 12 datasets.

Graph Generator	Cora	Citeseer	Pubmed	Cora-ML	Protein	PPI	Deezer	Facebook	3D Point Cloud	Autonom. System	EU Email	Google
E-R	0.204	$3.83e^{-2}$	$2.08e^{-2}$	0.999	$2.69e^{-4}$	0.564	$3.28e^{-4}$	2	$3.65e^{-5}$	2	2	0.77
W-S	0.616	0.107	$3.26e^{-2}$	1.91	$1.20e^{-2}$	1.51	$1.29e^{-2}$	2	$3.63e^{-4}$	2	2	0.968
B-A	0.177	$9.98e^{-2}$	$2.72e^{-3}$	0.105	0.349	$6.76e^{-2}$	$4.73e^{-2}$	2	0.134	2	2	0.254
RTG	2	2	2	2	2	2	2	2	2	2	2	-
BTER	$5.38e^{-2}$	$\mathbf{2.92}e^{-3}$	$1.89e^{-3}$	0.127	$2.89e^{-3}$	$2.77e^{-2}$	$\mathbf{2.45}e^{-5}$	0.481	$5.60e^{-5}$	2	2	0.138
SBM	0.146	$2.76e^{-2}$	$5.35e^{-3}$	0.565	$9.42e^{-4}$	0.207	$1.90e^{-4}$	1.99	$1.64e^{-4}$	2	2	0.435
DCSBM	$4.80e^{-2}$	$5.40e^{-3}$	$2.65e^{-4}$	0.231	$2.14e^{-3}$	$\mathbf{9.85}e^{-3}$	$5.83e^{-5}$	0.562	$1.76e^{-4}$	2	2	0.269
MMSB	$2.18e^{-2}$	$7.20e^{-3}$	-	$9.49e^{-2}$	$9.65e^{-4}$	$3.31e^{-2}$	-	-	$1.01e^{-4}$	2	-	-
R-MAT	1.83	1.75	1.91	2	1.92	1.99	2	2	1.99	2	2	2
Kronecker	$2.39e^{-2}$	$3.48e^{-3}$	$2.75e^{-2}$	0.322	$1.19e^{-2}$	1.49	$2.89e^{-3}$	2	$3.32e^{-4}$	2	1.91	0.141
VGAE	1.78	0.984	2	1.95	$6.31e^{-2}$	1.07	-	-	1.76	1.99	-	-
Graphite	1.97	0.979	1.62	1.98	$5.36e^{-2}$	1.61	-	-	$8.85e^{-2}$	2	-	-
SBMGNN	1.75	0.784	2	1.99	$5.30e^{-2}$	1.64	-	-	$2.10e^{-3}$	2	-	-
GraphRNN	-	-	-	-	$4.08e^{-5}$	-	-	-	-	-	-	-
GraphRNN-S	0.194	$6.70e^{-3}$	-	1.9	$\mathbf{1.51}e^{-5}$	0.237	-	-	-	-	-	-
GRAN	$6.57e^{-2}$	0.191	-	1.93	$2.51e^{-3}$	0.192	-	-	2	2	-	-
BiGG	0.181	$5.09e^{-2}$	-	0.802	$2.49e^{-4}$	0.421	-	-	$2.29e^{-4}$	2	-	-
ARVGA	1.02	0.968	2	2	0.559	1.47	-	-	0.929	2	-	-
NetGAN	$4.45e^{-2}$	$6.70e^{-3}$	-	$2.16e^{-2}$	$6.00e^{-4}$	$2.36e^{-2}$	-	-	$\mathbf{3.28}e^{-5}$	2	-	-
CondGEN	2	2	-	2	2	2	-	-	2	-	-	-
SGAE	$\mathbf{1.49}e^{-3}$	$3.31e^{-3}$	$\mathbf{2.57}e^{-4}$	$\mathbf{1.36}e^{-2}$	$1.65e^{-3}$	$9.69e^{-2}$	$3.02e^{-2}$	**0.37**	$3.35e^{-5}$	**0.577**	**1.78**	**0.107**

TABLE 2.8: Evaluation results on preserving community experiment.

Methods	Cora		Cora-ML		PPI		3D Point Cloud	
	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})	NMI(e^{-2})	ARI(e^{-2})
SBM	11.3±0.7	1.2±0.1	15.7±1.1	8.9±0.6	9.3±0.9	1.5±0.3	37.0±1.3	11.4±0.7
DCSBM	18.6±0.8	1.8±0.3	22.1±0.7	7.7±0.4	21.7±0.7	2.5±0.2	37.3±1.4	11.5±0.8
MMSB	15.4±0.6	0.8±0.4	23.3±0.6	13.2±1.1	22.7±0.7	6.3±1.0	7.1±0.4	1.3±0.3
VGAE	60.4±0.6	40.0±1.2	59.0±0.9	46.4±1.6	**49.1±0.5**	**16.6±1.8**	57.0±0.8	8.2±1.1
Graphite	**62.3±0.8**	**43.4±1.9**	60.8±0.7	51.0±1.2	41.1±0.5	-0.2±0.1	58.8±0.4	13.2±0.3
SBMGNN	61.9±0.4	41.0±1.6	**63.1±0.7**	**55.0±1.0**	47.8±0.7	13.3±0.1	59.2±0.9	15.9±1.1
ARVGA	41.8±0.8	9.3±1.3	33.1±0.7	14.9±1.1	32.6±0.9	0.1±0.2	47.2±0.8	4.1±0.5
NetGAN	5.2±0.5	0.2±0.1	37.2±1.4	32.7±2.6	8.5±0.8	4.4±0.8	**67.4±0.9**	**38.8±2.6**
SGAE	62.0±0.7	42.2±1.3	62.1±0.9	54.8±1.5	41.6±0.9	16.0±1.1	60.6±0.6	35.9±1.4

can beat other graph generators on all evaluating metrics in the Autonomous Systems dataset. This is not restricted to this dataset; we also find that, in six datasets (i.e., Cora, Citeseer, Cora-ML, PPI, 3D Point Cloud, Autonomous Systems), no graph generator beats others in all evaluating metrics. Due to space constraints, those experimental results which indicate similar findings are omitted in the text.

We also report the results of several representative evaluating metrics on all datasets. We provide the quality details of each generator for *Degree*, *Clustering Coefficient*, and *Orbit* in Table 2.5, Table 2.6, and Table 2.7, respectively. We can see that SGAE achieves seven out of twelve best results in *Degree* from Table 2.5, five out of twelve best results in *Clustering Coefficient* from Table 2.6, and seven best results in *Orbit* from Table 2.7. Therefore, SGAE outperforms other generators, especially when generating large datasets (e.g., Google, Deezer, EU email, Facebook).

SGAE cannot perform well when generating certain small-sized graphs (e.g., Protein, PPI, Cora, Citeseer). Neural network-based graph generators (e.g., GraphRNN, NetGAN, SBMGNN, etc) outperform SGAE in such datasets, but they cannot simulate large data distributions due to the limit of memory and FLOPS.

In terms of specific properties, Table 2.6 shows that BTER generates graphs with better clustering coefficient distribution than most generators. That is because BTER generates edges directly referring to the clustering coefficient by each node. Although SGAE outperforms BTER, the latter is much faster. The detailed speed comparison is introduced in Section 2.4.6.

In general, although there are several generators (e.g., GraphRNN) ranking higher SGAE when generating small graphs, these results still prove that SGAE fills the gap in learning the generative distribution of large networks. That is because SGAE breaks the scalability limitation of neural network-based graph generators.

2.4.4 Preserving Community Structure

In this subsection, we compare the performance of graph generators in terms of preserving community structures. We report experimental results on four representative datasets (Cora, Cora-ML, PPI, 3D Point Cloud) in Table 2.8. Some graph generators (e.g., E-R, RTG, and GRAN) are excluded because their optimization objectives cannot preserve node order and community membership.

We find that the autoencoder-based graph generators (i.e., VGAE, Graphite, SBMGNN, and SGAE) are superior in preserving community structures in Cora, Cora-ML, and PPI datasets. NetGAN performs well on 3D Point Cloud. This is because the autoencoder-based graph generators leverage the graph neural network (GNN) to infer the node representations, which will be decoded into a new graph. The GNN architecture has a reliable reasoning capability for community memberships and better generalization performance than NetGAN. In certain cases, users may prefer preserving community memberships, i.e., community structures of the observed graph. The autoencoder-based graph generator is the best choice to generate new graphs with similar community structures.

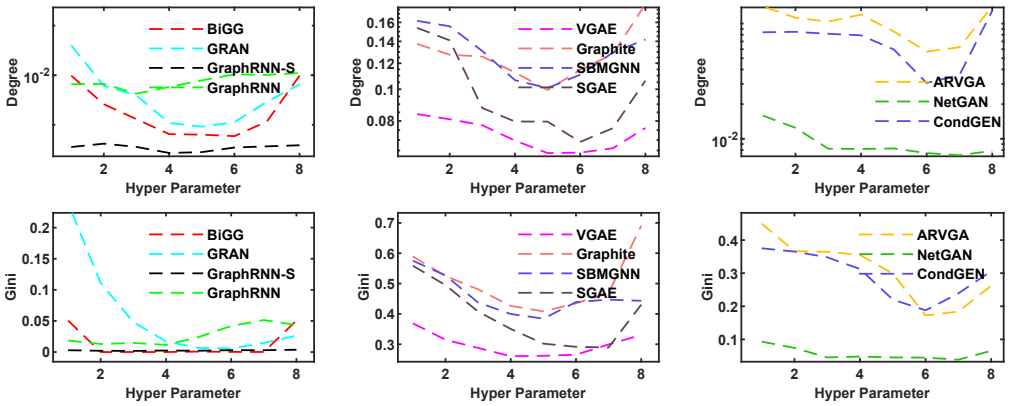


FIGURE 2.1: Parameter sensitivity experiment results. Lower is better.

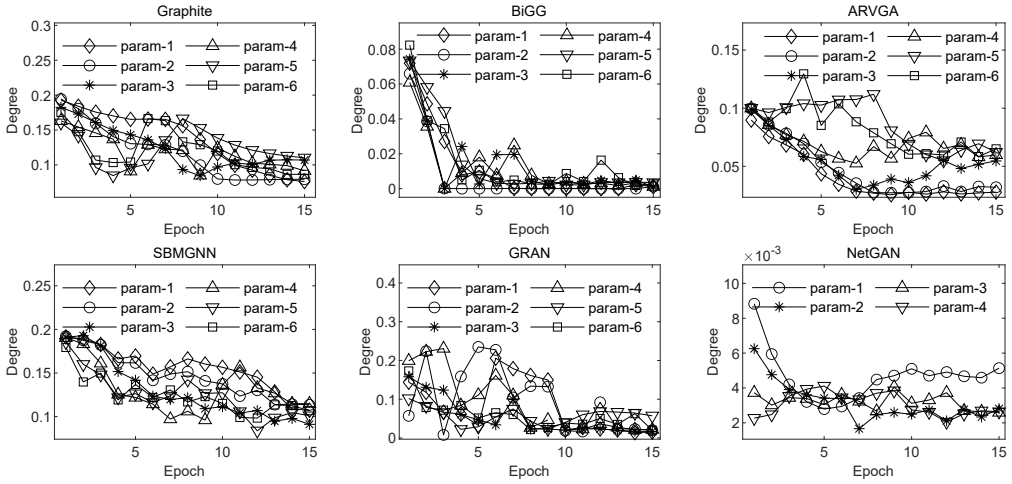


FIGURE 2.2: Model stability experiment results.

2.4.5 Parameter Sensitivity

Since different types of graph generators have different hyperparameters, we compare them in groups according to their similarity of model architecture and test their robustness and training difficulty.

2.4.5.1 Sensitivity

This section evaluates the hyper-parameter sensitivity of each neural network-based model and reports their performance by varying parameter settings. In Figure 2.1, due to space limitation, we provide detailed comparison results for two representative evaluating metrics (Degree and Gini Index) on Protein. According to the implementation of the model, neural networks-based graph

generators are divided into three categories, namely RNN-based, autoencoder-based, and GAN-based. Figure 2.1 shows experimental results of graph generators in these categories.

In the left part of Figure 2.1, we find that, for RNN-based graph generators, GraphRNN-S outperforms others in terms of sensitivity and performance. The full GraphRNN also has a flat curve, showing robustness as good as GraphRNN-S. GRAN and BiGG occasionally generate bad results which show their high sensitivity on parameter settings. The middle section of Figure 2.1, shows that VGAE generates graphs with the best robustness and quality. This is because VGAE has the simplest model architecture, resulting in better generalization. On the right side of Figure 2.1, NetGAN is shown clearly that it is not sensitive about parameter settings, and the quality of its generated graph is the best. The parameter changes of CondGEN and ARVGA significantly affect their generative performance compared with NetGAN. From this section, we can see that GraphRNN, VGAE, and NetGAN from three groups of graph generators are not sensitive about their parameter settings compared with the others.

2.4.5.2 Stability

We report the training stability of graph generators by varying parameter settings in Figure 2.2. We show the experimental results on Citeseer as a representative example.

We find that the curves of the model training process of autoencoder-based graph generators (e.g., Graphite, SBMGNN) fluctuate, but the overall trend is convergent. This is because updating parameters may have an impact on the degrees of all nodes. The RNN-based graph generators (e.g., GRAN, BiGG) are relatively stable, and the occasional collapse does not affect its tendency to continue converging to the optimal. The graph generation occasionally can not converge in GAN-based graph generators (e.g., ARVGA), while only the NetGAN with WGAN architecture has a more stable training process.

2.4.6 Scalability and Efficiency

We evaluate the scalability and efficiency of each generator in this subsection. Table 2.9 reports the time consumption of inferring a new graph. Note that because generating graphs by NetGAN requires one more step, i.e., assembling generated random walks into an adjacency matrix, generating one graph by NetGAN is more time-consuming. Table 2.10 reports the time consumption of

TABLE 2.9: Time consumption (seconds) per graph generation.

#Nodes	0.1k	1k	10k	100k	1000k
E-R	**4.6**e^{-4}	9.0e^{-3}	0.46	10.1	217
B-A	1.0e^{-3}	1.2e^{-2}	0.11	1.17	59.1
W-S	7.2e^{-4}	**7.1**e^{-3}	**0.08**	**0.81**	**8.63**
RTG	8.3e^{-3}	0.13	2.62	4.13	663
BTER	1.88e^{-3}	0.03	0.32	4.91	82.8
SBM	6.1e^{-3}	0.09	2.58	37.1	545
DCSBM	6.2e^{-3}	0.09	2.69	39.3	570
MMSB	6.1e^{-3}	0.09	2.56	-	-
R-MAT	8.5e^{-3}	0.09	0.98	9.71	99.1
Kronecker	8.5e^{-3}	0.08	1.00	9.69	99.2
GraphRNN	0.31	5.62	-	-	-
GraphRNN-S	0.27	4.74	63.6	-	-
GRAN	0.36	4.02	-	-	-
BiGG	0.33	2.03	60.4	-	-
VGAE	4.2e^{-3}	0.04	0.38	-	-
Graphite	6.1e^{-3}	0.06	0.64	-	-
SBMGNN	0.01	0.11	1.18	-	-
ARVGA	4.8e^{-3}	0.04	0.42	-	-
NetGAN	8.7e^{-3}	0.09	1.12	-	-
CondGEN	8.3e^{-3}	0.15	-	-	-
SGAE	4.5e^{-3}	0.04	0.48	43.8	4160

TABLE 2.10: Time consumption (minutes) of parameter updating during the training process.

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	**5.4**e^{-2}	0.44	18.0	-	-
Kronecker	9.6e^{-2}	0.35	**0.96**	**2.28**	**5.61**
GraphRNN	0.59	10.4	-	-	-
GraphRNN-S	0.56	6.45	61.8	-	-
GRAN	0.13	6.84	-	-	-
BiGG	0.11	2.81	33.7	-	-
VGAE	0.03	**0.10**	1.74	-	-
Graphite	0.04	0.11	2.11	-	-
SBMGNN	0.09	0.31	5.22	-	-
ARVGA	0.04	0.12	1.79	-	-
NetGAN	0.12	0.46	7.61	-	-
CondGEN	0.05	0.19	-	-	-
SGAE	0.04	0.11	1.76	8.63	79.8

parameter updating during the training process. Table 2.11 chronicles the time consumption of the entire training process. Table 2.12 details the peak memory usage during training process.

Simple model-based graph generators have the highest efficiency for generating large networks, incurring minor extra space cost and taking little time. Combining the experimental results on

TABLE 2.11: Time consumption (minutes) of the entire training process.

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	0.11	0.91	40.3	-	-
Kronecker	1.39	1.55	3.25	**4.73**	**21.3**
GraphRNN	2.45	28.9	-	-	-
GraphRNN-S	1.63	15.4	161	-	-
GRAN	1.36	14.3	-	-	-
BiGG	0.88	9.67	139	-	-
VGAE	**0.06**	0.42	9.75	-	-
Graphite	0.07	0.47	10.6	-	-
SBMGNN	0.08	0.63	12.4	-	-
ARVGA	0.07	0.50	10.3	-	-
NetGAN	0.27	2.80	31.1	-	-
CondGEN	0.18	25.3	-	-	-
SGAE	0.17	**0.34**	**3.15**	17.34	163.1

TABLE 2.12: Peak GPU memory usage (MiB) during training

#Nodes	0.1k	1k	10k	100k	1000k
MMSB	**1575**	**1709**	18529	OOM	OOM
GraphRNN	1915	3121	OOM	OOM	OOM
GraphRNN-S	1913	1959	5501	OOM	OOM
GRAN	1959	2677	OOM	OOM	OOM
BiGG	2043	3145	18985	OOM	OOM
VGAE	1719	1759	4799	OOM	OOM
Graphite	1719	1761	4819	OOM	OOM
SBMGNN	1719	1767	5243	OOM	OOM
ARVGA	1719	1762	4832	OOM	OOM
NetGAN	2237	2552	5008	OOM	OOM
CondGEN	1722	1789	-	-	-
SGAE	1721	1740	**1943**	**3971**	**24248**

Protein and Pubmed in Section 2.4.3, we can see that the scalability of simple model-based graph generators (e.g., BTER, R-MAT) is better than others'. This is because they are designed to generate a set of random graphs having specific properties, which is insensitive to the size of the graph. The efficiency of GraphRNN and other RNN-based graph generators is the worst since when generating large graphs, RNN needs to stack too many layers. Therefore, spending a lot of space in storing long-term memory exceeds RNN-based graph generators' ability to articulate the network. SGAE achieves the greatest efficiency in the training process and memory usage, proving the value of its decoder's improvements on time and space complexity.

TABLE 2.13: Overall performance evaluation of 20 representative general graph generators. The more amount of (★), the better the performance. Note that the symbol ★ is used to refine the ranks of the graph generators.

Graph Generator	Output Quality	Training Time	Inference Time	Scalability	Space	Robustness	Community Preserving	Tuning Difficulty
E-R	★	-	★★★★½	★★★★½	★★★★★	-	-	★★★★★
W-S	★	-	★★★★★	★★★★½	★★★★★	-	-	★★★★
B-A	★	-	★★★★½	★★★★★	★★★★★	-	-	★★★★★
RTG	★	-	★★★½	★★★★½	★★★★★	-	-	★★★★
BTER	★★½	-	★★★★½	★★★★★	★★★★½	-	-	★★★★
SBM	★★	-	★★★½	★★★★½	★★★★★	-	★★	★★★★
DCSBM	★★½	-	★★★½	★★★★★	★★★★½	-	★★½	★★★★
R-MAT	★★	-	★★★★½	★★★★★	★★★★★	-	-	★★★★
Kronecker	★★	★★★★★	★★★★½	★★★★★	★★★★★	★★★	-	★★★★
MMSB	★★	★★★½	★★★½	★★★	★★	★★★	★★	★★★★
VGAE	★★★½	★★★★	★★★★	★★★½	★★★	★★★★	★★★½	★★★★
Graphite	★★★½	★★★★	★★★½	★★★½	★★★	★★★	★★★½	★★★½
SBMGNN	★★★½	★★★½	★★★	★★★½	★★★	★★★	★★★★	★★★½
GraphRNN	★★★★★	★	★	★	★	★★★★	-	★★★★
GRAN	★★★★½	★★	★★	★★★	★★	★★★	-	★★★
BiGG	★★★★½	★★½	★★★	★★★½	★★★★	★★★	-	★★★
ARVGA	★★★	★★★★	★★★	★★★½	★★★	★★★	★★★	★★
NetGAN	★★★½	★★★	★★½	★★★½	★★★	★★★★	★★★	★★★
CondGEN	★★★	★★	★★★½	★★	★★★★	★★	-	★★
SGAE	★★★★	★★★★½	★★★★	★★★★	★★★★	★★★★	★★★½	★★★★

2.5 Recommendation

As shown in our comprehensive performance evaluation, there is no algorithm which can win under all metrics. This is because the sophisticated models are required to achieve a good simulation quality and this inevitably sacrifices the efficiency and scalability compared to the simple models. Moreover, as there are many different metrics to measure the simulation quality, it is difficult for a graph generator to win under all these metrics because: (1) we cannot directly optimize the distribution of the generated graphs according to the observed graphs; and (2) some simple algorithms only focus on optimizing a particular metric (e.g., degree distribution). Thus, it is critical to have a comprehensive recommendation for users with different requirements, and the algorithms which can achieve good trade-off among the metrics are welcomed in practice.

Table 2.13 quantitatively evaluates the performance of the 20 representative general graph generators from various perspectives including graph simulation quality, training time, inference time (i.e., the time used for generating a new graph), scalability, space (i.e., memory consumption), robustness, community preserving, and tuning difficulty for users. According to the comprehensive experimental evaluation in the subsections above, for each metric, we use the number of ★

to indicate the quality of every graph generator's performance, where ★★★★★ corresponds to the best performance (i.e., fastest, most scalable, uses least memory, easiest to tune parameters). Note that we use the number of parameters, stableness and sensitivity of the models to evaluate the user-friendliness of the parameter tuning. Therefore, Table 2.13 provides a roadmap of recommendations for researchers and practitioners in how to select general graph generators in different settings.

Below are some recommendations for users according to our comprehensive evaluations.

- The best graph generator tool for simulating a citation network is our proposed SGAE. According to the last row of Table 2.7, SGAE achieves three best performances for the four citation-network datasets (Cora, Citeseer, Pubmed, Cora-ML). Furthermore, according to Tables 2.5 and 2.6, SGAE also achieves the best performance for the Pubmed dataset. The reason is that the performance of deep graph generative models (except SGAE) will significantly degrade when generating graphs with more than 1k nodes.

- In terms of graph simulation quality, GraphRNN can achieve the best overall performance and can mimic the structural distribution of observed graphs adequately. It is a good choice if there are sufficient resources, and the relevant costs are acceptable for users. For instance, GraphRNN can be used for applications (e.g., protein graphs in bioinformatics) where the sizes of observed graphs and simulated graphs are small (e.g., less than 1000 nodes in our experiment environment). Note that GraphRNN also demonstrates a good performance in model robustness and parameter-tuning difficulty.

- If the efficiency and the scalability are of high priority, DCSBM and BTER are recommended because both are very efficient and scalable and have a good overall performance on graph simulation quality among approaches where the deep neural networks are not applied. Note that although BTER outperforms DCSBM in terms of inference time, DCSBM can better preserve the community structures of observed graphs.

- If users look for a good balance between graph simulation quality and efficiency (scalability), SGAE, as proposed in this chapter, is recommended. Compared to two RNN-based generators GRAN and BiGG, SGAE display competitive performance in graph simulation

quality, but outshines the others with better efficiency and scalability. Moreover, in our experiment settings, SGAE outperforms other autoencoder-based generators and GAN-based generators in both graph simulation quality and efficiency (scalability). As reported in Table 2.13, SGAE also achieves well in terms of model robustness, memory consumption, community preserving, and parameter tuning, compared to other deep neural networks-based approaches.

- When users are only interested in several specific properties of the observed graphs, other graph generators can be recommended. For instance, if users are only interested in the scale-free property of the graphs, the B-A model can comfortably fit this role. Similarly, the W-S model is recommended to simulate small-world graphs. When it is important to preserve the community structures of observed graphs, SBMGNN is a good choice.

2.6 Conclusion

Graph data simulation is fundamental in a wide range of applications such as social networks, e-commerce, and bioinformatics. In this chapter, we fill important gaps in this line of research by: (i) giving an overview of 20 representative general graph generators, including recently emerged deep learning-based approaches; (ii) conducting comprehensive experiments on 20 representative general graph generators and providing broad-spectrum recommendations for both researchers and practitioners; (iii) developing a new algorithm to achieve a good trade-off between graph simulation quality and efficiency; and (iv) implementing a user-friendly platform for researchers and practitioners such that they not only can easily apply a variety of existing general graph generators to their work but also immediately integrate their own general graph generators for comprehensive performance comparison and analytics.

Chapter 3

Community-Preserving Deep Graph Generation

3.1 Chapter Overview

This chapter introduces the Community-Preserving Generative Adversarial Network (CPGAN), a novel approach for simulating real-life graphs that balances efficiency and simulation quality while preserving essential structural properties, particularly community structures. This chapter has been published in [79]. Section 3.2 gives background and related articles of community-preserving graph generation. Section 3.3 introduces our proposed CPGAN. Section 3.4 reports the performance comparison between baselines and our CPGAN. Section 3.5 concludes the experimental results and discusses the contribution of this chapter.

3.2 BackGround and Related Works

Graphs have been used to model relationships in a wide spectrum of applications such as social science, biology, and information technology [80, 81]. In some scenarios, the real-life graphs are not available due to a variety of reasons such as incomplete observability, privacy concern, and company/government policy. Thus, many graph techniques have been developed to simulate real-life graphs in various tasks such as modeling physical and social interactions and constructing

knowledge graphs. For example, in financial fraud detection, generated graphs can be adopted to produce synthetic financial networks without divulging private information [82]. Moreover, graph generation techniques can also help us to better understand the distribution of graph structures and other features for the essential tasks. For instance, graph generators can be used to generate molecule [9] and formulas [16], which help to understand insights of the graph data.

3.2.1 Motivation.

Due to its importance in both academia and industry, there is a long history of study on graph generators in many domains such as database, data mining, and machine learning (Please refer to [1] for a recent survey). Among these studies, the arguably most important line is the *general graph generator* which aims to learn a generative model to capture the structural distributions of the observed graphs regardless of the domains. Unless otherwise specified, the graph generators referred to in this chapter are general graph generators. A large body of techniques have been developed in the literature which can be roughly divided into two categories: traditional graph generative models (e.g., [30, 42–45, 49]) and learning-based graph generative models (e.g., [21, 24, 39]). Generally speaking, the traditional approaches can efficiently generate large scale graphs based on some rules (e.g., [30, 43, 44]). However, the simulation quality of these models is not satisfactory for real-life graphs because they are hand-engineered to model some particular families of graphs and lack the capacity to learn a generative model directly from observed real-life graphs. With the advance of deep learning techniques, there is a clear trend in the literature where a variety of deep learning techniques such as recurrent neural network (RNN [21, 83]) and generative adversarial network (GAN [24, 25]) have been adopted to simulate real-life graphs. Though they have significantly improved the graph simulation quality by taking advantage of the sophisticated models, there are two major limitations.

(1) Community-preserving. Due to the complex nature of the graph, we cannot directly evaluate the goodness of the graph simulation by calculating similarity score between two graph distributions [1]. Thus, we have to resort to a variety of metrics each of which aims to quantitatively capture the likelihood of two graphs (graph distributions) from one perspective (e.g., degree distribution).

[1] Note that there are some similarity measures for two graphs such as graph edit distance [26] and maximum common subgraph [84]. But they cannot be applied to determine if two graphs are from the same distribution.

We notice that the community structure, which is one of the most unique and prominent features of the graph, is neglected by most of the graph generators. Stochastic block models (SBM [48]), as well as its variants DCSBM [50], MMSB [49] and SBMGNN [39], BTER [51] and Chung-Lu model [85] consider the community structure. But there are only a few parameters in their models, which cannot properly capture the community structure of real-life graphs due to the simplicity of their generative models. It is well-known that the community structure preserves the inherent high-order structural property of the graph and hence plays an important role in many downstream data analysis tasks such as link prediction and node classification. For instance, communities in a real-life network might represent real social groupings [86] and communities in a guarantee-loan network [87, 88] might represent dense loan relationships and financial institution groups (See Figure 3.1 as an example). This community information could help us to understand and exploit these networks more effectively [89]. Thus, in addition to the existing graph simulation quality evaluation metrics, we should also consider if a graph generator can well preserve the community structure of the observed real-life graph.

(2) Efficiency. Due to the complexity of the deep learning models, it is not a surprise that the emerging advanced deep learning-based general graph generative models are very time-consuming compared to the traditional models, and they can only handle small or medium-sized graphs in practice. For instance, GraphRNN [21] and GRAN [22] take RNN to generate the whole adjacency matrix, and NetGAN [24] uses random walks to assemble the whole graph, whose time complexity of training and inference procedures is $O(b \times n^2)$, where n is the number of graph nodes and b is the number of epochs. Considering that the efficiency and simulation quality are two important but contradictory requirements of graph generators in practice, it is desirable to develop a new deep learning model which can achieve a good trade-off between the efficiency (scalability) and the simulation quality.

3.2.2 Contribution.

One may wonder if we can simply modify the existing deep learning-based generative graph generator models to preserve the community structure of the observed graphs. As to our best knowledge, this is non-trivial because it is challenging to integrate the community-preserving property in the learning and optimization of the graph generative models without sacrificing simulation quality.

Thus, in this chapter, we aim to design a new graph generative model which can better preserve the community structures of the observed graphs and have competitive performance on other evaluation metrics.

Recent advances in generative adversarial networks (GAN) have shown great successes in the graph simulation task (e.g., [24, 25, 67]). In this chapter, we follow this line of research and propose a novel Community-Preserving Generative Adversarial Neural network (CPGAN) for the efficient and community-preserving graph generation. In particular, we propose a ladder network of the graph convolution and pooling layers as the permutation-invariant graph encoder. Before translating latent variables to a new graph, we leverage variational inference on the latent conjugate distributions to generate graphs naturally. The decoder (generator) is a full-connected network with a dot product to make link predictions once at all. Finally, the graph convolution and pooling layers are leveraged in an encoder (discriminator) to judge whether the community structure is well learned from the observed graphs.

In a nutshell, our principal contributions in this chapter are summarized as follows:

- We propose a new graph generation model, Community-Preserving Generative Adversarial Network (CPGAN). It can not only preserve the community structure as well as other important properties of the real-life graphs but also reduce the graph simulation time and improve the scalability compared to other learning-based graph generation models.
- We carefully design the generator and discriminator in a unified GAN framework, in which the generator is in a hierarchical graph variational autoencoder that could learn permutation-invariant representations of input graphs and could generate new graphs from node representations (embeddings), and the discriminator to judge whether the embeddings are from real or simulated graphs.
- We introduce a differentiable ladder-shaped network to enable graph pooling and message transmitting in different community structure levels, which is more efficient and effective than simply stacking deeper graph convolution layers.
- We conduct extensive experiments on both synthetic and real-world graphs. Results show that our proposed model can achieve a good trade-off between graph simulation quality and efficiency (scalability) compared to the baseline methods.

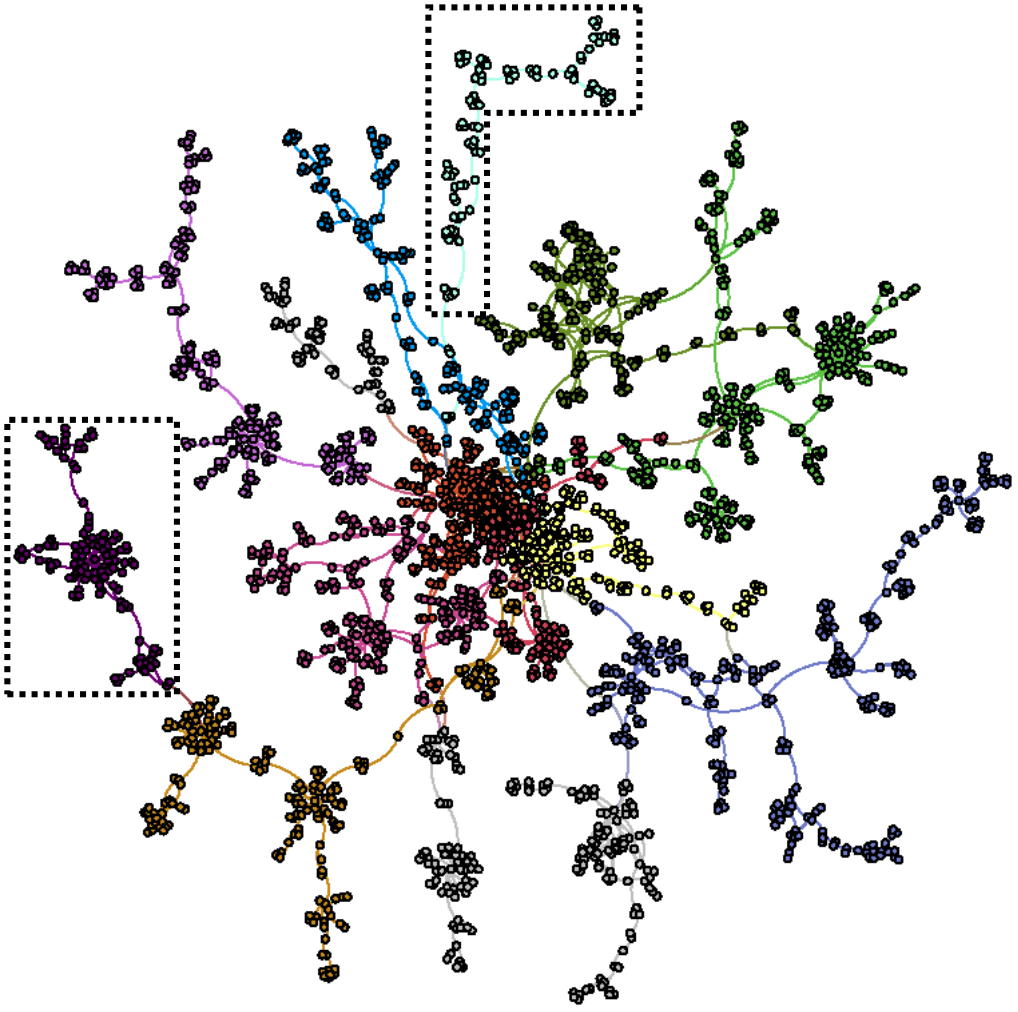


FIGURE 3.1: An illustration of the communities of a real-life network.

3.2.3 Problem Definition

We define a graph $G = (V, E)$ where V denotes a set of n nodes (vertices), and a set of m edges $E \subseteq V \times V$, where a tuple $e = (u, v) \in E$ represents an edge between two vertices u and v in V. The graph G can also be represented by an adjacency matrix $\mathbf{A} \in \{1, 0\}^{n \times n}$. Same as the literature, we assume G is an undirected graph, and hence the adjacency matrix of the graph is symmetric. Additionally, we denote the (optional) node-feature matrix associated with the graph as $\mathbf{X} \in \mathbb{R}^{n \times d}$ where n denotes the number of nodes and d denotes the dimension of the node feature. We summarize the notations in Table 3.1.

Problem Statement. Given an observed graph G, a graph generative model aims to capture the

TABLE 3.1: The summary of notations

Notation	Definition
$\mathcal{G}$	the graph
A	adjacency matrix
X	node features
Z_{rec}	the node features reconstructed from input graph
$\mathcal{N}(\mu, \text{diag}(\sigma^2))$	the normal distributions
n	total number of vertices
m	total number of edges
$\mathcal{E}, \mathcal{D}$	the encoder and decoder
G, D	the generator and discriminator
$z^{(k)}$	the node features of the k-th level's community structure

structural distribution of the graph, such that a set of new graphs $\{G'\}$ with similar structural distribution can be generated.

Ideally, a general graph generative model should be able to generate new graphs which have exactly the same distribution as the observed graph. However, it is notoriously difficult to tell if two graphs are from the same distribution due to the complex nature of graph structure [21]. In practice, we have to resort to representative evaluating metrics in our experiments, each of which aims to quantitatively capture the likelihood of two graphs from one perspective (e.g., degree distribution). Please refer to section 3.4.1 for more details. Below we introduce the evaluation metrics for the community-preserving which will be carefully considered by your graph generation model.

Evaluation of community-preserving.

In this chapter we aim to preserve the community structures of the training graphs. That is, we regard the community structures of the training graphs as the ground truth, and hopefully, the generated graph has the same community structure, where the goodness of the community-preserving is evaluated by two popular metrics: Adjusted Rand Index (namely **ARI**) and Normalized Mutual Information (namely **NMI**). Below are their detailed definitions.

Given a graph with n nodes and original community partition $Y_c = \{y_1, ..., y_c\}$, a community-preserving graph generative model generate a new graph with another community partition $X_r =$

$\{x_1, ..., x_r\}$. Assume there is a bijective mapping between nodes of two graphs, we can formulate the similarity of two community partitions using *Rand Index* (namely RI) as follows:

$$\text{RI} = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.1}$$

where TP is the number of true positives, i.e., the number of pairs of nodes that are in the same subsets in both X_r and Y_c, TN is the number of true negatives, i.e., the number of pairs of nodes that are in different subsets in X_r as well as Y_c. With the same rationale, FP and FN represent the number of false positives and the number of false negatives, respectively.

The Rand Index, however, suffers from one problem. For random data, it will be higher for low community counts than for high ones, because two nodes are more likely to be assigned together by chance. The ARI is a version of the RI corrected for the chance. According to Figure 3.2, a contingency table denotes the common nodes of two community partitions, with n_{ij} denoting the number of common nodes in the community x_i and y_j, $a_i = \sum_{j=1}^{c} n_{ij}$ and $b_j = \sum_{i=1}^{r} n_{ij}$. The ARI can be formulated in terms of the contingency table as follows:

$$\text{ARI} = \frac{\sum_{ij} \binom{n_{ij}}{2} - [\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}}{\frac{1}{2}[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}] - [\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}]/\binom{n}{2}} \tag{3.2}$$

By calculating the ARI, we can quantitatively evaluate the similarity between the community structure of the generated graph and the observed graph. We can also use Mutual Information (MI) as evaluating metrics for community-preserving. MI is formulated as follows:

$$\text{MI} = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{n_{ij}}{N} \log \frac{N n_{ij}}{a_i b_j} \tag{3.3}$$

where N denotes the number of nodes. In practice, we use NMI, i.e., the normalized version of MI as our evaluating metric.

	y_1	y_2	$\cdots$	y_c	Sum
x_1	n_{11}	n_{12}	$\cdots$	n_{1c}	a_1
x_2	n_{21}	n_{22}	$\cdots$	n_{2c}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
x_r	n_{r1}	n_{r2}	$\cdots$	n_{rc}	a_r
Sum	b_1	b_2	$\cdots$	b_c	

FIGURE 3.2: The contingency table of two community partitions X_r and Y_c.

3.2.4 Related Work

In this subsection, we summarize the related works in two main areas: traditional graph generation methods (Section 3.2.4.1) and deep graph generative models (Section 3.2.4.2). As our model follows the line of the GAN-based graph generative model, some details of the algorithms in this category are also presented in Section 3.2.4.3.

3.2.4.1 Traditional Graph Generation Methods

The graph generative models have been discussed in the previous chapters. In this chapter, we discuss the traditional graph generators with community-preserving properties. Note that SBM [48] and its variants DCSBM [50] and MMSB [49] also consider the community structure, but they are limited by the simplicity of the stochastic model, resulting in the poor performance in terms of community structure-preserving for real-life graphs. Specifically, only one parameter is used to capture each community (i.e., edges within this community), and one parameter is used to represent the connectivity probability for each pair of communities (i.e., edges between these two

communities). A simple example about using SBM to generate new graphs with three communities is:

$$B = \begin{pmatrix} p_1 & 0 & 0 \\ 0 & p_2 & 0 \\ 0 & 0 & p_3 \end{pmatrix} \tag{3.4}$$

with this block matrix B, all edges will be generated within the communities and there is no edge across the communities. And the i-th community is equivalent to an E-R graph with edge probability p_i.

3.2.4.2 Deep Graph Generative Methods

In recent years, some techniques based on deep neural networks (e.g., VGAE [17], DeepGMG [90], GraphRNN [21], Graphite [38], GRAN [22], CondGen [25]) are proposed to tackle the problem of graph generation. They significantly improve the quality of graph generation compared to the traditional approaches. For instance, Graphite and VGAE use variational autoencoders (VAE) technique [17, 38] in which graph neural networks are applied for inference (encoding) and generation (decoding). As Graphite and VGAE assume a fixed set of vertices, they can only learn from a single graph. NetGAN performs more efficiently than VGAE by learning the graph's random walks, but it is not scalable because of the generation of the fixed size of graphs. In DeepGMG, graph neural networks are used to express probabilistic dependencies among nodes and edges of the graph, which can properly learn distributions over any arbitrary graph. However, it takes $O(mn^2D(G))$ operations to generate a graph with m edges, n vertices, and graph diameter $D(G)$, which also suffers from the scalability issue.

GraphRNN [21] generates a graph sequentially through recurrent neural networks (RNNs). But it is not permutation-invariant since computing the likelihood requires marginalizing out the possible permutations of the node orderings for the adjacency matrix. GRAN [22] improves the scalability of GraphRNN by generating one block of nodes and associated edges at each step in auto-regressive methods, which is still not permutation-invariant. CondGen [25] overcomes this permutation-invariance challenge by leveraging a GCN as the encoder and handling the graph generation problem in embedding spaces. Graph U-Nets [91] chooses specific nodes to realize

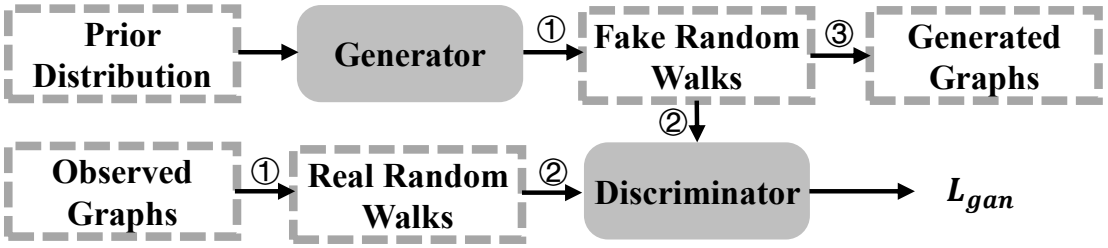


FIGURE 3.3: A brief summary of NetGAN's architecture. NetGAN generates new graphs via three steps: (1) sampling random walks; (2) model training based on GAN framework; and (3) assembling the adjacency matrix.

upsampling and downsampling graphs to obtain graph representations. However, they do not consider the community structures of the observed graphs in the learning procedure. SBMGNN [39] is a variant of SBM equipped with deep learning techniques, but its graph neural networks are used to infer the parameters of the overlapping stochastic blockmodel, which is not directly relevant to the community-preserving property. Thus, there is no performance improvement in terms of community-preserving compared to other deep learning-based graph generative models.

3.2.4.3 GAN-based Graph Generator

Generative adversarial networks (GANs) [92] have shown remarkable results in various tasks such as image generation [93], image translation [94], super-resolution imaging [95], and multimedia synthesis [96]. GANs have also been utilized in network science tasks recently, such as network embedding [97], semi-supervised learning [98], and graph generation [21, 25]. For the graph generation task, prior structure knowledge specified by the sample dataset is crucial for graph generation, especially when preserving community structure. For the graph's community structure, some models using pooling strategy [91, 99] can be trained to represent communities (clusters) each time, but it is still challenging to represent and generate these community structures together. For instance, NetGAN generates graphs via random walks, which is nontrivial to preserve community structure. As to the time complexity of graph generation, generating a graph from NetGAN requires three steps according to Figure 3.3. The first and second steps require $O(kw)$ time complexity, where k denotes the number of walks, and w denotes the length of each walk. The third step requires $O(n^2)$ time. In practice, to well simulate the real-life graphs, steps 1 and 2 may require far more time than $O(n^2)$ due to the irreversible bias of random walks [100].

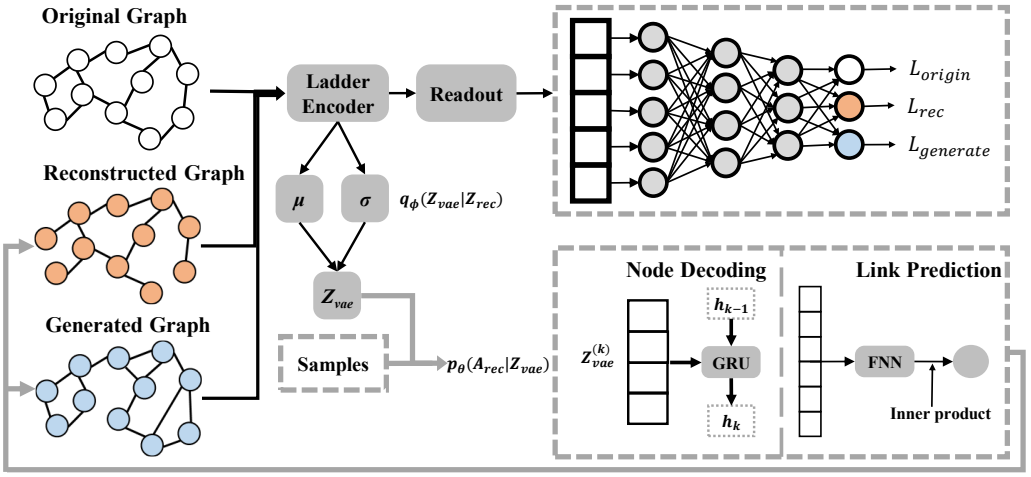


FIGURE 3.4: The Framework of CPGAN

3.3 Community-Preserving Graph Generative Model

The key idea of CPGAN is that we can extract and reconstruct the community structure of a set of graphs by providing a community-preserving model regardless of the input permutation of each node. In this section, we outline the main challenges, show the framework and implementation details of our models, and introduce how to design the training process to reconstruct and generate new graphs.

3.3.1 Challenges

In this work, we hope to (1) learn the community structure of graphs using adjacency matrices A in the training set and reconstruct new graphs with a similar structure, and (2) generate new graphs using trained decoder $\mathcal{D}$ and samples from prior distributions $\mathcal{N}$. Below, we show four intrinsic challenges in community structure preserved graph generation, together with our unique contributions and the motivations of our solutions.

3.3.1.1 Community Structure Preserved Graph Generation

We notice that the community structure, which is one of the most unique and prominent features of the graph, is neglected by most of the graph generators. Inspired by hierarchical clustering, nodes with similar features can be clustered to the same class in a bottom-up fashion. On the

other hand, in the representation learning of nodes, the clustering results can be used as important representation information about the community structure of nodes. Therefore, in practice, we introduce a hierarchical encoder to extract different levels of community structure as the input of the discriminator for the encoder of CPGAN, we need to use clustering results to enhance the graph discrimination level. For the decoding process of the generator, each node needs to exploit its community structure to enhance the generative performance. An intuitive solution is merging the node representations from different levels of community structure as the input of the decoder.

3.3.1.2 Permutation-Invariance

In our task, every graph with n nodes has $n!$ different permutations, which will result in an enormous number of adjacency matrices representing the same graph. Given a permutation matrix $\forall P \in \{0,1\}^{n\times n}$, we need the encoder and decoder to meet the requirements of the following equations to maintain permutation-invariance:

$$\begin{aligned} &\textbf{Encoder: } \mathcal{E}(PAP^T) = \mathcal{E}(A); \\ &\textbf{Decoder: } D(G(PZ)) = D(G(Z)). \end{aligned} \tag{3.5}$$

where Z denotes the samples from prior distributions. The adjacency matrices of different permutations may result in different graph representations if the model does not have permutation-invariance. Moreover, all permutation matrices need to be trained to learn the underlying representation of graphs. To address this issue, we implement our learning-based model through an all permutation-invariant architecture. Specifically, we ensure that all layers and objective functions are permutation-invariant to achieve this goal.

3.3.1.3 Scalability and Efficiency

The learning-based graph generative model can obtain better graph simulation quality. But there are some problems such as low efficiency and poor scalability, resulting in the incapability to generate real-life graphs. Therefore, it is desirable to develop a new deep learning model which can achieve a good trade-off between the efficiency (scalability) and the simulation quality. In

practice, our can sample nodes without replacement to assemble subgraphs during training process to achieve a good trade-off between efficiency (scalability) and quality.

3.3.1.4 Differentiable Community Information Transmission

When extracting the community structure of a graph, Diffpool [99] can coarsen graphs layer by layer, similar to hierarchical clustering. And the model architecture of Diffpool for graph classification is consistent with that of our discriminator. Therefore, our model encodes graph representation through hierarchical clustering in a set of graphs and transmits this information to the decoder to generate graphs with a similar community structure. Moreover, our pooling layer is differentiable to achieve the objective function to preserve the community structure of observed graphs.

3.3.2 Model Architecture

Figure 3.4 illustrates the architecture of CPGAN. It includes the encoder ($\mathcal{E}$), the generator (G)/decoder ($\mathcal{D}$), and the discriminator (D). The upper right part is the discriminator, which gives its judgments on whether the graph is from real datasets. The lower right part is the generator, which decodes the community structures of a graph and reconstructs a new graph. For graph generation tasks, samples from prior distributions will be directly decoded to generate new graphs. And for graph reconstruction tasks, encoders in discriminator and generator share their parameters. Graphs reconstructed and generated will be fed into this model again to "fool" the discriminator. L_{origin}, L_{rec}, and $L_{generate}$ represent the loss of original graph, reconstructed graph, and generated graph, respectively.

As illustrated in Figure 3.4, for a graph $\mathcal{G}$, given its adjacency matrix A and feature matrix X, the structural information of the graph can be obtained by a ladder encoder. Regarding the community information, we assume that the observed graphs have ground truth community label Y_c, which can be otherwise obtained by applying existing community detection algorithms (e.g., [55]). The output of assignment matrix (will be introduced in Section 3.3.3.2) can be seen as the predicted node community assignments X_r. And the assignment matrix will be constrained by these ground

truth labels. The community information and graph representation will be fed into the discriminator to determine whether the input graph is fake. At the same time, the coarsened graphs of each level will distribute their community structure features to original nodes through a differentiable hierarchical message transmission process. Then the series of community information of each node is decoded to enhance the reconstruction of the graph structure. The node representations sampled from the prior distribution can also be used to generate new graphs.

3.3.3 Ladder Message Transmission Encoder

Here we introduce a ladder-shaped encoder for this task so that our model can adaptively adjust the pooling strategy and extract the community structure information of nodes. We leverage node features X and adjacency matrix $A \in \{0,1\}^{n\times n}$ as the input of our proposed encoder. For each graph $\mathcal{G}$, we use identity matrix as its default node features X. Given node features $X \in \mathbb{R}^{n\times d}$ with d-dimension feature per node and adjacency matrix A, input graph $\mathcal{G}$ will be coarsened using stacked convolution and pooling layers.

3.3.3.1 Graph Convolution

As mentioned in Section 3.2.4.2, the classical message transmitting model is expressed by graph convolution networks (GCN) [25]. Post-transmitted message $Z \in \mathbb{R}^{n\times d'}$ can be calculated as follows:

$$Z = \sigma(GCN(X,A)) = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}XW) \tag{3.6}$$

where $\tilde{D} \in \mathbb{R}^{n\times n}$ denotes the degree matrix of $\tilde{A}$ with $\tilde{D}_{ii} = \sum_{j=0}^{n}\tilde{A}_{ij}$, $\tilde{A}$ denotes adjacency matrix containing self-loop with $\tilde{A} = A + I_n$, $W \in \mathbb{R}^{d\times d'}$ is trainable parameters in graph convolution layer with its kernel size d' and σ denotes activation function (default is the Rectified Linear Unit), and X denotes the node features derived from spectral embeddings of the adjacency matrix A with $X = X(A)$. Information can flow among nodes faster if we use some variants of $\tilde{A}$ (e.g. $\tilde{A} = A + A^2$) to improve the connectivity of graphs. The time complexity of graph convolution is $O(m+n)$, where m denotes the number of edges.

3.3.3.2 Graph Pooling

According to our experience, simply transmitting the message through GCNs can cause a very deep network to capture structure information, especially when we encounter large and sparse graphs with low connectivity. We need an effective way to obtain the hierarchical representations of a graph. Inspired by *Diffpool* [99], we can coarse a graph hierarchically and learn a strategy to coarse a set of graphs through a series of assignment matrices $S = \{S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}, 1 \leq l < k\}$, where n_l, n_{l+1}, and k denote the numbers of input nodes, output nodes, and layers respectively. The assignment matrix is calculated as follows:

$$\begin{aligned} Z^{(l)} &= \sigma(GCN_{l,embed}(X^{(l)}, A^{(l)})) \\ S^{(l)} &= softmax(GCN_{l,pool}(Z^{(l)}, A^{(l)})) \end{aligned} \tag{3.7}$$

where σ is Rectified Linear Unit activation function, $X^{(l)} \in \mathbb{R}^{n_l \times d_{l-1}}$ and $A^{(l)} \in \mathbb{R}^{n_l \times n_l}$ denote the feature matrix and adjacency matrix of n_l *cluster nodes*, respectively, $Z^{(l)} \in \mathbb{R}^{n_l \times d_l}$ denotes feature matrix with structure information of layer l, and two GCNs are leveraged to collect structure information and to infer the pooling strategy of layer l, respectively. Due to the multiple operations of graph convolution and pooling of one graph, we leverage a trick to use PairNorm [68] after each GCN to allow us to stack deep GCNs without over-smoothing. The assignment matrix can be seen as the predicted node community assignments. And the assignment matrix will be constrained by the ground truth labels (will be introduced in Section 3.3.6.2). Given assignment matrix $S^{(l)}$, the coarsened adjacency matrix $A^{(l+1)}$ and new embeddings $X^{(l+1)}$ can be generated as follows:

$$\begin{aligned} A^{(l+1)} &= {S^{(l)}}^T A^{(l)} S^{(l)} \\ X^{(l+1)} &= {S^{(l)}}^T Z^{(l)} \end{aligned} \tag{3.8}$$

Stacking graph convolution and pooling layers can obtain a series of node representations at different levels. In particular, if the layer k has just one node after pooling, the corresponding assignment matrix will be $\{1\}^{n_{k-1}}$, such that the graph pooling is equivalent to a graph readout `sum`. The overall time complexity of graph pooling is $O(m + n)$.

3.3.3.3 Graph Readout

Node representations of each graph are collapsed into a graph representation through graph readout. Therefore, the readout of output feature s_i of i-th level coarsened graph is calculated as follows:

$$\begin{aligned} s_i &= \frac{1}{n_i}\sum_{j=1}^{n_i} x_{ij} \\ s &= s_1 \oplus ... \oplus s_k \end{aligned} \tag{3.9}$$

where k is the number of layers for each graph, x_{ij} denotes the representation of j-th node of i-th level graph, and $\oplus ... \oplus$ denotes combining all representations in a new dimension. The time complexity of graph readout is $O(n)$. The final graph representation $s \in \mathbb{R}^{k\times d}$ is the input of the graph discriminator.

3.3.3.4 Graph Transposed Pooling

In order to reconstruct node representations, we need to properly depool a graph. Different from upsampling from a coarsened graph, we introduce a differentiable methodology to distribute information from the coarsened graph to a detailed graph. The proposed distributing method uses transposed versions of the similar assignment matrix. The transposed assignment matrix $S^{(l)}_{depool} \in \mathbb{R}^{n_{l+1},n_l}$ is calculated as follows:

$$S^{(l)}_{depool} = softmax(GCN_{l,depool}(Z^{(l)}, A^{(l)})^T) \tag{3.10}$$

Therefore, reconstructed node representations $Z_{rec} \in \mathbb{R}^{n\times k\times d}$ are calculated as follows:

$$\begin{aligned} Z^{(l)}_{rec} &= \begin{cases} Z^{(l)} & (l=1) \\ \prod_{i=1}^{l-1} S^{(i)T}_{depool} \times Z^{(l)} & (l>1) \end{cases} \\ Z_{rec} &= Z^{(1)}_{rec} \oplus ... \oplus Z^{(k)}_{rec} \end{aligned} \tag{3.11}$$

where $\oplus ... \oplus$ denotes combining all node representations in a new dimension. After that, Z_{rec} is ready to be the input of our proposed decoder $\mathcal{D}$. The overall time complexity of transposed

pooling is $O(m+n)$. Note that, in this work, we add a variational inference module to conjugate node latent distributions to control the output of our encoder.

3.3.4 Variational Inference

We leverage the variational inference before decoding node features for generating new graphs with observed hierarchical community structure distribution. We use $q_\phi(Z_{vae}|Z_{rec}) = \prod_{i=1}^{n} q_\phi(z_i|Z_{rec}).z_i \in Z_{vae}$ to achieve the mapping from the reconstructed features to the prior distributions $\mathcal{N}(\mu, \text{diag}(\sigma^2))$. And we choose a multi-layer perceptron (MLP) as our inference model. The inference process is formulated as follows:

$$\begin{aligned} g(Z_{rec}, \phi) &= \sigma(Z_{rec}\phi_0)\phi_1 \\ \bar{\mu} &= \frac{1}{n}\sum_{i=1}^{n} g_\mu(Z_{rec})_i \\ \bar{\sigma}^2 &= \frac{1}{n^2}\sum_{i=1}^{n} g_\sigma(Z_{rec})_i^2 \\ q_\phi(z_i|Z_{rec}) &\sim \mathcal{N}(\bar{z}|\bar{\mu}, \text{diag}(\bar{\sigma}^2)) \\ q_\phi(Z_{vae}|Z_{rec}) &= \sum_{i=1}^{n} q_\phi(z_i|Z_{rec}) \end{aligned} \tag{3.12}$$

where ϕ denotes the set of parameters in MLP, $g(\cdot)_i$ denotes the i-th row of $g(\cdot)$, and $Z_{vae} \in \mathbb{R}^{n\times k\times d'}$ is the output of variational inference module. The time complexity of inference module is $O(kn)$. According to [17], probabilistic variational reasoning can make the node representation far away from the zero-center, which intuitively makes the node representation more "sparse". We notice that this is helpful to preserve the node community structure.

After the variational inference module, we can select new node features from the prior distributions to generate new graphs. But we find that fully-connected networks alone cannot handle the task of generating graphs with complex and hierarchical community structure in section 3.4. So we proposed our graph decoder to address this issue.

3.3.5 Graph Decoder

Our proposed graph decoder consists of two steps: first decoding hierarchical graph representation sequences and then predicting node links. We embed hierarchical community structures with Gated Recurrent Unit (GRU) and obtain our node features h_k where k denotes the number of community structures. Decoded features are obtained by the following formula:

$$h_{l+1} = GRU(h_l, Z_{vae}^{(l+1)}). \quad (0 \leq l < k) \tag{3.13}$$

where h_l denotes the hidden state of the coarsened graph, h_0 is a zero matrix, $Z_{vae}^{(l)} \in \mathbb{R}^{n \times d'}$ denotes the node features of the l-th coarsened graph, and h_k denotes the decoded node features with hierarchical community information. After obtaining the node representations, we give the link predictions as follows:

$$\begin{aligned} g_\theta(h_k) &= \sigma(h_k\theta_0)\theta_1 \\ p_\theta(A_{ij}|h_{k,i}, h_{k,j}) &= \sigma(g_\theta(h_{k,i})^T g_\theta(h_{k,j})) \\ p_\theta(A_{rec}|Z_{vae}) &= \prod_{i=1}^{n}\prod_{j=1}^{n} p_\theta(A_{ij}|h_{k,i}, h_{k,j}) \end{aligned} \tag{3.14}$$

where $g_\theta(h_{k,i})$ is a two-layer MLP to extract community information to help generate edges, $h_{k,i}$ denotes the feature of the i-th node, and $A_{rec} \in \mathbb{R}^{n \times n}$ denotes the probability matrix of link prediction. When training decoder on large graphs, to accelerate this process, we sample n_s ($n_s \ll n$) nodes to obtain $A_{rec} \in \mathbb{R}^{n_s \times n_s}$. Specifically, we sample nodes without replacement to assemble subgraphs with a strategy according to node degrees as follows: $P_i = \frac{deg_i}{\sum_{i=1}^{n} deg_i}$, where P_i is the probability to select node i, and deg_i denotes the degree of node i. Therefore, the time complexity of the graph decoder is $O(kn + n_s^2)$.

3.3.6 Discriminator and Optimization

3.3.6.1 Graph Discriminator

Discrimination task requires graph features obtained by the encoder, i.e., output matrix $s \in \mathbb{R}^{k \times d}$ of graph readout layer in section 3.3.3.3 with $s = \mathcal{E}(A)$. We leverage a two-layer MLP classifier

as our discriminator D which is defined as

$$D_\phi(A) = \sigma(MLP(s, \phi)) \tag{3.15}$$

where ϕ denotes the parameters of MLP, and σ denotes the *sigmoid* activation function.

3.3.6.2 Discriminator Optimization

Formally, G and D play minimax game with value function $V(G, D)$ as follows:

$$\begin{aligned}\min_{\phi_G}\max_{\phi_D} V(D,G) =& \frac{1}{n}\sum_{i=1}^{n}\log(D(A_i)) \\ &+ \mathbb{E}_{p(Z_{vae})\sim q(\cdot|Z_{rec})}\log(1 - D(G(Z_{vae}))) \\ &+ \mathbb{E}_{p(Z_s)\sim\mathcal{N}(\cdot|\mathbf{0},\mathbf{I})}\log(1 - D(G(Z_s)))\end{aligned} \tag{3.16}$$

where Z_{vae} and Z_s are sampled from the approximate distributions and Gaussian prior distributions, respectively. Besides, to use the clustering results to enhance the level of discriminator, we introduce the clustering consistency $\mathcal{L}_{clus} = -\sum_{l=1}^{k}\text{logloss}(S^l, Y_c^l)$, where S^l denotes the assignment matrix introduced in Section 3.3.3.2, and Y_c^l denotes the ground-truth community partitions of observed graph. By default, we leverage *louvain* [55] community detection algorithm to obtain hierarchical community detection results as the ground-truth community partitions. In the training process, we need to update ϕ_D through ascending gradient by:

$$\nabla_{\phi_D} V(G, D) = \begin{cases} \nabla_{\phi_D}[\log(D(A)) + \mathcal{L}_{clus}] \\ \nabla_{\phi_D}\log(1 - D(G(Z))) \end{cases} \tag{3.17}$$

when judging graphs from real datasets, we update parameters using the upper part of equation 3.17. Note that, to ensure both community-structure preserving and other optimization objectives are well considered, our training process stops only when both $\mathcal{L}_{clus}$ and $\log(D(A))$ converge. When judging graphs generated, we update parameters using the lower part of equation 3.17.

3.3.6.3 Generator Optimization

The generator aims to minimize the log-probability that the discriminator correctly assigns to the graph reconstructed by G. Besides, to improve the performance of the decoder $\mathcal{D}$ and guarantee the permutation-invariance at the same time, we introduce the mapping consistency $\mathcal{L}_{rec}$ from CycleGAN [25] with $\mathcal{L}_{rec} = \frac{1}{n}\sum_{i=1}^{n}||\mathcal{E}(A_i) - \mathcal{E}(A_i')||^2$, where A_i' denotes the fake adjacency matrix reconstructed from A_i. In practice, the collapse of encoder $\mathcal{E}$ can be controlled by the mapping consistency. We propose computing the gradient of the decoder with respect to $\phi_{\mathcal{D}}$ by descending gradient:

$$\begin{aligned}
&\nabla_{\phi_{\mathcal{D}}}[V(G,D) - \mathcal{L}_{rec}(A', A)] \\
&= \nabla_{\phi_{\mathcal{D}}}[\mathbb{E}_{p(Z_{vae})\sim q(\cdot|Z_{rec})}\log(1 - D(G(Z_{vae}))) \\
&+ \mathbb{E}_{p(Z_s)\sim\mathcal{N}(\cdot|\mathbf{0},\mathbf{I})}\log(1 - D(G(Z_s))) \\
&- \frac{1}{n}\sum_{i=1}^{n}||\mathcal{E}(A_i) - \mathcal{E}(A_i')||^2]
\end{aligned} \tag{3.18}$$

where A' denotes the reconstructed adjacency matrices. After updating the decoder, we propose computing the gradient of the encoder with respect to $\phi_{\mathcal{E}}$ by descending gradient:

$$\begin{aligned}
&-\nabla_{\phi_{\mathcal{E}}}[\mathcal{L}_{prior}(q||p) + \mathcal{L}_{rec}(A', A)] \\
&= -\nabla_{\phi_{\mathcal{E}}}[D_{KL}(q(Z_{vae}|Z_{rec})||p(Z)) \\
&\quad + \frac{1}{n}\sum_{i=1}^{n}||\mathcal{E}(A_i) - \mathcal{E}(A_i')||^2]
\end{aligned} \tag{3.19}$$

where the Gaussian prior $p(Z)$ is set with $p(Z) = \prod_{i=1}^{n} p(z_i) = \mathcal{N}(\bar{z}|\mathbf{0},\mathbf{I})^n$, and $\mathcal{L}_{prior}(\cdot||\cdot)$ denotes calculating the Kullback-Leibler (KL) divergence between two distributions. With this modified encoder and decoder, the generation process can generate new graphs of arbitrary sizes and similar community structures.

3.3.7 Generating New Graphs

After the training, we sample n_s ($n_s \ll n$) nodes to obtain $A_{sub} \in \mathbb{R}^{n_s \times n_s}$ and assemble the output matrix $A_{out} \in \mathbb{R}^{n \times n}$ obtained from the generator and verified by the discriminator into a

generated adjacency matrix. Specifically, we initialize an empty A_{out} and fill in edges genrated in each subgraph's adjacency matrix A_{sub} until the number of generated edge meets requirement. The binarization strategy that choosing a threshold to determine each edge and sampling strategy through Bernoulli distributions parameterized by A_{out} might lead to leaving out low-degree nodes and high variance output, respectively. To address these issues, we use the following strategy: (1) generating one edge for node i through sampling from category distribution parameterized by the i-th row of A_{out}; and (2) selecting top-k entries of A_{out} until the number of edges reaches a pre-defined number. The overall time complexity of generating new graphs is $O(n^2)$.

3.3.8 Discussion

Scalability. Though our proposed method is much more efficient and scalable compared to other learning-based approaches, it cannot compete with traditional approaches (e.g., BTER and ER) in terms of efficiency and scalability because the efficiency of the deep learning based approach relies on GPU which has limited memory size compared to that of CPU. Note that, in terms of graph training, CPGAN can handle large-scale graphs since we use a sampled graph in each training iteration. But CPGAN assumes the whole graph can be accommodated in the GPU memory in the graph simulation procedure, which limits the scalability of CPGAN. It will be interesting to develop more scalable learning-based solutions by making use of CPU memory or even hard disks, which is non-trivial because the swapping between GPU memory and other storage medium is time consuming.

Difference with existing learning-based methods. Same as other learning-based graph generators, some classical models such as VAE and CycleGAN are also used in our method. We focus on two new goals: efficiency (scalability) and community preserving for the learning-based model, which need the development of new techniques. For instance, we leverage VAE to separately infer the hierarchical structure information, which is helpful for community-preserving. Besides, the mapping consistency from CycleGAN and VAE are used for permutation-invariant graph generation, which is essential for our scalable implementation of our sampling strategy.

TABLE 3.2: Detailed Stats of included datasets

	#Nodes	#Edges	#Comm.	d_{mean}	CPL	GINI	PWE
Citeseer	3327	4732	473	2.8446	5.9389	0.6769	2.8757
PubMed	19717	44338	2488	4.4974	6.3369	0.8844	1.4743
PPI	2361	6646	371	5.8196	4.3762	0.7432	1.9029
3D Point Cloud	5037	10886	1577	4.3224	32.40	0.8278	1.9276
Facebook	50515	819090	8010	32.43	14.41	0.7164	1.5033
Google	875713	4322051	9863	9.871	6.3780	0.6729	1.8251

TABLE 3.3: Performance evaluation of compared models for graph community structure preserving tasks in each dataset. NMI and ARI measure the similarity between community structures of generated graph and the one of observed graph, where the higher is better.

Graph	Citeseer		Pubmed		PPI		3D Point Cloud		Facebook		Google	
	NMI(e-2)	ARI(e-2)	NMI(e-2)	ARI(e-2)	NMI(e-2)	ARI(e-2)	NMI(e-2)	ARI(e-2)	NMI(e-2)	ARI(e-2)	NMI(e-2)	ARI(e-2)
SBM	19.7±0.9	1.9±0.1	4.4±0.2	0.3±0.1	11.3±0.7	1.2±0.1	37.0±1.3	11.4±0.7	14.5±2.0	2.1±0.3	24.4±0.9	1.3±0.4
DCSBM	27.1±0.8	1.7±0.1	18.9±0.2	0.3±0.1	18.6±0.8	1.8±0.3	37.3±1.4	11.5±0.8	17.5±1.5	1.9±0.3	29.4±0.6	5.7±0.5
BTER	27.3±0.7	1.8±0.1	19.1±0.2	0.3±0.1	19.0±0.7	1.7±0.1	38.1±1.2	12.1±0.8	17.9±1.2	2.1±0.2	30.3±0.7	5.8±0.5
MMSB	26.7±0.9	4.4±1.0	OOM	OOM	15.4±0.6	0.8±0.4	7.1±0.4	1.3±0.3	OOM	OOM	OOM	OOM
VGAE	63.0±0.4	29.0±1.5	42.0±0.3	15.0±0.4	50.4±0.6	40.0±1.2	57.0±0.8	8.2±1.1	OOM	OOM	OOM	OOM
Graphite	62.8±0.7	28.2±2.1	43.0±0.5	15.1±0.4	52.3±0.8	33.4±1.9	58.8±0.4	13.2±0.3	OOM	OOM	OOM	OOM
SBMGNN	62.6±0.5	21.5±1.0	39.3±0.5	14.1±0.5	56.9±0.4	31.0±1.6	59.2±0.9	15.9±1.1	OOM	OOM	OOM	OOM
NetGAN	57.9±0.5	20.1±0.3	OOM	OOM	55.2±0.5	30.2±0.3	67.4±0.9	37.8±2.6	OOM	OOM	OOM	OOM
CPGAN	**72.5±0.4**	**44.3±1.5**	**45.8±0.9**	**34.1±1.1**	**57.0±0.7**	**44.2±1.3**	**70.6±0.6**	**39.9±1.4**	**54.7±1.0**	**28.4±1.6**	**38.7±0.5**	**30.8±0.5**

3.4 Experiments

We perform extensive experiments to validate the effectiveness of our proposed methods. We describe the experimental settings first. Then, we give the experiment results of preserving community structure and generating realistic graphs compared with state-of-the-art baselines. At last, the model efficiency and memory consumption experiments are conducted, respectively.

3.4.1 Experiment Settings

Note that the source code and dataset used in the experiments are publicly available in GitHub (`https://github.com/xiangsheng1325/CPGAN`).

Dataset. We conduct experiments on six representative datasets in the literature including two

citation networks (Citeseer and Pubmed) [71] [2], PPI[74] [3], 3D point cloud [76] [4], Facebook [75] [5], and Google web pages [78] [6]. These datasets span multiple domains and have different community structures. The graph statistics of these datasets are introduced on Table 3.2, where "#Comm." denotes the number of communities. The detailed information of the six datasets are:

- **Citation Networks**. Citeseer and Pubmed are two typical citation networks, where nodes denote publications, and edges denote the citation relationships among publications. The Citeseer and Pubmed datasets contain 3327 and 19717 publications, and 4732 and 44338 citations, respectively.
- **PPI**. Protein-protein Interaction (PPI) network contains 2361 nodes and 6646 edges, each node representing one yeast protein. Edges are generated if there are interactions between two proteins.
- **3D Point Cloud**. Graph of points of household objects with 5037 nodes and 10886 edges, where nodes denote the objects, and edges are generated for k-nearest neighbors which are measured w.r.t Euclidean distance of the points in 3D space.
- **Facebook**. A real-world social network with 50515 nodes and 819090 edges, each node denoting one page. Edges are generated if there are mutual likes among them.
- **Google**. Web graph with 875713 nodes and 4322051 edges, where nodes represent web pages and edges represent hyperlinks between them.

Compared Methods. We compare our method with both the traditional models and recent deep graph generative models. All baseline models are designed to learn features on a set of graphs and generate new simulated graphs. The conventional baselines include: E-R [43], B-A [30], Chung-Lu [85], SBM [48], DCSBM [50], BTER [51], Kronecker [42], and MMSB [49]. The learning-based generative baselines are: VGAE [17], Graphite [38], SBMGNN [39], GraphRNN-S [21], NetGAN [24], and CondGen-R [25]. Note that we choose the scalable variant of GraphRNN and CondGen as baselines. Besides, we choose the representative baselines due to space limits.

[2] https://linqs.soe.ucsc.edu/data
[3] http://vlado.fmf.uni-lj.si/pub/networks/data/bio/Yeast/Yeast.htm
[4] http://www.first-mm.eu/data.html
[5] https://snap.stanford.edu/data/gemsec-Facebook.html
[6] https://snap.stanford.edu/data/web-Google.html

TABLE 3.4: Performance evaluation of compared models for graph generation tasks in each dataset. The evaluation results are the absolute differences from true measures, where the lower is better.

Graph	Citeseer					3D Point Cloud					Google				
	Deg.	Clus.	CPL	GINI	PWE	Deg.	Clus.	CPL	GINI	PWE	Deg.	Clus.	CPL	GINI	PWE
E-R	1.27e-2	1.71e-2	17.5	8.86e-2	0.12	0.349	2	25.6	0.237	13.6	6.24e-2	1.36	13.17	3.99e-2	0.221
B-A	1.40e-2	1.25e-2	19.4	0.159	1.43	0.546	2	27.7	0.331	12.2	1.94e-2	1.36	11.1	6.16e-2	0.54
Chung-Lu	1.47e-2	1.73e-2	18.5	9.83e-2	0.15	0.353	2	25.7	0.222	13.7	6.48e-2	1.29	13.32	7.31e-2	0.624
SBM	1.36e-2	4.94e-3	**12.4**	7.87e-2	5.13e-2	0.317	1.99	23.4	0.209	13.8	0.111	0.886	6.93	0.113	0.892
DCSBM	2.40e-2	3.44e-3	13.3	0.142	8.14e-2	0.309	1.98	23.4	0.218	13.8	8.48e-2	0.865	11.8	9.17e-2	0.595
BTER	1.21e-2	2.71e-3	13.1	7.73e-2	**3.03e-2**	**0.301**	2	22.6	**0.207**	13.6	1.85e-2	0.834	6.67	3.93e-2	0.210
Kronecker	2.58e-2	1.91e-2	18.5	0.132	3.12e-2	0.370	2	26.8	0.240	13.8	0.102	1.28	15.1	5.19e-2	1.2
MMSB	2.98e-2	1.84e-2	17.9	0.173	0.186	0.339	2	25.9	0.234	13.7			OOM		
VGAE	0.123	3.78e-2	18.2	0.477	0.126	0.731	1.96	30	0.864	13.8			OOM		
GraphRNN-S	1.34e-3	**1.48e-3**	17.3	7.32e-2	0.176			OOM					OOM		
CondGen-R	8.42e-2	0.14	20.8	0.362	0.295	0.604	1.73	30.4	0.658	14.1			OOM		
NetGAN	**1.07e-3**	1.51e-3	16.5	0.136	0.154	0.415	1.72	26.3	0.542	14.6			OOM		
CPGAN	1.25e-3	2.26e-3	15.3	**7.23e-2**	9.32e-2	0.410	**1.49**	**18.1**	0.355	**10.8**	**1.47e-2**	**0.672**	**6.45**	**3.43e-2**	**0.118**

Therefore, the SGAE proposed in Chapter 2 was excluded. In order to validate the effectiveness of CPGAN's sub-modules, we deploy some variants of our method, which are seperately denoted as CPGAN-C ("C" for "replacing the node decoding operation with a concatenation"), CPGAN-noV ("noV" for "not to use variational inference"), and CPGAN-noH ("noH" for "not to use hierarchical pooling") in the experiment. The CPGAN is our proposed graph generator in this chapter.

Parameter Settings and Evaluation Metrics. The included algorithms and evaluating scripts are implemented and compiled through Python-3.6, PyTorch-1.8.1, CUDA-11.1, and GCC-4.8.5 in our experiments. The experiments are operated on a machine with Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 80 GB RAM and NVIDIA RTX 3090 with 24 GB memory. We use one CPU core and one GPU for every algorithm. In the experiment, we set the graph convolution kernel size to 128 in the ladder message transmission encoder. The learning rate is set to 0.001, the graph pooling size to 256. For various baselines, we employ the original hyperparameters settings of the compared models. To evaluate the performance of all methods, we leverage the following benchmark metrics in the experiment:

Deg.: Maximum Mean Discrepancy (MMD) of degree distribution measuring for the difference between degree distributions of two graphs.

Clus.: MMD of clustering coefficient distribution measuring for the difference between clustering coefficient distributions of two graphs.

CPL: The difference of characteristic path length between two graphs.

GINI: The difference of GINI index between two graphs, where GINI is a common measure for inequality in a degree distribution.

PWE: The difference of power-law exponent between two graphs.

NMI and ARI: To evaluate the community-preserving property, we compare the community structure similarity between the observed graphs and the generated graphs. Our evaluating method is based on the *louvain* [55] community detection algorithm. *louvain*, which has a complexity of $O(m + n)$, can quickly and hierarchically detect the community structure of the graph, and obtain the community membership of nodes unsupervised based on maximizing modularity Q. The modularity of community memberships of a graph is defined as follows:

$$Q = \frac{1}{2m} \sum_{i,j} [A_{ij} - \frac{d_i d_j}{2m}] \delta(c_i, c_j) \tag{3.20}$$

where m represents the number of edges, d_i denotes the degree of node i, the $\delta(u, v)$ is 0 when $u = v$ and 1 otherwise and c_i denotes the community membership to which node i is assigned. We suppose that graphs having the same community structure should have the same community detection results, so we use two popular clustering metrics, Normalized Mutual Information (NMI) and Adjusted Rand Index (ARI), to quantitatively evaluate the community structure of the generated graphs.

3.4.2 Graph Generation

We introduce the experiments on several perspectives: preserving community structure, generative distribution distance, and parameter sensitivity. We conduct representative experiments and prove the superiority of our model in the task of graph generation, and some experiments with similar observations are excluded.

TABLE 3.5: Performance comparison for graph reconstruction tasks in each dataset.

Graph	PPI							Citeseer						
	Deg.	Clus.	CPL	GINI	PWE	Train NLL	Test NLL	Deg.	Clus.	CPL	GINI	PWE	Train NLL	Test NLL
VGAE	0.257	1.69	**6.11**	0.342	0.633	1.96	3.61	9.01e-2	1.6	1.45	0.263	0.149	2.26	3.78
Graphite	0.315	0.815	10.9	0.362	0.760	2.09	4.38	0.306	1.53	2.14	0.311	1.17	2.41	4.15
SBMGNN	0.356	1.61	10.9	0.397	0.777	2.20	4.00	0.217	1.32	2.14	0.358	0.517	2.31	4.26
CondGen	0.139	1.16	12.8	0.231	1.09	2.07	3.82	0.166	1.13	3.57	0.196	1.54	2.47	3.97
CPGAN	**6.21e-2**	**0.243**	11.31	**7.43e-2**	**0.437**	**1.84**	**3.52**	**8.49e-2**	**0.498**	**1.35**	**1.38e-2**	**3.16e-2**	**1.78**	**3.68**

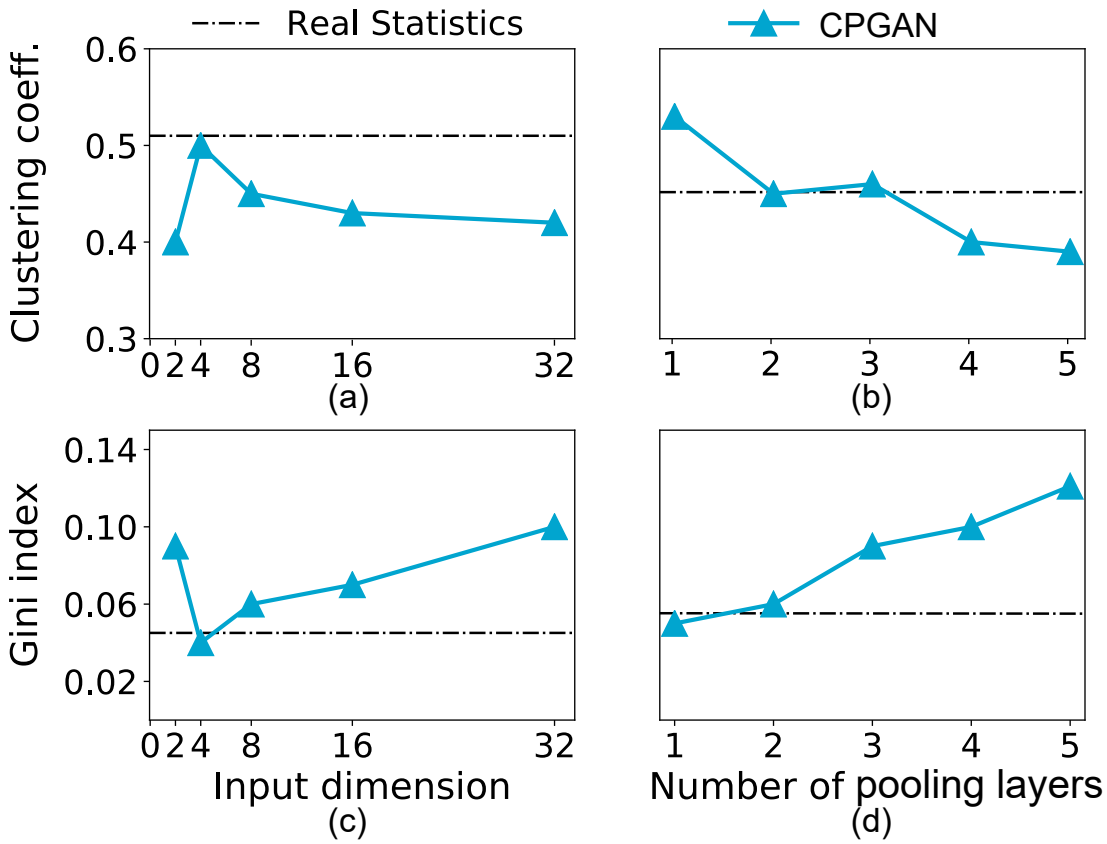


FIGURE 3.5: Parameter sensitivity experiment results. Points closer to the real statistics are better.

Preserving Community Structure. In this experiment, we first evaluate the generated graph by comparing community structures based on *louvain* [55] community detection algorithm. We compare the similarity of detection results between the observed graph and generated graph. Table 3.3 shows the performance of each method in preserving community structure, which is one of the main tasks of this chapter. Note that several baseline methods are excluded because of the unstable node permutations. Some algorithms cannot do the graph simulation on some graphs due to the limit of memory, and the corresponding results are marked as "OOM" in the table.

The first 8 rows are the results of baseline methods. As expected, CPGAN demonstrates the best performance among all graph generators evaluated, especially for the evaluation of ARI. It is

TABLE 3.6: Performance evaluation of sub-models for graph community preserving and graph generation tasks in 3 datasets. The NMI and ARI rows show the community-preserving measures of generated graphs, where the higher is better, while others are the absolute differences from true measures, where the lower is better.

Graph	PubMed				PPI				Facebook			
	NMI(e-2)	ARI(e-2)	Deg.	Clus.	NMI(e-2)	ARI(e-2)	Deg.	Clus.	NMI(e-2)	ARI(e-2)	Deg.	Clus.
CPGAN-C	32.1	14.5	2.38e-3	2.23e-3	51.2	39.3	2.47e-3	1.35e-2	53.3	26.1	1.20e-3	1.43e-2
CPGAN-noV	31.3	14.3	3.03e-3	5.14e-3	50.5	39.0	2.77e-3	1.76e-2	52.9	25.3	1.24e-3	1.56e-2
CPGAN-noH	28.8	13.2	3.96e-3	6.52e-3	49.7	38.4	3.49e-3	2.30e-2	50.1	23.2	1.96e-3	1.79e-2
CPGAN	**45.8**	**34.1**	**2.08e-3**	**1.81e-3**	**57.0**	**44.2**	**2.35e-3**	**1.12e-2**	**54.7**	**28.4**	**1.18e-3**	**1.35e-2**

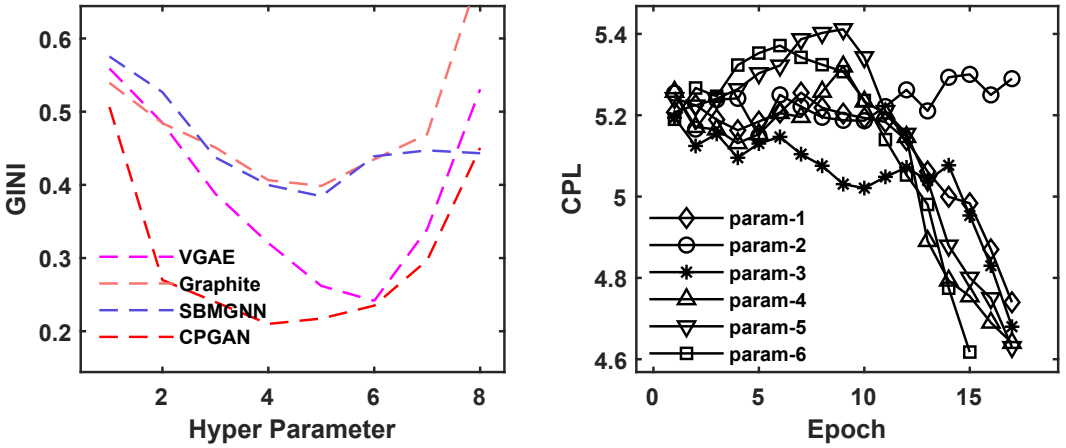


FIGURE 3.6: Model robust experiment results. The *left* part is the performance of several compared methods, and the *right* part is the performance of different hyper parameter settings. Points lower are better.

shown that BTER achieves the best performance compared with other traditional baselines. Nevertheless, its performance is not competitive compared to the learning-based models even most of them do not explicitly consider the community preserving property. As stressed in Section 3.2.4.2, SBMGNN does not utilize the deep neural network for the purpose of community preserving and hence does not show an advantage in the performance evaluation compared to other learning-based models.

Generative Distribution Distance. Table 3.4 shows the performance of different methods in graph generation. Each evaluation metric represents the difference between real graphs and generated graphs. The first 12 rows are the results of baseline methods. It is shown that BTER has the best performance of the traditional graph generators (first 6 Lines). The deep learning-based generative models (Lines 7-12) can significantly improve the performance. It is shown that, in addition to the best performance on community preserving, CPGAN also demonstrates competitive performance on other quality measures on large graphs compared to the baselines. In two datasets with

larger graph sizes, our method performs considerable improvements compared to the baselines. It is reported that CPGAN achieves one out of five best results in Citeseer, three out of five best results in 3D Point Cloud, and all five best results in the Google web graph dataset. According to the experiments, CPGAN achieves the best results on PubMED and FaceBook datasets, similar to the conclusions on Google datasets. And CPGAN shows the best performance on 3D Point Cloud data sets. Besides, CPGAN has competitive performance on PPI dataset, similar to the conclusions on 3D Point Cloud data sets. We notice that on PubMed, FaceBook, Google datasets, the compared learning-based baseline algorithms all lead to memory overflow due to their high space complexity. As expected, CPGAN always outperforms CPGAN-C, demonstrating the effectiveness of our new graph decoding module.

Parameters Sensitivity. Figure 3.5 illustrates the result of parameter sensitivity experiments. We report the statistical differences between the generated graph and the real graph according to the changes of spectral embedding dimension (Figures 3.5a and 3.5c) and the number of hierarchies ((Figures 3.5b and 3.5d)), where the number of hierarchies denotes the number of node clustering results in ladder encoder. It is clear the level of hierarchies around two achieves the best performance, which proves the essence of preserving community information in the learning process. Besides, the change of the dimension has no significant influence on the performance of the model. Based on the sensitivity experiment, we chose the best parameter (the input dimension of four and the level of hierarchical structures of two) in our experiment.

The left part of Figure 3.6 illustrates the comparison of model robustness and the training difficulty of our methods. We select the models with similar architecture and hyper-parameters to compare. When the hyper-parameters are traversed in the interval, our method is obviously more robust than other methods. Other experimental results on model robustness can draw the same conclusion. Based on the model robustness comparison experiment, we chose the best parameter settings of other baseline methods (the input dimensions and the hidden dimensions). The right part of Figure 3.6 illustrates the tuning difficulty of hyper-parameters. Our method is stable and has few collapses or instability in the case of different hyper-parameters. Based on the model tuning difficulty experiment, we chose the best training strategy (the learning rate of 0.001 and decay of 0.3 per 400 epochs) in our experiment.

TABLE 3.7: Time consumption (seconds) per graph generation.

#Nodes	0.1k	1k	10k	100k
E-R	**4.6**e^{-4}	9.0e^{-3}	0.46	10.1
B-A	1.0e^{-3}	1.2e^{-2}	**0.11**	1.17
Chung-Lu	7.2e^{-4}	2.5e^{-3}	0.18	2.38
SBM	6.1e^{-3}	0.09	2.58	37.1
DCSBM	6.2e^{-3}	0.09	2.69	39.3
BTER	1.28e^{-3}	**1.9**e^{-3}	0.16	**0.25**
MMSB	6.1e^{-3}	0.09	2.56	-
Kronecker	8.5e^{-3}	0.08	1.00	9.69
GraphRNN-S	0.27	4.74	63.6	-
VGAE	4.2e^{-3}	0.04	0.38	-
Graphite	6.1e^{-3}	0.06	0.64	-
SBMGNN	0.01	0.11	1.18	-
NetGAN	8.7e^{-3}	0.09	1.12	-
CondGEN-R	8.3e^{-3}	0.15	-	-
CPGAN	9.1e^{-3}	0.08	0.95	86.1

TABLE 3.8: Time consumption (minutes) of the entire training process.

#Nodes	0.1k	1k	10k	100k
MMSB	0.11	0.91	40.3	-
Kronecker	1.39	1.55	**3.25**	**4.73**
GraphRNN-S	1.63	15.4	161	-
VGAE	**0.06**	**0.42**	9.75	-
Graphite	0.07	0.47	10.6	-
SBMGNN	0.08	0.63	12.4	-
NetGAN	0.27	2.80	31.1	-
CondGEN-R	0.18	25.3	-	-
CPGAN	0.35	0.70	6.39	32.9

TABLE 3.9: Peak GPU memory usage (MiB) during training

#Nodes	0.1k	1k	10k	100k
MMSB	**1575**	**1709**	18529	OOM
GraphRNN-S	1913	1959	5501	OOM
VGAE	1719	1759	4799	OOM
Graphite	1719	1761	4819	OOM
SBMGNN	1719	1767	5243	OOM
NetGAN	2237	2552	5008	OOM
CondGEN-R	1722	1789	-	-
CPGAN	1728	1760	**2467**	**7930**

3.4.3 Graph Reconstruction

In this experiment, we use the complete dataset of PPI and Citeseer. We randomly select 80% edges of the dataset as the training set and employ the model to reconstruct the whole graph, including the rest 20% test set edges. We compute the negative log-likelihood (NLL) of the score

given by the discriminator and report the average number from train and test data sets. We exclude the E-R, B-A, and other methods that cannot employ to reconstruct graphs, and the experiments on other dataset preserves the conclusion that our method improves the performance among other baselines. Table 3.5 reports the experimental results. As we can see, our method achieves the most competitive results in the PPI dataset and perform the best in the Citeseer dataset with significant improvements to VGAE, Graphite, SBMGNN, and CondGen. The results are in accord with graph generation experiments, which prove the effectiveness of our method in realistic graph generation. We notice that the performance of the GAN-based models, including CPGAN, are not good for the measure of CPL on graphs with low CPL value (e.g., the CPL value of PPI dataset is 4.38 as shown in Table 3.2, which is the smallest one in 6 datasets). The reason is that the edges of graphs with low CPL value tends to be denser than ones with high CPL value. The dense edges may lead to a more complex discriminator. So the generator part of GAN will make more complex decisions to avoid the strong attack from discriminator part of GAN. Then the structure of dense graph generated by the GAN-based graph generative model is not stable in adversarial training process. And the CPL is a structure-sensitive measurement of generated graphs. Therefore, GAN-based models (e.g., CondGen and CPGAN) perform worse than other baseline models on graphs with small CPL value.

3.4.4 Ablation Study

We evaluate the effectiveness of each sub-module in our proposed method in this subsection. In Table 3.6, the first 3 rows show the performance of our proposed model's variations. It can be seen that our proposed method outperforms all the variants. The row 2 of Table 3.6 shows that the variant CPGAN-noV performs worse than our proposed CPGAN, which demonstrates the effectiveness of variantional inference. We can also find that the variant CPGAN-noH shows the worst performance among these variants, which demonstrates the effectiveness of our proposed model's components, especially for the ladder encoder with hierarchical pooling.

3.4.5 Model Efficiency and Scalability

We evaluate the efficiency and scalability of the graph generators in this subsection. Table 3.7 reports the time consumption of inferring a new graph where the number of nodes varies from 0.1K

to 100K. Table 3.8 chronicles the time consumption of the entire training process and Table 3.9 details the peak memory usage during the training process. As expected, the traditional graph generators such as BTER, Chung-Lu, E-R, B-A, SBM, DCSBM, and Kronecker outperform the learning-based graph generators in terms of efficiency and scalability, especially on large graphs. On the other hand, it is shown that CPGAN has the best efficiency and scalability among the learning-based approaches when the size of the graph grows. For instance, only CPGAN can handle the graph with size 100K in Tables 3.7, 3.8 and 3.9.

3.5 Conclusion

3.5.1 Summary of Experiments

It is reported that traditional graph generators (e.g., BTER and ER) significantly outperform learning-based graph generators in terms of efficiency and scalability, especially on large graphs. Due to the limitation of the learning-based methods as discussed in subsection 3.3.8, all learning-based methods, including CPGAN, cannot handle larger dataset with millions-scale nodes under current experiment setting (e.g., 24GB GPU memory). This suggests that BTER is the best choice when simulating large-scale graphs since it has the best graph simulation quality among all traditional graph generators and competitive performance in terms of efficiency and scalability. Nevertheless, traditional graph generators are not good choices in some applications where high simulation quality is required on real-life graphs. On the other hand, existing learning-based approaches can achieve good simulation quality, but have poor performance in terms of efficiency and scalability. It is shown that CPGAN has the best efficiency and scalability among the learning-based approaches when the size of the graph grows. Given the best performance on community-preserving and competitive performance on other simulation quality evaluation metrics, *we boast that CPGAN achieves a very good trade-off between graph simulation quality and efficiency (scalability) compared to other approaches*.

3.5.2 Discussion

In this chapter, we propose a deep generative model named CPGAN to simulate real-life graphs. Our model is designed to keep community structure together with other important properties in the graph simulation process. The simulation quality and efficiency (scalability) are two important but contradictory requirements of graph generators in practice, and our method can achieve a good trade-off, especially in the large real-world networks, compared to existing general graph generators.

Chapter 4

Graph Generation for Temporal Graphs

4.1 Chapter Overview

In this chapter, we proposed a learning-based solution for generating temporal graphs, which is better than State-of-the-art in terms of both efficiency and quality. This chapter has been accepted by IEEE ICDE 2025. Section 4.2 gives the background of simulating temporal graphs. Section 4.3 introduces the preliminaries of temporal graph generation. Section 4.4 shows our proposed temporal graph auto-encoders (TGAE). Section 4.5 reports the experimental results. Section 4.6 concludes this chapter.

4.2 Background

Due to the graph's strong expressive power, a host of researchers in fields such as e-commerce, cybersecurity, social networks, military, public health, and many more, are turning to graph modeling to support real-world data analysis [1, 4, 6, 67, 101]. An important line of research for graph processing and analytics is the simulation of graphs which is used for many purposes such as tackling the inaccessibility of the whole real-life graphs and a better understanding of the distribution of graph structures and other features [14, 48, 77]. There is a long history of study for the

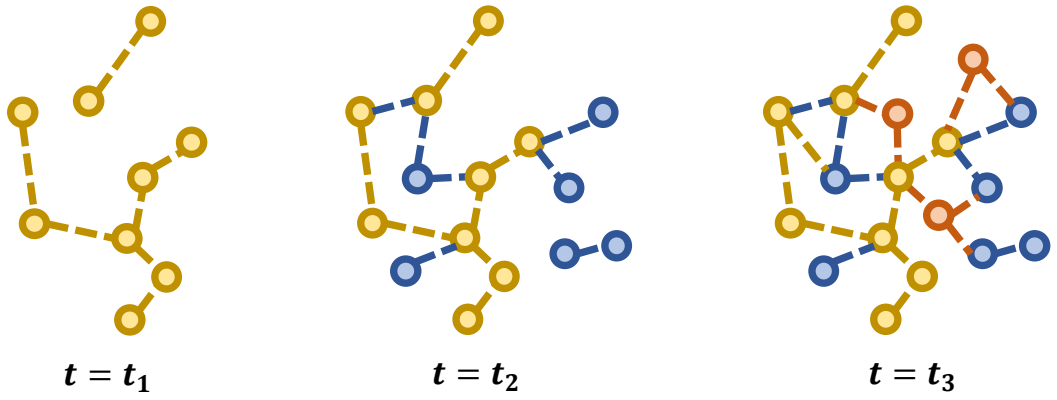


FIGURE 4.1: An example of time-evolving graph.

graph simulation in many research fields such as Database and Machine learning [67, 102, 103]. Recently, many research efforts have been devoted to design advanced generative models which can significantly enhance the simulation quality, thanks to the recent development of deep learning techniques. Nevertheless, we notice that most of the existing works aim to simulate the static graphs. While in many real-life applications, the data are naturally modeled as temporal graphs (a.k.a time-evolving graphs) where the graph evolves with the time. For instance, in the online finance networks and e-commerce networks [104, 105], the edges consists of a sequence of transactions with timestamps for users or products. In the location-based service networks with regarding to the Point of Interests (POIs) [106], an edge corresponds to a visit of an user towards a POI (e.g., a restaurant) at a particular time. In these applications, it is critical to capture the *evolution of the graphs* over the time; that is, the structure of the real graph will evolve with time, and hence the edge generative probability distribution of the nodes on the graph will change with the passage of time. For most graph simulators, which do not properly consider the temporal information of the graphs, a straightforward way is to simulate the temporal graphs based on some particular timestamps, i.e., learn the snapshots of the evolving graph at these timestamps. Unfortunately, this is not feasible in practice because it is cost-prohibitive to learn and simulate many snapshots of the graphs. Thus, to effectively and efficiently support temporal graph simulation in many key applications such as the generation of new drug molecules [10], chemical reaction pathway simulation [107], router load balancing [77], and pandemic trajectory generation [108], it is essential to develop advanced graph generative models which can capture both structure and time properties of the time-evolving graphs.

Motivation. We can store the time-evolving graph as a collection of graph snapshots (i.e. a series of time-stamped static graphs). The collection of graph snapshots contains all edges, nodes, and their corresponding timestamps. Specifically, as shown in Figure 4.1, after a period of time, several temporal nodes and edges are added to this time-stamped graph snapshot. The traditional way to model such a time-evolving graph is to aggregate timestamps into a series of snapshots. A recently developed state-of-the-art method [40], namely *Taggen*, reconstructs a set of temporal random walks to assemble a synthetic graph. However, a disadvantage comes from the inevitable bias of decomposing the time-evolving graphs into a set of temporal random walks. In this case, we have to live with the extra time and space required by a lot of operations of random walk sampling. If the number of walks is too large, the large number of training samples will bring unbearable computational costs when training deep generative models; if the number of walks is too small, the rich temporal structure properties may be lost during model training. The other disadvantage comes from the $O(T^2)$ factor in the time and space complexity analysis of *Taggen* where T is the number of distinct timestamps. complexity of time consumption and memory usage. This directly limits the efficiency and the scalability of the graph simulation model, especially when we need to simulate fine-grained time-evolving graphs.

This motives us to develop a new generative model for time-evolving graphs with better efficiency (scalability) and simulation quality. Particularly, to better capture both the structure and temporal information of the graph, we design a new temporal ego-graph based sampling approach for the temporal graph and we re-weight the input temporal nodes according to the degree of the node, so as to give priority to learning the local temporal structure of the representative nodes. For these representative nodes, we sample their ego-graphs to learn the representative temporal graph structure. We encode and reconstruct the ego-graphs' structure through the temporal graph attention mechanism. As to ego-graphs with k radius, we stack k temporal graph attention layers to layer by layer transmit messages from the periphery nodes to the center nodes. After that, we use the cross-entropy loss to learn directly from the representative temporal graph structure. We also designed a GPU-friendly parallel temporal ego-graph training strategy and the corresponding approximate objective function to reduce the number of steps of model training and achieve efficient model training. Specifically, we combine all the sampled ego-graphs to assemble k bipartite computation graphs to parallelize the computation process and reduce redundant repeats.

After our improvements, the number of steps of model training is $O(\frac{nT}{n_s})$, and the space consumption of model training is $O(n \times (T + n_s))$, where n_s denotes the hyper-parameter of the number of sampled initial nodes and n is the number of graph nodes. As demonstrated in the empirical study, our new approach can achieve much better simulation quality compared to the state-of-the-art technique. Moreover, the new approach also win out from space consumption, efficiency and scalability perspectives.

Contribution. Our principal contributions in this chapter are summarized as follows:

- We propose a new ego-graph based sampling strategy for the temporal graph, which has a stronger expressiveness to better capture the local structure and temporal properties of the temporal graph. We also design an efficient ego-graph sampling strategy and GPU-friendly parallelization ego-graph training strategy, as well as approximate optimization objectives, to achieve scalable model training.
- For better simulation of temporal graphs, we design a Temporal Graph Autoencoder (TGAE). Specifically, temporal node messages are passed from peripheral nodes to the central node in the ego-graph through a temporal graph attention mechanism. Then, the entire ego-graph is reconstructed evolutionarily from the central node through a variational autoencoder.
- Through the extensive experiments on both real-life and synthetic time-evolving graphs, we boast that our new approach significantly outperforms the state-of-the-art in terms of temporal graph simulation quality, efficiency, scalability and space consumption.

Roadmap. The rest of the chapter is organized as follows. In Section 4.3, we formally define the problem and provide the related works of this chapter. We then present the details of our proposed graph generator and introduce the optimizing targets in Section 4.4. Comprehensive experimental results for temporal graph generators are presented in Section 4.5. Section 4.6 concludes the chapter.

TABLE 4.1: The summary of symbols

Symbol	Definition
$\mathbf{X}$	the input features of node occurrences
$\hat{\mathbf{Y}}$	the probability of generating an edge
n	the total number of nodes
m	the total number of edges
T	the number of timestamps in $\tilde{G}$
$\tilde{V} = \{v_1^{t_{v_1}}, ..., v_n^{t_{v_n}}\}$	the set of temporal nodes
$\tilde{E} = \{e_1^{t_{e_1}}, ..., e_m^{t_{e_m}}\}$	the set of temporal edges
$\mathbf{N}(\cdot)$	the neighborhood function
d	the number of dimension

4.3 Preliminary

4.3.1 Problem Definition

Table 4.1 summarizes the symbols introduced in this chapter. In this section, we formalize the graph generation problem in the temporal graph [40, 109]. Given us a temporal graph $\tilde{G}$, we model the temporal graph as a series of graph snapshots $\{G^{t_1}, ..., G^T\}$, which include temporal nodes $\{v_1^{t_{v_1}}, ..., v_n^{t_{v_n}}\}$ and temporal edges $\{e_1^{t_{e_1}}, ..., e_m^{t_{e_m}}\}$. The definitions of temporal nodes and edges are as follows:

Definition 4.1. Temporal Nodes and Edges. *In a temporal graph, a node v_i is associated with a node id i and timestamps v_i occurred with $v_i = \{v_i^{t_1}, v_i^{t_2}, ...\}$. Same as temporal node v, an edge e_j is associated with a timestamp t_{e_j} and two temporal nodes $u^{t_{e_j}}$ and $v^{t_{e_j}}$. For temporal nodes and edges in the same timestamp t, the set of nodes and edges are defined as V^t and E^t, respectively.*

Different from previous temporal graph generation work [40], where the temporal graph was described as a set composed of timestamped edges, the temporal graph is described as a series of graph snapshots in this chapter. The definitions of temporal graph and graph snapshots are as follows:

Definition 4.2. Temporal Graph. *A temporal graph $\tilde{G} = \{G^{t_1}, ..., G^T\}$ is formed by a series of temporal graph snapshots $G^t = (V^t, E^t)$ with $t = 1...T$. And a snapshot G^t is associated with a timestamp t, temporal nodes $V^t = \{v_1^t, v_2^t, ...\}$, and temporal edges $E^t = \{e_1^t, e_2^t, ...\}$.*

In existing graph generatorss [21, 24, 100], the graph neighborhood $\mathbf{N}(v)$ of node v is defined as static one. Here, we generalize the definition of graph neighborhood to the temporal graph, which is defined as follows:

Definition 4.3. Temporal Neighborhood. *Given a temporal node v^t, the neighborhood of v^t is defined as $\mathbf{N}(v^t) = \{v_i^t | f_{sp}(u_i^t, v^t) \leq d_{\mathbf{N}}, |t_v - t_{u_i^t}| \leq t_{\mathbf{N}}\}$, where $f_{sp}(\cdot|\cdot)$ denotes the shortest path length between two nodes, $d_{\mathbf{N}}$ and $t_{\mathbf{N}}$ denote the path length and time window length, respectively, which are the hyper-parameters.*

The previous work defines k-length temporal walks [40, 110], and we relax this definition to the ego graph with a radius of k, so that we can capture the temporal neighbor structure of the observed graph. We consider all the neighbor nodes within the time window of the graph and take these nodes and their corresponding temporal edges as k-radius temporal ego-graph, which is defined as follows:

Definition 4.4. k-Radius Temporal Ego-Graph. *Given a temporal node v^t, a k-radius temporal ego-graph $\tilde{G}_{ego}(v^t)$ is composed of temporal nodes $\tilde{V}_{ego}(v^t)$ and edges $\tilde{E}_{ego}(v^t)$ corresponding to the temporal neighborhood of v, i.e., $\mathbf{N}(v^t)$.*

Problem statement. We formally define the temporal graph generation problem as follows:

Input: *We use a temporal graph $\tilde{G} = \{\tilde{V}, \tilde{E}\}$ for graph simulation.*

Output: *The generated temporal graph $\tilde{G}' = \{\tilde{V}', \tilde{E}'\}$ with highly preserved both structural and temporal properties.*

Evaluation of preserving temporal structure. The key of the temporal graph simulation is to keep the structure information along the whole involving procedure. Ideally, for every timestamp, the observed time-involving graph and the simulated corresponding graph have the same distribution. To verify the performance of preserving the temporal structure, we use the following two evaluation approaches for the temporal graph generative algorithms:

(1) calculating the difference of graph statistics for the snapshots under the same timestamp. Specifically, we randomly choose a timestamp and accumulate the nodes and edges generated from the initial timestamp to the current timestamp to get the generated graph snapshot. Similarly,

the snapshot of the original graph can be obtained by accumulating temporal edges and nodes in the same way. The difference between the generated snapshot and the original snapshot can be obtained by comparing the differences in graph statistics between the two. A distinct description of each graph statistic is introduced in Table 4.3 of Section 4.5.

(2) comparing the difference in temporal motif distribution. Specifically, after generating the last timestamp, we get the snapshot of the whole temporal graph. By computing the temporal motif distribution in graph snapshot (e.g., via counting the 3-edge temporal motifs [109]), we can get the motif distribution of the generated graph and the original graph. Then we use a Gaussian kernel of the total variation (TV) to measure the distance between the two motif distributions. After that, we use the popular evaluating metric Maximum Mean Discrepancy (MMD) to measure the similarity of two distributions. Assuming that p and q denote the original graph's motif distribution and the generated graph's, respectively, the measurement of two distribution is formulated as follows:

$$\begin{aligned} \text{TV}(p,q) =& \mathbb{E}_i[||\pi_p(i) - \pi_q(i)||] \\ \text{MMD}^2(p||q) =& \mathbb{E}_{x,y\sim p}[k(\text{TV}(x,y))] + \mathbb{E}_{x,y\sim q}[k(\text{TV}(x,y))] \\ & -2\mathbb{E}_{x\sim p,y\sim q}[k(\text{TV}(x,y))]. \end{aligned} \tag{4.1}$$

where k denotes the Gaussian kernel and $\pi_p(i)$ denotes probability of the i-th motif in temporal graph distribution p.

4.3.2 Related Works

This sub-section presents a review of recent literature on temporal graph learning and graph generative models.

Temporal Graph Learning. Recently, a large number of work has appeared in temporal graph learning. In these works, temporal graphs can be represented by a set of timestamped nodes and edges. For instance, Spatio-temporal graph convolution networks are leveraged on traffic forecasting [111]. The linkage evolution process is leveraged on temporal graph representation learning [112]. The graph attention mechanism is proposed to focus on the crucial part of the graph data for time-evolving graph learning [113]. Dynamic parameters of graph convolution networks are proposed to improve the temporal graph embeddings [114]. [115] decouples model

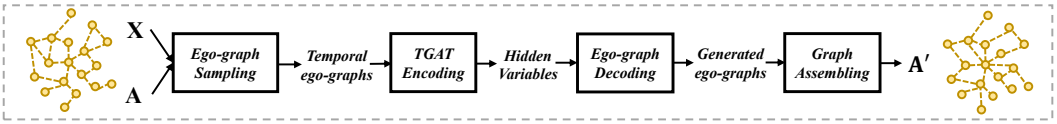


FIGURE 4.2: The framework of our proposed TGAE.

inference and graph computation to alleviate the damage of the heavy graph query operation to the speed of model inference. [116] proposes the hyperbolic temporal graph network (HTGN) that fully takes advantage of the exponential capacity and hierarchical awareness of hyperbolic geometry. However, separately focusing on temporal property and topological structure limits the embedding quality. To address this issue, our encoder aims to encode the temporal and topological structure together into one embedding. In this chapter, we propose to use temporal graph attention as the encoder component of our proposed temporal graph auto-encoder (TGAE).

Graph Generative Model. Early graph simulation problems were often solved by models for static graphs. The detailed discussion of graph generative models was included in previous Chapters. These generative models are all based on static graphs. However, these works ignore the dynamic nature of real-world graphs, i.e., the topology structure of a graph evolves over time. To solve this problem, Taggen [40] and TGGAN [117] were recently proposed to use generative adversarial networks (GANs) to simulate temporal graphs, and GANs achieved the best temporal graph simulation quality. Recently, there is a work on temporal graph simulation [102], which is essentiall different with our study, as they leverage simulation on temporal graph pattern matching. Different from generating random walks, our method directly learns the generative distribution of observed graphs. We also proposed an efficient learning strategy that achieves good trade-off between simulating quality and efficiency.

4.3.3 State of the Art

The state-of-the-art method Taggen [40] and TGGAN [117] simulate the real-world temporal graph in four steps: 1) sampling temporal random walks from observed temporal graphs. 2) new temporal random walks are synthesized by the generator. 3) the validity of the newly generated random walk is judged by the discriminator, and the invalid random walks are discarded. 4) assembling the valid generated temporal random walks into a new temporal graph.

Experiments show that TGGAN outperforms all baselines on generative quality in the task of generating temporal graphs. However, the quality and efficiency of temporal graph generation still remain a large space for improvement. For instance, we observe the time and space complexity of $O(kwb)$ are required by the first three steps of sampling-generation-discrimination, where k is the number of random walks, w is the length of random walks, and b denotes the number of iterations in model training. According to experimental practice and theoretical analysis [100], in the random walk-based generative model, $O(kw)$ can only simulate a high-quality temporal graph when it is close to $O(n^2t^2)$ where n is the number of graph nodes and t is the number of distinct timestamps. The fourth step of the assembly process requires $O(n^2t^2)$ time complexity and space complexity, which is actually very poor scalability for temporal graph generation tasks. Especially, $O(t^2)$ complexity cannot generate temporal graphs with a large number of timestamps, which makes it difficult for us to simulate a fine-grained time-evolving graph.

4.4 Our Approach

In this section, we introduce the detailed implementation of our proposed Temporal Graph Auto-Encoders (TGAE), including ego-graph sampling, temporal graph encoding, and ego-graph decoding, and show how our TGAE generate a new temporal graph from observed temporal graphs.

4.4.1 Model Architecture

As illustrated in Figure 4.2, our proposed TGAE is composed of four parts: ego-graph sampling, temporal graph encoding, ego-graph decoding, and temporal graph assembling. Given a temporal graph $\tilde{G} = \{G^{t_1}, ..., G^T\}$, we first convert all snapshots into one temporal adjacency matrix $\mathbf{A}_{t=1:T} \in \{1, 0\}^{T\times n\times n}$. Then we sample temporal ego-graphs for model training. After that, we leverage temporal graph attention networks (TGAT) on neighbor-level temporal structure to obtain hidden variables for each temporal node. Then ego-graph decoding module generate ego-graph corresponding to each temporal node's hidden variables. Finally, we use the generated edge probabilities to assemble a temporal graph score matrix $\mathbf{S}_{t=1:T} \in \mathbb{R}^{T\times n\times n}$. Given a temporal graph score matrix, we sample edges to generate new temporal graphs.

Algorithm 1: Sampling k-Radius Temporal Ego-Graph

Function `NodeSampling` (*nodeset, threshold*)
    Nodes$\leftarrow \emptyset$; i, $u \leftarrow 0$;
    if *length(nodeset)*$\leq$ *threshold* **then**
        return *nodeset*;
    else
        foreach $i \in 1$: *threshold* **do**
            $u \leftarrow$ random.choice(nodeset);
            Nodes.insert(u);
        end
        return *Nodes*;
    end
Function `k-EgoGraph` ($\tilde{G}$, v^t, k, th)
    ego, nodeset $\leftarrow \emptyset$;
    if $k \neq 1$ **then**
        nodeset $\leftarrow$ `NodeSampling` ($\mathbf{N}(v^t)$, th) ;
        foreach $u^t \in$ *nodeset* **do**
            ego $\leftarrow$ `k-EgoGraph` ($\tilde{G}$, u^t, $k-1$) ;
            nodeset.insert(ego.nodes);
        end
        ego $\leftarrow \tilde{G}$.subgraph(nodeset);
        return *ego*;
    else
        nodeset $\leftarrow$ `NodeSampling` ($\mathbf{N}(v^t)$, th) ;
        ego $\leftarrow \tilde{G}$.subgraph(nodeset);
        return *ego*;
    end
Function `EgoGraphDataLoader` ($\tilde{G}$, $\mathbf{X}_{t=1:T}$, k)
    EgoGraphs, Nodefeatures $\leftarrow \emptyset$;
    foreach $i \in 1 : T$ **do**
        $\mathbf{X}^{(i)} \leftarrow \mathbf{X}_{t=1:T}((n*(i-1)+1 : n*i), :)$;
        ego, nodefeat $\leftarrow \emptyset$;
        foreach $v \in 1 : n$ **do**
            ego $\leftarrow$ `k-EgoGraph` ($\tilde{G}$, v^i, k) ;
            EgoGraphs.insert(ego);
            nodefeat $\leftarrow \mathbf{X}^{(i)}$(ego.nodes, :);
            Nodefeatures.insert(nodefeat);
        end
    end
    return *EgoGraphs, NodeFeatures*

4.4.2 Ego-Graph Sampling

To extract local temporal structure, we leverage ego-graph sampling for temporal graphs. Given the temporal graph adjacency matrix $\mathbf{A}_{t=1:T}$, we first separately load each snapshot's node features $\mathbf{X}^{(t)} \in \mathbb{R}^{n \times d_{in}}$ corresponding to its timestamp t. Then, algorithm 1 shows the procedure of sampling a k-radius ego-graph for each temporal node u^t. Specifically, we first choose the representative temporal node as the central node of each ego-graph. Then we recursively sample the neighbor nodes, where the new nodes are sampled from the neighbors of the early sampled nodes. To reduce the training time consumption of our proposed model, we truncate the number of neighbors of nodes whose degrees exceed the threshold to achieve the tradeoff between efficiency and effectiveness. Specifically, when we sample the neighbors of the nodes, we sample several times with replacement and get th nodes as the neighbors of the node in the ego-graph. The process finishes when all the neighbors having less than k shortest path length with node u are included in this ego-graph. Algorithm 1 shows the detailed procedure of preparing the ego-graphs and node features as input data for our learning-based model. Note that our proposed method set the node identity numbers as default node features.

Initial Temporal Node Sampling. To model a complete temporal graph structure, we propose a strategy to select representative temporal nodes. A naive approach is to sample all temporal nodes according to a uniform distribution, however, this strategy tends to learn to generate unimportant edges [69]. To focus on important edges to generate a high-quality temporal graph, we propose to use the probability distribution based on temporal node degree as the initial temporal node sampling strategy. The sampling strategy is formulated as follows:

$$P(u^t) = \frac{deg_{u^t}}{\sum_{v^t \in \tilde{V}} deg_{v^t}} \tag{4.2}$$

where deg_{u^t} denotes the degree of temporal node u^t, i.e., the temporal neighbors associated with u^t. Assuming that in each epoch we sample n_s temporal nodes as initial temporal nodes, we sample n_s ego-graphs as the input of our encoding process. The set of initial nodes is represented as $\tilde{V}_s$.

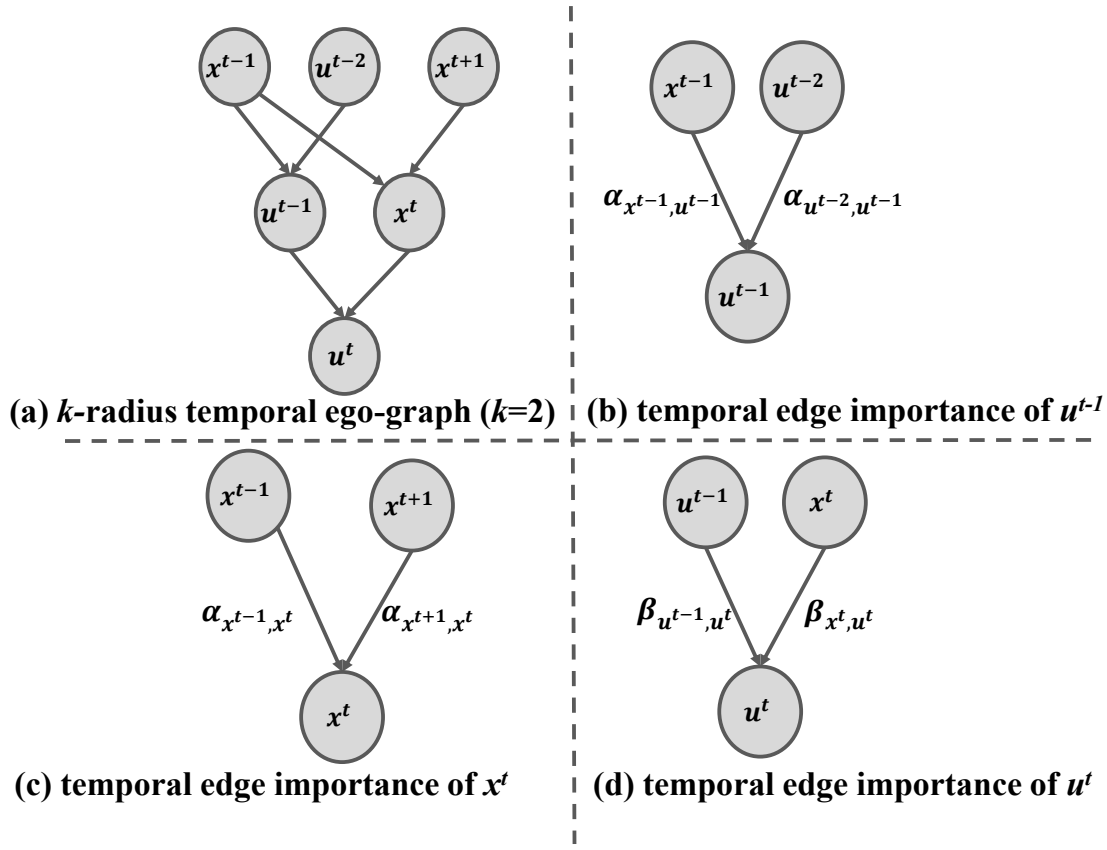


FIGURE 4.3: The illustration of the k-radius temporal ego-graph. The upper left part shows the ego-graph with the center temporal node u^t. The other three parts shows the edge importances calculated by k stacked temporal graph attention (TGAT) layers.

Unlike in random walk-based work, we reweight temporal nodes by their temporal degrees (i.e., the number of first-order temporal neighbors) to efficiently simulate high-quality temporal graphs. This re-weighting will allow our model to preferentially learn to generate neighbors of key temporal nodes. Besides, the neighbors of non-critical nodes contain a higher proportion of outlier points. Therefore, our initial node sampling strategy reduces the effect of outliers, resulting in efficient and effective model training with hardly sacrificing simulating quality.

4.4.3 Temporal Graph Attention Encoding

Given the temporal graph $\tilde{G}_{ego}(v^t) = (\tilde{V}_{ego}(v^t), \tilde{E}_{ego}(v^t))$ and temporal node features $\mathbf{X}_{ego} \in \mathbb{R}^{n_{ego} \times d_{in}}$, where n_{ego} denotes the node number of ego graph and d_{in} denotes the dimension of input features, we propose to employ temporal graph attention mechanism on our sampled ego graphs. In particular, we obtain the hidden variables of the center node u^t of the ego graph

through leveraging temporal attention mechanism to aggregate messages from graph structures and temporal neighbors, where d_{enc} denotes the dimension of hidden variables after encoding process. For each temporal ego graph, the message aggregating is formulated as follows:

$$\begin{aligned}\mathbf{h}_{u^t} =&\mathbf{TGAT}_{enc}(\mathbf{X}_{ego}|\tilde{V}_{ego}(u^t), \tilde{E}_{ego}(u^t)) \\ =&\text{Concat}(\text{TgaHead}_1, ..., \text{TgaHead}_{h_{tga}})\mathbf{W}_o\end{aligned} \tag{4.3}$$

where $\mathbf{h}_{u^t} \in \mathbb{R}^{1\times d_{att}}$ denotes one row of hidden variables of the temporal graph attention encoding layer, i.e., hidden variables on temporal node u^t, and $\mathbf{W}_o \in \mathbb{R}^{h_{tga}d_{enc}\times d_{att}}$ denotes the output projection matrix, h_{tga} denotes the number of heads, d_{att} is the dimension of attention vector, and each head of temporal graph attention layer $\text{TgaHead}_i \in \mathbb{R}^{1\times d_{enc}}$ is formulated as follows:

$$\text{TgaHead}_i = \sigma(\sum_{v^t\in\mathbf{N}(u^t)} \alpha^i_{u^t,v^t}\mathbf{h}_{u^t}) \tag{4.4}$$

where σ denotes the activation function and $\alpha^i_{u^t,v^t}$ denotes the importance of temporal edge (u^t, v^t) in i-th head, which is formulated as follows:

$$\alpha^i_{u^t,v^t} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}_i^T[\mathbf{h}_{u^t}||\mathbf{h}_{v^t}]))}{\sum_{k^t\in\mathbf{N}(v^t)}\exp(\text{LeakyReLU}(\mathbf{a}_i^T[\mathbf{h}_{k^t}||\mathbf{h}_{v^t}]))} \tag{4.5}$$

where $\mathbf{a}_i \in \mathbb{R}^{2d_{enc}}$ denotes the attention vector of the i-th attention head, and LeakyReLU denotes the non-linear activation function with a negative input slope $\alpha = 0.2$.

From the example in Figure 4.3, we can intuitively understand our temporal node coding process: (1) the input of the encoding process is the sampled ego-graph, which is shown in Figure 4.3 (a), where we assume that the value of k is 2; (2) the first TGAT layer calculates the importance α of second-order neighbors (such as x^{t-1} and u^{t-2} in Figure 4.3 (b)) to first-order neighbors (such as u^{t-1} in Figure 4.3 (b)); (3) the second TGAT layer calculates the importance β of first-order neighbors to the central node u^t; (4) in Figure 4.3 (d), the central node u^t outputs the representation

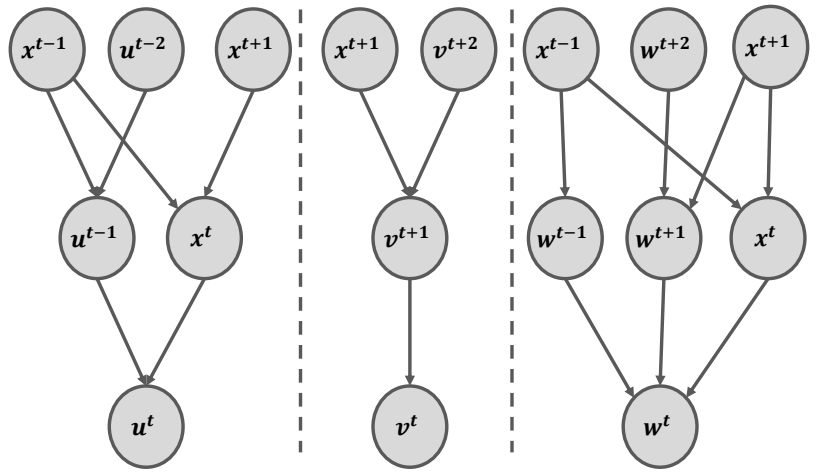
of this ego-graph. In the actual model training, we added self-loops to all temporal nodes to pass messages to themselves.

Parallel Ego-graph Training. To reduce the time consumption of the encoding process, we combine multiple ego-graphs for parallel node encoding to reduce computation steps from $O(nT)$ to $O(\frac{nT}{b})$, where b denotes the parallel number of temporal ego-graphs, i.e., batch size. For efficient training, we set the batch size as the size of initial sampled center node set with $b = |\tilde{V}_s| = n_s$. Therefore, the computation step is parallelized into $O(\frac{nT}{n_s})$. As shown in Figure 4.4, we put all the ego-graphs together and generate k-bipartite graphs by vertical splitting. These bipartite graphs represent a set of temporal ego-graph neighbors of order 1 to k. Specifically, we first use S_0 to represent the center node set of the temporal ego-graphs. Then, we use $S_1, ..., S_k$ to respectively accommodate the k-order neighbors of the center nodes of these temporal ego-graphs. After that, we index the source nodes of the bipartite graph in S_k and the target nodes of the bipartite graph in S_{k-1}. After getting the source and target, we get the k-bipartite computation graphs. We stack k TGAT layers to achieve message passing on the k-bipartite computation graph, and finally, get the representation of the central temporal node.

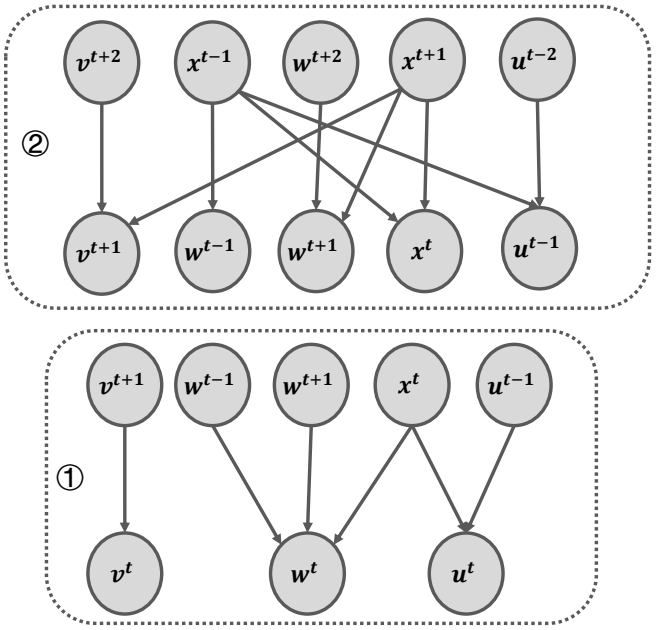
To further reduce the space consumption, we use a truncation mechanism to control space usage and ignore repeated nodes each time a new node is inserted into S_k. In Algorithm 1, to control the worst-case space requirement, we use th as the threshold. Once the total number of neighbors of a temporal node exceeds th, the algorithm converts from all neighbor sampling strategy to th-neighbor sampling strategy, and merge all the temporal ego-graphs into k-bipartite computation graphs for the sampled neighbor nodes instead of all the neighbor nodes.

4.4.4 Ego-Graph Decoding

To reconstruct local temporal structure distribution, we leverage ego-graph decoding process to infer the probablistic generative model for each temporal node. Given the hidden variables of temporal node $\mathbf{h}_{u^t}$, we first use two Multi-Layer Perceptrons (MLP) to infer the parameters μ and σ of the prior distribution $\mathcal{N}(\mu, \sigma^2)$. Then algorithm 2 shows the procedure of decoding the edge probabilities of a k-radius ego-graph for each temporal node u^t, where the $\mathbf{W}_{dec} \in \mathbb{R}^{d_{in} \times n}$ and $\mathbf{b}_{dec} \in \mathbb{R}^n$ are the output parameters for decoding process. When the categorical



(a) Merge all *k*-radius temporal ego-graphs (*k*=2)



(b) Convert ego-graphs into *k*-bipartite computation graph (*k*=2)

FIGURE 4.4: The illustration of the k-bipartite computation graphs. The upper part shows the initial k-radius temporal ego-graphs. The lower part shows the k-bipartite computation graphs, which are used for model training. In each bipartite computation graph, the results of the target nodes can be computed concurrently.

distribution of the $i-1$-order neighbors of u^t is generated, we generate i-order neighbors' edge probabilities. The process finishes when all the k-radius ego-graph's generative distributions are generated. Assuming that $\mathbf{H} \in \mathbb{R}^{nT \times d_{enc}}$ contains all hidden variables of all temporal nodes, we generate score matrix $\mathbf{S}_{t=1:T}$ by averaging all the edge probabilities generated by the ego-graphs.

Algorithm 2: Decoding k-Radius Temporal Ego-Graph

Function `k-EgoGraphDecoding` ($\tilde{G}_{ego}(u^t)$, $\mathbf{h}_{u^t}$, $\mathbf{X}_{ego}$, k)
  mu $\leftarrow$ MLP$_{\mu}(\mathbf{X}_{ego})$;
  sigma $\leftarrow$ MLP$_{\sigma^2}(\mathbf{X}_{ego})$;
  noise $\leftarrow$ random.normal($\mathbf{X}_{ego}$.shape);
  $\mathbf{Z} \leftarrow$ mu + sigma $*$ noise;
  return `EdgeProbability` ($\tilde{G}_{ego}(u^t)$, $\mathbf{h}_{u^t}$, $\mathbf{Z}$, k) ;
Function `EdgeProbability` ($\tilde{G}_{ego}(u^t)$, $\mathbf{h}_{u^t}$, $\mathbf{Z}$, k)
  p, $\mathbf{h}$, problist $\leftarrow \emptyset$;
  if *$k \neq 1$* **then**
    foreach $v^t \in \mathbf{N}(u^t)$ **do**
      $\mathbf{h} \leftarrow \mathbf{h}_{u^t} + \mathbf{Z}(v^t, :)$;
      p $\leftarrow$`EdgeProbability` ($\tilde{G}_{ego}(v^t)$, $\mathbf{h}$, $\mathbf{Z}$, $k-1$) ;
      problist.extend(p);
    end
    return *problist*;
  else
    $\mathbf{h} \leftarrow \mathbf{h}_{u^t} + \mathbf{Z}(u^t, :)$;
    p $\leftarrow$ softmax($\mathbf{h} \times \mathbf{W}_{dec} + \mathbf{b}_{dec}$);
    problist.insert(p);
    return *problist*;
  end

The decoding process has a space complexity of $O(n \times (T + n_s))$, where n_s denotes the number of initial temporal nodes.

4.4.5 Optimization Strategy

Batch Gradient Descending. We jointly optimize encoder and decoder's parameters by minimizing the variational lower bound as follows:

$$
\begin{aligned}
P(\mathbf{S}_{t=1:T}) &= \prod_{e_{i,j} \in \mathbf{A}_{t=1:T}} P(\mathbf{S}_{t=1:T})_{i,j} \\
\mathcal{L} &= -\frac{1}{NT} \sum_{t=1}^{T} \sum_{u=1}^{N} \mathbf{A}_{u^t} \log(P(\mathbf{S}_{t,u})) + \mathrm{KL}(q(\mathbf{Z}|\mathbf{X})||p(\mathbf{Z}))
\end{aligned}
\tag{4.6}
$$

where $P(\mathbf{S}_{t=1:T})$ denotes the generated score matrix from decoding module and $\mathrm{KL}(\cdot||\cdot)$ is the Kullback-Leibler divergence between two distributions.

Mini-batch Gradient Descending and Approximate Loss. In practice, it is more efficient to use a mini-batch gradient descending to update the model's parameters. Particularly, we optimize the model parameters through mini-batch data, i.e., randomly sampled ego-graphs and corresponding node features, achieve global parameter training, and can train a model with satisfactory generalization and robustness in less time. In addition, our KL-divergence calculation is still carried out on all nodes. Therefore, in our TGAE implementation, we update the parameters with an approximate loss function, which is formulated as follows:

$$\mathcal{L}_{tgae} = -\frac{1}{n_s} \sum_{u^t \in \tilde{V}_s} \mathbf{A}_{u^t} \log(P(\mathbf{S}_{t,u})) + \mathrm{KL}(q(\mathbf{Z}|\mathbf{X})||p(\mathbf{Z})) \tag{4.7}$$

where $\tilde{V}_s$ denotes the set of sampled initial temporal nodes and n_s denotes the size of $\tilde{V}_s$. By adjusting the value of n_s, we can achieve the trade-off between generating high-quality temporal graphs and fast model training.

4.4.6 Model Variants

Ego-Graph Sampling Variant. Our model can be generalized to random walk-based variants, only by reducing the neighbor threshold th to less than 2 in Algorithm 1. In this case, the ego-graph obtained by our temporal ego-graph sampling strategy is a chain structure, that is, a temporal random walk on the temporal graph. In this variant, we fix the whole architecture of TGAE so that it is consistent with the full version, except for the threshold of ego-graph neighbor sampling.

Initial Node Sampling Variant. In addition to ego-graph sampling strategy, our initial node sampling strategy can also be modified to a uniform distribution based sampling strategy. Under this node sampling strategy, our model learns to reconstruct every edge in the temporal graph without bias. In this variant, only the initial node sampling strategy is different from TGAE, and the rest are consistent with the proposed version.

Non-probablistic Variant. We also propose a non-probabilistic variant derived from full TGAE, in which the ego-graph sampling and temporal graph attention encoding are consistent with the full version. We modify the decoder of the full TGAE model to non-probabilistic version, which is formulated as follows:

TABLE 4.2: Statistics of the network data sets.

Network	#Nodes	#Edges	#Timestamps
DBLP	1,909	8,237	15
EMAIL	986	332,334	805
MSG	1,899	20,296	195
BITCOIN-A	3,783	24,186	1,902
BITCOIN-O	5,881	35,592	1,904
MATH	24,818	506,550	79
UBUNTU	159,316	964,437	88

$$\mathbf{Z} \leftarrow \text{MLP}_{\mu}(\mathbf{X}_{ego}) \tag{4.8}$$

where $\mathbf{X}_{ego}$ denotes the input features of the sampled central temporal nodes. Then, the calculation of the approximate loss is modified to fit this variant, which is formulated as follows:

$$\mathcal{L}_{tgae} = -\frac{1}{n_s} \sum_{u^t \in \tilde{V}_s} \mathbf{A}_{u^t} \log(P(\mathbf{S}_{t,u})) \tag{4.9}$$

After model training and parameter optimization, the temporal graph generation process of the non-probabilistic variant is consistent with the full TGAE.

4.4.7 Temporal Graph Generation

After the training process, we first generate all the ego-graphs to assemble the score matrix S. Then, we take score matrix as the parameters of the categorical distribution of each temporal edge with $p(t,u,v) = \frac{\mathbf{S}_{t,u,v}}{\sum_{i \in \mathbf{N}(u^t)} \mathbf{S}_{t,u,i}}$. Then we sample the corresponding temporal edges for each temporal node without replacement, which is formulated as $\mathbf{A}'_{u^t} \sim Cat(\prod_{i \in \mathbf{N}(u^t)} p(t,u,i))$, where Cat denotes categorical distribution. The generation process finishes when the generated temporal graph's edge amount meets the one observed graph has.

TABLE 4.3: Graph statistics for measuring network properties.

Metric Name	**Computation**	**Description**
Mean Degree	$\mathbb{E}[d(v)]$	Mean degree of nodes.
Claw Count	$\sum_{v \in V} \binom{d(v)}{3}$	# Claws of the graph.
Wedge Count	$\sum_{v \in V} \binom{d(v)}{2}$	# Wedges of the graph.
Triangle Count	$\frac{trace(A^3)}{6}$	# Triangles of the graph.
LCC	$max_{f \in F} \|\|f\|\|$	Size of the largest connected component.
PLE	$1 + n(\sum_{v \in V} log(\frac{d(u)}{d_{min}}))^{-1}$	Exponent of power-law distribution.
N-Component	$\|F\|$	# connected components.

TABLE 4.4: Median score $f_{med}(\cdot)$ comparison with seven metrics across seven temporal networks. (Smaller metric values indicate better performance)

Dataset	**Metric**	**TGAE**	**TIGGER**	**DYMOND**	**TGGAN**	**TagGen**	**NetGAN**	**E-R**	**B-A**	**VGAE**	**Graphite**	**SBMGNN**
DBLP	Mean Degree	2.41E-3	3.54E-3	2.98E-3	3.25E-3	**7.46E-4**	4.16E-3	5.52E-3	1.23E-1	1.79E-3	1.79E-3	1.79E-3
	LCC	**2.61E-3**	2.75E-3	2.71E-3	2.77E-3	2.78E-3	3.35E-1	7.27E-1	9.11E-2	5.11E-1	5.40E-1	4.62E-1
	Wedge Count	**4.15E-3**	3.08E-2	2.31E-2	5.38E-1	7.14E-1	5.05E-1	5.07E-1	3.74E-1	1.81E+0	2.15E+0	2.39E+0
	Claw Count	**7.29E-3**	2.64E-2	1.35E-2	2.98E+0	3.02E+0	9.27E-1	8.78E-1	4.52E+0	8.78E+0	1.21E+1	1.42E+1
	Triangle Count	**4.79E-3**	7.85E-2	3.77E-2	5.33E-1	5.44E-1	8.83E-1	9.94E-1	8.24E-1	9.27E+0	9.21E+0	8.66E+0
	PLE	**1.73E-3**	3.34E-2	9.15E-3	1.78E-1	1.79E-1	2.24E-1	1.65E-1	8.45E-2	4.01E-1	4.65E-1	4.25E-1
	N-Components	**3.05E-3**	3.07E-3	3.11E-3	3.39E-3	3.51E-3	2.13E-1	8.36E-1	5.06E-2	5.07E-1	5.49E-1	4.83E-1
MATH	Mean Degree	**2.69E-2**	1.05E-1	OOM	OOM	OOM	2.13E-1	2.29E-1	3.24E-2	2.39E-1	2.44E-1	3.72E-2
	LCC	8.72E-2	9.31E-2	OOM	OOM	OOM	**2.99E-2**	8.83E-1	1.24E-1	5.56E-1	5.30E-1	3.73E-1
	Wedge Count	**1.05E-1**	2.37E-1	OOM	OOM	OOM	2.42E-1	9.27E-1	3.15E-1	6.87E-1	7.50E-1	1.77E+0
	Claw Count	**2.59E-1**	3.75E-1	OOM	OOM	OOM	4.96E-1	9.99E-1	4.86E-1	2.95E+0	3.43E+0	8.13E+0
	Triangle Count	**9.79E-2**	8.78E-1	OOM	OOM	OOM	2.34E+0	1.00E+0	5.84E-1	1.74E+0	1.66E+0	2.24E+0
	PLE	**2.41E-2**	9.36E-1	OOM	OOM	OOM	1.11E+0	2.35E-1	7.81E-2	5.74E-1	5.40E-1	2.49E-1
	N-Components	**3.15E-2**	4.66E-2	OOM	OOM	OOM	3.50E-2	1.00E+0	1.35E-1	5.90E-1	5.61E-1	3.96E-1
UBUNTU	Mean Degree	**9.73E-2**	OOM	OOM	OOM	OOM	OOM	2.32E+1	5.29E-1	OOM	OOM	OOM
	LCC	**1.32E-1**	OOM	OOM	OOM	OOM	OOM	3.71E+0	2.98E+0	OOM	OOM	OOM
	Wedge Count	**3.16E-1**	OOM	OOM	OOM	OOM	OOM	1.45E+1	9.76E-1	OOM	OOM	OOM
	Claw Count	5.60E-1	OOM	OOM	OOM	OOM	OOM	**3.01E-1**	9.96E-1	OOM	OOM	OOM
	Triangle Count	**1.21E-1**	OOM	OOM	OOM	OOM	OOM	5.05E-1	1.00E+0	OOM	OOM	OOM
	PLE	**8.52E-2**	OOM	OOM	OOM	OOM	OOM	7.33E-1	5.31E-1	OOM	OOM	OOM
	N-Components	**2.64E-2**	OOM	OOM	OOM	OOM	OOM	1.00E+0	8.55E-1	OOM	OOM	OOM

4.5 Experiments

In this section, we describe the extensive experiments for evaluating the effectiveness of our proposed method. We first describe the experiment setup. Then, present the experimental results of temporal graph auto-encoder compared with other baselines, which is the main task of this chapter.

TABLE 4.5: Average score $f_{avg}(\cdot)$ comparison with seven metrics across nine temporal networks. (Smaller metric values indicate better performance)

Dataset	Metric	TGAE	TIGGER	DYMOND	TGGAN	TagGen	NetGAN	E-R	B-A	VGAE	Graphite	SBMGNN
DBLP	Mean Degree	2.33E-3	3.41E-3	2.78E-3	3.68E-3	**1.31E-3**	3.83E-3	2.12E-2	1.08E-1	8.93E-3	8.76E-3	8.92E-3
	LCC	**4.81E-3**	4.37E-2	8.76E-3	7.83E-2	8.62E-2	6.57E-1	6.23E-1	1.93E-1	5.09E-1	5.04E-1	4.08E-1
	Wedge Count	**7.46E-3**	6.81E-1	3.06E-2	7.25E-1	9.88E-1	5.63E-1	4.76E-1	3.50E-1	1.92E+0	2.11E+0	2.36E+0
	Claw Count	**1.14E-2**	2.98E-1	5.23E-2	3.22E+0	5.21E+0	1.26E+0	8.55E-1	4.63E+0	1.11E+1	1.24E+1	1.69E+1
	Triangle Count	**7.38E-3**	3.84E-1	2.95E-2	5.24E-1	6.86E-1	7.51E-1	9.92E-1	8.16E-1	8.99E+0	9.61E+0	9.20E+0
	PLE	**2.71E-3**	1.75E-1	3.64E-2	2.53E-1	2.50E-1	2.24E-1	1.83E-1	8.46E-2	3.98E-1	4.32E-1	4.00E-1
	N-Components	**3.07E-3**	3.77E-2	9.47E-3	4.20E-2	4.64E-2	2.29E-1	6.19E-1	1.02E-1	5.95E+0	6.27E+0	5.93E+0
MATH	Mean Degree	**2.64E-2**	6.39E-2	OOM	OOM	OOM	1.97E-1	2.08E-1	3.81E-2	2.05E-1	2.07E-1	3.97E-2
	LCC	8.08E-2	7.01E-1	OOM	OOM	OOM	**3.15E-2**	1.49E+0	2.41E-1	5.41E-1	5.16E-1	3.68E-1
	Wedge Count	**1.24E-1**	1.39E-1	OOM	OOM	OOM	2.57E-1	9.30E-1	3.29E-1	8.07E-1	8.59E-1	1.86E+0
	Claw Count	**2.74E-1**	2.98E-1	OOM	OOM	OOM	4.97E-1	9.99E-1	5.00E-1	3.45E+0	3.89E+0	8.50E+0
	Triangle Count	**1.20E-1**	4.18E-1	OOM	OOM	OOM	2.20E+0	1.00E+0	6.35E-1	1.88E+0	1.81E+0	2.25E+0
	PLE	**2.43E-2**	8.31E-2	OOM	OOM	OOM	9.66E-1	2.98E-1	1.09E-1	5.39E-1	5.11E-1	2.34E-1
	N-Components	**9.39E-2**	1.12E-1	OOM	OOM	OOM	1.34E-1	9.45E-1	3.19E-1	4.73E+0	4.52E+0	3.33E+0
UBUNTU	Mean Degree	**7.41E-2**	OOM	OOM	OOM	OOM	OOM	2.03E+1	8.89E-1	OOM	OOM	OOM
	LCC	**2.10E-1**	OOM	OOM	OOM	OOM	OOM	6.94E+3	6.02E+3	OOM	OOM	OOM
	Wedge Count	**3.06E-1**	OOM	OOM	OOM	OOM	OOM	5.07E+4	2.34E+4	OOM	OOM	OOM
	Claw Count	**5.14E-1**	OOM	OOM	OOM	OOM	OOM	3.10E+5	3.29E+6	OOM	OOM	OOM
	Triangle Count	**1.01E-1**	OOM	OOM	OOM	OOM	OOM	4.73E-1	7.95E-1	OOM	OOM	OOM
	PLE	**1.29E-1**	OOM	OOM	OOM	OOM	OOM	6.51E-1	5.37E-1	OOM	OOM	OOM
	N-Components	**8.40E-2**	OOM	OOM	OOM	OOM	OOM	9.97E-1	8.25E-1	OOM	OOM	OOM

After that, the efficiency and scalability of the proposed method are tested. Finally, we report the ablation study and parameter sensitivity experiments.

TABLE 4.6: Maximum mean discrepancy of instance counts of all 2- and 3-node, 3-edge $\boldsymbol{\delta}$-temporal motifs between raw and generated temporal networks ($\boldsymbol{\sigma}$ refers to the sigma value for Gaussian kernel)

Dataset	TGAE	TIGGER	DYMOND	TGGAN	TagGen	NetGAN	E-R	B-A	VGAE	Graphite	SBMGNN
DBLP	**2.65E-5**	9.68E-4	1.25E-4	2.08E-2	2.31E-2	2.21E-1	6.43E-2	1.08E+0	1.34E+0	1.95E+0	1.99E+0
MSG	**2.27E-5**	2.12E-4	3.77E-5	9.81E-3	1.09E-2	1.85E-2	1.83E-2	1.17E+0	1.98E+0	1.99E+0	1.65E+0
BITCOIN-A	**1.12E-6**	2.76E-5	OOM	OOM	OOM	OOM	1.90E+0	2.00E+0	3.88E-1	5.39E-1	1.08E-1
BITCOIN-O	**5.49E-6**	3.06E-5	OOM	OOM	OOM	OOM	1.80E+0	2.00E+0	1.82E+0	1.98E+0	5.22E-1
EMAIL	**2.12E-2**	7.65E-2	3.27E-2	OOM	OOM	9.78E-2	9.74E-1	1.95E+0	1.95E+0	1.07E+0	1.74E+0
MATH	**7.86E-4**	2.14E-3	OOM	OOM	OOM	5.11E-3	6.59E-3	2.74E-3	2.00E+0	1.89E+0	1.94E+0
UBUNTU	**1.27E-3**	OOM	OOM	OOM	OOM	OOM	1.52E+0	2.00E+0	OOM	OOM	OOM

4.5.1 Experiment Settings

We introduce the experimental datasets, comparison methods, metrics, and parameter settings in this subsection. Please note that this chapter is focusing on graph simulation of *temporal graphs*. Therefore, the experimental settings including datasets, compared methods, and evaluating metrics are all different from the previous chapters.

Datasets. We evaluate our temporal graph auto-encoder on seven real temporal networks. Specifically, DBLP [118] is a citation network that contains bibliographic information of the publications in IEEE Visualization Conference from 1990 to 2015; MSG [119] and EMAIL [109] are two communication networks, where a single edge represents a message/email sent from one person

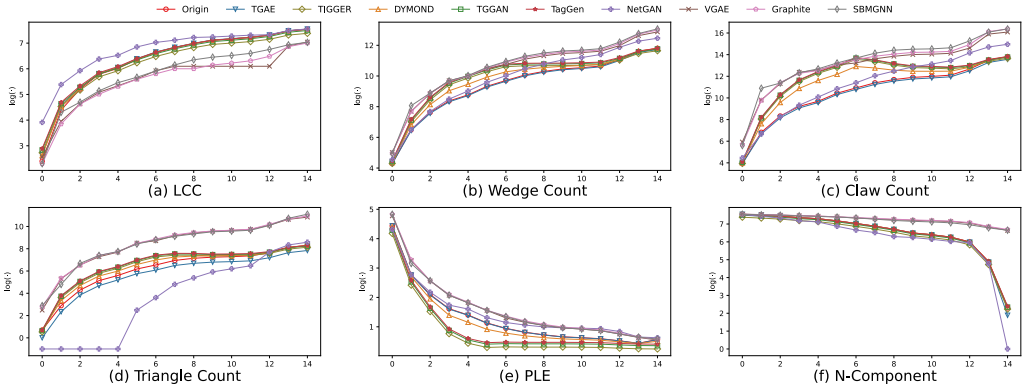


FIGURE 4.5: The comparison results on the seven evaluation metrics across 15 timestamps in DBLP data set. Best viewed in color. The algorithm better fitting the curve of the original graph (colored in blue) is better.

to another at a certain timestamps; BITCOIN-A and BITCOIN-O [120, 121] are two who-trusts-whom networks where people trade with bitcoins on Bitcoin Alpha and OTC platforms; MATH and UBUNTU [109] are temporal networks of interactions on the stack exchange web sites Math Overflow and Ask Ubuntu. The statistics of datasets are summarized in Table 4.2.

Compared methods. We compare TGAE with two state-of-the-art temporal graph generative models (TIGGER [122] and DYMOND [123]), three GAN-based graph generative models (TG-GAN [117], TagGen [40], and NetGAN [24]), two simple model-based generative models (E-R [124] and B-A [30]), and three auto-encoder-based generative models (VGAE [17], Graphite [38], and SBMGNN [39]). Note that NetGAN, simple model-based, and autoencoder-based models are not designed for temporal graph generation. To generate temporal networks, we separately generate snapshots of the temporal graph at each timestamp.

Evaluation metrics. We collected several popular evaluating metrics to measure the difference between the original temporal graph and the generated graph. The graph statistics for measuring graph properties are summarized in Table 4.3. As all of these metrics are designed for static graphs, we follow the practice of TagGen [40], who generalized the aforementioned metrics to the dynamic setting by calculating mean and median value of the metrics among all timestamps. Specifically, given a metric $f_m(\cdot)$, the real graph $\widetilde{G}$, and the synthetic one $\widetilde{G'}$, we construct a sequence of snapshots $\widetilde{S}^t$ $(\widetilde{S'}^t)$, $t = 1, \ldots, T$, of $\widetilde{G}$ $(\widetilde{G'})$ by aggregating edges from the initial timestamp to the current timestamp t. Then, we measure the average/median difference (in percentage) of the

given metric $f_m(\cdot)$ between two graphs as follows:

$$
\begin{aligned}
f_{avg}(\widetilde{G},\ \widetilde{G'},\ f_m) =& Mean_{t=1:T}(|\frac{f_m(\widetilde{S}^t) - f_m(\widetilde{S'}^t)}{f_m(\widetilde{S}^t)}|) \\
f_{med}(\widetilde{G},\ \widetilde{G'},\ f_m) =& Median_{t=1:T}(|\frac{f_m(\widetilde{S}^t) - f_m(\widetilde{S'}^t)}{f_m(\widetilde{S}^t)}|)
\end{aligned}
\tag{4.10}
$$

Parameter settings. As to baseline methods, we use the best parameter settings given by the original authors. Our proposed TGAE and evaluating metrics are implemented through Python-3.7, PyTorch-1.8, and CUDA-11.1 in our experiments. The experiments are operated on a machine with Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz, 62 GB RAM, and NVIDIA Tesla V100 with 32 GB memory. We use one CPU core and one GPU for every algorithm.

4.5.2 Temporal Graph Generation

We compare our proposed TGAE with ten baseline models across seven temporal graph datasets. For the static methods, we apply them to generate one static graph at each timestamp and construct a series of graph snapshots by aggregating all static graphs. The results of seven evaluating metrics in the form of $f_{avg}(\cdot)$ and $f_{med}(\cdot)$ are shown in Tables 4.5 and 4.4.

Evaluation with graph statistics As shown in Tables 4.5 and 4.4, TGAE outperforms all the baseline methods in at least six of seven evaluating metrics. As to the DBLP dataset, the state-of-the-art baseline DYMOND achieves the second-best performance. Besides, Taggen achieves the best performance on *mean degree* measurement. As to MATH dataset, TIGGER achieves the second-best performance. According to the seventh and eighth columns, simple model-based generative methods (i.e., E-R and B-A) has the worst generative performance on temporal graph generation. According to the sixth column and the last three columns, we can see that static graph generative methods (i.e., NetGAN, VGAE, Graphite, and SBMGNN) are consistently worse than temporal graph generative methods. TGAE significantly outperforms TIGGER, DYMOND, TGGAN, and TagGen with all metrics except a slightly worse with Mean Degree, which shows that TGAE is better at capturing most graph properties. TGAE also outperforms other methods in the other datasets (e.g., MSG dataset). Due to space limits, we put the representative results in this manuscript. Please note that most of the learning-based methods cannot simulate large temporal

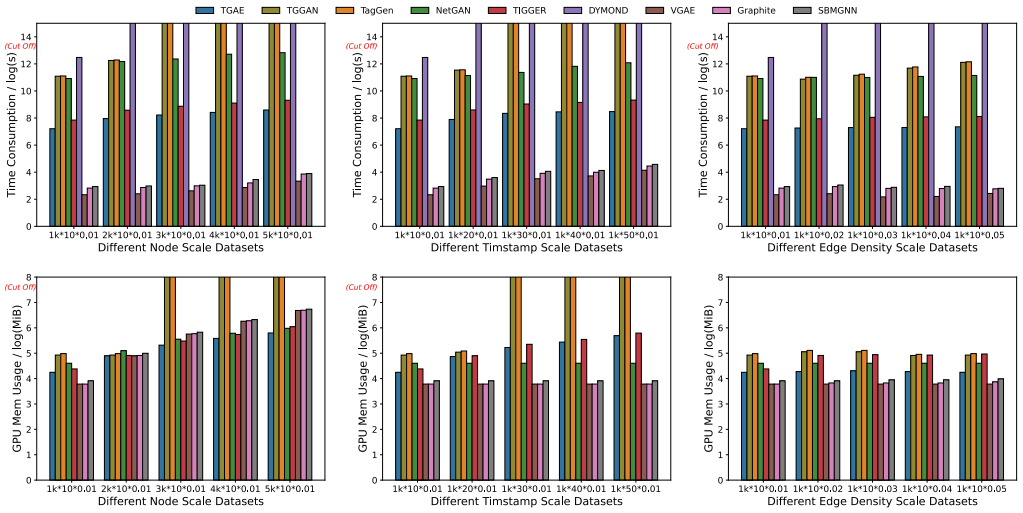


FIGURE 4.6: The comparison results on the time consumption and GPU memory usage in data sets designed for scalability test. The x-axis label implies the complexity of input temporal graphs in the form of Number of Nodes $*$ Timestamps $*$ Edge Density.

graphs (e.g., the UBUNTU dataset, containing about 14 million temporal nodes) due to their high requirements for GPU memory usage. Our proposed TGAE can simulate these large temporal graphs with affordable time consumption, which will be introduced in Section 4.5.5.

TABLE 4.7: Results of ablation study on TGAE and its variants. (Smaller metric values indicate better performance)

Dataset	Metric	TGAE	TGAE-g	TGAE-n	TGAE-p
MSG	Degree	**1.61E-2**	3.66E-2	1.73E-2	1.85E-2
	Motif	**2.27E-5**	8.14E-5	4.67E-5	4.93E-5
BITCOIN-A	Degree	**5.18E-3**	1.27E-2	7.33E-3	7.91E-3
	Motif	**1.12E-6**	4.33E-6	2.98E-6	2.35E-6
BITCOIN-O	Degree	**1.11E-2**	2.33E-2	1.65E-2	1.73E-2
	Motif	**5.49E-6**	2.10E-5	1.09E-5	1.13E-5

4.5.3 Temporal Attribute Preservation

We further study the generative performance from detailed comparison based on preserving temporal attributes, such as temporal motifs and temporal tendency.

Temporal motifs are recurring subgraph patterns over time in a temporal graph. [109] By evaluating our model using temporal motifs, we can assess how well our model captures and reproduces these

fundamental patterns in the generated graphs. Temporal tendency refers to the propensity of a graph's structure to evolve over time [40]. By evaluating our model using temporal tendency, we can assess how well our model captures and reproduces these dynamic changes in the generated graphs.

If our model can accurately generate graphs with similar temporal motif distributions as the original graph, it indicates that our model has successfully learned the underlying structural and temporal patterns in the data. And if our model can accurately generate graphs with similar temporal tendencies as the original graph, it indicates that our model has successfully learned the underlying dynamic behaviors in the data.

Temporal Motif Preservation. To evaluate the capacity of preserving temporal pattern information in the observed data, we also count the instances of all 2- and 3-node, 3-edge temporal motifs [109] and calculate the motif distributions *maximum mean discrepancy* [21] between the generated graphs and raw temporal graphs. The results on motif distribution are shown in Table 4.6.

According to the first column of Table 4.6, TGAE achieves the best performance in preserving the motif distribution in simulated temporal networks, which demonstrates its ability to capture both temporal and topological information. According to the triangle count row of Tables 4.4 and 4.5, we also find that the result of preserving motif distribution shows a similar trend to the triangle count. The results indicates that our proposed TGAE can simulate temporal graphs with similar motif distribution compared with observed graphs.

Temporal Tendency Visualization. To visualize the temporal tendency and show the similarity of simulated graphs, we experiment with the DBLP dataset and measure the statistics of observed graphs and generated graphs from all algorithms in each timestamp. By putting results together with the original graph, we can explore more information on the variation tendency of different methods on 15 timestamps in the DBLP dataset. The experimental results are reported in Figure 4.5, where the X-axis represents the timestamp, and the Y-axis represents the value of a metric. According to Figurea 4.5 (a) and (f), most of all the methods can have similar number of connected components compared to observed graphs. According to Figures 4.5 (b) and (c), TGAE (colored in blue) constantly performs better than the baseline methods as it better fits the triangle and claw count variation trends of the original graph (colored in red). Significantly worse performance of

simple model-based algorithms (e.g., E-R) in motif metrics (e.g., Triangle Count) proves their extremely weak expressive power. Our model has surpassed TIGGER, DYMOND, TGGAN, and TagGen in almost all measurements. The results demonstrate that TGAE is the best learning-based temporal graph generative model in terms of generative quality.

4.5.4 Ablation Study.

To validate the effectiveness of each component and our proposed sampling strategy, we report the ablation study results in Table 4.7. TGAE-g denotes the variant that the ego-graph sampling strategy is blocked. TGAE-n denotes the node sampling strategy is changed to uniform sampling. TGAE-p denotes the non-probabilistic variant. According to the first two columns of Table 4.7, we can see that if we use the random walk instead of ego-graphs to model the temporal graph, the generative performance degrades significantly. The results on the other two variants show similar observations. Therefore, the results on three datasets and two measurements demonstrate that all the included components are effective.

4.5.5 Scalability and efficiency

We also evaluated the scalability and efficiency of our model and the baseline methods. The first row of Figure 4.6 reports the time consumption of inferring a new graph, whose independent variables are the number of nodes, timestamps, and edge density respectively, while the second row reports the peak memory usage. Note that the B-A and E-R methods are not compared in GPU memory usage, because they are not implemented with the deep learning-based approach.

Simple model-based graph generators (B-A and E-R) have the highest efficiency for generating large temporal networks, incurring minor extra space costs and taking little time. All the learning-based methods (including our proposed TGAE) are more time-consuming than simple model-based methods.

As for temporal graph generative models, TGAE achieved much better results than DYMOND, TGGAN, and TagGen in terms of time consumption and memory usage. As can be seen in Table 4.5, 4.4, and Figure 4.6, GAN-based methods, including TGGAN, TagGen, and NetGAN, cannot run through most of the datasets due to its high time and memory consumption. As the number

of nodes and timestamps increase, TGAE has a linear increase of time consumption and memory usage. Compared with other learning-based temporal graph generative methods, TGAE is the best model choice for the efficient and effective temporal graph generation. Compared with all the baseline methods, TGAE can achieve a good trade-off between simulating quality and efficiency.

4.6 Conclusion

Temporal graph simulation can help to mimic real-life graphs in many applications, including biology, information technology, and social science. However, most of the graph simulation works focus on static graph simulation, ignoring the temporal evolving property of real-life graphs. In this chapter, we proposed temporal graph autoencoders (TGAE) to simulate real-life graphs and reproduce the temporal and structural properties from observed graph data. Besides, existing learning-based approaches are limited by their high GPU memory usage. Therefore, we propose initial node sampling and ego-graph sampling strategies to achieve efficient model training. Extensive experiment results on simulating quality and model efficiency show that our proposed TGAE achieves the best generative performance compared with other learning-based baselines. TGAE also achieves a good trade-off between quality and efficiency. In the future, we aim to scale the learning-based approaches to simulate large graphs with billion nodes.

Part II

Efficient Graph Generation: Applications

Chapter 5

Credit Card Fraud Detection

5.1 Chapter Overview

In this chapter, we show how synthesized graphs (such as temporal transaction graph) help downstream tasks, such as credit card fraud detection. This chapter has been published in [125]. In section 5.2, we give the background and related works on graph data-driven financial fraud detection. Sections 5.3 and 5.4 give the proposed solutions for graph-based fraud detection tasks. Section 5.5 shows the performance comparison between ours and baselines. Section 5.6 concludes this chapter.

5.2 Background and Related Works

5.2.1 Background

The great losses caused by financial fraud have attracted continuous attention from academia, industry, and regulatory agencies. Ensuring the security of financial transactions is crucial for protecting the privacy and assets of customers, preventing fraud and identity theft, maintaining trust and confidence in the financial system, and complying with the relevant laws and regulations. However, fraudulent behaviors against online payments, such as illegal card swiping, have caused

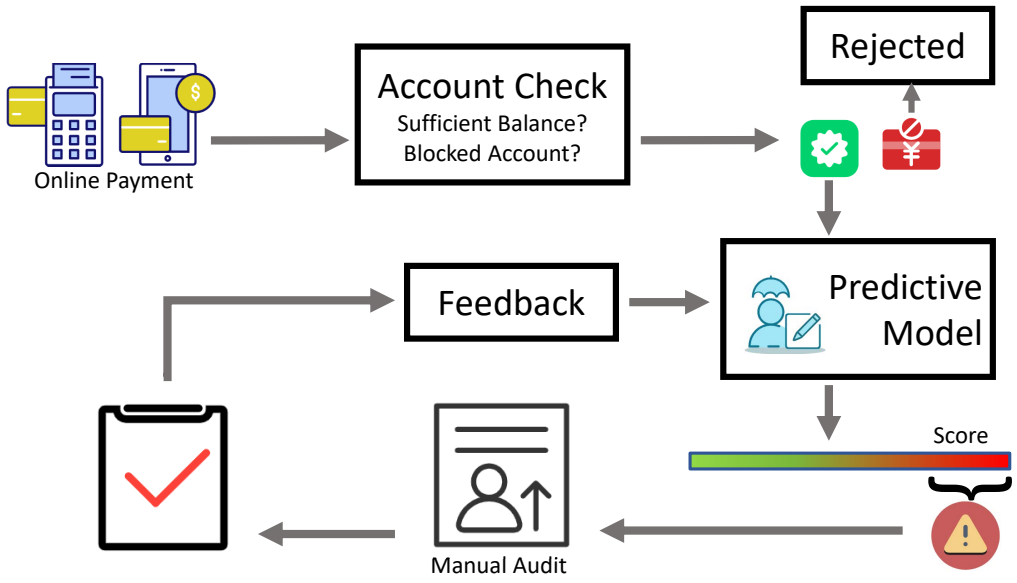


FIGURE 5.1: The framework of credit card fraud detection. The card issuer assesses each transaction with an online predictive model once it has passed account checking.

property losses to online payment users [126]. An effective financial fraud detection method can reduce the operating costs of service providers and protect the property of bank users.

An important line of research in financial fraud detection is credit card fraud detection, where credit card fraud is a general term for the unauthorized use of funds in a transaction, typically by means of a credit or debit card [126]. Figure 5.1 shows a typical fraud detection framework deployed in the commercial system [127]. A direct way to detect fraud is to match each transaction according to specific rules such as card blacklists and budget checking. However, criminals will also obtain knowledge of vulnerabilities from the response of the pre-designed rule system, thus invalidating the original system. To solve the invalidation problem, the predictive model is designed to automatically detect fraud patterns and produces a fraud risk score. Domain experts then can thereby focus on high-risk transactions.

In the literature, many existing predictive models have been extensively studied to deal with fraud transactions (e.g., [128–130]), which can be classified into two categories: (1) *Rule-based methods* directly generate sophisticated rules by domain experts to identify suspicious transactions. For instance, authors in [131] proposed an association rule method for mining frequent fraud rules; (2) *Machine learning-based methods* learn static models by exploring large amounts of historical data. For example, authors in [132] extracted features based on neural networks and built supervised

classifiers for detecting fraudulent transactions. Authors in [129] advanced the usage of automatic feature engineering in a convolutional neural network (CNN). Recently, graph machine learning-based methods have been proposed [133, 134] where the transactions are modeled as a graph, and the advanced graph embedding techniques are deployed.

The state-of-the-art fraud detection techniques [104, 133–136] can well capture the temporal or graph-based patterns of the transactions and significantly advance the performance of credit card fraud detection. However, they have at least one of the following main limitations: (1) ignoring unlabeled data containing rich fraud pattern information; (2) ignoring the importance of categorical attributes, which are ubiquitous in the real production environment; (3) requiring too much time on feature engineering, especially for temporal and categorical features.

In our preliminary work [125], we proposed a gated temporal attention network to address the above challenges and gained competitive results. Particularly, to capture the relationships among the credit card transactions associated with temporal information, we leverage a temporal transaction graph to model the time-relevant patterns. Besides, labeling the transactions is time-consuming and cost-expensive. Only a tiny proportion (much less than 10%) of transactions are labeled in billions of real-life transactions, which contain many fraud patterns that have not been detected. Therefore, it is crucial to exploit the natural features from unlabeled data. To mitigate the issue of insufficient utilization, risk embedding is introduced to unify the propagation of feature and label information and fully exploit the risk information. Additionally, categorical attributes are ubiquitous and useful in real applications. In response to this, we devise an attribute learning layer for preprocessing the transaction attributes.

However, an increasing number of criminals are organized like enterprises, which can be far-reaching and move quickly from place to place, to conduct conspiracy frauds to covet money from innocent consumers [137]. To fight against human brain-armed criminal behavior, existing graph neural-based methods still face significant challenges in capturing these complicated fraud patterns. Therefore, in this chapter, we substantially improved our previous work by proposing a risk-aware graph network to represent the high-order adjacent fraud patterns via a cross-attentional mechanism on multi-hop neighbors. In practice, we aggregate degree and risk information from multi-hop neighbor transactions, which is then processed by a convolutional embedding layer and a structural attention layer, which can extract the local risk information and make our model

aware of higher-order risk structures. The substantial contributions of our work are summarized as follows:

- We model credit card behaviors as a temporal transaction graph and formulate a credit card fraud detection problem as a semi-supervised node classification task.
- We present a novel attribute-driven temporal graph neural network for credit card fraud detection. Specifically, we propose a gated temporal attention network to extract temporal and attribute information.
- We make our network aware of high-order adjacent risk patterns via a risk-aware representation learning layer, which gathers degree and risk information from multi-hop neighborhoods and learns local risk structure representations.
- Extensive experiments conducted on three datasets show the superiority of our proposed RGTAN on fraud detection. Semi-supervised experiment results show that, when leveraging rich information from the abundance of unlabeled data and a bit of labeled data, our proposed method detects more fraud transactions than baselines. The real-world case studies demonstrate our proposed method's effectiveness in detecting real-world fraud patterns.

A preliminary version of this manuscript appeared in [125]. To further capture the high-order adjacent fraud patterns, this journal version proposed a risk-aware gated temporal attention network in Section 4 (new section) to enhance the capacity of the existing graph neural model. In the preliminary submission [125], historical fraud labels and attribute features are concatenated directly as encodings for downstream tasks. While in this extension, we proposed a risk-aware graph network to represent the high-order adjacent fraud patterns via a cross-attentional mechanism on multi-hop neighbors, which could overcome the 1-WL test capacity limitations of the existing graph neural model [138]. We thoroughly evaluated the new proposed substantial improvement approach, compared with the preliminary work and the state-of-the-art baselines in Section 5 (updated section). The experimental results prove the superior performance of our new contribution in detecting complicated inter-connected fraud patterns. In addition, we added empirical studies on real-world application scenarios after system deployment and reported our knowledge discovery in Section 6 (new section).

The rest of the chapter is organized as follows. In Section 5.2.2, we conduct a survey on the previous works regarding credit card fraud detection, graph-based methods, and graph structure learning. In Section 5.3, we present the Graph Temporal Graph Attention (GTGA) mechanism designed for extracting temporal fraud patterns as well as the attribute embedding layer. In Section 5.4, we detailedly introduce risk embedding and neighbor risk-aware embedding, as well as how they equip the network with awareness of risk structures. Comprehensive experimental results for our proposed method are presented in Section 5.5. Section 5.5.6 studies two typical risk patterns and validates the risk-aware capacity of our model. Section 5.6 concludes the chapter.

5.2.2 Related Works

5.2.2.1 Credit Card Fraud Detection

A variety of machine learning methods have been suggested in previous studies for solving the problem of credit card fraud detection. Bayesian Belief Networks (BBN) and Artificial Neural Networks (ANN) were used on a real dataset from Europay International in [139]. Neural network based models and decision tree models were contrasted in [140]. In [141], decision trees and support vector machines (SVMs) are applied to a real-world national bank dataset. [129] demonstrated that using a convolution model to extract spatial patterns can achieve higher accuracy than non-convolution neural networks. Recently, graph-based fraud detection techniques have emerged and gained increasing popularity. For example, CARE-GNN in [135] was designed to handle fraud detection on relational graphs, and PC-GNN [134] was designed for imbalanced supervised learning on graphs. AO-GNN [142] introduced reinforcement learning to pursue an optimal edge pruning strategy, so as to combat the label imbalance issue. H2-FDetector [143] leveraged both homophilic and heterophilic connections and introduced a prototype prior to guiding the identification of fraudsters. In [104, 127], authors proposed joint feature learning based on spatial and temporal patterns. However, they modeled the fraud patterns by using only one transaction/cardholder, thereby not being able to exploit the unlabeled data in real-life credit card transactions. The approach we present in this chapter is radically different, as we employ a semi-supervised architecture, where the fraud patterns on both unlabeled and labeled data are jointly learned within an attribute-driven graph neural network framework.

5.2.2.2 Graph-based Semi-supervised Learning

Many recent works have shown the benefit of using unlabeled node attributes in graph neural networks for a wide range of prediction tasks [144, 145], such as text classification [146], molecule property prediction [147] or language understanding [148]. For instance, graph convolutional networks (GCN) were employed on partially labeled citation networks for node property prediction [62]. GraphSAGE [149] was proposed to generate low-dimensional embeddings for previously unseen data. Graph attentive network model and random walks [133] were deployed on social graphs to link the unlabeled and labeled data and pass messages among them. SPC-GNN [150] used a self-paced label augmentation strategy to assist the co-training paradigm and gained competitive results in semi-supervised node classification tasks. However, they still face at least one of the following three limitations: (1) cannot scale up to real-world graphs over millions of nodes (e.g., vanilla graph attention networks [57] have a space complexity $O(N^2)$, where N denotes the number of nodes, which is unaffordable for tasks with millions of nodes); (2) cannot propagate and learn the categorical attribute embeddings, especially for risk embeddings; (3) suffer from insufficient attention and utilization of graph structural information. Differently, our approach addresses the fraud detection problem via a message-passing model where categorical attributes and risk structural information are jointly aggregated and exploited. We propose an attribute-driven model and semi-supervised graph neural networks to find more fraud patterns, which significantly improve the accuracy of credit card fraud detection.

5.2.2.3 Graph Structure Learning

Graph structure learning (GSL) is a research area that aims to learn more effective graph structures and representations for downstream tasks [151] and involves inferring optimal graph structures and representations from data that are generated by or correlated with the graph [152]. Most approaches in this realm are inspired by [153], which employs *persistent homology* to calculate topological features(e.g., cycle, path, connected components). [154] proposes a new kernel and an optimization framework to learn the topological summaries of data and achieve competitive results in graph classification. [155] augment the subtree features of the Weisfeiler–Lehman graph kernel with topological information so as to improve the performance of graph-level classification. [156] enables deep neural networks to capture topological structure via inputting features obtained from

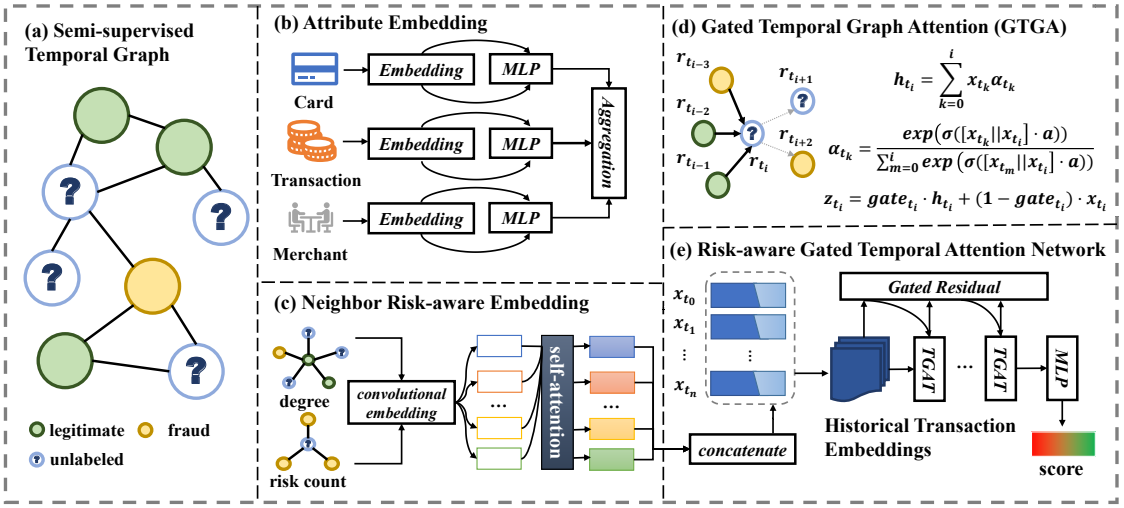


FIGURE 5.2: The illustration of the proposed graph neural network model. Raw transaction records are processed by attributed embedding and attribute aggregation to combine each semantic representation. Degree and risk information is collected from the multi-hop neighborhood and concatenated into node features after convolutional embedding and self-attention operations. Afterward, the learned representations are fed into a risk-aware gated temporal attention network (RGTAN) for representation learning. The transaction representation is then fed into a multi-layer perceptron for fraud detection. Attentional weights are jointly optimized in an end-to-end mechanism with graph neural networks and fraud detection networks.

persistent homology. PEGN in [157] designs a persistence layer to enrich graph representations, and [158] further makes graph neural networks topological-aware via a topological graph layer (TOGL). A subgraph isomorphism counting layer is raised in GSN to capture higher-order structural information [138]. It is worth paying attention to such structural information in the realm of fraud detection. [159] utilized a HGAR attention mechanism to select risk pattern candidates(i.e. 4-vertex-motif structures). However, the aforementioned approaches are not readily applicable to the domain of fraud detection. Purely structure-aware methods fail to leverage label information, thus hindering their ability to detect fraud patterns that are intimately associated with fraud labels. In this chapter, we innovatively introduce the idea of graph structural learning into fraud detection. Specifically, we devise a risk-aware learning layer, which adopts the idea of 'structure-ware', to model high-order adjacent fraud patterns which are proven to be conducive to improving the expressiveness and performance of graph neural networks.

5.3 Gated Temporal Graph Attention

In this section, we first introduce the framework of our proposed Gated Temporal Graph Attention (GTGA) mechanism. After that, we present the process of feature engineering and the gated temporal attention networks. The optimization strategy and learning objective are defined at the end.

5.3.1 Model Architecture

The general model architecture of our proposed method is illustrated in Figure 5.2. Methods discussed in this section include *(b)Attribute Embeeding* and *(d) Gated Temporal Graph Attention*. Raw attributes of transaction records are first learned by the attribute embedding look-up and feature learning layer, which includes feature aggregation with a multi-layer perception (MLP). In our implementation, the attributes of the card include the card type, cardholder type, card limit, remaining limit, etc. The transaction attributes include the channel ID, currency ID, transaction amount, etc. The merchant attributes contain merchant type, terminal type, merchant location, sector, charge ratio, etc.

Based on the transaction data, thanks to the proposed graph generators in previous Chapters, we leverage rule-based and learning-based approaches to generate the temporal transaction graph. Then, we devise a gated temporal attention network to aggregate and learn the importance of historical transaction embeddings. Afterward, we leverage a two-layer MLP to learn the fraud probability from these representations. The whole model can be optimized in an end-to-end mechanism jointly with the existing stochastic gradient descent algorithm.

5.3.2 Attribute Embedding and Feature Learning

This subsection introduces our preprocessing of transaction attributes. For given transaction records $\mathbf{r} = (r_1, r_2, \cdots, r_N)$, each record r_i contains card attributes f_c^i, transaction attributes f_r^i, and merchant attributes f_m^i as $r_i = \{f_c^i, f_r^i, f_m^i\}$. In preprocessing, different from [104, 127], we do not filter out any cards or merchants that have few authorized transaction records. As the number of cards and merchants which have never been checked manually is much larger than checked, we

adopt full transaction records of users so as to maintain all potential frauds during preprocessing. Afterward, we construct the numerical attribute representation of each record into tensor format $\mathbf{X}_{num} \in \mathbb{R}^{N \times d}$, where N denotes the number of transactions, and d denotes the dimensions of features. Besides, we extract the card, transaction, and merchant category attributes $\mathbf{X}_{cat} \in \mathbb{R}^{N \times d}$ separately through attribute embedding layers, which can be formulated as follows:

$$\begin{aligned} e_{attr} =& \text{onehot}(f_{attr}) \odot \mathbf{E}_{attr}, \\ x_{cat,i} =& \text{MLP}_i(\sum_{\forall j \in table_i} e_j), i \in \{card, trans, mchnt\}, \end{aligned} \tag{5.1}$$

where $j \in table_i$ denotes the column j in our input table data i, $e_{attr} \in \mathbb{R}^{1 \times d}$ denotes the embedding of attribute $attr$, onehot$(\cdot)$ denotes the one-hot encoding, f_{attr} denotes the single attribute of one transaction, and $\mathbf{E}_{attr} \in \mathbb{R}^{m \times d}$ denotes the embedding matrix of attribute $attr$, where m denotes the maximum number of attribute $attr$.

After obtaining the embedding vector of each attribute in the card, transaction, and merchant tables, we aggregate these embeddings to obtain each transaction's categorical embedding through add-pooling with $x_{cat}^{(u)} = \sum_i x_{cat,i}^{(u)}, i \in \{card, trans, mchnt\}$, where $x_{cat}^{(u)} \in \mathbb{R}^{1 \times d}$ denotes the category embedding vector of the u-th transaction record. Compared with previous baselines, our method reduces space complexity from $O(Nac)$ to $O(Na + acd)$, where N denotes the number of transactions in our dataset, c denotes the number of categories, and a denotes the average number of unique attribute values in each categorical column. Therefore, our method can handle a large number of categorical attributes in real application environments. Besides, due to the heterogeneity of categorical attributes, our proposed feature learning layer can model all categorical attributes and project them to a unified spatial dimension, which is beneficial to our attribute-driven graph learning model.

5.3.3 Gated Temporal Graph Attention Mechanism

In order to learn the temporal fraud patterns, we generate the temporal transaction graph and aggregate messages on this graph to update the embedding of each transaction. Particularly, the directed temporal edges are generated with the previous transactions as the source and the current ones

as the target, as illustrated in Figure 5.2(c). Then we aggregate messages through the Temporal Graph Attention. The number of generated temporal edges per node is a hyper-parameter, which will be studied in the experiment section.

Temporal Graph Attention. After the feature engineering and attribute embedding, we leverage a series of transaction embeddings $\mathbf{X} = \{x_{t_0}, x_{t_1}, ...x_{t_n}\}$ to learn the temporal embedding of each transaction record. First, we combine categorical attributes and numerical attributes as the input of RGTAN with $x_{t_i} = x_{num}^{(t_i)} + x_{cat}^{(t_i)}$. At the first GTGA layer, we set $\mathbf{H}_0 = \mathbf{X}$ as the input embedding matrix. Afterward, we leverage multi-head attention to separately calculate the importance of each neighbor and update embeddings, which can be formulated as follows:

$$\mathbf{H} = \text{Concat}(\text{Head}_1, ..., \text{Head}_{h_{att}})\mathbf{W}_o, \tag{5.2}$$

where h_{att} denotes the number of heads, $\mathbf{W}_o \in \mathbb{R}^{d \times d}$ denotes learnable parameters, $\mathbf{H}$ denotes the aggregated embeddings with $\mathbf{H} = \{h_{t_0}, h_{t_1}, ..., h_{t_n}\}$, and each attention head is formulated as follows:

$$\begin{aligned} \text{Head} &= \sum_{x_i \in \mathcal{X}} \sigma\Big(\sum_{x_t \in \mathcal{N}(x_i)} \alpha_{x_t, x_i} x_t \Big), \\ \alpha_{x_t, x_i} &= \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T [x_t || x_i))}{\sum_{x_j \in \mathcal{N}(x_t)} \exp(\text{LeakyReLU}(\mathbf{a}^T [x_t || x_j]))}, \end{aligned} \tag{5.3}$$

where $\mathcal{N}(x_i)$ denotes the temporal neighbors of the i-th transaction, α_{x_t, x_i} denotes the importance of temporal edge (x_t, x_i) in each attention head, and $\mathbf{a} \in \mathbb{R}^{2d}$ denotes the weight vector of each head. In practice, to avoid extra space consumption in extreme cases (such as high-frequency transactions in a short period), we use a neighbor sampling and truncation strategy to control the number of neighbor nodes $|\mathcal{N}(x_t)|$ (i.e., the number of associated temporal edges per node) through which the temporal graph attention layer propagates messages. Besides, to avoid borrowing future information, the neighbor transactions sampled for each transaction must be the past transactions from the same cardholder so that we can model the temporal fraud pattern through message passing on the temporal transaction graph.

Attribute-driven Gated Residual. To further improve the effectiveness and interpretability of our method, after obtaining aggregated embeddings, we leverage the embeddings and raw attributes to infer the importance of the aggregated embeddings and raw attributes after the temporal graph attention process, which can be formulated as follows:

$$\begin{aligned} \text{gate}_{t_i} =& \sigma([x_{cat,t_i}||x_{num,t_i}||h_{t_i}]\beta_{t_i}), \\ z_{t_i} =& \text{gate}_{t_i} \cdot h_{t_i} + (1 - \text{gate}_{t_i}) \cdot x_{t_i}, \end{aligned} \tag{5.4}$$

where $\text{gate}_{t_i} \in [0, 1]$ denotes the gate variable of the t_i-th transaction, σ denotes the sigmoid activation function, $\beta_{t_i} \in \mathbb{R}^{3d\times 1}$ denotes the gate vector, and z_{t_i} denotes the output vector of each GTGA layer, which is fed into the next layer as input. According to our framework, if we stack a new GTGA layer with the attribute-driven gated residual mechanism, we use the output of the k-th gating mechanism as the input of the $k+1$-th GTGA. In this stacking framework, the bottom-up k-th GTGA layer weighs the importance of the k-th order neighbor transactions. In addition, the bottom-up k-th attribute-driven gated residual mechanism weighs the importance of each transaction's k-th order neighbor transaction embedding and its own embedding. Algorithm 3 shows the detailed computation process of message passing in one GTGA layer.

5.3.4 Fraud Risk Prediction

After obtaining the aggregated embeddings of transactions, we leverage a two-layer MLP to predict the fraud risk, which is formulated as follows:

$$\hat{\mathbf{y}} = \sigma(\text{PReLU}(\mathbf{H}\mathbf{W}_0 + \mathbf{b}_0)\mathbf{W}_1 + \mathbf{b}_1), \tag{5.5}$$

where $\hat{\mathbf{y}} \in \mathbb{R}^{N\times 1}$ denotes the risk prediction results of all transactions, and $\mathbf{W}$ and $\mathbf{b}$ denote the learnable parameters of MLP. Afterward, we calculate the objective function $\mathcal{L}$ via binary cross-entropy, which is formulated as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=0}^{N}[\mathbf{y}_i \cdot \log p(\hat{\mathbf{y}}_i|\mathbf{X}, \mathbf{A}) + (1-\mathbf{y}_i) \cdot \log(1 - p(\hat{\mathbf{y}}_i|\mathbf{X}, \mathbf{A}))], \tag{5.6}$$

where $\mathbf{y}$ denotes the ground-truth label of transactions. The proposed network structure can be optimized through the standard SGD-based algorithms.

5.4 Risk-aware Gated Temporal Attention Network

In this section, we present the details of risk propagation and neighbor risk-aware embedding. Practically, manually annotated labels are adopted as categorical features and integrated into the original features after embedding transformations. Risk information from multi-hop neighborhoods is encoded into features by adding the degree and risk count from multi-hop neighbors. Both are jointly passed and updated through GTGA, which we call *risk-aware message passing*. To avoid possible label leakage, we leverage a masked fraud detection strategy.

Algorithm 3: Steps of computation in a GTGA layer

Input : $G(V, E)$: the given transaction graph
$\mathbf{H}_k$: the embedding matrix from the k^{th} layer
x_{cat}: categorical features
x_{num}: numerical features
Output: $\mathbf{H}_{k+1}$: updated embedding feature matrix as input of $(k+1)^{\text{th}}$ layer
for $i \leftarrow 1$ **to** N **do**
  for $h \leftarrow 1$ **to** h_{att} **do**
    $\text{Head}_h^i \leftarrow \sigma(\sum_{x_t \in \mathcal{N}(x_i)} \alpha_{x_t, x_i} x_t)$;
    $\alpha \leftarrow \frac{\exp(\sigma(\mathbf{a}^T[x_t||x_i]))}{\sum_{x_j \in \mathcal{N}(x_i)} \exp(\sigma(\mathbf{a}^T[x_t||x_j]))}$;
  end
  $h_i \leftarrow \text{Concat}(\text{Head}_1^i, ..., \text{Head}_{h_{att}}^i)\mathbf{W}_o$;
  $v_i \leftarrow \text{Concat}(x_{cat,i}, x_{num,i}, h_i)$;
  $\text{gate}_i \leftarrow \sigma(v_i \beta_i)$;
  $z_i \leftarrow \text{gate}_i \cdot h_i + (1 - \text{gate}_i) \cdot \mathbf{H}_{k,i}$;
end
$\mathbf{H}_{k+1} \leftarrow [z_1||...||z_N]$;

5.4.1 Risk Propagation Representation

The manual-annotated labels are expensive in real-world fraud detection practice. With labeled risk information, we can effectively model more fraud patterns, such as risk propagation. Therefore, inspired by unifying label propagation with feature propagation [160], we propose to take the manually annotated label as one of the categorical attributes of the transaction and get the embedding of this categorical attribute, which we call *risk embedding*. Specifically, we take the manually annotated label as the risk feature of each transaction, where the category of unlabeled data is 'unlabeled', and the category of the rest of the data is 'fraud' or 'legitimate'. Then, we add this feature to the transaction data as one of our input categorical attributes. Due to concerns about label leakage, this attribute has not been used in previous fraud detection solutions. We will discuss the techniques for avoiding label leakage later. Then, we propose to embed the partially observed risk attributes (i.e., labels) into the same space as the other node features, which consist of the risk embedding vectors for labeled nodes and zero embedding vectors for the unlabeled ones. Then, we simply add the node features and risk embeddings together as input node features with $x_{t_i} = x_{num}^{(t_i)} + x_{cat}^{(t_i)} + \tilde{y}^{(t_1)}\mathbf{W}_r$, where $\mathbf{W}_r$ denotes the learnable parameters of risk embedding. [160] have proved that by mapping partially-labeled $\hat{\mathbf{Y}}$ and node features $\mathbf{X}$ into the same space and adding them up, we can use one graph neural network to achieve both attribute propagation and label propagation. Therefore, our fraud detection model can jointly model the temporal fraud patterns and fraud risk propagation just by adding the transaction label as one of the transaction categorical attributes.

5.4.2 Neighbor Risk-aware Attentional Embedding

Fraud transactions often fabricate noisy information to make them difficult to be recognized and therefore result in redundant link information around fraudulent nodes, which can to some extent weaken the power of neighborhood aggregation [134]. For example, fraudulent transactions might be deliberately connected to numerous legitimate transactions so that spammers could hide among legitimate users. Such risk patterns are relatively difficult to be captured by vanilla graph-based methods. In this chapter, we leverage the degree information of multi-hop neighbor nodes and count of risky neighbors as node features, which can inherently reflect the local risk structure,

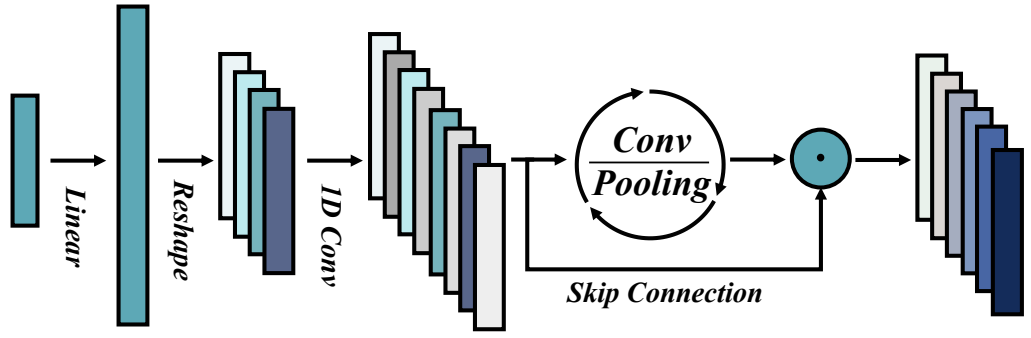


FIGURE 5.3: Convolutional Embedding. The final output of the convolutional embedding layers is represented as $\mathbb{R}^{N\times r\times d}$, with each output channel corresponding to a previous neighbor risk feature.

to alleviate the problems described above. Risk-aware representations are further learned by a convolutional embedding layer and structural attention layer.

Convolutional Embedding. For given transaction records $\mathbf{r} = (r_1, r_2, \cdots, r_N)$, each record r_i have neighbor degree feature $f^i_{degree,k}$ and risk neighbor count feature $f^i_{risk,k}$, where k denotes k-hop neighborhood. Besides, to avoid future label leakage, we merely gather information from past transaction records belonging to the same cardholder of r_i. The above two features can be formulated as follows:

$$\begin{aligned} f^i_{degree,k} &= \sum_{u\in\mathcal{N}^i_k} \text{Degree}(u), \\ f^i_{risk,k} &= \sum_{u\in\mathcal{N}^i_k, \mathbf{y}_u=1} \mathbf{y}_u \end{aligned} \tag{5.7}$$

where $\mathcal{N}^i_k$ denotes the filtered k-hop neighbor nodes of r_i, $\text{Degree}(u)$ counts the in degree of r_u and $\mathbf{y} \in \mathbb{R}^N$ denotes labels of all transactions. After obtaining the neighbor risk information for each transaction record, we stack these numerical features and construct neighborhood risk-aware representation through a convolutional embedding layer. [161] has proven such embedding numerical features can be conducive to many backbone structures. We construct the neighbor risk features into tensor format $\mathbf{X}_{nei} \in \mathbb{R}^{N\times r}$ where r denotes the number of neighbor risk features, which later will be transformed into the risk-aware embedding matrix $\mathbf{X}_{rsk} \in \mathbb{R}^{N\times r\times d}$ as Figure 5.3 shows.

Specifically, $\mathbf{X}_{nei} \in \mathbb{R}^{N \times r}$ is expanded via a linear layer and reshaped into c channels, which we denote as $\mathbf{X}_{conv} \in \mathbb{R}^{N \times c \times L}$ with L referring to the length of each channel. Thereafter, multiple 1D convolutional layers are leveraged to extract the risk information representations. We denote each channel of record i at layer l as $\mathbf{X}_{conv}^{i,j,l}(1 <= j <= r)$, and each channel vector is convolved by a filter vector $\mathbf{w}$ of length H with zero-padding after the batch norm operation.

$$\mathbf{X}_{conv}^{i,j,l+1}[m] = \sigma(\sum_{h=0}^{H-1} \mathbf{w}[h]\mathbf{X}_{conv}^{i,j,l}[m-h+P] + b), \quad m \in [0, L-1] \tag{5.8}$$

where b is the bias term and P is the padding size that satisfies $P = \frac{H-1}{2}$. At the final convolutional layer, the number of channels and the vector length are adjusted to r and d, via an output convolutional layer and an adaptive pooling layer respectively, with each channel corresponding to a previous neighbor risk feature. The skip connection strategy is leveraged to alleviate the problem of gradient vanishing and exploit spatial correlations and translations between risk features. $\mathbf{X}_{conv}^{L} \in \mathbb{R}^{N \times r \times d}$ are adopted as the risk-aware embedding.

Structural Attention. Afterwards, we introduce a structure-aware self-attention module to separately calculate the importance of each neighbor risk-aware feature and update the embeddings, which can further exploit the neighborhood risk information and help learn fraud patterns. For a certain transaction record r_i, we denote its embedding matrix as $\mathbf{X}^i \in \mathrm{R}^{r \times d} = \{f_1, f_2, ..., f_r\}$. The calculation process can be formulated as follows:

$$\mathbf{H}^i = \text{Concat}(\text{Head}_1, ..., \text{Head}_{num})\mathbf{W}_o, \tag{5.9}$$

where num denotes the number of attention heads, $\mathbf{W}_o \in \mathrm{R}^{d \times d}$ denotes learnable parameters, $\mathbf{H}$ denotes the updated risk-aware embeddings and each attention head is formulated as follows:

Algorithm 4: Neighbor risk-aware embedding

Input : $G(V, E)$: the given transaction graph
Output: $\mathbf{X}_{rsk}$: neighbor risk-aware embeddings
for $i \leftarrow 1$ **to** N **do**
  for $k \leftarrow 1$ **to** K **do**
    $f^i_{de,k} = \sum_{u \in \mathcal{N}^i_k} \text{Degree}(u)$,
    $f^i_{ri,k} = \sum_{u \in \mathcal{N}^i_k} \mathbf{y}_u$ **if** $\mathbf{y}_u = 1$; **then**
  end
  $f^i_{nei} \leftarrow [f^i_{de,1}, ..., f^i_{de,K}, f^i_{ri,1}, ..., f^i_{ri,K}]$
  $h^i_0 \leftarrow \texttt{Reshape}(f^i_{nei}\mathbf{W}_0$,(c,L))
  for $l \leftarrow 1$ **to** L **do**
    for $c \leftarrow 1$ **to** C **do**
      for $m \leftarrow 1$ **to** L **do**
        $h^i_{l,c}[m] \leftarrow \sigma^{H-1}_{s=0}(\mathbf{w}^{l,c}[s]h^i_{l-1,c}[m - s + P] + b) + h^i_{l-1,c}$
      end
    end
  end
  $\{f_1, f_2, ...f_r\} \leftarrow \{h_{L,1}, h_{L,2}, ...h_{L,r}\}$
  for $h \leftarrow 1$ **to** num **do**
    $\text{Head}^i_h \leftarrow \sigma(\sum_{t \in [1,r]} \alpha_{f_t,f_i} x_t)$;
    $\alpha_{f_t,f_i} \leftarrow \frac{\exp(\sigma(\mathbf{a}^T[f_t||f_i]))}{\sum_{j \in [1,r]} \exp(\sigma(\mathbf{a}^T[f_j||f_i]))}$;
  end
  $\mathbf{H}^i \leftarrow \text{Concat}(\text{Head}^i_1, ..., \text{Head}^i_{num})\mathbf{W}_1$;
end
$\mathbf{X}_{rsk} \leftarrow \{\mathbf{H}^1, \mathbf{H}^2, ..., \mathbf{H}^N\}$

$$\begin{aligned} \text{Head} &= \sum_{i \in [1,r]} \sigma(\sum_{m \in [1,r]} \alpha_{f_m,f_i} f_i), \\ \alpha_{f_m,f_i} &= \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[f_m||f_i]))}{\sum_{n \in [1,r]} \exp(\text{LeakyReLU}(\mathbf{a}^T[x_t||x_n]))} \end{aligned} \tag{5.10}$$

where α_{f_m,f_i} denotes the importance between feature f_m and f_i in each attention head, and $\mathbf{a} \in \mathrm{R}^{2d}$ denots the weight vector of each head. The output embedding matrix $\mathbf{H}^i \in \mathbb{R}^{r \times d}$ is then reduced in the last dimension and then concatenated to the input node features, which can be formulated as follows:

$$x_{t_i} = \text{CONCAT}(x_{num}^{(t_i)} + x_{cat}^{(t_i)} + \tilde{y}^{(t_1)}\mathbf{W}_r, \text{SQUEEZE}(H^i\mathbf{W}_{proj})) \tag{5.11}$$

where $\mathbf{W}_{proj} \in \mathbb{R}^{d\times 1}$ and SQUEEZE denotes squeezing matrix in the last dimension. Therefore, our fraud detection model is enhanced by the integration of neighbor risk-aware embeddings and is capable of better modeling temporal fraud patterns. Algorithm 4 demonstrates the detailed computation of risk-aware embeddings.

5.4.3 Loss Function and Model Optimization

Previous works on credit card fraud detection did not consider using the partially observed labels $\hat{\mathbf{Y}}$ in both training and inference stages and label information collected from the neighborhood is also rarely exploited. They only took those risk information as optimization objectives to supervise their fraud detection model training with given transaction attributes. Different from previous solutions for credit card fraud detection, we semi-supervise our proposed RGTAN model through propagating transaction attributes, risk embeddings, and neighbor risk-aware features among labeled and unlabeled transactions to train our model. Simply using an unmasked objective for our fraud detection model will result in label leakage in the training process. In this case, our model will directly take observed labels and neglect the complicated hidden fraud patterns, which are not able to be generalized in predicting future fraud transactions.

Therefore, we propose to learn from the risk information of each transaction's neighbor transactions instead of learning from its own label. Specifically, a masked fraud detection training strategy is leveraged to train our model. During training, at each step, we randomly sample a batch of nodes, namely center nodes, along with the neighbor nodes corresponding to each center node. Then, we convert the $\hat{\mathbf{Y}}$ into $\tilde{\mathbf{Y}}$ by masking all the center nodes' risk embeddings to zero embeddings and keeping the others unchanged. Furthermore, to ulteriorly model fraud patterns from neighbor transactions, risk information from the neighborhood is directly collected and transformed as node features. The operation above theoretically may result in label leakage, and a multi-hop masking strategy is leveraged to tackle the risks. Particularly, we denote the risk feature from k-hop neighbor as f_{risk}^k, and in the training process, f_{risk}^k belonging to k-hop neighbor of

center nodes is masked to zeros embeddings. Then, our objective function is to predict $\hat{\mathbf{Y}}$ with given $\mathbf{X}$, $\tilde{\mathbf{Y}}$ and $\mathbf{A}$:

$$\begin{aligned}\mathcal{L} = -\frac{1}{|V|}\sum_{i=0}^{|V|}[\mathbf{y}_i \cdot \log p(\hat{\mathbf{y}}_i|\mathbf{X}, \tilde{\mathbf{Y}}, \mathbf{A}) + \\ (1-\mathbf{y}_i) \cdot \log(1 - p(\hat{\mathbf{y}}_i|\mathbf{X}, \tilde{\mathbf{Y}}, \mathbf{A}))],\end{aligned} \tag{5.12}$$

where $|V|$ represents the number of center nodes with masked labels. In this way, we can train our model without the self-loop leakage of risk information; and during inference, we employ all observed labels $\hat{\mathbf{Y}}$ as input categorical attributes to predict the risk of the transactions out of the training set. So far, the optimization objective of our model can be intuitively summarized as: modeling the fraud patterns by the attribute information (including risk categorical information and neighbor risk-aware embeddings) of neighboring transaction nodes and the attribute information (excluding risk information) of itself.

5.5 Experiments

In this section, we first describe the datasets used in the experiments, then compare our fraud detection performance with other state-of-the-art baselines on two supervised graph-based fraud detection datasets and one semi-supervised dataset. Then, we perform ablation studies by evaluating two variants of the proposed RGTAN, which demonstrates the effectiveness of our proposed method and attribute-driven mechanism. Finally, real-world case studies show that our proposed RGTAN significantly outperforms other baselines for detecting typical fraud patterns.

5.5.1 Experiment Settings

5.5.1.1 Datasets

To the best of our knowledge, we did not find any public semi-supervised credit card fraud detection dataset. Therefore, we collect the partially labeled records from our collaborated partners,

TABLE 5.1: Statistics of the three fraud detection datasets.

Dataset	YelpChi	Amazon	FFSD
#Node	45,954	11,948	1,820,840
#Edge	7,739,912	8,808,728	31,619,440
#Fraud	6,677	821	33,858
#Legitimate	39,277	11,127	141,861
#Unlabeled	-	-	1,645,121

TABLE 5.2: Fraud detection performance of various methods on three datasets: YelpChi, Amazon, and FFSD. The evaluation metrics used are the area under the roc curve (AUC), macro average of F1 score (F1-macro), and average precision (AP). Among the methods, RGTAN stands out with the highest AUC and AP scores on all three datasets. RGTAN's excellent performance on YelpChi dataset with AUC score of 0.9498, F1-macro score of 0.8492*, and AP score of 0.8241 is particularly noteworthy.

Dataset	YelpChi			Amazon			FFSD		
	AUC	F1	AP	AUC	F1	AP	AUC	F1	AP
GEM	0.5270	0.1060	0.1807	0.5261	0.0941	0.1159	0.5383	0.1490	0.1889
Player2Vec	0.7003	0.4121	0.2473	0.6185	0.2451	0.1291	0.5278	0.2147	0.2041
FdGars	0.7332	0.4420	0.2709	0.6556	0.2713	0.1438	0.6965	0.4089	0.2449
Semi-GNN	0.5161	0.1023	0.1811	0.7063	0.5492	0.2254	0.5473	0.4485	0.2758
GraphSAGE	0.5364	0.4508	0.1712	0.7502	0.5795	0.2624	0.6527	0.5370	0.3844
GraphConsis	0.7060	0.6041	0.3331	0.8782	0.7819	0.7336	0.6579	0.5466	0.3876
CARE-GNN	0.7934	0.6493	0.4268	0.9115	0.8531	0.8219	0.6623	0.5771	0.4060
PC-GNN	0.8174	0.6682	0.4810	0.9581	0.9153	0.8549	0.6795	0.6077	0.4487
GTAN	0.9241	0.7988	0.7513	0.9630	**0.9213**	0.8838	0.7616	0.6764	0.5767
RGTAN	**0.9498***	**0.8492***	**0.8241***	**0.9705***	0.9198	**0.8925***	**0.7680***	**0.6800***	**0.5786***

namely Finacial Fraud Semi-supervised Dataset (**FFSD**). We collected it from a major global financial institution, which comprises real-world credit card transaction records spanning ten months. The ground truth labels are obtained on cases reported by consumers and confirmed by domain experts in the financial institution. If a transaction is reported by a cardholder or identified by financial experts as fraudulent, we label it as 1; otherwise, it is labeled as 0.

Besides, we also experimented on two public supervised fraud detection datasets. The **YelpChi** graph dataset [162] contains a selection of hotel and restaurant reviews on Yelp. Nodes in the graph of the YelpChi dataset are reviews with 32-dimensional features, and the edges are the relationships among reviews. The **Amazon** graph dataset [163] includes product reviews of musical instruments. The nodes in the graph are users with 25-dimensional features, and the edges are the relationships among reviews. Some basic statistics of the three datasets are shown in Table 5.1.

5.5.1.2 Compared Methods.

The following methods are compared to highlight the effectiveness of the proposed GTAN.

- *GEM.* Heterogeneous GNN-based model proposed in [164]. We set the learning rate to 0.1 and the number of hops of neighbors to 5.
- *FdGars.* Fraudster detection via the graph convolutional networks proposed in [165]. We set the learning rate to 0.01 and the hidden dimension to 256.
- *Player2Vec.* Attributed heterogeneous information network proposed in [166]. We set the same parameters as the FdGars model.
- *Semi-GNN.* A semi-supervised graph attentive network for financial fraud detection proposed in [133]. We set the learning rate to 0.001.
- *GraphSAGE.* The inductive graph learning model proposed in [149]. We set the embedding dimension to 128.
- *GraphConsis.* The GNN-based model tackling the inconsistency problem, proposed in [136]. We used the default parameters suggested by the original paper.
- *CARE-GNN.* The GNN-based model tackling fraud detection on a relational graph [135]. We used default parameters from the original paper.
- *PC-GNN.* A GNN-based model remedying the class imbalance problem, proposed in [134]. We used the default parameters from the original paper.
- GTAN. The attribute-driven semi-supervised attention network proposed in [125]. We used the default parameters from the original paper.
- **RGTAN.** The proposed risk-aware gated temporal attention network model[1]. We also evaluate three variants of our model, RGTAN-A, RGTAN-R, and RGTAN-N, in which the temporal graph attention component, risk embedding component and risk-aware representations are not considered, respectively. In practice, we learn risk-aware embeddings from 2-hop neighborhood. We set the batch size to 128, the learning rate to 0.002, the input dropout

[1]The sources of our proposed methods will be available at https://github.com/finint/antifraud.

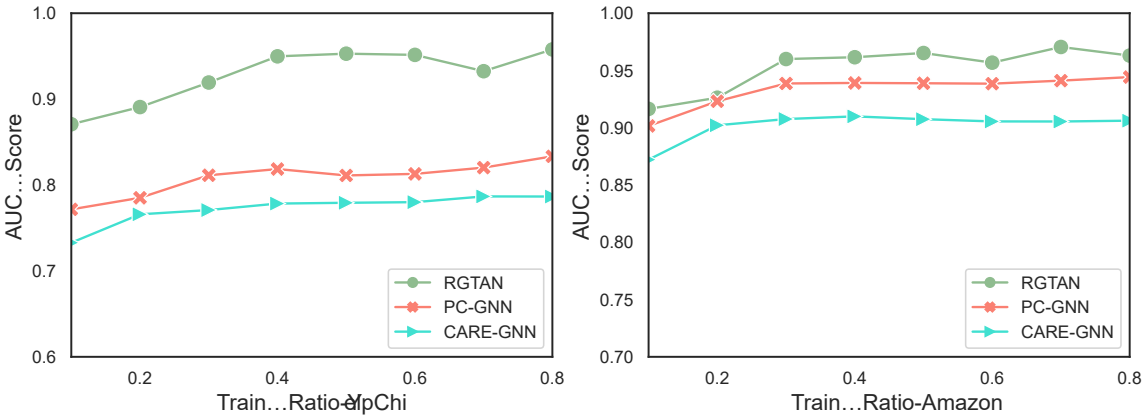


FIGURE 5.4: The result of semi-supervised experiments with different ratios of labeled training data. The left is the performance of CRAE-GNN, PC-GNN and RGTAN on YelpChi dataset, with the training ratio ranging from 0.1 to 0.8, which generally displays an upward trend with more data used for training. The right is the compared performance on Amazon dataset and roughly exhibits the same trend.

ratio to 0.2, the number of heads to 4, and the hidden dimension d to 256, and train the model with the Adam optimizer for 25 epochs with early stopping.

5.5.1.3 Evaluation Metrics

We evaluate the experimental results on credit card fraud detection and opinion fraud datasets by the area under the ROC curve (AUC), macro average of F1 score (F1-macro), and average precision (AP), which are calculated as follows:

We count the number of True Positive N_{TP} (i.e. correct identification of positive labels), False Positive N_{FP} (i.e., incorrect identification of positive labels), and False Negatives N_{FN} (i.e., incorrect identification of negative labels). Then, F1 score and AP as formulated with $F1_{macro} = \frac{1}{l}\sum_{i=1}^{l}\frac{2\times P_i \times R_i}{P_i+R_i}$ and $AP = \sum_{i=1}^{l}(R_i - R_{i-1})P_i$, where $P_i = N_{TP}/(N_{TP}+N_{FP})$ and $R_i = N_{TP}/(N_{TP}+N_{FN})$. We also report the AUC (the area under the ROC curve) in our experiments.

5.5.2 Fraud Detection Experiment

In YelpChi and Amazon datasets, the training-to-testing ratio was set to 2:3. For the FFSD dataset, the first seven months' transactions are utilized as training data, and the following three months (August, September, and October of 2021) are used to detect fraudulent transactions. Each method

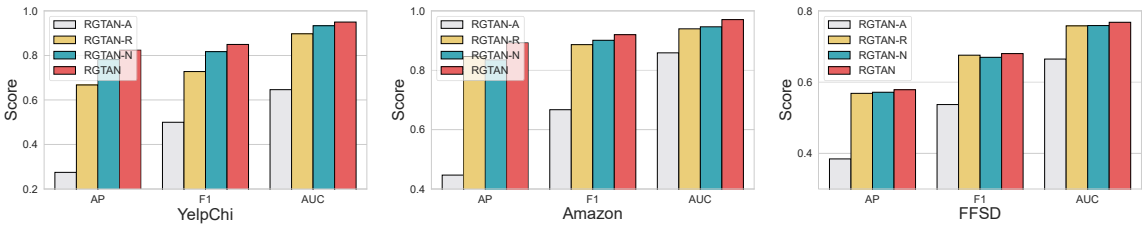


FIGURE 5.5: The ablation study results on three datasets. Gray bars represent the RGTAN-A variant, yellow bars represent the RGTAN-R variant, blue bars represent the RGTAN-N model and red bars represent the RGTAN model. The removal of GTGA component across three datasets in RGTAN-A lead to a dramatic performance drop. Discarding either risk embedding or neighbor risk-aware embedding results in a certain degree of performance deterioration.

is tested ten times, and the average performance is presented in Table 5.2. The statistical significance of the improvement is denoted by * and determined by a paired t-test with a p-value of less than 0.01.

The first five rows of Table 5.2 present the outcomes of some traditional graph-based techniques, such as GEM, Player2Vec, FdGars, Semi-GNN, and GraphSASE. The results indicate that GEM's performance was not satisfactory, highlighting the inadequacy of a shallow model in tackling complex fraud patterns. Player2Vec and FdGars show improved performance, partially due to their enlarged model capacity. Semi-GNN and GraphSAGE are close in performance and superior to the previous three methods, demonstrating the effectiveness of deep graph learning-based models in detecting fraudulent transactions.

By incorporating transaction graphs into the learning process, PC-GNN achieves more competitive results than the aforementioned baselines. The effectiveness of employing graph features in detecting fraudulent transactions is strongly demonstrated. GTAN outperforms the previous methods across all three datasets, validating the expressiveness of the temporal gated attention mechanism. The last row of Table 5.2 presents the results of our proposed method, RGTAN, which successfully outperforms all baselines with at least 2.5%, 0.75%, and 0.64% AUC improvements across the three datasets, respectively. Furthermore, RGTAN outperforms other baselines by at least 7.3%, 0.9%, and 0.2% AP improvements across the three datasets, respectively, strongly demonstrating the effectiveness of our risk-aware embeddings to capture multi-hop risk structure and higher-order risk patterns.

5.5.3 Risk-aware Semi-supervised Experiment

To assess the effectiveness of semi-supervised learning, we conducted experiments with varying ratios of labeled and unlabeled data in the training set. To streamline the presentation of our findings, we focus on two of the most competitive baselines, namely PC-GNN and CARE-GNN, and use them as a point of comparison for the subsequent semi-supervised experiments. Specifically, we vary the proportion of training nodes from 10% to 80% in increments of 10%, while keeping the remaining nodes as the test set for each experiment. Our experiments are conducted on fully annotated datasets, YelpChi and Amazon, to enable us to explore a wider range of labeled data ratios. The results of our experiments are displayed in Figure 5.4.

Our analysis of the YelpChi dataset reveals that RGTAN consistently outperforms the other models under different training ratios. Even in scenarios with a limited number of labeled data (i.e., 10% training ratio), RGTAN performs well. Moreover, as the number of labeled data increases, the performance of RGTAN steadily improves despite some minor fluctuations. Similarly, our experiments on the Amazon dataset show that RGTAN consistently achieves the best performance across different training ratios. Compared to the YelpChi dataset, the RGTAN model exhibits less sensitivity to changes in the training ratio on the Amazon dataset, with no more than a 5% variation in AUC. These results suggest that RGTAN can perform well even with a small proportion of labeled data (as low as 10%).

In conclusion, our experiments demonstrate the robustness of the RGTAN model to changes in training ratio and its consistent superiority over PC-GNN and CARE-GNN in semi-supervised learning. These findings underscore the effectiveness of the RGTAN model in this domain.

5.5.4 Ablation Study

To test the effectiveness of each key component in our model, we evaluate three variants: RGTAN-A, RGTAN-R and RGTAN-N. RGTAN-A ablates the GTGA component and aggregates messages from neighboring nodes with equal weights, which means it fails to utilize the temporal graph attention mechanism to adaptively adjust the weights of neighbor nodes based on their importance for fraud detection. RGTAN-R ablates the risk embedding component and only uses the original node attributes $\mathbf{X}$ without considering the risk propagation process. This variant does not capture

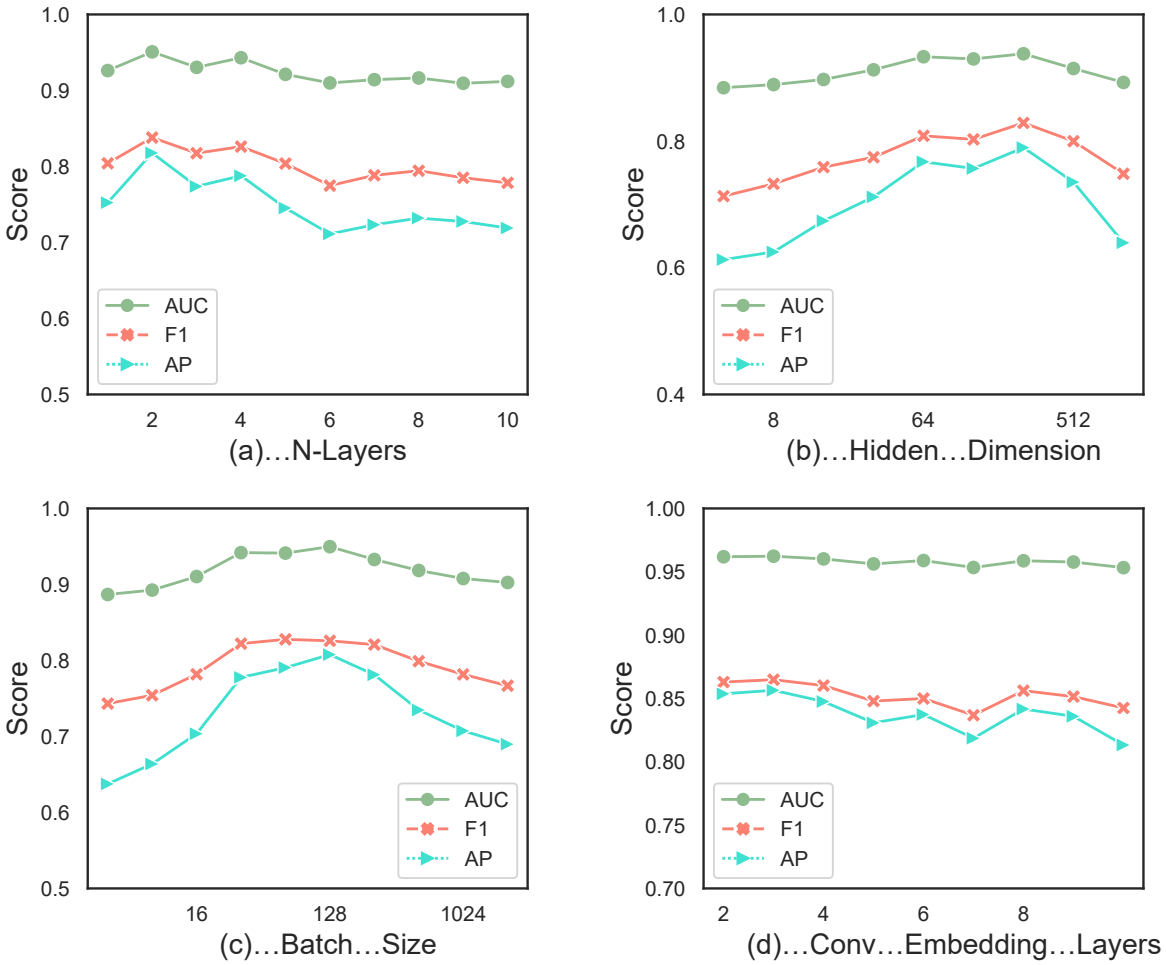


FIGURE 5.6: Parameter sensitivity analysis with respect to (a) the number of GNN layers; (b) the number of temporal edges per node; (c) hidden dimension; and (d) the batch size. (a) shows that the performance of our model reaches the highest when the number of GNN layers is set to 2, and it slightly decreases when the number of layers continues to increase possibly due to the over-smoothing problem. (b) demonstrates that our model is robust to the choice of hidden dimension, with the overall AUC fluctuating by less than 5%. (c) illustrates that RGATN achieves the optimal performance when the batch size is 128, yet it is not sensitive to the choice of batch size, with the AUC under all settings varying by less than 3%. (d) exhibits high robustness to the setting of convolutional embedding layers, with the overall AUC fluctuating by less than 1%.

the credit card fraud patterns from transaction risk propagation and only models the transaction embeddings from other attributes that are not related to risk propagation. In the RGTAN-N model, neighbor risk-aware embedding layers are removed to test its usability and validity. This variant RGTAN-N is incapable of capturing the high-order adjacent fraud patterns.

Figure 5.5 compares the performance of our model (RGTAN) and its three variants that ablate different components: RGTAN-A, RGTAN-R and RGTAN-N. The grey bars indicate that RGTAN-A, which removes the temporal graph attention mechanism, has the lowest scores among the four

models. This demonstrates that the temporal graph attention mechanism is crucial for reweighting the temporal transaction neighbors and capturing their importance for fraud detection. The yellow bars show that RGTAN-R, which removes the risk embedding component, also performs worse than RGTAN, indicating that the risk embedding is conducive to modeling credit card fraud patterns from transaction risk propagation. The blue bars represent RGTAN-N, which removes the neighbor risk-aware embedding layers. RGTAN-N gains the second-highest scores among the four models, which also validates the effectiveness of neighbor risk-aware embedding in enhancing the capacity of RGTAN to detect fraud patterns. In summary, removing either component deteriorates the performance of RGTAN, which proves that the temporal graph attention mechanism, risk embedding and neighborhood risk representation are effective in graph-based credit card fraud detection.

5.5.5 Parameter Sensitivity Experiment

In this section, we investigate the sensitivity of the model parameters by varying the depth of temporal graph attention layers, the hidden dimension, the batch size, and the number of convolutional embedding layers. The results of our experiments on the YELP dataset are presented in Figure 5.6.

We examine the effect of changing the depth of temporal graph attention layers by testing depths ranging from 1 to 10. As depicted in Figure 5.6(a), our model's performance remains stable up to 10 GNN layers. As the depth of hidden layers increases, our model aggregates temporal information from larger neighborhoods. We observe that our model achieves the best performance with two GNN layers when the AUC and AP metrics reach their peak. Therefore, we set the default depth to 2. If we increase the depth of GTGA layers further, we notice a slight degradation in model performance, which might be due to over-smoothing of transaction embeddings caused by deeper GNNs [68].

We investigate the impact of varying the hidden dimension from 4 to 1024 on our proposed model's performance in Figure 5.6(b). We observe that our model maintains stable performance across a range of hidden dimensions, and achieves the relative performance peak at 256. Figure 5.6(c) demonstrates that our RGTAN model performs best with a batch size of 64. Although the model is not sensitive to hyperparameters such as hidden dimension and batch size, with less than 3% variation in AUC across values ranging from 16 to 1024, we set the batch size to 128 considering

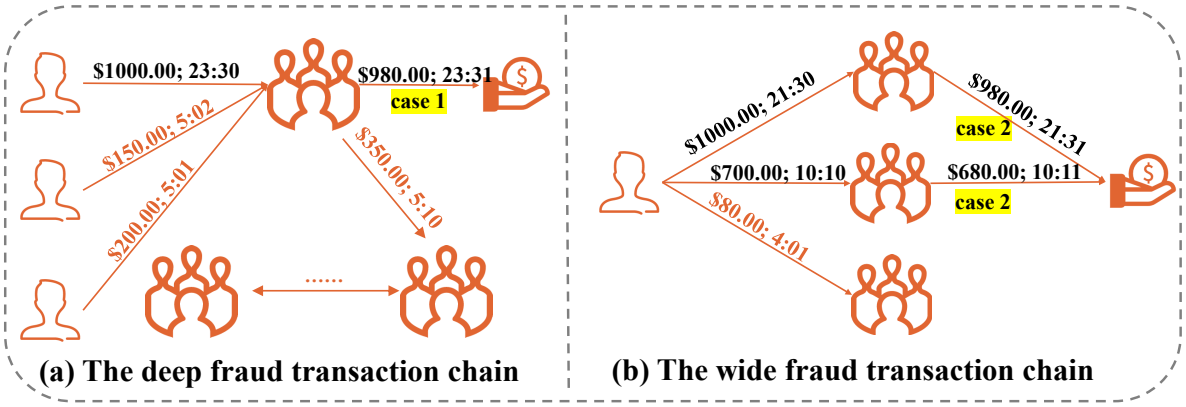


FIGURE 5.7: The case studies on two typical fraud patterns: (a) fraudulent transactions can be hidden by a deep transaction chain; (b) over multiple links, multiple fraudulent transactions let money separately enter the same cardholder.

the training efficiency of the model. Finally, we evaluate the robustness of our risk-aware representation learning modules by varying the number of convolutional embedding layers from 2 to 10. As illustrated in Figure 5.6(d), the performance of our RGTAN model remains stable, with less than a 1% fluctuation in the overall AUC.

5.5.6 Case Studies

With the development of rule-based and machine learning-based fraud detection systems, today's credit card fraud is not simply an individual risk. For example, one of the cardholder's credit cards fell to the ground and was picked up by others. Instead, today's credit card fraud patterns have multi-hop temporal connections that form a *transaction chain*. As shown in Figure 5.7, according to the practical experience in credit card fraud detection, there are two typical types of fraud *transaction chains*. According to Figure 5.7(a), the money flows between multi-hop neighbors, and the end of this deep transaction chain is hidden by a long series of *'legitimate'* transactions. In Figure 5.7(b), the money flows through multiple pipelines, and the end of this wide transaction chain is hidden by multi-source *'legitimate'* transactions. In these cases, cardholders and rule-based fraud detection systems can only report the beginning of the transaction chains, while the end transactions of such fraud chains are difficult to detect by traditional machine learning algorithms due to their incapability to model relations among transaction chains. Therefore, detecting such cases requires a large number of effort from reviewers in card issuers.

TABLE 5.3: Evaluation results on detecting the end of two types of *fraud transaction chains*. Our proposed model outperforms other baselines significantly in detecting these two fraud patterns.

	Case 1		Case 2	
	AUC	AP	AUC	AP
GraphConsis	0.6178	0.3625	0.6626	0.4002
CARE-GNN	0.6274	0.3576	0.6715	0.4128
PC-GNN	0.6431	0.4251	0.6933	0.4656
GTAN	0.7932	0.6108	0.8321	0.6298
RGTAN	**0.8167***	**0.6325***	**0.8518***	**0.6504***

We conduct case studies in a world-leading card issuer to validate the performance of detecting typical fraud patterns. Specifically, we select all transactions matching end-of-chain fraud patterns (i.e., case 1 and case 2 illustrated in Figure 5.7) from manually annotated cases. Then, an equal number of legitimate transactions are randomly selected among all transactions as the legitimate samples of the two cases, respectively. Then, we calculate the AUC and AP of the predicted results in these two groups of transactions, respectively. Table 5.3 reports the performance of 5 methods in detecting the end of chains of two typical types of fraud. Based on the first four rows, GTAN far outperforms all previous baselines with 16.0% and 14.9% AUC improvements and 19.5% and 17.4% AP improvements. According to the last two rows of Table 5.3, the results show that our method RGTAN outperforms GTAN with 2.3% and 2.0% AUC improvements and 2.2% and 2.1% AP improvements. This demonstrates the effectiveness of our proposed RGTAN for identifying real-world human brain-armed credit card fraud patterns. The high capability in detecting complex fraud patterns may be the main source of performance gain of our substantial improvement approach.

5.6 Conclusion

This chapter addressed the crucial real-world issue of identifying credit card fraud. We devised an efficient semi-supervised technique that models data on temporal transaction graphs and employs attribute-driven gated temporal attention networks, given the laborious and costly nature of labeling fraud transactions. We proposed an attribute representation and risk propagation mechanism to capture the fraud patterns precisely, given the pervasive categorical attributes and human-annotated

labels. We innovatively leveraged neighborhood risk-aware representations to augment the capability of RGTAN to capture local risk information, given the importance of adjacent risk structures for fraud detection. The extensive experiments exhibited the superiority of our suggested methods over other baselines in three fraud detection datasets. Semi-supervised experiments illustrated the outstanding fraud detection performance of our model with only a minuscule fraction of manually annotated data. Moreover, we uncovered fraud patterns by examining the chain propagation among transactions in the case studies. The outcomes of case studies indicated that our models could detect real-world fraud patterns. Our approach has been on the way to becoming a fundamental component in a real-world credit card fraud analysis system hosted by world-leading credit card issuers, serving more than one hundred million users.

Chapter 6

Financial Time Series Prediction

6.1 Chapter Overview

In this chapter, we show the application of synthesized graphs on financial time series prediction tasks (specifically, univariate time series prediction). The company relation graphs are generated day-by-day to help promote the proposed graph neural networks for stock price movement prediction. This chapter has been published in [167]. Chapter 6.2 introduces the background of financial time series prediction tasks. Chapter 6.3 gives the related articles about graph data-driven stock prediction. Chapter 6.4 shows the process of graph generation in financial time series prediction. Chapter 6.5 introduces the main solutions of our proposed temporal and heterogeneous graph neural network (THGNN). Chapter 6.6 reports the performance comparison results among other graph-based methods. Chapter 6.7 concludes this chapter.

6.2 Background

Stock market is a financial ecosystem that involves transactions between companies and investors, with a global market capitalization of more than $83.5 trillion as of 2020 [168]. The Efficient Market Hypothesis [169] points out that the stock price represents all the available information, but the stock price is volatile in nature, resulting in difficulty on predicting its movement [170]. In recent years, deep learning has been widely used in predicting the price movement of stocks [171].

Researchers also explore to improve the prediction performance by incorporating more sources as model inputs, including more technical indicators [169], factors [172], financial status [173], online news [174], social media posts [175], etc.

Traditional learning methods treat the time series as independent and identically distributed to each other, which is not coincident to the real situation in the financial market [176]. For example, two stocks in the same sector may have a higher correlation than those in different fields [177]. Therefore, recent studies employ knowledge graphs to represent the internal relations among entities [178] and leverage graph learning for the price movement prediction [179]. These works have shown the effectiveness by integrating stock's relations in the prediction models [180]. Thus, graph-based methods could benefit from learning meaningful representations of inputs, resulting in better prediction accuracy [181, 182].

However, generating relation graphs of entities is very challenging because it is fluctuate, noisy, amphibolous and dynamically changed [183]. For example, financial knowledge graphs mainly include the supply chain, primary business and investment relations of entities [184], which is labeled by domain experts or extracted from unstructured texts. But different experts have different knowledge background which may lead to different relation graphs. Besides, the current nature language processing (NLP) techniques are still facing significant shortcomings in high accuracy relation extraction [185]. In other words, the relations may be misled by either unilateral text news or inaccurate extracting models. In addition, these relations may dynamically change in time series. For example, the main business of a company would change according to the market demands, and the supply chain graph would be upgraded because of the technique evolution [186]. Existing graph learning price prediction methods are inevitably suboptimal in learning these fluctuate and dynamical situations.

To address the above challenges, we propose a novel temporal and heterogeneous graph neural network-based method for financial time series prediction. Specifically, we directly model the relations of price time series of entities based on historical data and represent them in a temporal and heterogeneous graph, i.e., company relational graph. After obtaining the company relational graph, we leverage sequential transformers encode the historical prices and graph neural networks to encode internal relations of each company. Specifically, we update each company's representations by aggregating information from their neighbors in company relational graphs in two steps.

The first step is a time-aware graph attention mechanism. The second is a heterogeneous graph attention mechanism. We thoroughly evaluate our approach on both the S&P 500[1] and CSI 300[2] dataset in the United States and China's stock markets. The experimental results show that our method significantly outperforms state-of-the-art baselines. In order to keep the sententious of our model, we conduct ablation studies to prove the effectiveness and essential of the each component of our method, including transformer encoder, time-aware graph attention, heterogeneous graph attention. Finally, we deploy our model in real-world quantitative algorithm trading platform, hosted in EMoney Inc.[3], a leading financial service provider in China. The cumulative returns of portfolios contributed by our approach is significantly better than existing models in financial industry. We will release the dataset as well as the source codes of the proposed techniques along with this chapter. In conclusion, our principle contributions are summarized as follows:

- We propose a graph learning framework to effectively model the internal relations among entities for financial time series prediction, which fits the dynamical market status and is concordant with the ground-truth price movements.
- We design a temporal and heterogeneous graph neural network model to learn the dynamic relationships by two-stage attention mechanisms. The proposed model is concise and effective in joint and automatically learning from historical price sequence and internal relations.
- Our proposed THGNN is simple and can be easily implemented in the industry-level system. Extensive experiments on both the Unite States and China stock markets demonstrate the superior performance of our proposed methods. We also extensively evaluated its effectiveness by real-world trading platform.

6.3 Related Works

6.3.1 Financial Time Series Learning

It is widely known that price movements of stocks are affected by various aspects in financial market [172]. In previous studies, a common strategy is to manually construct various factors as

[1] https://www.spglobal.com/spdji/en/indices/equity/sp-500
[2] http://www.cffex.com.cn/en_new/CSI300IndexOptions.html
[3] http://www.emoney.cn/

feature inputs [187, 188]. For example, Michel et. al, [173] integrate market signals with stock fundamental and technical indicators to make decisions. Li et. al, [183] establish a link between news articles and the related entities for stock price movement forecasting. A large number of existing methods employ recurrent neural network and its variants, such as LSTM [65] and GRU [189], to learn the sequential latent features of historical information and employ them for downstream prediction task [190].

In these works, the market signals processing of each stock is carried out independently. However, this inevitably ignores the internal relationship among stocks and would lead to suboptimal performance. Some works [191] leverage the correlation information as model inputs, but cannot automatically capture the dynamic changes of relations. In this article, we model the relationship between stocks as dynamic company relation graphs and joint learn the graph relation and historical sequence feature automatically for future price movement prediction.

6.3.2 Graph Learning for Stock Prediction

Researchers have shown that the price movement of stocks is not only related to its own historical prices, but also connect to its linked stocks [180]. The link relation includes suppliers and customers, shareholders and investor, etc. Existing works normally employ knowledge graphs to store and represent these relations [187, 192]. Recently, graph neural network (GNN) [4] is proposed to effectively learn on graph-structured data, which has shown its superior performance in various domains, including fraud detection [83, 127], computer vision [193, 194], etc. Researchers also introduce the advanced GNN-based approaches in the stock price prediction task. For example, Chen et. al, [177] models the supply chain relationships of entities into knowledge graphs and uses graph convolution networks to predict stock movements. Ramit et. al, [182] leverage attentional graph neural network on the connections constructed by social media texts and company correlations. Cheng et.al, [187] leverage multi-modality graph neural network on the connections constructed by historical price series, media news, and associated events.

However, the graph constructed by these methods are limited by constant, predefined corporate relationships, which is powered by handcraft editing or nature language processing techniques,

suffering from heavy resources labeling and low extracting accuracy [195]. But the actual corporate diagram evolves frequently over time. Besides, the company relation graph is also heterogeneous, which means there are multiple relation types among entities. Therefore, existing methods cannot exploit the full information from real-life company relation graphs. In this chapter, we construct the relation graph dynamically and automatically based on their ground-truth historical price sequences and then propose a novel temporal and heterogeneous graph neural network methods to jointly learn their sequential and relational features for more accurate stock price prediction. We demonstrate the effectiveness of our methods by extensive experiments and real-world trading platform.

6.4 Temporal and Heterogeneous Graph Generation

6.4.1 Problem Definition

Different from traditional graph-based methods that construct static and homogeneous graphs by handcraft labeling or nature language processing techniques to infer stock movements, our model represents the company relation graph as a collection of temporal and heterogeneous graphs, which are automatically generated by historical price sequences. In graphs, the node denotes each equity and edge represents their relations in sequential. Temporal graphs are composed of timestamped edges and timestamped nodes [40, 109]. Each node might be associated with multiple relationships and multiple timestamped edges on different trading days. There are multiple types of edge $E = \{E_1, ..., E_r\}$ in company relation graphs. And the occurences of node $V = \{V^{t_1}, ..., V^T\}$ and edge $E = \{E^{t_1}, ..., E^T\}$ are different on different trading days. Table 6.1 summarizes the symbols introduced in this chapter.

Temporal and Relational Occurrence. *In a temporal company relation graph, an edge e is associated with a series of temporal occurrences $e = \{e^{t_1}, e^{t_2}, ...\}$, which indicate the occurrences of edge e at trading days $\{t_1, t_2, ...\}$ in the company relation graph. Each type of relational occurrence is associated with a series of temporal occurrences with $E = \{E_{r_1}, E_{r_2}, ...\}$, which indicate the occurrences of edge e in different relationships $\{r_1, r_2, ...\}$. Same as temporal occurrences of edge e, a node v is associated with a set of temporal occurrences with $v = \{v^{t_1}, v^{t_2}, ...\}$.*

Temporal and Heterogeneous Graph. *A temporal and heterogeneous company relation graph* $\tilde{G} = (\tilde{V}, \tilde{E})$ *is formed by a set of temporal nodes* $\tilde{V} = \{v_1^{t_{v_1}}, ..., v_n^{t_{v_n}}\}$ *and a series of sets of temporal edges* $\tilde{E} = \{\tilde{E}_1, ..., \tilde{E}_r\}$*, where* $\tilde{E}_r = \{e_1^{t_{e_1}}, ..., e_m^{t_{e_m}}\}_r$ *denotes the edges of relation* r*, and* $e_i^{t_{e_i}} = (u_{e_i}, v_{e_i})^{t_{e_i}}$ *denotes the temporal edge.*

In existing works [177, 180, 184], the graph neighborhood $\mathcal{N}(v)$ of node v is defined as static or homogeneous. Here, we generalize the definition of graph neighborhood to the temporal and heterogeneous graph, which is set as follows:

Temporal and Heterogeneous Graph Neighborhood. *Given a temporal node* v*, the neighborhood of* v *is defined as* $\mathcal{N}(v) = \{v_i | f_{sp}(v_i, v) \leq d_{\mathcal{N}}, |t_v - t_{v_i}| \leq t_{\mathcal{N}}\}$, where $f_{sp}(\cdot|\cdot)$ denotes the shortest path length between two nodes, $d_{\mathcal{N}}$ and $t_{\mathcal{N}}$ denote the hyper-parameters. As for heterogeneous graph, we define $\mathcal{N}_r(\cdot)$ as the neighborhood function of relation r.

Finally, we formally define the stock movement prediction problem as follows:

Input: *Historical price sequences of listed companies* $\mathbf{X} = \{x_1, x_2, \cdots, x_n\}$*, where each* $x_i = \{x_{i,1}, x_{i,2}, \cdots, x_{i,t}\}$ *denotes the historical price sequences. We then use the* $\mathbf{X}$ *to generate the temporal and heterogeneous company relation graph* $\tilde{G}$*, with multiple types of temporal edges* $\{\tilde{E}_{r_1}, \tilde{E}_{r_2}, ...\}$*, for downstream tasks .*

Output: *The probability* $\hat{\mathbf{Y}}$ *of price movements of each equity.*

6.4.2 Stock Correlation Graph Generation

In this subsection, we report the process of the temporal and heterogeneous graph construction. As mentioned in previous studies [177, 196], there may be multiple relationships between companies (such as suppliers and customers, shareholders and invested companies). Different from conventional knowledge graph-based relations that construct relation by human labeling or NLP techniques, generating relations directly based on market trend signals has proved to be effective [67, 183] in practical, which does not require additional ambiguity domain knowledge or text news sources, and is easy to be implemented.

Therefore, we need to use rule-based or learning-based approaches to generate the correlation graph. In this chapter, we obtain the correlation matrix by calculating the correlation coefficients

between ground-truth stock historical market signals directly. Then, the relationship between companies is determined according to the value of each element of the correlation matrix. The relationship between companies may be positive (correlation $>$ threshold) or negative (correlation $<$ -threshold). In order to reduce noise, we connect the edges whose absolute value is greater than the threshold, and the rest of the edges are considered not to be connected. So far, the edges E of company relation graph are generated. Therefore, we model the inter-company relation graph as a heterogeneous graph with two relationships, i.e., $G = (V, \{E_{r_1}, E_{r_2}, ...\})$, with $r \in \{pos, neg\}$.

As the relationships between companies tend to be dynamic, we generate the company relation graph in temporal format as our model's inputs. In particular, within T trading days, we generate temporal and heterogeneous company relation graphs with T timestamps, which is formulated as $\tilde{G} = (\tilde{V}, \{\tilde{E}_{pos}, \tilde{E}_{neg}\})$. Finally, the generated graphs and original sequence inputs are fed to downstream learning task simultaneously.

6.5 Temporal and Heterogeneous Graph Neural Networks

In this section, we introduce the framework of our proposed temporal and heterogeneous graph neural network and their each component in detail. Our model takes the historical price sequence as inputs and infer the probability of stock movements as outputs. We represent the relation of stocks in a dynamic heterogeneous graph with two types of edges. We then jointly encode the historical and relation features by transformers and heterogeneous graph attention network. We report the problem definition first and each module of our method in turn.

6.5.1 Historical Price Encoding

The input stock movement feature of price sequences is defined as $\mathbf{X}^t \in \mathbb{R}^{n \times T \times d_{feat}}$ on trading day t, where n denotes the number of stocks, T means the number of trading days before t, and d_{feat} denotes the dimension of historical price features. We first leverage a linear transformation and positional encoding (PE) [144, 197] on trading features to obtain the input tensor

TABLE 6.1: The summary of symbols

Symbol	Definition
$\mathbf{X}$	the historical price of listed companies
$\hat{\mathbf{Y}}$	the probability of price movements
n	the total number of nodes
m	the total number of edges
r	the number of relationships in graph $\tilde{G}$
T	the number of trading days in $\tilde{G}$
$\tilde{G} = (\tilde{V}, \{\tilde{E}_{r_1}, \tilde{E}_{r_2}, ...\})$	the temporal and heterogeneous graph
$\tilde{V} = \{v_1^{t_{v_1}}, ..., v_n^{t_{v_n}}\}$	the set of temporal nodes
$\tilde{E}_r = \{e_1^{t_{e_1}}, ..., e_m^{t_{em}}\}_r$	the set of temporal edges of relation r
$\mathcal{N}(\cdot)$	the neighborhood function
d	the number of dimension

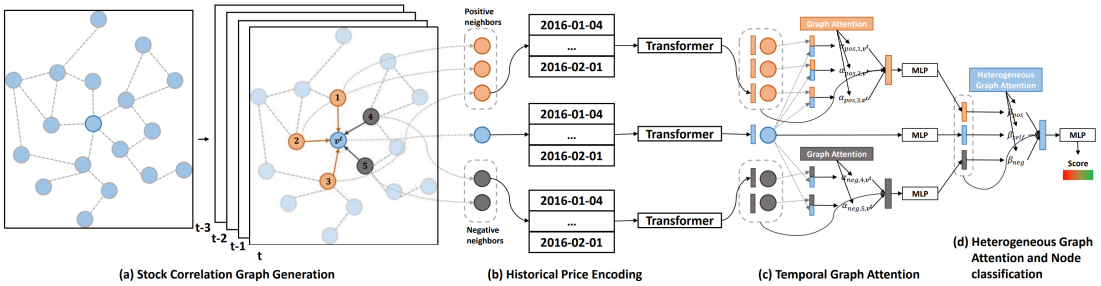


FIGURE 6.1: The proposed Temporal and Heterogeneous Graph Neural Networks architecture for stock movement predictions. The first part is the generation of a stock correlation graph, which builds dynamic relations for stocks in the market every trading day. The second part is the historical price encoding, which selects a temporal node v^t and its neighbor nodes to encode the historical price information. Transformer encoders share their parameters. The third part is the graph attention layer, which adaptively calculates the importance of the neighbors and aggregates the information according to the neighbors' importance. The fourth part is the heterogeneous graph attention layer, which adaptively calculates the importance and aggregates information from different types of neighbors. Then, we leverage a multi-layer perception to give the prediction of each stock's future movement.

$\mathbf{H}^t \in \mathbb{R}^{n \times T \times d_{in}}$, which is formulated as follows:

$$\begin{aligned} \hat{\mathbf{H}}^t &= \mathbf{W}_{in}\mathbf{X}^t + \mathbf{b}_{in} \\ \mathbf{H}^t &= \hat{\mathbf{H}}^t + \text{PE} \\ \text{PE}(p, 2i) &= sin(p/10000^{2i/d_{in}}) \\ \text{PE}(p, 2i+1) &= cos(p/10000^{2i/d_{in}}) \end{aligned} \tag{6.1}$$

where $p \in \{1, 2, .., T\}$ is the trading day position, i is the dimension, d_{in} denotes the dimension of input features, and $\mathbf{W}_{in} \in \mathbb{R}^{d_{feat} \times d_{in}}$ and $\mathbf{b}_{in} \in \mathbb{R}^{d_{in}}$ denote the learnable parameters. After linear transformation, we proposed to leverage multi-head attentional transformer to encode the input feature for each stock in each day. Then, the proposed encoder outputs $\mathbf{H}_{enc}^t \in \mathbb{R}^{n \times T \times d_{enc}}$ for downstream tasks, where d_{enc} denotes the output dimension of the encoder. Mathematically, we formulate the historical feature encoder's output $\mathbf{H}_{enc}^t$ as follows:

$$\mathbf{H}_{enc}^t = \text{Concat}(\text{EncHead}_1^t, \ldots, \text{EncHead}_{h_{enc}}^t)\mathbf{W}_o \tag{6.2}$$

where $\mathbf{W}_o \in \mathbb{R}^{h_{enc} d_v \times d_{enc}}$ denotes the output projection matrix, h_{enc} denotes the number of heads in the encoder, d_v denotes the output dimension of each head, Concat denotes a concatenation of the output of heads, and $\text{EncHead}_i^t \in \mathbb{R}^{n \times T \times d_v}$ denotes the output of encoder head with $\text{EncHead}_i^t = \text{Attention}(\mathbf{Q}_i^t, \mathbf{K}_i^t, \mathbf{V}_i^t)$, which is formulated as follows:

$$\begin{aligned} \mathbf{Q}_i^t = \mathbf{H}^t \mathbf{W}_i^Q, \mathbf{K}_i^t =& \mathbf{H}^t \mathbf{W}_i^K, \mathbf{V}_i^t = \mathbf{H}^t \mathbf{W}_i^V \\ \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) =& \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{in}}})\mathbf{V} \end{aligned} \tag{6.3}$$

where $\mathbf{W}_i^Q \in \mathbb{R}^{d_{in} \times d_{hidden}}$, $\mathbf{W}_i^K \in \mathbb{R}^{d_{in} \times d_{hidden}}$, $\mathbf{W}_i^V \in \mathbb{R}^{d_{in} \times d_v}$ denote the projection matrices, and d_{hidden} denotes the dimension of hidden layer.

6.5.2 Temporal Graph Attention Mechanism

Given the historical sequence encoder output $\mathbf{H}_{enc}^t$ and temporal relation graph $\tilde{G}$, we propose to employ graph attention mechanism on the sequential and heterogeneous inputs. In particular, we flatten the embeddings of all nodes to $\mathbf{H}_{enc}^t \in \mathbb{R}^{n \times T d_{enc}}$ and leverage two-stage temporal attention mechanism to aggregate messages from graph structures and temporal sequences, which is illustrative reported in Figure 6.1 (c). The two-stage temporal graph attention layers could aggregate messages from both the postive and negative neighbors simultaneously. For each relationship $r \in \{pos, neg\}$, the message aggregating is formulated as follows:

$$\mathbf{H}_r^t = \text{Concat}(\text{TgaHead}_1, ..., \text{TgaHead}_{h_{tga}})\mathbf{W}_{o,r} \tag{6.4}$$

where $\mathbf{H}_r^t \in \mathbb{R}^{n \times d_{att}}$ denotes the output of the temporal graph attention layer on trading day t, and $\mathbf{W}_{o,r} \in \mathbb{R}^{h_{tga} T d_{enc} \times d_{att}}$ denotes the output projection matrix, h_{tga} denotes the number of heads, and each head of temporal graph attention layer $\text{TgaHead}_i \in \mathbb{R}^{n \times T d_{enc}}$ is formulated as follows:

$$\text{TgaHead}_i = \sum_{v^t \in \tilde{V}} \sigma\Big(\sum_{u^t \in \mathcal{N}_r(v^t)} \alpha^i_{u^t, v^t} \mathbf{h}_{u^t} \Big) \tag{6.5}$$

where σ denotes the activation function, $\mathbf{h}_{u^t} \in \mathbb{R}^{T d_{enc}}$ denotes the u^t-th row of historical price embedding $\mathbf{H}^t_{enc}$, and $\alpha^i_{u^t, v^t}$ denotes the importance of temporal edge (u^t, v^t) in i-th head, which is formulated as follows:

$$\alpha^i_{u^t, v^t} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T_{r,i}[\mathbf{h}_{u^t} || \mathbf{h}_{v^t}]))}{\sum_{k^t \in \mathcal{N}_r(v^t)} \exp(\text{LeakyReLU}(\mathbf{a}^T_{r,i}[\mathbf{h}_{k^t} || \mathbf{h}_{v^t}]))} \tag{6.6}$$

where $\mathbf{a}_{r,i} \in \mathbb{R}^{2 T d_{enc}}$ denotes the weight vector of relation r and i-th head.

6.5.3 Heterogeneous Graph Attention Mechanism

As shown in Figure 6.1 (d), we already have messages from different types of neighbors through the two-stage attention mechanism. Then, we propose the heterogeneous graph attention network to learn from different relationships in relation graphs.

We define message sources as three types of embeddings, namely, messages from ourselves $\mathbf{H}^t_{self}$, positive neighbors $\mathbf{H}^t_{pos}$, and negative neighbors $\mathbf{H}^t_{neg}$, respectively. $\mathbf{H}^t_{self} \in \mathbb{R}^{n \times d_{att}}$ is derived from $\mathbf{H}^t_{enc}$ through a linear transformation with $\mathbf{H}^t_{self} = \mathbf{W}_{self} \mathbf{H}^t_{enc} + \mathbf{b}_{self}$, where $\mathbf{W}_{self} \in \mathbb{R}^{T d_{enc} \times d_{att}}$ and $\mathbf{b}_{self} \in \mathbb{R}^{d_{att}}$ denote the learnable parameters. $\mathbf{H}^t_{pos}$ and $\mathbf{H}^t_{neg}$ are derived from the graph attention mechanism in section 6.5.2. Taking three groups of node embeddings as input, we can adaptively generate the importance of different relationships through attention mechanism. The weights of three relationships $(\beta_{self}, \beta_{pos}, \beta_{neg})$ can be shown as follows:

$$(\beta_{self}, \beta_{pos}, \beta_{neg}) = \text{HGA}(\mathbf{H}^t_{self}, \mathbf{H}^t_{pos}, \mathbf{H}^t_{neg}) \tag{6.7}$$

We first use three Multi-Layer Perceptrons (MLP) to transform these three embeddings. Then we measure the importance of each embedding using a heterogeneous attention vector $\mathbf{q}$. Furthermore, we average the importance of all node embeddings, which can be explained as the importance of each company relation. The importance of each company relation, denoted as $r \in \{self, pos, neg\}$, is shown as follows:

$$w_r = \frac{1}{|\tilde{V}|} \sum_{v^t \in \tilde{V}} \mathbf{q}^T \tanh(\mathbf{W}\mathbf{h}_{v^t,r} + \mathbf{b}) \tag{6.8}$$

where $\mathbf{W} \in \mathbb{R}^{d_{att} \times d_q}$ and $\mathbf{b} \in \mathbb{R}^{d_q}$ are the parameters of MLP, $\mathbf{q} \in \mathbb{R}^{d_q}$ denotes the attention vector, and $\mathbf{h}_{v^t,r}$ denotes the v_t-th row of $\mathbf{H}_r^t$. Note that all above parameters are shared for all relationship of node embeddings. After obtaining the importance of each relationship, we calculate the contribution of each relationship and obtain the final embedding $\mathbf{Z}^t \in \mathbb{R}^{n \times d_{att}}$ as follows:

$$\begin{aligned} \beta_r &= \frac{\exp(w_r)}{\sum_{r \in \{self,pos,neg\}} \exp(w_r)} \\ \mathbf{Z}^t &= \sum_{r \in \{self,pos,neg\}} \beta_r \cdot \mathbf{H}_r^t \end{aligned} \tag{6.9}$$

For a better understanding of the aggregating process of heterogeneous graph attention layer, we give a brief explanation in Figure 6.1 (d). Then we apply the final embedding to a semi-supervised node classification task.

6.5.4 Optimization Objectives

Here we give the implementation of objective functions. We model the stock movement prediction task as a semi-supervised node classification problem. Specifically, we selected 200 stocks of which future movements are ranked in top-100 or bottom-100, and labeled the corresponding nodes as 1 and 0, respectively. Then, we use one layer of MLP as the classifier to get the classification results of labeled nodes. Furthermore, we use binary cross-entropy to calculate the objective function L, which is formulated as follows:

$$\begin{aligned} \hat{\mathbf{Y}} &= \sigma(\mathbf{W}\mathbf{Z}_l^t + \mathbf{b}) \\ \mathcal{L} &= -\sum_{l \in \mathcal{Y}_t} [\mathbf{Y}_l^t \log(\hat{\mathbf{Y}}_l) + (1 - \mathbf{Y}_l^t) \log(1 - \hat{\mathbf{Y}}_l)] \end{aligned} \tag{6.10}$$

where $\mathcal{Y}_t$ denotes the set of labeled nodes, $\mathbf{Y}_l^t$ and $\mathbf{Z}_l^t$ denote the label and embedding of the labeled node l, respectively, σ denotes the Sigmoid activation function, and $\mathbf{W}$ and $\mathbf{b}$ are the parameters of MLP. With the guide of labeled data, we use Adam [198] optimizer to update the parameters of our proposed method. Please note that we use this objective function to jointly optimize the parameters of historical price encoder, temporal and heterogeneous graph neural network and node classifier.

6.6 Experiments

In this section, we first introduce the datasets and experimental settings. Then we detail report the experimental results in real-world dataset and applications.

6.6.1 Experimental Setttings

Datasets. Extensive experiments are conducted in both the Unite States and Chinna's stock markets by choosing the constituted entities in S & P 500 and CSI 300 index. The historical price data from 2016 to 2021 are chosen as our datasets. In addition to historical price data, our input data also includes company relation graphs. The graphs are generated by the stock price correlation matrix, which is introduced in Section 6.4.2. The stock price correlation matrix of each day is determined by the historical price movement of past 20 trading days. Specifically, we compare the opening price, closing price, trading volume, and trading volume of each pair of two stocks, calculate the correlation coefficient between them and take the mean value as the element of the correlation matrix.

Parameter Settings. Our temporal graph $\tilde{G}$ contains company relationships for 20 trading days. The $d_{\mathcal{N}}$ and $t_{\mathcal{N}}$ of neighborhood function $\mathcal{N}(\cdot)$ are both set as 1. During the graph generation process, the threshold for generating one edge is set as 0.6. The historical price data of the previous 20 trading days are used as input features. The feature dimension d_{feat} of the encoding layer is 6, the input dimension d_{in} and encoding dimension d_{enc} of the encoding layer are both 128. The hidden dimension d_{hidden} is 512, the dimension of value d_v is 128, and the number of heads h_{enc} is 8. In temporal graph attention layer, d_{att} is 256, and the number of heads h_{tga} is 4. The dimension of the attention vector in the heterogeneous graph attention layer, d_q, is 256.

TABLE 6.2: Performance evaluation of compared models for financial time series prediction in S&P 500 and CSI 300 datasets. ACC and ARR measure the prediction performance and portfolio return rate of each prediction model, respectively, where the higher is better. AV and MDD measure the investment risk of each prediction model where the lower absolute value is better. ASR, CR, and IR measure the profit under a unit of risk, where the higher is better.

	S&P 500							CSI 300						
	ACC	ARR	AV	MDD	ASR	CR	IR	ACC	ARR	AV	MDD	ASR	CR	IR
LSTM	0.532	0.377	0.449	-0.382	0.842	0.989	0.954	0.515	0.291	**0.318**	-0.240	0.915	1.213	0.877
GRU	0.530	0.362	**0.445**	-0.379	0.813	0.955	0.934	0.517	0.312	0.320	-0.243	0.975	1.284	0.932
Transformer	0.533	0.385	0.454	-0.384	0.848	1.005	0.960	0.518	0.327	0.322	-0.245	1.016	1.335	0.969
eLSTM	0.534	0.434	0.454	-0.373	0.955	1.163	1.041	0.520	0.330	0.323	-0.239	1.022	1.381	0.991
LSTM+GCN	0.538	0.470	0.442	-0.354	1.062	1.326	1.103	0.523	0.351	0.320	**-0.217**	1.097	1.618	1.119
LSTM+RGCN	0.565	0.558	0.463	-0.366	1.205	1.522	1.203	0.536	0.509	0.326	-0.235	1.561	2.166	1.537
TGC	0.552	0.528	0.455	**-0.344**	1.163	1.535	1.180	0.531	0.453	0.323	-0.224	1.402	2.022	1.412
MAN-SF	0.551	0.527	0.467	-0.357	1.130	1.478	1.157	0.527	0.418	0.334	-0.225	1.251	1.858	1.282
HATS	0.541	0.494	0.466	-0.387	1.060	1.277	1.110	0.525	0.385	0.332	-0.249	1.160	1.546	1.116
REST	0.549	0.502	0.466	-0.359	1.079	1.398	1.117	0.528	0.425	0.331	-0.228	1.284	1.864	1.298
AD-GAT	0.564	0.535	0.457	-0.371	1.170	1.444	1.187	0.539	0.537	0.329	-0.240	1.632	2.238	1.596
THGNN	**0.579**	**0.665**	0.468	-0.369	**1.421**	**1.804**	**1.340**	**0.551**	**0.632**	0.336	-0.237	**1.881**	**2.667**	**1.875**

Trading Protocols. On the basis of [199], we use the daily buy-hold-sell trading strategy to evaluate the performance of stock movement prediction methods in terms of returns. During each trading day during the test period (from January 1, 2020 to December 31, 2020), we use simulated stock traders to predict transactions:

1. When the trading day t closes, traders use this method to get the prediction score, that is, the ranking of the predicted rate of return of each stock.
2. When the trading day $t + 1$ opens: the trader sells the stock bought on the trading day t. Meanwhile, traders buy stocks with high expected returns, i.e., the stocks with top-k scores.
3. Please note that if a stock is continuously rated with the highest expected return, the trader holds the stock.

In calculating the cumulative return on investment, we follow several simple assumptions:

1. Traders spend the same amount on each trading day (for example, \$50,000). We made this assumption to eliminate the time dependence of the testing process in order to make a fair comparison.
2. There is always enough liquidity in the market to satisfy the opening price of the order on the $t + 1$ day, and the selling price is also the opening price on the $t + 1$ day.

3. Transaction costs are ignored because the cost of trading US stocks through brokers is relatively cheap, whether on a transaction-by-transaction basis or on a stock-by-stock basis.Fidelity Investments (Fidelity Investments) and Interactive Brokerage (Interactive Broker), for example, charge $4.95 and $0.005 per transaction, respectively.

Compared Baselines. We compared our proposed method with state-of-the-art sequential-based models as well as the graph-based approaches. They are 1) Non-graph-based Methods, includes LSTM [65], GRU [189], Transformer [144], eLSTM [171]. 2) Graph-based Methods: LSTM-GCN [177], LSTM-RGCN [183], TGC [184], MAN-SF [182], HATS [196], REST [200] and AD-GAT [180].

Evaluating Metrics. Since the goal is to accurately select the stocks with highest returns appropriately, we use seven metrics: prediction accuracy (ACC), annual return rate (ARR), annual volatility (AV), maximum draw down (MDD), annual sharpe ratio (ASR), calmar ratio (CR), and information ratio (IR) to report the performance of baselines and our proposed model. Prediction accuracy is widely used to evaluate classification tasks, such as stock movement prediction [171, 201], so we calculate the prediction accuracy of all stocks for each trading day during the test period. Because ARR directly reflects the effect of stock investment strategy, ARR is our main measure, which is calculated by adding up the returns of selected stocks on each test day in a year. AV directly reflects the average risk of investment strategy per unit time. MDD reflects the maximal draw down of investment strategy during the whole test period. ASR reflects the benefits of taking on a unit of volatility with $ASR = \frac{ARR}{AV}$. CR reflects the benefits of taking on a unit of draw down with $CR = \frac{ARR}{\text{abs}(MDD)}$. IR reflects the excess return under additional risk. The smaller the absolute value of AV and MDD is, the higher the value of ACC, ARR, ASR, CR, and IR is, the better the performance is. For each method, we repeated the test process five times and reported average performance to eliminate fluctuations caused by different initialization.

6.6.2 Financial Prediction

In this section, we evaluate the performance of financial time series prediction and portfolio, which is the main task of this chapter. Table 6.2 reports the performance through evaluation metrics such as ACC and ARR for each method in two datasets. The first four rows of Table 6.2 show the

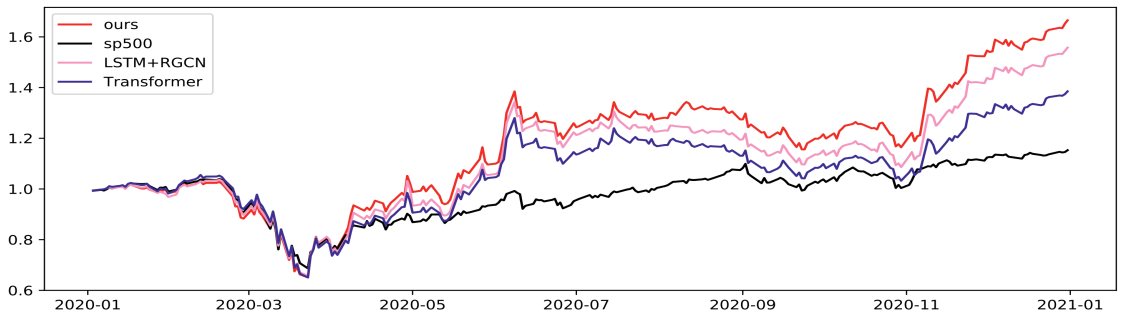


FIGURE 6.2: The accumulated returns gained in the test set (2020) by our proposed THGNN and selected baselines. For better illustration, we select one baseline from non-graph-based model and graph-based model, respectively.

TABLE 6.3: Performance evaluation of ablated models for financial time series prediction in S&P 500 dataset. ACC, ARR, ASR, CR, and IR measure the prediction performance and portfolio return rate of each prediction model, where the higher is better.

	ACC	ARR	ASR	CR	IR
THGNN-noenc	0.548	0.571	1.201	1.524	1.082
THGNN-notemp	0.539	0.486	0.964	1.292	0.946
THGNN-nohete	0.553	0.600	1.279	1.618	1.198
THGNN	**0.579**	**0.665**	**1.421**	**1.804**	**1.340**

performance of models that do not use graph-based technology. It it clear that, none of these four methods is satisfactory, and their performance is all lower than that of other baselines. This proves that models without using company relationship data cannot achieve optimal performance.

Lines 5 to 11 of Table 6.2 show the performance of the baseline models using graph-based technology. According to line 6, LSTM+RGCN performs best. This proves the effectiveness of using heterogeneous graphs of inter-company relationships. Note that according to line 7, TGC's performance is also competitive and its investment strategy is less volatile. This proves the effectiveness of using the dynamic relationship between companies.

According to the previous observation, the financial prediction model can be improved by using the heterogeneous or dynamic relationships of the company relation graph. Therefore, it is necessary to design an innovative model to improve the prediction performance of financial series from both dynamic and heterogeneous graph structure. According to the last row of Table 6.2, our proposed THGNN outperforms all baselines and proves the superiority of temporal and heterogeneous graph neural network in financial time series prediction.

6.6.3 Ablation Study

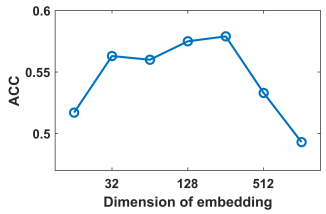
In this section, we conduct the ablation experiments, that is, evaluating the performance of our methods that one part is dropped.

According to the first row of Table 6.3, THGNN-noenc can not achieve the best performance after dropping the historical price encoding layer. This is because the encoder is responsible for extracting the time correlation in the historical price information. According to the second row, THGNN-notemp achieves unsatisfactory performance after dropping the temporal graph attention layer. This is because the temporal graph attention layer is responsible for dynamically adjusting the relationship between companies. Moreover, the relationship between companies changes dynamically over time, especially over a long period of time. According to the third row, THGNN-nohete cannot achieve the best performance after dropping the heterogeneous attention mechanism. This is because the heterogeneous graph attention layer is responsible for weighing the importance of messages from different relationships.

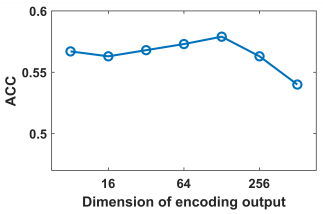
6.6.4 Performance of the Portfolio

In the performance evaluation of the portfolio strategy, we reported 6 widely-used evaluating metrics for portfolio, e.g., the annualized rate of return (ARR), annual sharp ratio (ASR), calmar ratio (CR), and information ratio (IR). Then, we show the accumulative return curve to compare the investment portfolio of our model and baselines during the test period. According to the trading protocols mentioned by section 6.6.1, we use the output of the prediction model to adjust our position day by day.

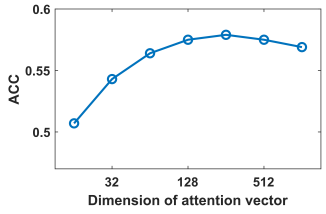
Table 6.2 reports the performance of our model's and baselines' portfolio strategies. It is clear that our method has achieved the best performance in four of the six investment evaluating metrics. Specifically, our method performs best in terms of ARR, ASR, CR and IR. TGC and LSTM+GCN perform better than our model in terms of AV and MDD. This shows that our proposed THGNN takes the initiative to take more risks to pursue higher risk-return ratio than other baselines'. According to Table 6.3, THGNN outperforms its sub-models in terms of portfolio strategy return evaluating metrics (e.g., ARR, ASR, CR, and IR). Therefore, our model shows strong effectiveness in building profitable portfolio strategies.



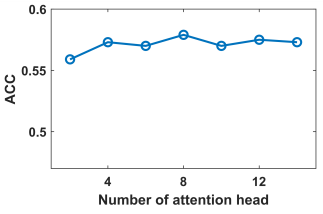
(a) Dimension of final embedding d_{att}



(b) Dimension of encoding output d_{enc}



(c) Dimension of the attention vector $\mathbf{q}$



(d) Number of attention head h

FIGURE 6.3: Prediction performance of THGNN in terms of dimension of final embedding d_{att}, dimension of encoding output d_{enc}, dimension of the attention vector $\mathbf{q}$, and number of attention head h.

In order to further establish and evaluate the returns of our model during the test time, we calculate the cumulative returns for each trading day during the test period. We report the cumulative return curve of our model and other models on each trading day in Figure 6.2. Due to space constraints, we select some representative baselines to compare with our model. We can observe that all baselines outperform the S&P500 index. None of the models beat the others in the first four months. After starting in May, our model began to stay ahead of other models. In the following time, our model gradually widened the gap with other models. THGNN remained in the lead in the last month of 2020 and eventually achieved a profit on investment of more than 60 per cent. Experimental results on other baselines can draw similar conclusion.

6.6.5 Parameter Sensitivity

In this section, we report the experimental results of parameter sensitivity on the financial prediction task on S&P500 dataset with various parameters in Figure 6.3.

According to Figure 6.3 (a), we can observe that the performance of the model increases with the increase of embedded dimensions. The performance of the model peaked at 256 and then degraded rapidly. This is because embedded information needs a suitable dimension to reduce information loss and noise. According to Figure 6.3 (b), the performance of the model slowly

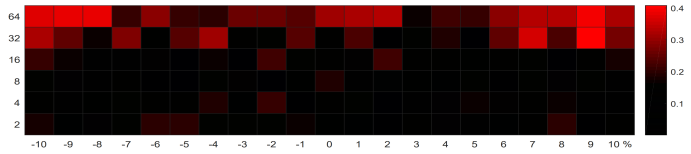
improves as the coding output dimension increases. The performance of the model begins to deteriorate after the dimension reaches 128. This is because the input information dimension is low, so the low-dimensional output can also achieve better performance. According to Figure 6.3 (c), the performance of the model increases with the increase of the attention vector dimension. When the dimension reaches 256, the performance of the model reaches its peak. Continuing to increase the dimension will lead to overfitting, which will degrade the model. According to Figure 6.3 (d), we can observe that the fluctuation of the model performance is low. We choose to pay attention to the number of force heads as 8. We also note that increasing the number of attention heads will make the model training more stable.

6.6.6 Interpretability of Graph Neural Network

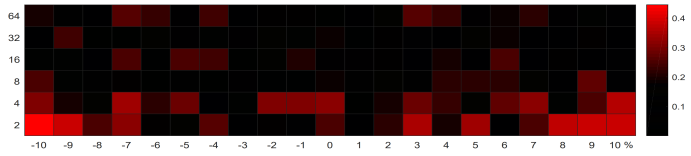
The stock price fluctuation of listed companies is not only related to the historical price information of the company, but also affected by the companies related to it. Therefore, we need to integrate dynamic and heterogeneous relevant information as input to the prediction model. In our technology, the relationship between each company changes dynamically over time. The strength of the relationship between companies, that is, the proportion of contribution to the message also changes over time. Our time chart attention mechanism can dynamically adjust the importance of each company in the diagram. In addition, our heterogeneous attention mechanism can dynamically adjust the importance of each source. Therefore, our model can help to predict stock price volatility more accurately, and the experimental results verify the superiority of our model performance.

Then, in order to explore the interpretability of our proposed model, we extract the attention weight of the graph in the process of the model prediction. We counted the attention weight of all nodes in the process of message delivery on the relational graph. Under different daily returns and node degrees, we take the mean value of attention weights and visualize the statistical results, which are reported in Figure 6.4.

Figure 6.4 (a) shows the attention weights on the pos diagram. We can see (on the y-axis) that the nodes with higher degrees have higher average attention weights. This shows that in the process of message delivery on the pos graph, the degree is higher, that is, the nodes with more neighbors will contribute more messages to their neighbors. We also found that (on the x-axis), companies

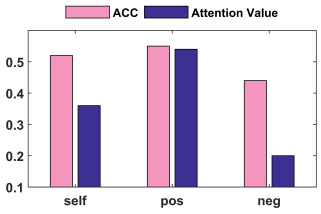


(a) Graph Attention in pos Relationship

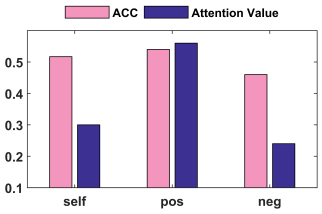


(b) Graph Attention in neg Relationship

FIGURE 6.4: Visualization of attention weights, X-axis denotes the daily return of stocks. Y-axis denotes the average degree in each company relation graph.



(a) ACC values on S&P500.



(b) ACC values on CSI300.

FIGURE 6.5: Prediction performance of single message source and corresponding attention value.

with larger fluctuations in daily returns also had higher average attention weights. This shows that more volatile price changes will contribute more information to their neighbors, which also means that price fluctuations will produce momentum spillover effects. According to Figure 6.4 (b), we can see that on the neg graph, with a lower degree node, the average attention weight is higher. This indicates that during message delivery on the neg graph, nodes with lower degrees contribute more messages to their neighbors.

For more interpretable experimental results, we also visualize each relationship's attention weights and show the corresponding performance when using only one relationship. Specifically, we trained our proposed THGNN at two datasets, and counted the mean value of the attention weights in heterogeneous graph attention layer. Then, we used each single relationship's message as the input of the prediction model to obtain the prediction performance, as illustrated in Figure 6.5.

It is clear that $self$ and pos message resources achieve better prediction performance than neg. Moreover, the pos message resource contribute importantly to the prediction model, which proves the contribution to the prediction model. The reason might be that the influence between companies in the similar price movement is relatively useful for prediction future price movement.

(a) List of selected stocks

(b) Historical price and buy/sell signals

(c) Related listed companies

FIGURE 6.6: The desktop interface of investment portfolio based on our proposed THGNN method. It includes price-relevant listed companies from historical data and visualization of how does our method makes predictions on buying and selling. The part (a) lists the stocks held by our method in order from highest to lowest. And part (b) shows the 'buy' and 'sell' signals generated by our trading protocols. Then part (c) lists the listed companies related to China National Petroleum Corporation (CBPC: 601857) and shows which ones have higher correlation to this company according to our generated stock correlation graph.

Although the neg message resource has an unsatisfactory performance when predicting price movement single, it still contributes to our proposed THGNN in achieving the state-of-the-art performance according to Table 6.3. We can see that temporal graph attention layer can reveal the difference between nodes and weights then adequately, and the heterogeneous graph attention can adjust the contribution of each message resource adaptively. The result demonstrates the effectiveness of graph-structure information and the interpretability of proposed graph neural network model.

6.6.7 System Implementation

In this section, we introduce the implementation detail of our proposed methods. We first show the settings of model employment and training strategy. Then we show the web-based application, which shows how our proposed method gives customers informative advice. Our proposed THGNN is re-trained every trading day. To handle the large-scale dataset, we leverage mini-batch

gradient descent optimization with a batch size of 256 and a learning rate of 3e-4. The model is implemented by PyTorch and deployed in Python and Java. Besides, we use distributed Scrapy [202] to obtain historical stock data and utilize Neo4j [203] as the graph database to store relational graphs.

Figure 6.6 shows the interface of our desktop application. The upper left part of Figure 6.6 is the list of stocks to be held based on the THGNN strategy. The lower left part of Figure 6.6 reports the price change curve of China National Petroleum Corporation (CBPC: 601857). It contains buy and sell points suggested by our THGNN. B denotes buying and S denotes selling. It can be seen that our model provides three buy signals and two sell signals. The lower right part of Figure 6.6 reports the relevant companies of CBPC. Companies with high stock price volatility correlations are marked with red arrowhead. The results shows that our investment strategy provides informative advice through a relational graph approach.

6.7 Conclusion and Discussion

In this chapter, a new temporal and heterogeneous graph neural network model is proposed for financial time series prediction. Our method addresses the limitations of the existing works of graph neural networks by adjusting the message contribution ratio of each node through temporal and heterogeneous graph attention mechanism. We evaluate the effectiveness of the proposed method comprehensively by comparing it with the most influential graph-based and non-graph-based baselines. In addition, THGNN performs better than other baselines in the actual investment strategy, and the results show that our approach based on dynamic heterogeneous graph can obtain a more profitable portfolio strategy than those based on static or homogeneous graph structure.

In conclusion, this chapter is the first time to model the inter-company relationship into a heterogeneous dynamic graph, and apply it to the financial time series prediction problem. This is beneficial to the more extensive research and innovation of graph-based technology in the financial field. On the one hand, we model the company relation graph as the real dynamic heterogeneous graph; on the other hand, we improve the financial time series prediction model through the latest graph neural network technology. Besides, there is still room for improvement in our work on generating real-life company relation graphs. In the future, we will focus on improving the modeling

of the company's relation to help the prediction model obtain more accurate training input graph data.

Chapter 7

EPILOGUE

In this thesis, we explored and addressed the challenges of efficiently generating graphs with learning-based methods, and promoting generated graph data into downstream tasks. We started by introducing the concept of graph generation. Then we propose a benchmark of graph generative models utilizing model-based or deep learning-based approaches. Based on our graph generation benchmark, we find the research gap, that most previous researchers focus on simulating large graphs with simple models or generating high-quality small graphs with deep learning models. In other words, there are few works on finding a trade-off between efficiency and performance when generating realistic graphs with learning-based methods.

In the first part of our thesis, we proposed three solutions for efficient learning-based graph generation. The first solution (chapter two) is to achieve a trade-off between efficiency and performance, which is the first learning-based method to generate a realistic graph with one million nodes. The second one (chapter three) addresses the challenge of simulating large graphs with community structures. The third one (chapter four) focuses on the learning-based solution for simulating temporal graphs, which achieves state-of-the-art performance on both efficiency and generative quality.

In the second part of our thesis, we extended our learning-based approach to focus on downstream applications, such as enhancing attribute-driven fraud detection with risk-aware graph representation (chapter five), and temporal and heterogeneous graph neural networks for financial time series

prediction (chapter six). These techniques showcased the usage of generated graphs in tackling complex, real-world issues beyond graph generation.

A crucial element of this thesis was our emphasis on scalability. Our developed methods bypass the limitations of existing deep learning-based techniques, allowing efficient generation of large-scale graphs. Besides, the properties of real-world graphs, such as community structure and dynamics are important to capture. The works described in this thesis, encapsulated in a series of published journal articles and conference papers, have demonstrated superior performance compared to existing state-of-the-art graph generation techniques in terms of efficiency and quality. In the future, the field of learning-based graph generation is still ripe for further exploration. Future research directions include generating graphs with edge weights, refining our models for additional applications, exploring other deep learning architectures, and developing methods to handle even larger graph structures. The current work lays a solid foundation for these future investigations, pushing the boundaries of what is possible in learning-based large graph generation. In conclusion, this thesis presents significant contributions to the field of learning-based graph generation, fostering advancements in the simulation of realistic network structures and stimulating novel approaches for data representation and analysis in numerous applications.

Bibliography

[1] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr. Graph generators: State of the art and open challenges. *ACM Comput. Surv.*, 2020.

[2] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[3] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, and Bo Long. Graph neural networks for natural language processing: A survey. *CoRR*, 2021.

[4] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[5] Feng Xia, Ke Sun, Shuo Yu, Abdul Aziz, Liangtian Wan, Shirui Pan, and Huan Liu. Graph learning: A survey. *CoRR*, abs/2105.00696, 2021.

[6] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 34:249–270, 2022.

[7] Tengfei Ma, Jie Chen, and Cao Xiao. Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *NeurIPS*, page 7113–7124, 2018.

[8] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *ICANN*, pages 412–422. Springer, 2018.

[9] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS*, page 6412–6422. Curran Associates Inc., 2018.

[10] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *ICML*, pages 2328–2337, 2018.

[11] Garry Robins, Pip Pattison, Yuval Kalish, and Dean Lusher. An introduction to exponential random graph (p*) models for social networks. *Social networks*, pages 173–191, 2007.

[12] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *SIGKDD*, pages 701–710. ACM, 2014.

[13] Christopher L. Barrett, Richard J. Beckman, Maleq Khan, V. S. Anil Kumar, Madhav V. Marathe, Paula Elaine Stretz, Tridib Dutta, and Bryan L. Lewis. Generation and analysis of large synthetic social contact networks. In *WSC*, pages 1003–1014. IEEE, 2009.

[14] Julia Stoyanovich, Bill Howe, and H. V. Jagadish. Responsible data management. *Proc. VLDB Endow.*, pages 3474–3488, 2020.

[15] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *ICLR*. OpenReview.net, 2019.

[16] Jiaxuan You, Haoze Wu, Clark W. Barrett, Raghuram Ramanujan, and Jure Leskovec. G2SAT: learning to generate SAT formulas. In *NeurIPS*, pages 10552–10563, 2019.

[17] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *NeurIPS*, 2016.

[18] Han Xiao, Minlie Huang, and Xiaoyan Zhu. Transg : A generative model for knowledge graph embedding. In *ACL*. The Association for Computer Linguistics, 2016.

[19] Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *ICCV*, pages 1284–1293. IEEE, 2019.

[20] Jiaxuan You, Jure Leskovec, Kaiming He, and Saining Xie. Graph structure of neural networks. In *ICML*, pages 10881–10891, 2020.

[21] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *ICML*, pages 5694–5703, 2018.

[22] 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. In *NeurIPS*, pages 4255–4265, 2019.

[23] Hanjun Dai, Azade Nazi, Yujia Li, Bo Dai, and Dale Schuurmans. Scalable deep generative modeling for sparse graphs. In *ICML*, pages 2302–2312, 2020.

[24] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. In *ICML*, pages 610–619, 2018.

[25] Carl Yang, Peiye Zhuang, Wenhan Shi, Alan Luu, and Pan Li. Conditional structure generation through graph variational generative adversarial nets. In *NeurIPS*, 2019.

[26] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13: 353–362, 1983.

[27] Andrea Marcelli, Stefano Quer, and Giovanni Squillero. The maximum common subgraph problem: A portfolio approach. *CoRR*, 2019.

[28] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, pages 79–86, 1951.

[29] Cayley. On Monge's "Mémoire sur la Théorie des Déblais et des Remblais.". *Proceedings of the London Mathematical Society*, pages 139–143, 1882.

[30] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, page 47, 2002.

[31] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, pages 27:1–27:27, 2011.

[32] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter W. Battaglia. Learning deep generative models of graphs. *CoRR*, 2018.

[33] Shih-Yang Su, Hossein Hajimirsadeghi, and Greg Mori. Graph generation with variational recurrent neural network. *CoRR*, 2019.

[34] Davide Bacciu, Alessio Micheli, and Marco Podda. Edge-based sequential graph generation with recurrent neural networks. *Neurocomputing*, pages 177–189, 2020.

[35] Davide Bacciu, Alessio Micheli, and Marco Podda. Graph generation by sequential edge prediction. In *ESANN*, 2019.

[36] Marco Podda, Davide Bacciu, and Alessio Micheli. A deep generative model for fragment-based molecule generation, 2020.

[37] Arindam Sarkar, Nikhil Mehta, and Piyush Rai. Graph representation learning via ladder gamma variational autoencoders. *AAAI*, pages 5604–5611, 2020.

[38] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. In *ICML*, pages 2434–2444, 2019.

[39] Nikhil Mehta, Lawrence Carin Duke, and Piyush Rai. Stochastic blockmodels meet graph neural networks. In *International Conference on Machine Learning*, pages 4466–4474. PMLR, 2019.

[40] Dawei Zhou, Lecheng Zheng, Jiawei Han, and Jingrui He. A data-driven graph generative model for temporal interaction networks. In *SIGKDD*, pages 401–411. ACM, 2020.

[41] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding. In *IJCAI*, pages 2609–2615, 2018.

[42] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *JMLR*, pages 985–1042, 2010.

[43] P Erdős and A Rényi. On random graphs i. publicationes mathematicae (debrecen). 1959.

[44] Dj Watts and Sh Strogatz. Collective dynamics of 'small-world' networks (see comments). *Nature*, pages P.440–442, 1998.

[45] Leman Akoglu and Christos Faloutsos. Rtg: A recursive realistic graph generator using random typing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 13–28. Springer, 2009.

[46] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[47] Sebastian Moreno, Jennifer Neville, and Sergey Kirshner. Tied kronecker product graph models to capture variance in network populations. *ACM TKDD*, 2018.

[48] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, pages 109–137, 1983.

[49] Edoardo M Airoldi, David M Blei, Stephen E Fienberg, and Eric P Xing. Mixed membership stochastic blockmodels. *JMLR*, pages 1981–2014, 2008.

[50] Brian Karrer and M. E. J. Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, page 016107, 2011.

[51] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and Seshadhri Comandur. A scalable generative graph model with community structure. *SIAM J. Sci. Comput.*, 36, 2014.

[52] Amit Krishna Joshi, Pascal Hitzler, and Guozhu Dong. Linkgen: Multipurpose linked data generator. In *ISWC*, Lecture Notes in Computer Science, pages 113–121, 2016.

[53] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE TKDE*, pages 856–869, 2017.

[54] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Phys. Rev. E*, page 036113, 2005.

[55] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, page P10008, 2008.

[56] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NeurIPS*, 2014.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *ICLR*, 2018.

[58] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[59] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, pages 1556–1566. The Association for Computer Linguistics, 2015.

[60] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[61] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *ICML*, JMLR Workshop and Conference Proceedings. JMLR.org, 2014.

[62] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[63] Yee Whye Teh, Dilan Grür, and Zoubin Ghahramani. Stick-breaking construction for the indian buffet process. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, pages 556–563. PMLR, 2007.

[64] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[65] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[66] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. In *NeurIPS*, pages 5767–5777, 2017.

[67] Sheng Xiang, Dong Wen, Dawei Cheng, Ying Zhang, Lu Qin, Zhengping Qian, and Xuemin Lin. General graph generators: experiments, analyses, and improvements. *The VLDB Journal*, pages 1–29, 2021.

[68] Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. *ICLR*, 2020.

[69] Guillaume Salha-Galvan, Romain Hennequin, Jean-Baptiste Remy, Manuel Moussallam, and Michalis Vazirgiannis. Fastgae: Scalable graph autoencoders with stochastic subgraph decoding. *Neural networks : the official journal of the International Neural Network Society*, 142:1–19, 2021.

[70] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, 2008.

[71] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, pages 93–106, 2008.

[72] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Inf. Retr.*, pages 127–163, 2000.

[73] D. Paul Dobson and J. Andrew Doig. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, pages 771–783, 2003.

[74] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic acids research*, pages 2443–2450, 2003.

[75] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. Gemsec: Graph embedding with self clustering. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*, pages 65–72. ACM, 2019.

[76] marion neumann, plinio moreno, laura antanas, roman garnett, and kristian kersting. Graph kernels for object category prediction in task-dependent robot grasping. 2013.

[77] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *SIGKDD*, pages 177–187. ACM, 2005.

[78] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.*, pages 29–123, 2009.

[79] Sheng Xiang, Dawei Cheng, Jianfu Zhang, Zhenwei Ma, Xiaoyang Wang, and Ying Zhang. Efficient learning-based community-preserving graph generation. *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1982–1994, 2022.

[80] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864. ACM, 2016.

[81] Chengxi Zang, Peng Cui, Christos Faloutsos, and Wenwu Zhu. Long short memory process: Modeling growth dynamics of microscopic social connectivity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 565–574. ACM, 2017.

[82] Eliezer M Fich and Anil Shivdasani. Financial fraud, director reputation, and shareholder wealth. *Journal of Financial Economics*, 86(2):306–336, 2007.

[83] Dawei Cheng, Zhibin Niu, and Liqing Zhang. Delinquent events prediction in temporal networked-guarantee loans. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[84] Viggo Kann. On the approximability of the maximum common subgraph problem. In *STACS*, 1992.

[85] Fan R. K. Chung and Linyuan Lu. The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences of the United States of America*, 99:15879 – 15882, 2002.

[86] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proc Natl Acad U S A*, 99(12):7821–7826, 2002.

[87] Dawei Cheng, Zhibin Niu, Yi Tu, and Liqing Zhang. Prediction defaults for networked-guarantee loans. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 361–366. IEEE, 2018.

[88] Zhibin Niu, Runlin Li, Junqi Wu, Dawei Cheng, and Jiawan Zhang. iconviz: Interactive visual exploration of the default contagion risk of networked-guarantee loans. In *2020 IEEE conference on visual analytics science and technology (VAST)*, pages 84–94. IEEE, 2020.

[89] Dawei Cheng, Chen Chen, Xiaoyang Wang, and Sheng Xiang. Efficient top-k vulnerable nodes detection in uncertain graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[90] Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in neural information processing systems*, pages 3581–3589, 2014.

[91] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *ICML*, 2019.

[92] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[93] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *ICML*, pages 214–223, 2017.

[94] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8789–8797, 2018.

[95] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. In *International Conference on Learning Representations*, 2018.

[96] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[97] Hongchang Gao, Jian Pei, and Heng Huang. Progan: Network embedding via proximity generative adversarial network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1308–1316. ACM, 2019.

[98] Ming Ding, Jie Tang, and Jie Zhang. Semi-supervised learning on graphs with generative adversarial nets. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 913–922. ACM, 2018.

[99] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, pages 4800–4810, 2018.

[100] Luca Rendsburg, Holger Heidrich, and Ulrike Von Luxburg. Netgan without gan: From random walks to low-rank approximations. In *International Conference on Machine Learning*, pages 8073–8082. PMLR, 2020.

[101] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2021.01.001.

[102] Yuliang Ma, Ye Yuan, Meng Liu, Guoren Wang, and Yishu Wang. Graph simulation on large scale temporal graphs. *GeoInformatica*, 24:199–220, 2019.

[103] Yaochen Xie, Zhao Xu, Zhengyang Wang, and Shuiwang Ji. Self-supervised learning of graph neural networks: A unified review. *IEEE transactions on pattern analysis and machine intelligence*, PP, 2022.

[104] Dawei Cheng, Sheng Xiang, Chencheng Shang, Yiyi Zhang, Fangzhou Yang, and Liqing Zhang. Spatio-temporal attention-based neural network for credit card fraud detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 362–369, 2020.

[105] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[106] Karim Keramat Jahromi, Matteo Zignani, Sabrina Gaito, and Gian Paolo Rossi. Simulating human mobility patterns in urban areas. *Simul. Model. Pract. Theory*, 62:137–156, 2016.

[107] Matthew J McDermott, Shyam S. Dwaraknath, and Kristin A. Persson. A graph-based network for predicting chemical reaction pathways in solid-state materials synthesis. *Nature Communications*, 12, 2021.

[108] Marta C. González, César A. Hidalgo, and A L Barabasi. Understanding individual human mobility patterns. *Nature*, 453:779–782, 2008.

[109] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*, pages 601–610, 2017.

[110] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion Proceedings of the The Web Conference 2018*, pages 969–976, 2018.

[111] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

[112] Taisong Li, Jiawei Zhang, S Yu Philip, Yan Zhang, and Yonghong Yan. Deep dynamic network embedding for link prediction. *IEEE Access*, 6:29219–29230, 2018.

[113] Zhining Liu, Dawei Zhou, and Jingrui He. Towards explainable representation of time-evolving graphs via spatial-temporal graph attention networks. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pages 2137–2140, 2019.

[114] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, and Charles E. Leisersen. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *AAAI*, 2020.

[115] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2628–2638, 2021.

[116] Menglin Yang, Min Zhou, Marcus Kalander, Zengfeng Huang, and Irwin King. Discrete-time temporal network embedding via implicit hierarchical learning in hyperbolic space. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1975–1985, 2021.

[117] Liming Zhang, Liang Zhao, Shan Qin, Dieter Pfoser, and Chen Ling. Tg-gan: Continuous-time temporal graph deep generative models with time-validity constraints. In *Proceedings of the Web Conference 2021*, WWW '21, page 2104–2116, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383127. doi: 10.1145/3442381.3449818.

[118] Dawei Zhou, Kangyang Wang, Nan Cao, and Jingrui He. Rare category detection on time-evolving graphs. In *2015 IEEE International Conference on Data Mining*, pages 1135–1140. IEEE, 2015.

[119] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932, 2009.

[120] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. Edge weight prediction in weighted signed networks. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 221–230. IEEE, 2016.

[121] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and VS Subrahmanian. Rev2: Fraudulent user prediction in rating platforms. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 333–341, 2018.

[122] Shubham Gupta, Sahil Manchanda, Srikanta Bedathur, and Sayan Ranu. Tigger: Scalable generative modelling for temporal interaction graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(6):6819–6828, Jun. 2022. doi: 10.1609/aaai.v36i6.20638.

[123] Giselle Zeno, Timothy La Fond, and Jennifer Neville. Dymond: Dynamic motif-nodes network generative model. In *Proceedings of the Web Conference 2021*, WWW '21, page 718–729, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383127. doi: 10.1145/3442381.3450102.

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

[125] Sheng Xiang, Mingzhi Zhu, Dawei Cheng, Enxia Li, Ruihui Zhao, Yi Ouyang, Ling Chen, and Yefeng Zheng. Semi-supervised credit card fraud detection via attribute-driven graph representation. In *AAAI*, 2023.

[126] Siddhartha Bhattacharyya, Sanjeev Jha, Kurian Tharakunnel, and J Christopher Westland. Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3): 602–613, 2011.

[127] D. Cheng, X. Wang, Y. Zhang, and L. Zhang. Graph neural network for fraud detection via spatial-temporal attention. *IEEE TKDE*, pages 1–1, 5555.

[128] Raghavendra Patidar, Lokesh Sharma, et al. Credit card fraud detection using neural network. *International Journal of Soft Computing and Engineering (IJSCE)*, 1(32-38), 2011.

[129] Kang Fu, Dawei Cheng, Yi Tu, and Liqing Zhang. Credit card fraud detection using convolutional neural networks. In *International Conference on Neural Information Processing*, pages 483–490. Springer, 2016.

[130] Nuno Carneiro, Gonçalo Figueira, and Miguel Costa. A data mining based system for credit-card fraud detection in e-tail. *Decision Support Systems*, 95:91–101, 2017.

[131] KR Seeja and Masoumeh Zareapoor. Fraudminer: A novel credit card fraud detection model based on frequent itemset mining. *The Scientific World Journal*, 2014, 2014.

[132] Ugo Fiore, Alfredo De Santis, Francesca Perla, Paolo Zanetti, and Francesco Palmieri. Using generative adversarial networks for improving classification effectiveness in credit card fraud detection. *Information Sciences*, 2017.

[133] 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.

[134] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. Pick and choose: a gnn-based imbalanced learning approach for fraud detection. In *Proceedings of the Web Conference 2021*, pages 3168–3177, 2021.

[135] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 315–324, 2020.

[136] Zhiwei Liu, Yingtong Dou, Philip S. Yu, Yutong Deng, and Hao Peng. Alleviating the inconsistency problem of applying graph neural network to fraud detection. In *Proceedings of the 43nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[137] Arjan Reurink. Financial fraud: a literature review. *Journal of Economic Surveys*, 32(5): 1292–1325, 2018.

[138] Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):657–668, 2022.

[139] Sam Maes, Karl Tuyls, Bram Vanschoenwinkel, and Bernard Manderick. Credit card fraud detection using bayesian and neural networks. In *Proceedings of the 1st international naiso congress on neuro fuzzy technologies*, pages 261–270, 2002.

[140] Mohammed J Zaki, Wagner Meira Jr, and Wagner Meira. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press, 2014.

[141] Yusuf G Şahin and Ekrem Duman. Detecting credit card fraud by decision trees and support vector machines. 2011.

[142] Mengda Huang, Yang Liu, Xiang Ao, Kuan Li, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. Auc-oriented graph neural network for fraud detection. In *Proceedings of the ACM Web Conference 2022*, pages 1311–1321, 2022.

[143] Fengzhao Shi, Yanan Cao, Yanmin Shang, Yuchen Zhou, Chuan Zhou, and Jia Wu. H2-fdetector: a gnn-based fraud detector with homophilic and heterophilic connections. In *Proceedings of the ACM Web Conference 2022*, pages 1486–1494, 2022.

[144] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[145] Zixing Song, Xiangli Yang, Zenglin Xu, and Irwin King. Graph-based semi-supervised learning: A comprehensive review. *IEEE transactions on neural networks and learning systems*, PP, 2022.

[146] Dan Xu, Wei Wang, Hao Tang, Hong Liu, Nicu Sebe, and Elisa Ricci. Structured attention guided convolutional neural fields for monocular depth estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3917–3925, 2018.

[147] Qingji Guan, Yaping Huang, Zhun Zhong, Zhedong Zheng, Liang Zheng, and Yi Yang. Diagnose like a radiologist: Attention guided convolutional neural network for thorax disease classification. *arXiv preprint arXiv:1801.09927*, 2018.

[148] Tao Shen, Tianyi Zhou, Guodong Long, Jing Jiang, Shirui Pan, and Chengqi Zhang. Disan: Directional self-attention network for rnn/cnn-free language understanding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[149] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[150] Maoguo Gong, Hui Zhou, AK Qin, Wenfeng Liu, and Zhongying Zhao. Self-paced co-training of graph neural networks for semi-supervised node classification. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[151] Yanqiao Zhu, Weizhi Xu, Jinghao Zhang, Yuanqi Du, Jieyu Zhang, Qiang Liu, Carl Yang, and Shu Wu. A survey on graph structure learning: Progress and opportunities.

[152] Qingyun Sun, Jianxin Li, Hao Peng, Jia Wu, Xingcheng Fu, Cheng Ji, and S Yu Philip. Graph structure learning with variational information bottleneck. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 4165–4174, 2022.

[153] Herbert Edelsbrunner and John L Harer. *Computational topology: an introduction*. American Mathematical Society, 2022.

[154] Qi Zhao and Yusu Wang. Learning metrics for persistence-based summaries and applications for graph classification. *Advances in Neural Information Processing Systems*, 32, 2019.

[155] Bastian Rieck, Christian Bock, and Karsten Borgwardt. A persistent weisfeiler-lehman procedure for graph classification. In *International Conference on Machine Learning*, pages 5448–5458. PMLR, 2019.

[156] Christoph Hofer, Roland Kwitt, Marc Niethammer, and Andreas Uhl. Deep learning with topological signatures. *Advances in neural information processing systems*, 30, 2017.

[157] Qi Zhao, Ze Ye, Chao Chen, and Yusu Wang. Persistence enhanced graph neural network. In *International Conference on Artificial Intelligence and Statistics*, pages 2896–2906. PMLR, 2020.

[158] Max Horn, Edward De Brouwer, Michael Moor, Yves Moreau, Bastian Rieck, and Karsten Borgwardt. Topological graph neural networks. *arXiv preprint arXiv:2102.07835*, 2021.

[159] Dawei Cheng, Yi Tu, Zhenwei Ma, Zhibin Niu, and Liqing Zhang. Risk assessment for networked-guarantee loans using high-order graph attention representation. In *IJCAI*, pages 5822–5828. AAAI Press, 2019.

[160] Yunsheng Shi, Zhengjie Huang, Wenjin Wang, Hui Zhong, Shikun Feng, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. In *IJCAI*, 2021.

[161] Yury Gorishniy, Ivan Rubachev, and Artem Babenko. On embeddings for numerical features in tabular deep learning, 2022. URL `https://arxiv.org/abs/2203.05556`.

[162] Shebuti Rayana and Leman Akoglu. Collective opinion spam detection: Bridging review networks and metadata. In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*, pages 985–994, 2015.

[163] Julian John McAuley and Jure Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *Proceedings of the 22nd international conference on World Wide Web*, pages 897–908, 2013.

[164] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. Heterogeneous graph neural networks for malicious account detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2077–2085, 2018.

[165] Jianyu Wang, Rui Wen, Chunming Wu, Yu Huang, and Jian Xion. Fdgars: Fraudster detection via graph convolutional networks in online app review system. In *Companion Proceedings of The 2019 World Wide Web Conference*, pages 310–316, 2019.

[166] Yiming Zhang, Yujie Fan, Yanfang Ye, Liang Zhao, and Chuan Shi. Key player identification in underground forums over attributed heterogeneous information network embedding framework. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 549–558, 2019.

[167] Sheng Xiang, Dawei Cheng, Chencheng Shang, Ying Zhang, and Yuqi Liang. Temporal and heterogeneous graph neural network for financial time series prediction. *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022.

[168] World Federation of Exchanges database. Market capitalization of listed domestic companies, 2021. URL `https://data.worldbank.org/indicator/CM.MKT.LCAP.CD`. accessed 25-July-2021.

[169] Simone Merello, Andrea Picasso Ratto, L. Oneto, and E. Cambria. Ensemble application of transfer learning and sample weighting for stock market prediction. *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[170] Klaus Adam, J. Nicolini, and A. Marcet. Stock market volatility and learning. *IO: Theory*, 2008.

[171] Qing Li, Jinghua Tan, Jun Wang, and HsinChun Chen. A multimodal event-driven lstm model for stock prediction using online news. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020. doi: 10.1109/TKDE.2020.2968894.

[172] Fuli Feng, Huimin Chen, Xiangnan He, Ji Ding, Maosong Sun, and Tat-Seng Chua. Enhancing stock movement prediction with adversarial training. In *IJCAI*, 2019.

[173] Michel Ballings, D. V. Poel, Nathalie Hespeels, and Ruben Gryp. Evaluating multiple classifiers for stock price direction prediction. *Expert Syst. Appl.*, 42:7046–7056, 2015.

[174] Xin Liang, Dawei Cheng, Fangzhou Yang, Yifeng Luo, Weining Qian, and Aoying Zhou. F-hmtc: Detecting financial events for investment decisions based on neural hierarchical multi-label text classification. In *IJCAI*, pages 4490–4496, 2020.

[175] Yaowei Wang, Qing Li, Zhexue Huang, and Mark Junjie Li. EAN: event attention network for stock price trend prediction based on sentimental embedding. In *Proceedings of the 11th ACM Conference on Web Science, WebSci 2019, Boston, MA, USA, June 30 - July 03, 2019*, pages 311–320.

[176] Zhige Li, Derek Yang, Li Zhao, Jiang Bian, Tao Qin, and Tie-Yan Liu. Individualized indicator for all: Stock-wise technical indicator optimization with stock embedding. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[177] Yingmei Chen, Zhongyu Wei, and Xuanjing Huang. Incorporating corporation relationship via graph convolutional neural networks for stock price prediction. *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018.

[178] Dawei Cheng, Fangzhou Yang, Xiaoyang Wang, Y. Zhang, and Liqing Zhang. Knowledge graph-based event embedding framework for financial quantitative investments. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[179] Qikai Liu, Xiang Cheng, Sen Su, and Shuguang Zhu. Hierarchical complementary attention network for predicting stock price movements with news. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 1603–1606.

[180] Rui Cheng and Qing Li. Modeling the momentum spillover effect for stock prediction via attribute-driven graph attention networks. In *AAAI*, 2021.

[181] Dawei Cheng, Yiyi Zhang, Fangzhou Yang, Yi Tu, Zhibin Niu, and Liqing Zhang. A dynamic default prediction framework for networked-guarantee loans. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2547–2555. ACM, 2019.

[182] Ramit Sawhney, Shivam Agarwal, Arnav Wadhwa, and Rajiv Shah. Deep attentive learning for stock movement prediction from social media text and company correlations. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8415–8426, 2020.

[183] Wei Li, Ruihan Bao, Keiko Harimoto, Deli Chen, Jingjing Xu, and Qi Su. Modeling the stock relation with graph network for overnight stock movement prediction. In *IJCAI*, 2020.

[184] Fuli Feng, Xiangnan He, Xiang Wang, Cheng Luo, Yiqun Liu, and Tat-Seng Chua. Temporal relational ranking for stock prediction. *ACM Transactions on Information Systems (TOIS)*, 37:1 – 30, 2019.

[185] Xiao Ding, Yue Zhang, Ting Liu, and Junwen Duan. Using structured events to predict stock price movement: An empirical investigation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 2014, October 25-29, 2014*, pages 1415–1425.

[186] Joel F Houston, Chen Lin, and Zhongyan Zhu. The financial implications of supply chain changes. *Management Science*, 62(9):2520–2542, 2016.

[187] Dawei Cheng, Fangzhou Yang, Sheng Xiang, and Jin Liu. Financial time series forecasting with multi-modality graph neural network. *Pattern Recognition*, 121:108218, 2022. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2021.108218.

[188] Xiao-Yang Liu, Jingyang Rui, Jiechao Gao, Liuqing Yang, Hongyang Yang, Zhaoran Wang, Christina Dan Wang, and Guo Jian. FinRL-Meta: Data-driven deep reinforcementlearning in quantitative finance. *Data-Centric AI Workshop, NeurIPS*, 2021.

[189] Kyunghyun Cho, B. V. Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *ArXiv*, abs/1406.1078, 2014.

[190] Zhigang Jin, Ya-Chi Yang, and Yuhong Liu. Stock closing price prediction based on sentiment analysis and lstm. *Neural Computing and Applications*, 32:9713–9729, 2019.

[191] Gartheeban Ganeshapillai, John Guttag, and Andrew Lo. Learning connections in financial time series. In *International Conference on Machine Learning*, pages 109–117. PMLR, 2013.

[192] Weiwen Liu, Yin Zhang, Jianling Wang, Yun He, James Caverlee, Patrick PK Chan, Daniel S Yeung, and Pheng-Ann Heng. Item relationship graph neural networks for e-commerce. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[193] Yi Tu, Li Niu, Weijie Zhao, Dawei Cheng, and Liqing Zhang. Image cropping with composition and saliency aware aesthetic score map. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12104–12111, 2020.

[194] Y. Tu, Li Niu, Junjie Chen, Dawei Cheng, and Liqing Zhang. Learning from web data with self-organizing memory module. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12843–12852, 2020.

[195] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[196] Raehyun Kim, Chan Ho So, Minbyul Jeong, Sanghoon Lee, Jinkyu Kim, and Jaewoo Kang. Hats: A hierarchical graph attention network for stock movement prediction. *ArXiv*, abs/1908.07999, 2019.

[197] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform bad for graph representation? In *NeurIPS*, 2021.

[198] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[199] M. Dixon, D. Klabjan, and J. Bang. Classification-based financial markets prediction using deep neural networks. *Algorithmic Finance*, 6:67–77, 2017.

[200] Wentao Xu, Weiqing Liu, Chang Xu, Jiang Bian, Jian Yin, and Tie-Yan Liu. Rest: Relational event-driven stock trend forecasting. *Proceedings of the Web Conference 2021*, 2021.

[201] Chi Chen, Li Zhao, Jiang Bian, Chunxiao Xing, and Tie-Yan Liu. Investment behaviors can tell what inside: Exploring stock intrinsic properties for stock trend prediction. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[202] Daniel N. Myers and James W. McGuffee. Choosing scrapy. 2015.

[203] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *EDBT '13*, 2013.