A Systematic Survey on Deep Generative Models for Graph Generation

Xiaojie Guo [1] and Liang Zhao [1]

[1]Affiliation not available

October 30, 2023

Abstract

Graphs are important data representations for describing objects and their relationships, which appear in a wide diversity of real-world scenarios. As one of a critical problem in this area, graph generation considers learning the distributions of given graphs and generating more novel graphs. Owing to its wide range of applications, generative models for graphs have a rich history, which, however, are traditionally hand-crafted and only capable of modeling a few statistical properties of graphs. Recent advances in deep generative models for graph generation is an important step towards improving the fidelity of generated graphs and paves the way for new kinds of applications. This article provides an extensive overview of the literature in the field of deep generative models for graph generation. Firstly, the formal definition of deep generative models for the graph generation as well as preliminary knowledge is provided. Secondly, two taxonomies of deep generative models for unconditional, and conditional graph generation respectively are proposed; the existing works of each are compared and analyzed. After that, an overview of the evaluation metrics in this specific domain is provided. Finally, the applications that deep graph generation enables are summarized and five promising future research directions are highlighted.

A Systematic Survey on Deep Generative Models for Graph Generation

XIAOJIE GUO, Department of Information Science and Technology, George Mason University, USA
LIANG ZHAO*, Department of Computer Science, Emory University, USA

Graphs are important data representations for describing objects and their relationships, which appear in a wide diversity of real-world scenarios. As one of a critical problem in this area, graph generation considers learning the distributions of given graphs and generating more novel graphs. Owing to its wide range of applications, generative models for graphs have a rich history, which, however, are traditionally hand-crafted and only capable of modeling a few statistical properties of graphs. Recent advances in deep generative models for graph generation is an important step towards improving the fidelity of generated graphs and paves the way for new kinds of applications. This article provides an extensive overview of the literature in the field of deep generative models for graph generation. Firstly, the formal definition of deep generative models for the graph generation as well as preliminary knowledge is provided. Secondly, two taxonomies of deep generative models for unconditional, and conditional graph generation respectively are proposed; the existing works of each are compared and analyzed. After that, an overview of the evaluation metrics in this specific domain is provided. Finally, the applications that deep graph generation enables are summarized and five promising future research directions are highlighted.

Additional Key Words and Phrases: graph generation, graph neural network, deep generative models for graphs.

1 INTRODUCTION

Graphs are ubiquitous in the real world, representing objects and their relationships such as social networks, citation networks, biology networks, traffic networks, etc. Graphs are also known to have complicated structures that contain rich underlying values [8]. Tremendous effort has been made in this area, resulting in a rich literature of related papers and methods to deal with various kinds of graph problems, which can be categorized into two types: 1) predicting and analyzing patterns on given graphs. 2) learning the distributions of given graphs and generating more novel graphs. The first type covers many research areas including node classification, graph classification, link prediction, and community detection. Over the past few decades, a significant amount of work has been done in this domain. More recently, representation learning methods, such as deep neural networks for graphs, have also been applied to this aspect. In contrast to the first type, the second type is related to graph generation problem, which is the focus of this paper.

Graph generation entails modeling and generating real-world graphs, and it has applications in several domains, such as understanding interaction dynamics in social networks [47, 128, 129], link

*Corresponding author

Authors' addresses: Xiaojie Guo, xguo7@gmu.edu, Department of Information Science and Technology, George Mason University, 4400 University Drive, Fairfax, VA, USA; Liang Zhao, liang.zhao@emory.edu, Department of Computer Science, Emory University, 201 Downtown Drive, Atlanta, GA, USA.

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

prediction [70, 113], and anomaly detection [109]. Owing to its many applications, the development of generative models for graphs has a rich history, resulting in famous models such as random graphs, small-world models, stochastic block models, and Bayesian network models, which generate graphs based on apriori structural assumptions [98]. These graph generation models [1, 77, 111] are engineered towards modeling a pre-selected family of graphs, such as random graphs [33], small-world networks [132], and scale-free graphs [1]. However, they have limitations. First, due to their simplicity and hand-crafted nature, these random graph models generally have limited capacity to model complex dependencies and are only capable of modeling a few statistical properties of graphs. For example, ErdosâĂŞRÃľnyi graphs do not have the heavy-tailed degree distribution that is typical of many real-world networks. Second, the utilization of the apriori assumption limits these traditional techniques from exploring more applications in larger scale of domains, where the apriori knowledge of graphs are always not available.

Considering the limitations of the traditional graph generation techniques, a key open challenge is developing methods that can directly learn generative models from an observed set of graphs. Developing generative models that can learn directly from data is an important step towards improving the fidelity of generated graphs, and it paves the way for new kinds of applications, such as novel drug discovery [107, 142], and protein structure modeling [3, 5, 34]. Recent advances in deep generative models, such as variational autoencoders (VAE) [68] and generative adversarial networks (GAN) [43], indicate important progress in generative modeling for complex domains, such as image and text data. Building on these approaches, a number of deep learning models for generating graphs have been proposed, which formalized the promising area of *Deep Generative Models for Graph Generation*, which is the focus of this survey.

1.1 Formal Problem Definition

A graph is defined as $G(\mathcal{V}, \mathcal{E}, F, E)$, where $\mathcal{V}$ is the set of N nodes, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of M edges. $e_{i,j} \in \mathcal{E}$ is an edge connecting nodes $v_i, v_j \in \mathcal{V}$. The graph can be conveniently described in matrix or tensor form using its (weighed) adjacency matrix A. If the graph is node-attributed or edge-attributed, there are node attribute matrix $F \in \mathbb{R}^{N \times D}$ assigning attributes to each node or edge attribute tensor $E \in \mathbb{R}^{N \times N \times K}$ assigning attributes to each edge $e_{i,j}$. K is the dimension of the edge attributes, and D is the dimension of the node attributes.

Given a set of observed graphs $\mathbb{G} = \{G_1, ...G_s\}$ sampled from data distribution $p(G)$, where each graph G_i may have different numbers of nodes and edges, the goal of learning generative models for graphs is to learn the distribution of the observed set of graphs. By sampling a graph $G \sim p_{model}(G)$, new graphs can hence be achieved, which is known as deep graph generation, the short form of deep generative models for graph generation. Sometimes, the generation process can be conditioned on additional information y, such that $G \sim p_{model}(G|y)$, in order to provide extra control over the graph generation results. The generation process with such conditions is called conditional deep graph generation.

1.2 Challenges

The development of deep generative models for graphs poses unique challenges. In order to address these challenges, in recent years, numerous research works have been carried out to develop the domain of deep graph generation. These challenges are mainly listed below.

Non-unique Representations. In the general deep graph generation, the aim is to learn the distributions of possible graph structures without assuming a fixed set of nodes (e.g., to generate candidate molecules of varying sizes). In this general setting, a graph with n nodes can be represented by up to $n!$ equivalent adjacency matrices, each corresponding to a different, arbitrary node ordering.

Such high representation complexity is challenging to model, which makes it expensive to compute and, thereafter, optimize objective functions, like reconstruction error, during training.

Complex Dependency. The nodes and edges of a graph have complex dependency and relationships. For example, in many real-world graphs two nodes are more likely to be connected if they share common neighbors. Therefore, the generation of each node or edge cannot be modeled as an independent event, but need to be generated jointly. One way to formalize the graph generation is to make auto-regressive decisions, which naturally accommodate complex dependencies inside the graphs through sequential formalization of graphs.

Large and Various Output Spaces. To generate a graph with n nodes the generative model may have to output n^2 values to specify its structure, which makes it expensive, especially for large-scale graph. However, it is common to find graphs containing millions of graphs in real-world, such as citation and social networks. Also, the numbers of nodes and edges vary between different graphs. Consequently, it is important for generative models to scale to large-scale graphs for realistic graph generation and to accommodate such complexity and variability in the output space.

Discrete Objects by Nature. The standard machine learning techniques, which were developed primarily for continuous data, do not work off-the-shelf, but usually need adjustments. A prominent example is the back-propagation algorithm, which is not directly applicable to graphs, since it works only for continuously differentiable objective functions. To this end, it is usual to project graphs (or their constituents) into a continuous space and represent them as vectors/matrix. However, reconstructing the generated graphs from the continuous representations remains a challenge.

Conditional Generation. Sometimes, it is crucial to guide the graph generation process by conditioning it on extra contextual information. For example, in Natural Language Processing (NLP) domain, Abstract Meaning Representation (AMR) structures and dependency graphs [88, 145] are generated conditioning on an input sequence. The other example is about molecular optimization [59], which generate the target graph conditioning on an input graph. Thus, the deep graph generation problems can face a more challenging problem setting, which requires learning the conditional distribution of the observed graphs given the condition.

Evaluation for Implicit Properties Evaluating the generated graphs is a very critical but challenging issue, due to the unique properties of graphs which with complex and high-dimensional structure and implicit features. Existing methods use different evaluation metrics. For example, some works [51, 124, 142] compute the distance of the graph statistic distribution of the graphs in the test set and graphs that are generated, while other works [34, 83] indirectly use some classifier-based metrics to judge whether the generated graphs are of the same distribution as the training graphs. It is important to systematically review all the existing metrics and choose the approximate ones based on their strengths and limitations according to the application requirements.

Various Validity Requirements. Modeling and understanding graph generation via deep learning involve a wide variety of important applications, including molecule designing [57, 107], protein structure modeling [3], AMR parsing in NLP [88, 145], et al. These inter-discipline problems have their unique requirements for the validity of the generated graphs. For example, the generated molecule graphs need to have valency validity, while the semantic parsing in NLP requires Part-of-Speech (POS)-related constraint. Thus, addressing the validity requirements for different applications is crucial in enabling wider applications of deep graph generation.

Black-box with Low Reliability. Compared with the traditional graph generation area, deep learning based graph modeling methods are like black-boxes which bear the weaknesses of low interpretability and reliability. Improving the interpretability of the deep graph generative models could be a vital issue in unpacking the black-box of the generation process and paving the way for wider application domains, which are of high sensitivity and require strong reliability, such as

smart health and automatic driving. In addition, semantic explanation of the latent representations can further enhance the scientific exploration of the associated application domains.

1.3 Our Contributions

Though recently emerged, deep graph generation has attracted great attentions. Various advanced works on deep graph generation have been conducted, ranging from the one-shot graph generation to sequential graph generation process, accommodating various deep generative learning strategies. These methods aim to solve one or several of the above challenges by works from different fields, including machine learning, bio-informatics, artificial intelligence, human health and social-network mining. However, the methods developed by different research fields tend to use different vocabularies and solve problems from different angles. Also, standard and comprehensive evaluation procedures to validate the developed deep generative models for graphs are lacking. A comprehensive and systematic survey covering the research on deep generative models for graph generation as well as its applications, evaluations, and open problems is imperative yet missing.

To this end, this paper provides a systematic review of deep generative models for graph generation. We categorize methods and problems based on the challenges they address, discuss their underlying assumptions, and compare their advantages and disadvantages. The goal is to help interdisciplinary researchers choose appropriate techniques to solve problems in their applications domains, and more importantly, to help graph generation researchers understand the basic principles as well as identify open research opportunities in deep graph generation domain. As far as we know, this is the first comprehensive survey on deep generative models for graph generation. Below, we summarize the major contributions of this survey:

- We propose a taxonomy of deep generative models for graph generation categorized by problem settings and methodologies. The drawbacks, advantages, relations, and difference among different subcategories have been introduced.
- We provide a detailed description, analysis, and comparison of deep generative models for graph generation as well as the deep generative models on which they are based.
- We summarize and categorize the existing evaluation procedures and metrics of deep generative models for graph generation.
- We introduce existing application domains of deep generative models for graph generation as well as the potential benefits and opportunities they bring into the application domains.
- We suggest several open problems and promising future research directions in the field of deep generative models for graph generation.

1.4 Relationship with Related Surveys

There are three types of related surveys. The first type mainly centers around the traditional graph generation by classic graph theory and network science [13], which does not focus on the most recent advancement in deep generative neural networks in artificial intelligence. The second type is about representation learning on graphs [45, 135, 146]. This is a very hot domain in machine learning, especially deep learning. It can benefit a number of downstream tasks including node and graph classification, link prediction, and graph generation. This domain focuses on learning graph embedding given existing graphs. Few works include a handful of deep generative models that could be used for representation learning tasks. The last type is specific to particular applications such as molecule design by deep learning, instead of for this generic technical domain. To the best of our knowledge, there is no systematic survey on deep generative models for graph generation.

1.5 Outline of the Survey

The rest of this survey is organized as follows. In Section 2, we first introduce the preliminary of the existing deep generative models that are used as the base model for learning graph distributions.

Then we introduce the definitions of the basic concepts required to understand the deep graph generation problem as well as its extensive problem, conditional deep graph generation. In the next two sections, we provide the taxonomy of deep graph generation, and the taxonomy structure is illustrated in Fig.1. Section 3 compares related works of unconditional deep graph generation problem and summarizes the challenges faced in each. In Section 4, we categorize the conditional deep graph generation in terms of three sub-problem settings. The challenges behind each problem are summarized, and a detailed analysis of different techniques is provided. Lastly, we summarize and categorize the evaluation metrics in Section 5. Then we present the applications that deep graph generation enables in Section 6. At last, we discuss five potential future research directions and conclude this survey in Section 7.

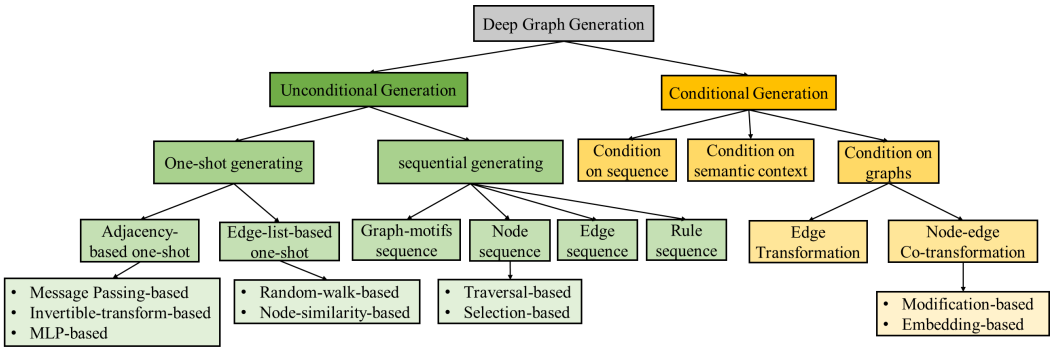

Fig. 1. Classification of deep generative models for graph generation problems

2 PRELIMINARIES KNOWLEDGE

In recent years, there has been a resurgence of interest in deep generative models, which have been at the forefront of deep unsupervised learning for the last decade. The reason for that is because they offer a very efficient way to analyze and understand unlabeled data. The idea behind generative models is to capture the inner probabilistic distribution that generates a class of data to generate similar data [103]. Emerging approaches such as generative adversarial networks (GANs) [43], variational auto-encoders (VAEs) [68], generative recursive neural network (generative RNN) [126] (e.g., pixelRNNs, RNN language models), flow-based learning [104], and many of their variants and extensions have led to impressive results in myriads of applications. In this section, we provide a review of five popular and classic deep generative models for learning the distributions by observing large amounts of data in any format. They include VAE, GANs, generative RNN, flow-based learning, and Reinforcement Learning, which also form the backbone of the base learning methods of all the existing deep generative models for graph generation.

2.1 Variational Auto-encoders

VAE [68] is a latent variable-based model that pairs a top-down generator with a bottom-up inference network. Instead of directly performing maximum likelihood estimation on the intractable marginal log-likelihood, training is done by optimizing the tractable evidence lower bound (ELBO). Suppose we have a dataset of samples x from a distribution parameterized by ground truth generative latent codes $z \in \mathbb{R}^c$ (c refers to the length of the latent codes). VAE aims to learn a joint distribution between the latent space $z \sim p(z)$ and the input space $x \sim p(x)$.

Specifically, in the probabilistic setting of a VAE, the encoder is defined by a variational posterior $q_\phi(z|x)$, while the decoder is defined by a generative distribution $p_\theta(x|z)$, as represented by the two orange trapezoids in Fig. 2(a). ϕ, θ are trainable parameters of the encoder and decoder. The VAE

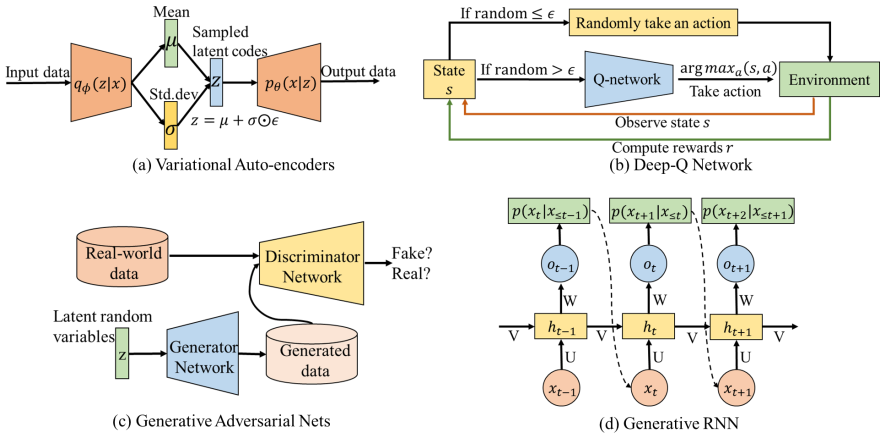

Fig. 2. Abstract architecture of deep generative models: (a) Variational auto-encoders; (b) Deep Q-network; (c) Generative adversarial nets; (d) generative RNN.

aims to learn a marginal likelihood of the data in a generative process as: $\max_{\phi,\theta} \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$. Then the marginal likelihoods of individual data points can be rewritten as follows:

$$\log p_\theta(x|z) = D_{KL}(q_\phi(z|x)||p(z)) + \mathcal{L}(\phi, \theta; x, z), \tag{1}$$

where the first term stands for the non-negative KullbackâĂŞLeibler divergence between the true and the approximate posterior; the second term is called the (variational) lower bound on the marginal likelihood. Thus, maximizing $\mathcal{L}(\phi, \theta; x, z)$ is to maximize the lower bound of the true objective:

$$\mathcal{L}(\phi, \theta; x, z) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p(z)). \tag{2}$$

In order to make the optimization of the above objective tractable in practice, we assume a simple prior distribution $p(z)$ as a standard Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$ with a diagonal co-variance matrix. Parameterizing the distributions in this way allows for the use of the âĂIJreparameterization trickâĂİ to estimate gradients of the lower bound with respect to the parameter ϕ, where each random variable $z_i \sim q_\phi(z_i|x)$ is parameterized as Gaussian with a differentiable transformation of a noise variable $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$, that is, z is computed as $z = \mu + \sigma \odot \epsilon$, where μ and σ are outputs from the encoder.

2.2 Generative Adversarial Nets

GANs were introduced as an alternative way to train a generative model [43]. GANs are based on a game theory scenario called the min-max game, where a discriminator and a generator compete against each other. The generator generates data from stochastic noise, and the discriminator tries to tell whether it is real (coming from a training set) or fabricated (from the generator). The absolute difference between carefully calculated rewards from both networks is minimized so that both networks learn simultaneously as they try to outperform each other.

Specifically, the architecture of GANs consists of two âĂŸadversarialâĂŹ models: a generative model $\mathcal{G}_\theta$ which captures the data distribution $p(x)$, and a discriminative model $\mathcal{D}_\phi$ which estimates the probability that a sample comes from the training set rather than $\mathcal{G}_\theta$, as shown in Fig.2(c). Both $\mathcal{G}_\theta$ and $\mathcal{D}_\phi$ could be a non-linear mapping function, such as a multi-layer perceptron [125] parameterized by parameters θ and ϕ. To learn a generator distribution $p_{model}(x)$ of observed data x, the generator builds a mapping function from a prior noise distribution $p_z(z)$ to data space as $\mathcal{G}_\theta(z)$. And the discriminator, $\mathcal{D}_\phi(x)$, outputs a single scalar representing the probability that the input data x came form the training data rather than sampled from $p_{model}(x)$.

The generator and discriminator are both trained simultaneously by adjusting the parameters of $p_{model}(x)$ to minimize $log(1-\mathcal{D}_\phi(\mathcal{G}_\theta(z))$ and adjusting the parameters of $\mathcal{D}_\phi$ to minimize $log\mathcal{D}_\phi(x)$, as if they are following the two-player min-max game with value function $V(\mathcal{G}_\theta, \mathcal{D}_\phi)$:

$$\min_{\mathcal{G}_\theta}\max_{\mathcal{D}_\phi} V(\mathcal{G}_\theta, \mathcal{D}_\phi) = \mathbb{E}_{x\sim p_{model}(x)}[log\mathcal{D}_\phi(x)] + \mathbb{E}_{z\sim p_z(z)}[log(1-\mathcal{D}_\phi(\mathcal{G}_\theta(z))], \tag{3}$$

The training of the generator and discriminator is kept alternating until the generator can hopefully generate real-like data that is difficult to discriminate from real samples by a strong discriminator.

In general, GANs show great power in generating data such as image [26, 43], audio [20], and texts [97]. In contrast to VAE, GANs learn to generate samples without assuming an approximate distribution. By utilizing the discriminator, GANs avoid optimizing the explicit likelihood loss function, which may explain their ability to produce high-quality objects as demonstrated by Denton et al. [26]. However, GANs still have drawbacks. One is that they can sometimes be extremely hard to train in adversarial style. They may fall into the divergence trap very easily by getting stuck in a poor local minimum. Mode collapse is also an issue, where the generator produces samples that belong to a limited set of modes, which results in low diversity. Moreover, alternatively training and large computation workloads for two networks can result in long-term convergence process.

2.2.1 Generative Recursive Neural Network. RNN [94] is a straightforward adaptation of the standard feed-forward neural network by using their internal state (memory) to process variable length sequential data. At each step, the RNN predicts the output depending on the previous computed hidden states and updates its current hidden state, that it, they have a âĂIJmemoryâĂİ that captures information about what has been calculated so far. The RNNâĂŹs high dimensional hidden state and nonlinear evolution endow it with great expressive power to integrate information over many iterative steps for accurate predictions. Even if the non-linearity used by each unit is quite simple, iterating it over time leads to very rich dynamics [126].

A standard RNN is formalized as follows: given a sequence of input vectors $(x_1, ..., x_T)$, the RNN computes a sequence of hidden states $(h_1, ..., h_T)$ and a sequence of outputs $(o_1, ..., o_T)$ by iterating the following equations from $t = 1$ to T:

$$h_t = \tanh(Ux_t + Vh_{t-1} + b_h); \qquad o_t = Wh_t + b_o \tag{4}$$

where U, V, and W are learning weight matrices; the vectors b_h and b_o are biases for calculating the hidden states and output at each step, respectively. The expression Vh_{t-1} at step $t = 1$ is initialized by a vector, h_0, and the tanh non-linearity activation function is applied coordinate-wise.

The RNN model can be modified to a generative model for generating the sequential data, as shown in Fig. 2(d). The goal of modeling a sequence is to predict the next element in the sequence given the previous generated elements. More formally, given a training sequence $(x_1, ..., x_T)$, RNN uses the sequence of its output vectors $(o_1, ..., o_T)$ to parameterize a sequence of predictive distributions $p(x_{t+1}|x_{\leq t})$. The distribution type of $p(x_{t+1}|x_{\leq t})$ need to be assumed in advance. For example, to determine the category of the discrete data x_{t+1}, we can assume a softmax distribution as $p(x_{t+1} = j) = exp(o_t^{(j)})/\sum_K o_t^{(K)}$, where j refers to one of the categories of the object, $o_t^{(j)}$ refers to the j-th variable in the output vector o_t and K refers to the total number of categories of the objects. The objective of modeling sequential data is to maximize the total log likelihood of the training sequence $\sum_{t=0}^{T-1} \log p(x_{t+1}|x_{\leq t})$, which implies that the RNN learns a joint probability distribution of sequences. Then we can generate a sequence by sampling from $p(x_{t+1}|x_{\leq t})$ stochastically, which is parameterized by the output at each step.

2.3 Flow-based Learning

Normalizing flows (NFs) [27] are a class of generative models that define a parameterized invertible deterministic transformation between two spaces z and x. $z \sim p_z(z)$ is a latent space that follows

distribution such as Gaussian, and $x \sim p_x(x)$ is a real-world observational space of objects such as images, graphs, and texts. Let $f_\theta : z \rightarrow x$ be an invertible transformation parameterized by θ. Then the relationship between the density function of real-world data x and that of z can be expressed via the change-of-variables formula:

$$p_x(x) = p_z(f_\theta^{-1}(x))|\det(\partial f_\theta^{-1}(x)/\partial x)|. \tag{5}$$

There are two key processes of normalizing flows as a generative model: (1) Calculating data likelihood: given a datapoint x, the exact density $p_x(x)$ can be calculated by inverting the transformation $z = f_\theta^{-1}(x)$; (2) Sampling: x can be sampled from the distribution $p_x(x)$ by first sampling $z \sim p_z(z)$ and then performing the transformation $x = f_\theta(z)$. To efficiently perform the above mentioned operations, f_θ is required to be invertible with an easily computable Jacobian determinant.

Autoregressive flow (AF), originally proposed in [104], is a variant of normalizing flow by providing an easily computable triangular Jacobian determinant. It is specially designed for modeling the conditional distributions in the sequence. Formally, given $x \in \mathbb{R}^D$ (D is the dimension of observed sequential data), the autoregressive conditional probabilities for the d-th element in the sequence can be parameterized as Gaussian:

$$p(x_d|x_{1:d-1}) = \mathcal{N}(\mu_d, (\sigma_d)^2), \quad where \quad \mu_d = g_\theta(x_{1:d-1}), \quad \sigma_d = g_\phi(x_{1:d-1}) \tag{6}$$

where g_θ and g_ϕ are unconstrained and positive scalar functions of $x_{1:d-1}$ respectively for computing the mean and deviation. In practice, these functions can be implemented as neural networks. The affine transformation of AF can be written as follows:

$$f_\theta(z_d) = x_d = \mu_d + \sigma_d \cdot z_d; \quad f_\theta^{-1}(x_d) = z_d = (x_d - \mu_d)/\sigma_d, \tag{7}$$

where z_d is the randomly sampled value from standard Gaussian. The Jacobian matrix here is triangular, since $\partial x_i/\partial z_j$ is non-zero only for $j \leqslant i$. Therefore, the determinant can be efficiently computed through $\prod_{d=1}^{D} \sigma_d$. Specifically, to perform density estimation, we can apply all individual scalar affine transformations in parallel to compute the base density, each of which depends on previous variables $x_{1:d-1}$; to sample x, we can first sample $z \in \mathbb{R}^D$ and compute x_1 through the affine transformation, and then each subsequent x_d can be computed sequentially based on $x_{1:d-1}$.

2.4 Reinforcement Learning and Deep Q-Network

Reinforcement learning (RL) is a commonly used framework for learning controlling policies by a computer algorithm, the so-called agent, through interacting with its environment [118, 127]. Here, we give a brief introduction of this learning strategy as well as its typical form deep Q-learning networks (DQNs) [96] for data generation.

In RL process, an agent is faced with a sequential decision making problem, where interaction with the environment takes place at discrete time steps. The agent takes action a_t at state s_t at time t, by following certain policies or rules, which will result in a new state s_{t+1} as well as a reward r_t. If we consider infinite horizon problems with a discounted cumulative reward objective $R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}$ ($\gamma \in [0, 1]$ is the discount factor), the aim of the agent is to find an optimal policy $\pi : s \rightarrow a$ by maximizing its expected discounted cumulative rewards. Q-Learning [131] is a value-based method for solving RL problems by encoding policies through the use of action-value functions:

$$Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]. \tag{8}$$

The optimal value function is denoted as $Q^*(s, a) = \max_\pi Q^\pi(s, a)$, and an optimal policy π^* can be easily derived by $\pi^*(s) \in \text{argmax}_a Q^*(s, a)$. Typically, Q-value function relies on all possible state-action pairs, which are often impractical to obtain. One solution for addressing this challenge is to approximate $Q(s, a)$ using a parameterized function [127].

Based on recent advances in deep learning techniques, Mnih et al. [96] introduced the DQN. The DQN approximates the Q-value function with a non-linear deep convolutional network, which also automatically creates useful features to represent the internal states of the RL, as shown in Fig. 2(b). In DQN, the agent interacts with the environment in i discrete iterations, aiming to maximize its long term reward. DQN has shown great power in generating sequential objects by taking a series of actions [78]. A sequential object is generated based on a sequence of actions that are taken.

During the generation, DQN selects the action at each step using an ϵ-greedy implementation. With probability ϵ, a random action is selected from the range of possible actions, otherwise the action which results in high Q-value score is selected. To perform experience replay, the agentâĂŹs experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t are stored in a data set $D_t = \{e_1, \text{fi}, e_t\}$. At each iteration i in the learning process, the updates of the learning weights are applied on samples of experience $(s_t, a_t, r_t, s_{t+1}) \sim U(D)$, drawn randomly from the pool of stored samples, with the following loss function:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim U(D)}[(r_t + \gamma \underset{a_{t+1}}{max} Q(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s, a; \theta_i))^2], \tag{9}$$

where θ_i refers to the parameters of the Q-network at iteration i and θ_i^- refers to the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters θ_i every several steps and are held fixed between individual updates. The process of generating the data after training is similar to that of the training process.

3 UNCONDITIONAL DEEP GENERATIVE MODELS FOR GRAPH GENERATION

The goal of unconditional deep graph generation is to learn the distribution $p_{\text{model}}(G)$ based on a set of observed realistic graphs being sampled from the real distribution $p(G)$ by deep generative models. Based on the style of the generation process, we can categorize the methods into two main branches: (1) **Sequential generating**: this generates the nodes and edges in a sequential way, one after another, (2) **One-shot generating**: this refers to building a probabilistic graph model based on the matrix representation that can generate all nodes and edges in one shot. These two ways of generating graphs have their limitations and merits. Sequential generating performs the local decisions made in the preceding one in an efficient way with time complexity of only $O(N)$, but it has difficulty in preserving the long-term dependency. Thus, some global properties (e.g., scale-free property) of the graph are hard to include. Moreover, existing works on sequential generating are limited to a predefined ordering of the sequence, leaving open the role of permutation. One-shot generating methods have the capacity of modeling the global property of a graph by generating and refining the whole graph (i.e. nodes and edges) synchronously through several iterations, but most of them are hard to scale to large graphs since the time complexity is not less than $O(N^2)$.

Table 1. Deep Generative-based Methods for Unconditional Graph Generation

Generating Style	Techniques		Reference
Sequential Generating	Node-sequence-based	Traversal-based	[4, 23, 67, 107, 123, 142, 144]
		Selection-based	[66, 79, 82, 86]
	Edge-sequence-based		[5, 6, 11, 44]
	Graph-Motif-sequence-based		[48, 57, 81, 105]
	Rule-sequence-based		[22, 73]
One-shot Generating	Adjacency-based	MLP-based	[3, 24, 34, 89, 106, 119]
		Message-Passing-based	[16, 36, 49, 101]
		Invertible-transform-based	[54, 92]
	Edge-list-based	Random-walk-based	[11, 18, 38, 143]
		Node-similarity-based	[47, 69, 85, 113, 117, 150]

3.1 Generating a Graph Sequentially

This type of methods treats the graph generation as a sequential decision making process, wherein nodes and edges are generated one by one (or group by group), conditioned on the sub-graph already generated. By modeling graph generation as a sequential process, these approaches naturally accommodate complex local dependencies between generated edges and nodes. A graph G is represented into a sequence of components $S = \{s_1, ..., s_N\}$, where each $s_i \in S$ can be regarded as a generation unit. The distribution of graphs $p(G)$ can then be formalized as the joint (conditional) probability of all the components in general. While generating graphs, different components will be generated sequentially, by conditioning on the other parts already generated. One core issue is how to break down the graph generation into sequential generation of its components. Thus, regarding the formalization the unit s_i for sequentialization, there are four common ways: *node-sequence-based*, *edge- sequence-based*, *graph-motif-sequence-based* and *rule-sequence-based*, as shown on Fig. 1 (left).

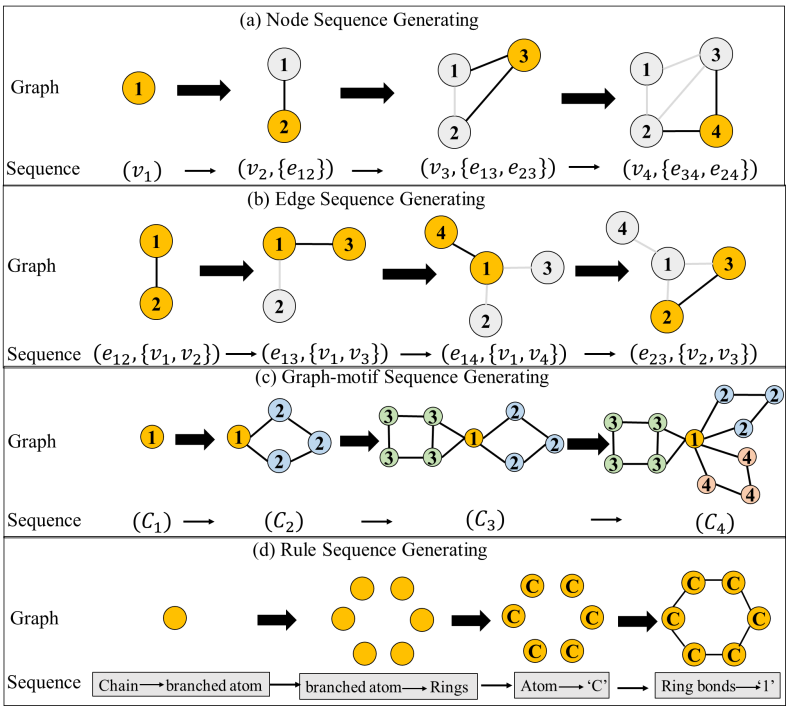

Fig. 3. Four categories in sequential generating: graph line refers to the immediate graph that are generated per step; Sequence line refers to the sequence S_i that is generated per step.

3.1.1 Node-Sequence-based. Node-sequence-based methods essentially generate the graph by generating one node and its associated edges per step, as shown in Fig. 3 (a). Specifically, the graph can be modeled by a sequence based on a predefined ordering π on nodes. Each unit s_i in the sequence of components S is represented as a tuple $s_i = (v_i^\pi, \{e_{i,j}\}_{j<i})$, indicating that at each high-level step, the generator generates one node v_i^π and all its associated edges set $\{e_{i,j}\}_{j<i}$. Here we omit the node and edge attribute symbol for clarity, but we should bear in mind that the generated node and edges can all have attributes (i.e. type, label). Given a newly generated node v_i^π, existing methods for the generation of its associated edges $\{e_{i,j}\}_{j<i}$ can be grouped into two:

1) traversal-based, where the edges are formed when traversing the newly generated node v_i and all the existing nodes, and 2) selection-based, which entails determining whether there is an edge between the newly generated node v_i and any of the existing nodes.

Traversal-based. When treating a graph as a sequence of node tuples each of which is denoted as $s_i = (v_i^\pi, \{e_{i,j}\}_{j<i})$, several approaches [23, 67, 107, 117, 123, 142, 144] represent each node's associated edges by the adjacent vector $A^\pi_{i,.}$ (we assume that the graph is undirected, without the loss of generality), which covers all the potential edges from the newly added node v_i to the previously generated nodes. Thus, we can further represent each unit as $s_i = (v_i^\pi, A^\pi_{i,.})$. And the sequence can be represented as $Seq(G, \pi) = \{(v_1^\pi, A^\pi_{1,.}), ..., (v_N^\pi, A^\pi_{N,.})\}$. The aim is to learn the distribution as:

$$p(\mathcal{V}^\pi, A^\pi) = \prod_{i=1}^{n+1} p(v_i^\pi | v^\pi_{<i}, A^\pi_{<i,.}) p(A^\pi_{i,.} | v^\pi_{\leq i}, A^\pi_{<i,.}), \tag{10}$$

where $v^\pi_{<i}$ refers to the nodes generated before v_i^π and $A^\pi_{<i,.}$ refers to the adjacent vectors generated before $A^\pi_{i,.}$. Such joint probability can be implemented by sequential-based architectures such as generative RNN models [83, 107, 142, 144] and auto-regressive flow-based learning models [117], which are introduced subsequently.

In the generative RNN-based models, the node distributions $p(v_i^\pi | v^\pi_{<i}, A^\pi_{<i,.})$ are typically assumed as a multivariate Bernoulli distribution that is parameterized by $\phi_i \in \mathbb{R}^T$, where T refers to the number of node categories. The edge existence distribution $p(A^\pi_{i,.} | v^\pi_{\leq i}, A^\pi_{<i,.})$ can be assumed as the joint of several dependent Bernoulli distributions as follows:

$$p(A^\pi_{i,.} | A^\pi_{<i,.}) = \prod_{j=1}^{i-1} p(A^\pi_{i,j} | A^\pi_{i,<j}, A^\pi_{<i,.}), \tag{11}$$

where $p(A^\pi_{i,.} | A^\pi_{<i,.})$ is parameterized by $\theta_i \in \mathbb{R}^{i-1}$ and the distribution of $p(A^\pi_{i,j} | A^\pi_{i,<j}, A^\pi_{<i,.})$ is parameterized by each entry $\theta_{i,j}$ in θ_i. The architecture for implementing Eq. 10 and 11 can be regarded as a hierarchical-RNN, where the outer RNN is used for generating the nodes and the inner RNN is used for generating each node's associated edges. After either a node or edge is generated, a graph-level hidden representation of the already generated sub-graph is calculated and updated through a message passing neural network (MPNN) [40]. Specifically, at each Step i, a parameter ϕ_i will be calculated through a multilayer perceptron (MLP)-based function based on the current graph-level hidden representation. The parameter ϕ_i is used to parameterize the Bernoulli distribution of node existence, from which node v_i^π is sampled. After that, the adjacent vector $A^\pi_{i,.}$ is generated by sequentially generating each of its entry. Specifically, at each step j in generating $A^\pi_{i,.}$, the edge $A^\pi_{i,j}$ is generated by sampling based on the conditional parameter $\theta_{i,j}$, which is also calculated through a MLP-based function based on the current graph-level hidden representation.

In addition to RNN-based methods, now we introduce some representative works based on auto-regressive flow-based learning models [117]. Shi et al. [117] achieved conditional generation via the flow-based learning as introduced in Section 2.3. Based on the idea to first transform discrete data into continuous data with real-valued noise and *dequantization techniques* [27], specifically, the discrete unit $s_i = (v_i^\pi, A^\pi_{i,.})$ is pre-processed into continuous data $z_i = (z_i^F, z_i^A)$:

$$z_i^F = F_i^\pi + u; \quad z^A_{i,j} = A^\pi_{i,j} + u, \quad u \sim U[0, 1), \tag{12}$$

where F_i^π refers to the category of node v_i^π and $U[0, 1)$ refers to a uniform distribution [71]. Then the conditional distributions for the continuous data z_i^F and $z^A_{i,j}$ are assumed as Gaussian distribution:

$$p(z_i^F | z^F_{<i}, z^A_{<i,.}) = \mathcal{N}(\mu_i^F, (\sigma_i^F)^2); \quad P(z^A_{i,j} | z^F_{\leq i}, z^A_{<i,.}, z^A_{i,<j}) = \mathcal{N}(\mu^A_{i,j}, (\sigma^A_{i,j})^2), \tag{13}$$

where the mean μ_i^F, $\mu^A_{i,j}$ and standard deviation σ_i^F, $\sigma^A_{i,j}$ of the Gaussian distribution for node and edge generation are calculated based on the MLP-based networks whose input is the hidden representations of the already generated graph. The hidden representations of the graph are typically calculated through MPNN.

Several additional works are based on VAE, yet their latent representations are generated sequentially. Su et al. [123] propose a graph recurrent neural network with variational Bayes to learn the conditional distributions. It uses the conditional VAE (CVAE) [120] and utilizes three MLP-based networks for modeling three distributions of the generation process, namely prior distribution $p(z_i|v^\pi_{<i}, A^\pi_{<i},)$, node generation distribution $p(v^\pi_i|z_i, v^\pi_{<i}, A^\pi_{<i,.})$, and edge generation distribution $p(A^\pi_{i,.}|z_i, v^\pi_{\leq i}, A^\pi_{<i,.})$. Here z_i refers to the latent representation at Step i. During the generation process, at each step i, the prior network is first used to draw samples z_i from the learnt prior distribution $p(z_i|v^\pi_{<i}, A^\pi_{<i})$, which is parameterized by the output of an MLP-based function with the input of the already generated graph. Then the node v^π_i and its associated edges $A^\pi_{i,.}$ are generated by sampling from $p(v^\pi_i|z_i, v^\pi_{<i}, A^\pi_{<i,.})$ and $p(A^\pi_{i,.}|z_i, v^\pi_{\leq i}, A^\pi_{<i,.})$, respectively, which are parameterized by the outputs of two MLP-based functions with the input of z_i and the already generated graph.

Selection-based. The selection-based methods generate the nodes in the same way as the traversal-based method, but have a different way of generating the associated edge set. Traversing all the existing nodes to generate the associated edge set for each newly generated node v^π_i is time-consuming and potentially low in efficiency, especially for sparse graphs. It is efficient to directly generate the edge set $\{e_{i,j}\}_{j<i}$ of v^π_i by only selecting the neighboring nodes from the already generated nodes. Specifically, for each newly generated node v^π_i, the selection-based methods generate its $\{e_{i,j}\}_{j<i}$ relying on two functions: an *addEdge* function to determine the size of the edge set $\{e_{i,j}\}_{j<i}$ of node v^π_i and a *selectNode* function to select the neighboring nodes sequentially from the partially generated graph [66, 79, 82, 86].

Specifically, at Step i, after generating a node v^π_i, an *addEdge* function is used to output a parameter as $f_{addEdge}(h^\pi_{v_i})$, following a Bernoulli distribution indicating whether we want to add an edge to the node v^π_i. Here $h^\pi_{v_i}$ refers to the node-level hidden states of v^π_i which is calculated through a node embedding function, e.g., MPNN based on the already-generated parts of the graph. If an edge is determined to be added, the next step is selecting the neighboring node v^π_j from the existing nodes. To select this neighboring node, we can compute a score m^π_j (as Eq.14) for each existing node v^π_j based on *selectNode* function $f_{selectNode}$, which is then passed through a softmax function [10] to be properly normalized into a distribution of nodes:

$$m^\pi_{i,j} = f_{selectNode}(h^\pi_{v_i}, h^\pi_{v_j}), \quad wherej < i. \tag{14}$$

$$p(e_{i,j}|v^\pi_{<i}, \{e_{<i,j}\}_{j<i}) = softmax(m^\pi_{i,j}). \tag{15}$$

The MLP-based function $f_{selectNode}$ maps pairs of node-level hidden states $h^\pi_{v_i}$ and $h^\pi_{v_j}$ to a score $m^\pi_{i,j}$ for connecting node v^π_j to the new node v^π_i. This can be extended to handle discrete edge attributes by making $m^\pi_{i,j}$ a vector of scores with the same size as the number of the edge attribute's categories, and taking the softmax over all categories of the edge attribute. Based on the aforementioned procedure, the two functions $f_{addEdge}$ and $f_{selectNode}$ are iteratively executed to generate the edges within the edge set $\{e_{<i,j}\}_{j<i}$ of node v^π_i until the terminal signal from function $f_{addEdge}$ indicates that no more edges for node v_i are yet to be added.

3.1.2 Edge-Sequence-based. Edge-sequence-based methods represent the graph as a sequence of edges and generate an edge as well as its two related nodes per step, as shown in Fig. 3 (b). It defines an ordering of the edges in the graph and also an ordering function $\alpha(\cdot)$ for indexing the nodes. Then the graph G can be modeled by a sequence of edges [5, 6, 44] and each unit in the sequence is a tuple represented as $s_i = (\alpha(u), \alpha(v), F_u, F_v, E^i_{u,v})$, where each element of the sequence consists of a pair of nodes' indexes $\alpha(u)$ and $\alpha(v)$ for node u and v, node attribute F_u, F_v, and the edge attribute $E^i_{u,v}$ for the edge at Step i. The edge-sequence-based methods usually employ two parallel networks for generating two related nodes of the edge respectively. The key problem in generating graphs by

a sequence of edges is to pre-define the ordering index function $\alpha(\cdot)$ for nodes; thus, based on the index of the generated nodes, the graph can be constructed from the generated sequence of edges.

Goyal et al. [44] used depth first search (DFS) algorithm [137] as the ordering index function $\alpha(\cdot)$ to construct graph canonical index of nodes by performing a DFS. The conditional distribution for generating each edge in graph G can be formalized as follows:

$$p(s_i|s_{<i}) = p(\alpha(u)|s_{<i})p(\alpha(v)|s_{<i})p(F_u|s_{<i})p(F_v|s_{<i})p(E^i_{u,v}|s_{<i}), \tag{16}$$

where $s_{<i}$ refers to the already generated edges and nodes. A customized long short-term memory (LSTM) is designed which consists of a transition state function f_{trans} for transferring the hidden state of the last step into that of the current step (in Eq.17), an embedding function f_{emb} for embedding the already generated graph into latent representations (in Eq. 17), and five separate output functions for the above five distribution components (in Eq 17 to Eq. 20). It is assumed that the five elements in one tuple are independent of each others, and thus the inference is operated as:

$$h^{(i)}_G = f_{\text{trans}}(h^{(i-1)}_G, f_{\text{emb}}(s_{i-1})) \tag{17}$$

$$\alpha(u) \sim Cat(\theta_{\alpha(u)}); \quad \theta_{\alpha(u)} = f_{\alpha(u)}(h^{(i)}_G); \quad \alpha(v) \sim Cat(\theta_{\alpha(v)}); \quad \theta_{\alpha(v)} = f_{\alpha(v)}(h^{(i)}_G) \tag{18}$$

$$F_u \sim Cat(\theta_{F_u}); \quad \theta_{F_u} = f_{F_u}(h^{(i)}_G); \quad F_v \sim Cat(\theta_{F_v}); \quad \theta_{F_v} = f_{F_v}(h^{(i)}_G) \tag{19}$$

$$E^i_{u,v} \sim Cat(\theta_{E^i_{u,v}}); \quad \theta_{E^i_{u,v}} = f_{E^i_{u,v}}(h^{(i)}_G), \tag{20}$$

where s_{i-1} refers to the generated tuple at Step $i-1$ and is represented as the concatenation of all the component representations in the tuple. $h^{(i)}_G$ is a graph-level LSTM hidden state vector that encodes the state of the graph generated so far at Step i. Given the graph state $h^{(i)}_G$, the output of five functions $f_{\alpha(u)}$, $f_{\alpha(v)}$, f_{F_u}, f_{F_v}, $f_{E_{u,v}}$ model the categorical distribution of the five components of the newly formed edge tuple, which are paramerized by five vectors $\theta_{\alpha(u)}$, $\theta_{\alpha(v)}$, θ_{F_u}, θ_{F_v}, $\theta_{E_{u,v}}$ respectively. Finally, the components of the newly formed edge tuple are sampled from the five learnt categorical distributions.

In contrast to the methods mentioned above, which assume that the elements in each tuple $s_i = (\alpha(u), \alpha(v), F_u, F_v, E^i_{u,v})$ are independent of each other, Bacciu et al. [5] assume the existence of node dependence in a tuple. This method deals with homogeneous graphs without considering the node/edge categories, by representing each tuple in the sequence as $s_i = (\alpha(u), \alpha(v))$ and formalizing the distribution as $p(s_i|s_{<i}) = p(\alpha(u)|s_{<i})p(\alpha(v)|\alpha(u), s_{<i})$. Then, the first node is sampled in the same way as in Eq. 18, while the second node in the tuple is sampled as follows:

$$\alpha(v) \sim Cat(\theta_{\alpha(v)}); \quad \theta_{\alpha(v)} = f_{\alpha(v)}(h^{(i)}_G, g_{\text{emb}}(\alpha(u))), \tag{21}$$

where the function g_{emb} is used for embedding the index of the first generated node u in the pair.

3.1.3 Graph-motif-sequence-based. Several methods [48, 57, 81, 105] are proposed to represent a graph G as a sequence of graph motifs as $Seq(G) = \{C_1, ..., C_M\}$, where a block of nodes and edges that constitute each graph motif C_i are generated at each step, as shown in Fig. 3 (c). The varying size of graph motif(i.e. the number of nodes in a graph motif) along with the sampling overlap size (i.e. the overlap between two graph motifs) can allow for the exploration of the efficiency-quality trade-off of the generation model. A key problem in graph motif-based methods is how to connect the newly generated graph motif to the graph portion that has already been generated, considering that there are many potential ways in linking two sub-graphs. This is mainly depends on the definition of the graph motifs. Currently, there are two ways to solve this problem.

One of the ways is designed for generating general graphs; it is similar to the traversal-based node-sequence generation by generating the adjacent vectors for each edge, such as the GraphRNN [142], except for the generation of several nodes instead of one per step. As described in Section 3.1.1, a graph G is represented as a sequence of node-based tuples as $G = \{s_1, ..., s_N\}$, where $s_i = (v^\pi_i, A^\pi_{i,\cdot})$

is generated per step. Based on this node sequence, Liao et al. [81] (GRANs) regard every B recursive nodes tuples as a graph motif C_i and generates each block per step. In this way, the generated nodes in the new graph motif follow the ordering of the nodes in the whole graph and contain all the connection information of the existing and newly generated nodes. To formalize the dependency among the existing and newly generated nodes, GRANs proposes an MPNN-based model to generate the adjacent edge vectors. Specifically, for the t-th generation step, a graph G_t that contains the already-generated graph with $B \cdot (t-1)$ nodes and the edges among these nodes, as well as the B nodes in the newly generated graph motif is constructed. For these new B nodes, edges are initially fully added to connect them with each other and the previous $B \cdot (t-1)$ nodes. The node-level hidden states of the newly added B nodes are all initialized with 0. Then an MPNN-based graph neural network (GNN) [115] on this augmented graph is used to update the nodes' hidden states by encoding the graph structure. After several rounds of message passing implementation based on a GNN, the node-level hidden states of both the existing and newly added nodes are used to infer the final distribution of the newly added edges as follows:

$$p(C_t|C_{<t}) = \prod_{B(t-1)<i\leq B} \prod_{1\leq j\leq i} p(A^{\pi}_{i,j}|C_{<t}) = \beta \prod_{B(t-1)<i\leq B} \prod_{1\leq j\leq i} \theta_{i,j}, \tag{22}$$

$$\theta_{i,j} = Sigmoid(MLP_{\theta}(h_{v_i} - h_{v_j})), \tag{23}$$

where $\theta_{i,j}$ parameterizes the Bernoulli distribution for the edge existence through the MLP-based function MLP_{θ}, which takes the node-level hidden states as input.

The definition of graph motifs can also involve domain knowledge, such as in the situation of molecules (i.e., graph of atoms) [57, 105], where the sequence of the graph motifs is generated based on an RNN model. Jin et al. [57] propose the Junction-Tree-VAE by first generating a tree-structured scaffold over chemical substructures, and then combining them into a molecule with an MPNN. Specifically, a Tree Decomposition of Molecules algorithm [110] tailored for molecules to decompose the graph G into several graph motifs C_i is followed, and each C_i is regarded as a node in the tree structure T. To generate a graph G, a T is first generated and then converted into the final graph. The decoder for generating a T consists of both *topology prediction function* and *label prediction function*. The topology prediction function models the probability of the current node to have a child, and the label prediction function models a distribution of the labels of all types of C_i. When reproducing a molecular graph G that underlies the predicted junction tree T, since each motif contains several atoms, the neighboring motifs C_i and C_j can be attached to each other as sub-graphs in many potential ways. To solve this, a scoring function over all the candidates graphs is proposed, and the optimal one that maximizes the scoring function is the final generated graph.

Podda et al. [105] also deal with the molecule generation problem but with a different way of defining the graph motifs. To break a molecule into a sequence of fragments C_i , they leverage the breaking of retrosynthetically interesting chemical substructures (BRICS) algorithm [25], which breaks strategic bonds in a molecule that matches a set of chemical reactions. Specifically, their fragmentation algorithm works by scanning atoms in a sequence from left to right in the order imposed by the simplified molecular input line entry system (SMILES) encoding [133]. As soon as a breakable bond (according to the BRICS rules) is encountered during the scanning process, the molecule is broken into two at that bond. After that, the leftmost fragment is collected, and the process repeats on the rightmost fragment in a recursive fashion. Since the fragment extraction is ordered from left to right according to the SMILES representation, it is possible to reconstruct the original molecule from a sequence of fragments. In this way, they successfully represent a molecule as a sequence, and view a sequence of fragments as a âĂIJsentenceâĂİ; in addition they learn to generate the sentence similar to the work proposed by Bowman et al. [15] based on skip-gram embedding methods [75] and gated recurrent units (GRUs) [21].

3.1.4 Rule-Sequence-based. Several methods [22, 73] chose to generate a sequence of production rules or commands, guided by which graph can be constructed sequentially. There are some structured data that often come with formal grammars (e.g. molecule), which results in strict semantic constrain. Thus, to enforce the semantic validity of the generated graphs, graph generation is transformed into generating their parse trees that are derived from context free grammar (CFG), while the parse tree can be further expressed as a sequence of rules based on a pre-defined order.

Kusner et al. [73] propose generating a parse tree that describes a discrete object (e.g. arithmetic expressions and molecule) by a grammar; they also proposed a graph generation method named GrammerVAE. An example of using the parse tree for molecule generation: to encode the parse tree, they decompose it into a sequence of production rules by performing a pre-ordered traversal on its branches from left-to-right, and then convert these rules into one-hot indicator vectors, where each dimension corresponds to a rule in the SMILES grammar. The deep convolutional neural network is then mapped into a continuous latent vector z. While decoding, the continuous vector z is passed through an RNN which produces a set of unnormalized log probability vectors (or âĂŸlogitsâĂŹ). Each dimension of the logit vectors corresponds to a production rule in the grammar. The model generates the parse trees directly in a top-down direction, by repeatedly expanding the tree with its production rules. The molecules are also generated by following the rules generated sequentially, as shown in Fig. 3 (d). Although the CFG provides a mechanism for generating syntactic valid objects, it is still incapable of guaranteeing the model for generating semantic valid objects [73]. To deal with this limitation, Dai et al. [22] propose the syntax-directed variational autoencoder (SD-VAE), in which a semantic restriction component is advanced to the stage of syntax tree generator. This allows for a the generator with both syntactic and semantic validity.

3.2 Generating a Graph in One Shot

These methods learn to map each whole graph into a single latent representation which follows some probabilistic distribution in latent space. Each whole graph can then be generated by directly sampling from this probabilistic distribution in one step. The core issue of these methods is usually how to jointly generate graph topology together with node/edge attributes (if at all). Considering that the graph topology can usually be represented in terms of adjacency matrix and edge list, the existing methods can be categorized as adjacency-based and edge-list based. The former one focuses on directly generating the whole adjacency matrix, while the latter generates the graph topology by examining the existence of edges corresponding to different pairs of nodes.

3.2.1 Adjacency-based. Adjacency one-shot method assumes complex dependence among the graphs and generates the whole graph in one step but considering the interactions among nodes and edges. Adjacency one-shot method varies based on the decoding techniques where the adjacent matrix A^π or edge attributes tensor E^π and node attribute matrix F^π are jointly generated from a graph-level latent representation z. The main challenge is how to ensure correlation among elements of a graph in order to pursue of global properties. In terms of the techniques to tackle this challenge, there are three categories of adjacency one-shot methods elaborated as follows.

MLP-based methods. Most of the one-shot graph generation techniques involves simply constructing the graph decoder $g(z)$ using MLP [3, 24, 34, 89, 106, 119], where the models' parameters can be optimized under common frameworks such as VAE and GAN. The MLP-based models ingest a latent graph representation $z \sim p(z)$ and simultaneously output adjacent matrix A^π and node attribute F^π, as shown in Fig. 4 (a). Specifically, the generator $g(z)$ takes D-dimensional vectors $z \in \mathbb{R}^D$ sampled from a statistical distribution such as standard normal distribution and outputs graphs. For each z, $g(z)$ outputs two continuous and dense objects: $\tilde{A}^\pi$, which defines edge attributes and $\tilde{F}^\pi$, which denotes node attributes through two simple MLPs. Both $\tilde{A}^\pi$ and $\tilde{F}^\pi$ have a probabilistic interpretation since each node and edge attribute is represented with probabilities of categorical

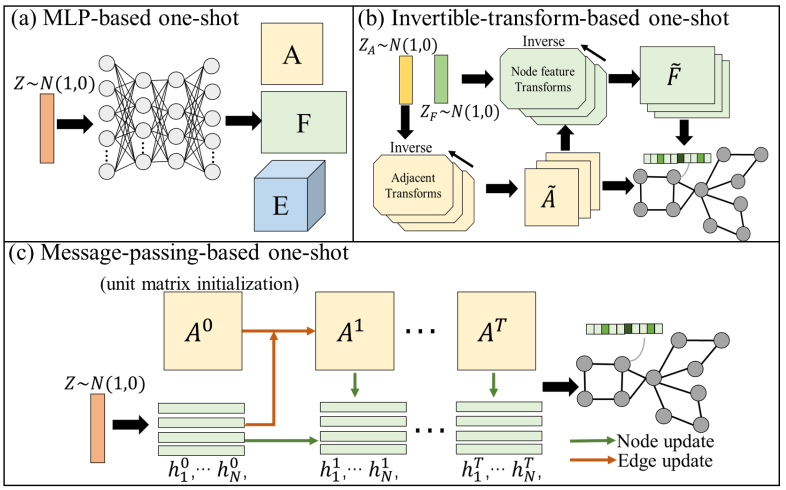

Fig. 4. Three common techniques for adjacent-based one-shot generation

distributions of types. To generate the final graph, it is required to obtain the discrete-valued objects A^π and F^π from $\tilde{A}^\pi$ and $\tilde{F}^\pi$, respectively. The existing works have two ways to realize this step detailed as follows.

In the first way, the existing works [3, 89, 119] use sigmoid activation function to compute A^π and F^π during the training time. At test time, the discrete-valued estimate A^π and F^π can be obtained by taking edge- and node-wise argmax in $\tilde{A}^\pi$ and $\tilde{F}^\pi$. In the other way, existing works [24, 34, 106] leverage categorical reparameterization with the Gumbel-Softmax [55, 91], which is to sample from a categorical distribution during the forward pass (i.e., $F_i^\pi \sim Cat(\tilde{F}_i^\pi)$ and $A_{ij}^\pi = Cat(\tilde{A}_{ij}^\pi)$) and the original continuous-valued $\tilde{A}^\pi$ and $\tilde{F}^\pi$ in the backward pass. In this way, these methods can perform continuous-valued operations during the training procedure and do the categorical sampling procedure to finally generate the F and A.

Message-passing-based methods. Message-passing-based methods generate graphs by iteratively refining the graph topology and node representations of the initialized graph through the MPNN. Specifically, based on the latent representation z sampled from a simple distribution (e.g., Gaussian), we usually first generate an initialized adjacent matrix A^0 and the initialized node latent representations $H^0 \in \mathbb{R}^{N\times L}$, where L refers to the length of each node representation (here we omit the node ordering symbol π for clarity). Then A^0 and H^0 are updated though MPNN with multiple layers for generating the final graph, as shown in Fig. 4 (c). Existing methods leverage common generative frameworks such as VAE and GANs [16, 36, 49], or have a plain framework based on the score-based generative process [101].

For works utilizing common generative frameworks such as VAE and GAN, the decoder is implemented as follows [16, 36, 49]. Normally, the first step is about projecting the initial latent representation z from the fixed dimensional latent space to an initial state h_i^0 for each node through MLP-based or RNN-based networks. A fully-connected graph is also initialized with the same latent values of each entry $A_{i,j}^0$ in A^0. Next, using the initialized graph we can perform message passing on both the node and edge representations for updating $A_{i,j}^{l+1}$ and h_i^{l+1} at layer $l+1$ as:

$$A_{i,j}^{l+1} = A_{i,j}^l + ReLu(v_1 A_{i,j}^l + v_2 h_i^l + v_3 h_j^l); \quad h_i^{l+1} = h_i^l + ReLu(w_1 h_i^l + \sum\nolimits_j^N \eta_{i,j} w_2 h_j^l), \tag{24}$$

where v_1, v_2, v_3, w_1 and w_2 are trainable parameters. Finally, after T layers' updating, the outputs $A_{i,j}^T$ and F_i^T are used to parameterize the categorical distributions of each edge and node, based on which each edge $A_{i,j}$ and node F_i are generated through categorical sampling introduced above.

For the score-based generative modeling process, the core is to design a plain graph generation framework based on score function [101]. Specifically, existing methods usually first sample N, which is the number of nodes to be generated, from the empirical distribution of the number of nodes in the training dataset. Then they sample the adjacent matrix A with annealed Langevin dynamics [121]. Specifically, they first initialize the adjacent matrix as A^0 with each element following a Normal distribution. Then, they update the adjacent matrix by iteratively sampling from a series of trained conditional score models $\{s_\theta(A;\sigma_i)\}_{i=1}^K$ (i.e. a function parameterized by θ) using Langevin dynamics. Here $\{\sigma_i\}_{i=1}^K$ is a sequence of noise levels and K refers to the number of noise levels. To implement the score function $s_\theta(A;\sigma_i)$, MPNN-based score networks, as described in Eq. 24 are introduced. Formally, the output of the score function is given:

$$\hat{A}_{i,j} = \text{Concate}(A_{i,j}^{(l+1)} | l = 0, ..., T); \quad s_\theta(A;\sigma_i) = \text{ReLu}((W\hat{A} + b)\alpha_i + \beta_i) \tag{25}$$

where T is the number of MPNN layers of the score function network, and α_i and β_i are learnable parameters of MLP-based output layer for each noise level σ_i as $\sigma_i = (\alpha_i, \beta_i)$. W and b are shared weights and bias respectively of the output layers of all score function models. Concate in the above equation refers to the operation that concatenates all $A_{i,j}^{(l+1)}$ into a vector.

Invertible-transform-based methods Flow-based generative methods can also do one-shot generation, by a unique invertible function between graph G and the latent prior z sampling from a simple distribution (e.g., Gaussian), as shown in Fig. 4 (b). Based on vanilla flow-based learning techniques introduced in Section 2.3, special forward transformation $G \longrightarrow z$ and backward transformation $z \longrightarrow G$ needs to be designed.

Madhawa et al. [92] propose the first flow-based one-shot graph generation model called GraphNVP. To get $z = (z^F, z^A)$ from $G = (A, F)$ in the *forward transformation*, they first convert the discrete variable A and F into continuous variable A' and F' by adding real-valued noise (same as that in Eq. 12), which is known as *dequantization*. Then two types of reversible affine coupling layers: adjacency coupling layers and node attribute coupling layers are utilized to transform the adjacency matrix A' and the node attribute matrix F' into latent representations z_A and z_F, respectively. The lth reversible coupling layers are designed as follows:

$$z_F^l[i] = z_F^{l-1}[i] \odot \exp(s_F(z_F^{l-1}[i], A)) + t_F(z_F^{l-1}[i], A) \tag{26}$$

$$z_A^l[i,j] = z_A^{l-1}[i,j] \odot \exp(s_A(z_A^{l-1}[i,j])) + t_A(z_A^{l-1}[i,j]) \tag{27}$$

where $z_F^0 = X'$ and $z_A^0 = A'$. $z_F^l[i]$ refers to the ith entry of z_F^l; $\odot$ denotes element-wise multiplication. Functions $s_A(\cdot)$ and $t_A(\cdot)$ stand for scale and translation operations which can be implemented based on MPNN, and $s_F(\cdot)$, $t_F(\cdot)$ can be implemented based on MLP networks. To get $G = (F, A)$ from $z = (z_F, z_A)$ in the *backward transformation*, the reversed operation is conducted based on the above forward transformation operation in Eq. 26 and 27. Specifically, after drawing random samples z_A and z_F, a sequence of inverted adjacency coupling layers is applied on z_A for a probabilistic adjacency matrix $\tilde{A}$, from which a discrete adjacency matrix A is constructed by taking node-wise and edge-wise argmax operation. Next a probabilistic feature matrix $\tilde{F}$ is generated given the sampled z_F and the generated adjacency matrix A through a sequence of inverted node attribute coupling layers. Likewise, the node-wise argmax of $\tilde{F}$ is used to get discrete feature matrix F.

Honda et al. [54] propose a graph residual flow (GRF) with more flexible and complex non-linear mappings than the above mentioned coupling flows in GraphNVP. The forward transformation is designed as follows:

$$z_F^l = z_F^{l-1} + \mathcal{R}_F^l(z_F^{l-1}, A); \quad z_A^l = z_A^{l-1} + \mathcal{R}_A^l(z_A^{l-1}), \tag{28}$$

where $\mathcal{R}_F^l$ and $\mathcal{R}_A^l$ are the residual blocks for node attribute matrix F and adjacency matrix A at lth layer. The residual block is implemented based on GCNs [70] and is proved to be invertible. The

backward transformation process is similar to the GraphNVP except for the computation of the inverse of the z_A and z_F by the fixed-point iteration [9] based on the invertible residual.

3.2.2 Edge-list-based. This category typically requires a generative model that learns edge probabilities, based on which all the edges are generated independently. These methods are usually used in learning from one large-scale graph and learning to generate the synthetic one given the known nodes. In terms of how the edge probability are generated, existing works are further categorized into two, namely *random-walk-based* [11, 18, 38, 143] and *node-similarity-based* [47, 69, 85, 113, 150].

Random-walk-based. This type of methods generate the edge probability based on a score matrix, which is calculated by the frequency of each edge that appears in a set of generated random walks. Bojchevski et al. [11] propose NetGAN to mimic the large-scale real-world networks. Specifically, at the first step, a GAN-based generative model is used to learn the distribution of random walks over the observed graph, and then it generates a set of random walks. At the second step, a score matrix $S \in \mathcal{R}^{N\times N}$ is constructed, where each entry denotes the counts of an edge that appears in the set of generated random walks. At last, based on the score matrix, the edge probability matrix $\tilde{A}$ is calculated as $\tilde{A}_{i,j} = S_{i,j}/\sum_{u,v}^{N} S_{u,v}$, which will be used to generate individual edge $A_{i,j}$, based on efficient sampling processes.

Following this, some works propose improving the NetGAN, by changing the way to choose the first node in starting a random walk [18] or learning spatial-temporal random walks for spatial-temporal graph generation [143]. Gamage et al. [38] generalize the NetGAN by adding two motif-biased random-walk GANs. The edge probability is thus calculated based on the score matrices from three sets of random walks (i.e. $S^{(1)}$, $S^{(2)}$, and $S^{(3)}$) that are generated from the three GANs. To sample each edge, one view $S^{(k)}$ is randomly selected from the three scores matrices. Based on $S^{(k)}$, edge probability $\tilde{A}_{i,j}$ is calculated as $\tilde{A}_{i,j} = S^{(k)}_{i,j}/\sum_{u,v}^{N} S_{u,v}$.

Node-similarity-based. These methods generate the edge probability based on pairwise relationships between the given or sampled nodes' embedding (as in [69]). Specifically, the probability adjacent matrix $\tilde{A}$ is generated given the node representations $Z \in \mathcal{R}^{N\times L}$, where $Z_i \in \mathcal{R}^{L}$ refers to the latent representation for node v_i. $\tilde{A}$ will be used to generate individual edge $A_{i,j}$, based on efficient sampling processes. Existing methods differ on how to calculate $\tilde{A}$.

Several works [47, 69, 150] compute $\tilde{A}_{i,j}$ based on the inner-product operations of two node embedding Z_i and Z_j. This reflects the idea that nodes that are close in the embedding space should have a high probability of being connected. These works require a setting where node set is pre-defined and the node attribute F is known in advance. Specifically, by first sampling node latent representation Z_i from the standard normal distribution, Kipf and Welling [69] calculate the probability adjacent matrix as $\tilde{A} = \text{Sigmoid}(ZZ^T)$. The adjacent matrix A is then sampled from $\tilde{A}$ which parameterizes the Bernoulli distribution of the edge existence, as similar to work by Zou and Lerman [150]. To further consider the complex dependence among generated edges, Grover et al. [47] propose an iterative two-step approach that alternates between defining an intermediate graph and then gradually refining the graph through message passing. Formally, given a latent matrix Z and an input feature matrix F, they are iterated over the following sequence of operations:

$$\widehat{A} = (ZZ^T)/\parallel Z \parallel^2 + \mathbf{1}\mathbf{1}^T; \quad Z^* = GNN(\widehat{A}, Cancate(Z, X))), \tag{29}$$

where the first step constructs an intermediate weighted adjacency matrix $\widehat{A}$ by operating an inner-product and adding an additional constant vector of value(i.e., $\mathbf{1}$) to ensure that entries are non-negative; the second step performs a pass through a parameterized graph neural network (GNN) (as shown in Eq. 29). The above sequence is repeated to gradually refine the node attribute matrix Z^*. The distribution of the individual edge $A_{i,j}$ is also assumed as a Bernoulli distribution, which is parameterized by the value calculated through $\text{Sigmoid}(Z^*_i Z^*_j)$.

Other works [85, 113] compute $\tilde{A}_{i,j}$ by measuring the closeness of two node latent representations with the ℓ_2 norm. Specifically, Liu et al. [85] propose a decoder for calculating $\tilde{A}_{i,j}$ as:

$$\tilde{A}_{i,j} = 1/(1 + exp(C(\| Z_i - Z_j \|_2^2 - 1))), \tag{30}$$

where C is called a temperature hyperparameter. Salha et al. [113] propose a gravity-inspired decoding schema in the generative model as follows:

$$\tilde{A}_{i,j} = \text{Sigmoid}(m_j - log \| Z_i - Z_j \|_2^2), \tag{31}$$

where m_j is the gravity scale of node v_j learned from the input graph by its featured encoder.

Shi et al. [117] propose computing the probability adjacent matrix $\tilde{A}$ by generating the triad edges among three nodes together based on the well-known *triadic closure property* which is exhibited in many real-world networks: for any three nodes v_i, v_j and v_k in a graph, if there are edges between v_i, v_j and v_k, v_i, it is likely that an edge also exists between v_k and v_j. When generating, they first randomly sample N node representations from normal distribution. Then K triads are randomly sampled from these node representations. Finally, the predictions are averaged over all K triads. Specifically, to sample a triad, first, a node v_i is randomly selected from the node set. With a predefined probability p, the next node v_j is randomly selected from $Ne(i)$ (neighbors of node v_i based on the current generated graph), and with a probability $1 - p$, v_j is randomly selected from a far-away node not in $Ne(i)$. Likewise, the third node v_k is sampled based on node v_j. After getting a triad, a triad decoder is used to predict the elements in $\tilde{A}$ corresponding to the three constituent edges. The triad decoder $f(\cdot)$ consists of fully connected layers and the classic convolution layers with inputs of Z_i, Z_j and Z_k. The output of the decoder is finally merged with the three inner products constructed from Z_i, Z_j and Z_k as follows:

$$[\tilde{A}_{i,j}, \tilde{A}_{i,k}, \tilde{A}_{j,k}] = \text{Sigmoid}(f(Z_i, Z_j, Z_k) + [Z_i Z_j^T, Z_i Z_k^T, Z_j Z_k^T]). \tag{32}$$

4 CONDITIONAL DEEP GENERATIVE MODELS FOR GRAPH GENERATION

The goal of conditional deep graph generation is to learn a conditional distribution $p_{\text{model}}(G|y)$ based on a set of observed realistic graphs G along with their corresponding auxiliary information, namely a condition y by deep generative models. The auxiliary information could be category labels, semantic context, graph from other distribution spaces, etc.

Compared with unconditional deep graph generation, in addition to the challenge in generating graphs, conditional generation needs to consider how to extract the features from the given condition and integrate it into the generation of graphs. Thus, to systematically introduce the existing conditional deep graph generative models, we mainly focus on describing how these methods deal with the condition information. Since the conditions could be any form of auxiliary information, we categorize the problem in terms of types of conditions, including **graphs**, **sequence**, and **semantic context**, shown as the yellow parts of the taxonomy tree in Fig. 1.

Table 2. Deep Generative-based Methods for Conditional Graph Generation

Conditioning objects		Techniques of encoding conditions	References
Graphs	Edge Transformation	Adjacent-based edge convolution	[28, 39, 51, 147]
	Node-edge Co-transformation	Embedding-based	[59, 62, 93, 124]
		Editing-based	[58, 140, 148]
Sequence		RNN-based encoding	[19, 84, 130, 139]
Context Semantics		Concatenation with latent representation	[60, 79, 80, 138]

4.1 Conditioning on Graphs

The problem of deep graph generation conditioning on another graph can also be called as deep graph transformation (also known as deep graph translation) problem [51]. It aims at translating

an input graph G_S in the source domain to the distribution of corresponding output graphs G_T in the target domain based on deep graph generative models. Considering the entities that are being transformed during the translation process, there are two categories of works in the domain of deep graph generation conditioning on graphs: *edge transformation* and *node-edge-co-transformation*[1].

4.1.1 Edge Transformation. The problem of edge transformation is to generate the graph topology and edge attributes of the target graph conditioning on the input graph. It requires the edge set $\mathcal{E}$ and edge attributes E to change while the graph node set and node attributes are fixed during the translation process as: $\mathcal{T} : G_S(\mathcal{V}, \mathcal{E}_S, F, E_S) \rightarrow G_T(\mathcal{V}, \mathcal{E}_T, F, E_T)$. The edge transformation problem has a wide range of real-world applications, such as modeling chemical reactions [140], protein folding [3] and malware cyber-network synthesis [51]. Existing works adopt different frameworks to model the translation process.

Some works utilize the encoder-decoder framework by learning abstract latent representation of the input graph through the encoder and then generating the target graph based on these hidden information through the decoder [39, 51]. Guo et al. [51] propose a GAN-based model for graph topology transformation. The proposed GT-GAN consists of a graph translator and a conditional graph discriminator. The graph translator includes two parts: graph encoder and graph decoder. A graph convolution neural net [65] is extended to serve as the graph encoder in order to embed the input graph into node-level representations while a new graph deconvolution net is used as the decoder to generate the target graph. Specifically, the encoder consists of edge-to-edge and edge-to-node convolution layers, which first extracts latent edge-level representations and then node-level representations $\{H_i\}_{i=1}^N$, where $H_i \in \mathbb{R}^L$ refers to the latent representation of node v_i. The decoder consists of node-to-edge and edge-to-edge deconvolution layers to first get each edge representations $\hat{E}_{i,j}$ based on H_i and H_j, and then the final edge attribute tensor E based on $\hat{E}$. They also leverage the skip-net structure [112] between the encoder and decoder so that the sample-specific representations in the encoder can be directly passed over through skip connection to the decoderâĂŹs layers while the sample invariant mapping will be learned in the encoder-decoder structure. To further handle the situation when the pairing information of the input and output is not available, Gao et al. [39] utilize the same encoder and decoder in GT-GAN and propose dealing with the unpaired graph transformation problems based on Cycle-GAN [149].

Zhou et al. [147] propose modeling the underlying distribution of graph structures of the input graph at different levels of granularity, and then âĂIJtransferringâĂİ such hierarchical distribution from the graphs in the source domain to a unique graph in the target domain. The input graph is characterized as several coarse-grained graphs by aggregating the strongly coupled nodes with a small algebraic distance to form coarser nodes. Overall, the framework can be separated into three stages. At the first step, the coarse-grained graphs at K levels of granularity are constructed from the input graph adjacent matrix A_S. The adjacent matrix of the coarse-grained graph $A_S^{(l)} \in \mathbb{R}^{N^{(l)} \times N^{(l)}}$ at the kth layer is defined as follows:

$$A_S^{(k)} = {P^{(k-1)}}^T \dots {P^{(1)}}^T A_S P^{(1)} \dots P^{(k-1)}, \tag{33}$$

where $P^{(k)} \in \mathbb{R}^{N^{(l)} \times N^{(l)}}$ is a coarse-grained operator for the kth level and $N^{(l)}$ refers to the number of nodes of the coarse-grained graph at level l. In the next stage, each coarse-grained graph at each level k will be reconstructed back into a fine graph adjacent matrix $A_T^{(k)} \in \mathbb{R}^{N^{(l)} \times N^{(l)}}$ as follows:

$$A_T^{(k)} = {R^{(1)}}^T \dots {R^{(k-1)}}^T A_S^{(k)} R^{(k-1)} \dots R^{(1)}, \tag{34}$$

where $R^{(k)} \in \mathbb{R}^{N^{(l)} \times N^{(l)}}$ is the reconstruction operator for the kth level. Thus all the reconstructed fine graphs at each layer are in the same scale. Finally, these graphs are aggregated into a unique

[1]Node transformation is not the focus of our survey since the graph topology has not been changed. That domain is highly related to node embedding using techniques such as graph convolution neural networks [40, 70]

one by a linear function to get the final adjacent matrix as follows: $A_T = \sum_{k=1}^{K} w^k A_T^{(k)} + b^k$, where $w^k \in \mathbb{R}$ and $b^k \in \mathbb{R}$ are weights and bias.

Do et al. [28] propose a plain transformation framework named graph transformation policy network (GTPN) for the chemical reaction prediction by formalizing the graph transformation process as a Markov decision process and modifying the input source graph through several iterations. Thus, given a graph of reactant molecule as input graph, G_S, they predict a set of reaction triples that transforms G_S into a graph of product molecule G_T. This process is modeled as a sequence consisting of tuples like $(\zeta^t, v_i^t, v_j^t, b^t)$ where v_i^t and v_j^t are the selected nodes from node set at step t whose connection needs to be modified; b^t is the new edge type of (v_i^t, v_j^t), and ζ^t is a binary signal that indicates the end of the sequence. At every step of the forward pass, the GTPN model performs seven major steps: 1) computing the atom representation vectors through MPNN, 2) computing the most possible K reaction atom pairs, 3) Predicting the continuation signal ζ, 4) predicting the reaction atom pair (v_i, v_j), 5) predicting a new bond type b^t of this atom pair, 6) updating the atom representations, and 7) updating the recurrent state.

4.1.2 Node-edge Co-transformation. The problem of node-edge co-transformation (NECT) is generating the node and edge attributes of the target graph conditioning on those of the input graph. It requires that both the nodes and edges can vary during the transformation process between the source graph and the generated target graph as follows: $\mathcal{T} : G_S(\mathcal{V}_S, \mathcal{E}_S, F_S, E_S) \rightarrow G_T(\mathcal{V}_T, \mathcal{E}_T, F_T, E_T)$. In terms of the techniques on how the input graph is assimilated to generate the target graph, there are two categories: one is embedding-based and the other is editing-based.

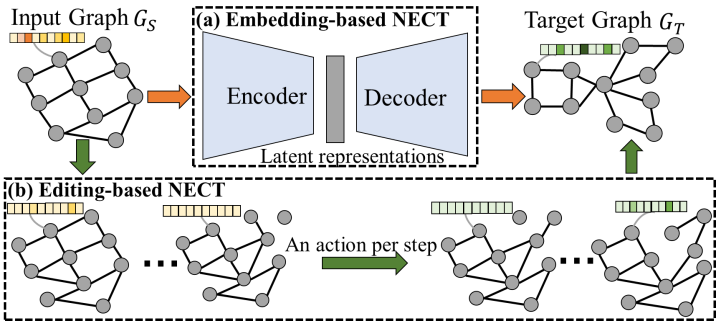

Fig. 5. Embedding-based NECT vs Modification-based NECT

Embedding-based NECT. The embedding-based NECT normally encodes the source graph into latent representations containing higher-level rich information of the input graph by an encoder, which is then decoded into the target graph by a decoder, as shown in Fig. 5 (a) [58, 59, 62, 93, 124]. These methods are usually based on conditional VAEs [120] and conditional GANs [95].

Kaluza et al. [62] propose exploring the latent spaces of directed acyclic graphs (DAGs) and develops a neural network-based DAG-to-DAG translation model, where both the domain and the range of the target function are DAG spaces. The encoder M_{encode} is borrowed from the deep-gated DAG recursive neural network (DG-DAGRNN) [2], which generalizes stacked RNNs on sequences to DAG structures. Each layer of the DG-DAGRNN consists of gated recurrent units (GRUs), which are repeated for each node v_i. The encoder outputs an embedding $h = M_{\text{encode}}(G_S)$, which serves as the input of the DAG decoder. The decoder follows the local-based node-sequential generation style as described in Section 3.1.1. Specifically, first, the number of nodes N of the target graph is predicted by an MLP network with the input of h. Also, the hidden state of the target graph is initialized with h. Then at each step, a node v_i as well as its corresponding edge set $\{e_{i,j}\}_{j<i}$ are generated based on the hidden state at each step until an end node is added to the graph or the

number of nodes exceeds a predefined threshold. Sun and Li [124] propose a general graph-to-graph model by first formalizing the graph into a DAG without loss of information and utilize recurrent based model to translate this DAG. They embeds the topology of the input graph into the node representations by exerting a topology constraint, which results in a topology-flow encoder. Their decoder follows the same node sequential-based generation as proposed by You et al. [142].

There are also some embedding-based graph translation methods that represent the graph as a set of graph motifs, which are usually targeted for the task of molecule optimization [59, 93]. Jin et al. [59] extend the junction-tree variational auto-encoder (JT-VAE) [57] to an encoder-decoder architecture for learning graph-to-graph mappings. In terms of model architecture, the encoder is a graph message passing network that embeds nodes in both the tree and graph into continuous vectors. The decoder consists of a tree-structured decoder for predicting junction trees, and a graph decoder that learns to combine clusters in the predicted junction tree into a molecule. Their key departures from JT-VAE include a unified encoder architecture for trees and graphs, along with an attention mechanism in the tree decoding process. Maziarka et al. [93] also employ the encoder and decoder mechanism of the JT-VAE, but build these components on the CycleGAN [149].

Editing-based NECT. Different from the encoder-decoder framework, modification-based NECT directly modifies the input graph iteratively to generate the target graphs [52, 140, 148], as shown in Fig. 5 (b). There are two ways to realize the process of editing the source graph. One is utilizing an RL agent to sequentially modify the source graph based on a formulated Markov decision process[140, 148] as described in Section 2.4. The modification at each step will be selected from the defined action set, including "add nodeâĂİ, "add edge", "remove bonds" et al. The other is to update nodes and edges from the source graph synchronously in a one-shot manner through the MPNN using several iterations [52].

You et al. [140] propose the graph convolutional policy network (GCPN), a general graph convolutional network based model for goal-directed graph generation through reinforcement learning. The model is trained to optimize the domain-specific property of the source molecule through policy gradient, and acts in an environment that incorporates domain-specific rules. They define a distinct, fixed-dimension and homogeneous action space amenable to reinforcement learning, where an action is analogous to link prediction. Specifically, they first define a set of scaffold subgraphs $\{C_1, ..., C_s\}$ based on the source graph. This set acts as a subgraph vocabulary that contains the subgraphs to be added into the target graph during graph generation. Given the modified graph G_t at step t, they define the corresponding extended graph as $G_t \cup C_i$. Based on this definition, an action can either correspond to connecting a new subgraph C_i to a node in G_t or connecting existing nodes within graph G_t.

Zhou et al. [148] also present a framework, named molecule deep Q-networks (MolDQN), for molecule optimization by combining domain knowledge of chemistry and state-of-the-art reinforcement learning techniques (double Q-learning and randomized value functions). They directly define modifications of molecules, thereby ensuring 100% chemical validity. Intuitively, their modification or optimization of a molecule can be done in a step-wise fashion, where each step belongs to one of the following three categories: (1) *atom addition*, (2) *bond* addition, and (3) *bond removal*. Specifically, in the action of *atom addition*, they first define an empty set of atoms $\mathcal{V}_T$ for the target molecule graph. Then they define a valid action as adding an atom in $\mathcal{V}_T$ and also a bond between the added atom and the original molecule wherever possible. In the action of *bond addition*, a bond is added between two atoms in $\mathcal{V}_T$. If there is no bond between the two atoms, the actions between them consist of adding a single, double, or triple bond. If there already exist a bond, additional action changes the bond type by increasing the index of the bond type by one or two. In the action of *bond removal*, they define the valid bond removal action set as the actions that decrease the

bond type index of an existing bond. The transitions include: (1) Triple bond → {Double, Single, No} Bond, (2) Double bond → {Single, No} Bond, and (3) Single bond → {No} Bond.

Guo et al. [52] follow the second way which edits the source graph iteratively, through the generation process similar to the *MPNN-based adjacency-based one-shot method* in Section 3.2.1 and Fig. 5 (c) for unconditional deep graph generation, except for taking the graph in the source domain as input instead of the initialized graph. The transformation process is modeled by several stages and each stage generates an immediate graph. Specifically, at each stage t, there are two paths, namely node translation and edge translation paths. In node translation path, an MLP-based *influence-function* is used for calculating the influence $I_i^{(t)}$ on each node v_i from its neighboring nodes, and another MLP-based *updating-function* is used for updating the node attribute as $F_i^{(t)}$ with the input of influence $I_i^{(t)}$. The edge translation path is constructed in the same way as the node translation path, where each edge is generated by the influence from its adjacent edges.

4.2 Conditioning on Sequence

The problem of deep graph generation conditioning on a sequence can be formalized as the deep sequence-to-graph transformation problem. It aims to generate the target graph G_T conditioning on an input sequence X. The deep sequence-to-graph problem is usually observed in domains such as NLP [19, 130] and time series mining [84, 139].

The existing methods handle the semantic parsing task [19, 130] by transforming a sequence-to-graph problem into a sequence-to-sequence problem and utilizing the classical RNN-based encoder-decoder model to learn this mapping. Chen et al. [19] propose a neural semantic parsing approach named *Sequence-to-Action*, which models semantic parsing as an end-to-end semantic graph generation process. Given a sentence $X = \{x_1, ..., x_m\}$, the *Sequence-to-Action* model generates a sequence of actions $Y = \{y_1, .., y_m\}$ for constructing the correct semantic graph. A semantic graph consists of nodes (including variables, entities, types) and edges (semantic relations), with some universal operations (e.g., argmax, argmin, count, sum, and not). To generate a semantic graph, they define six types of actions: *Add Variable Node*, *Add Entity Node*, *Add Type Node*, *Add Edge*, *Operation Function* and *Argument Action*. In this way, the generated parse tree is represented as a sequence, and the sequence-to-graph problem is transformed into a sequence-to-sequence problem. Then the attention-based sequence-to-sequence RNN model [7] with an encoder and decoder is utilized, where the encoder converts the input sequence X to a sequence of context sensitive vectors $\{b_1, ..., b_m\}$ using a bidirectional RNN and a classical attention-based decoder generates action sequence Y based on the context sensitive vectors. Wang et al. [130] also represent the generation of parse tree as a sequence of actions and borrow the successful idea from the Stack-LSTM neural parsing model [30], They present two non-trivial improvements, namely Bi-LSTM subtraction and incremental tree-LSTM, to better learn a sequence-to-sequence mapping.

Other methods handle the problem of Time Series Conditioned Graph Generation [84, 139]: given an input multivariate time series, the aim is to infer a target relation graph to model the underlying interrelationship between the time series and each node. Yang et al. [139] explore GANs in the conditional setting and propose the novel model of time series conditioned graph generation-generative adversarial networks (TSGG-GAN) for time series conditioned graph generation. Specifically, the generator in a TSGG-GAN adopts a variant of recurrent neural network called simple recurrent units (SRU) [76] to extract essential information from the time series, and uses an MLP to generate the directed weighted graph.

4.3 Conditioning on Semantic context

The problem of deep graph generation conditioning on semantic context aims to generate the target graph G_T conditioning on an input semantic context, which can be usually represented as

additional meta-features. The semantic context can refer to the category, label, modality or any additional information that can be intuitively represented as a vector C. The main issue is deciding where to concatenate or embed the condition representation into the generation process. As a summary, the conditioning information can be added in terms of one or multiple of the following modules: (1) the node state initialization module, (2) the message passing process for MPNN-based decoding, and (3) the conditional distribution parameterization for sequential generating.

Yang et al. [138] propose a novel unified model of graph variational generative adversarial nets, where the conditioning semantic context is inputted into the node state initialization module. Specifically, in the generation process, they first model the embedding Z_i of each node with separate latent distributions. Then, a conditional graph VAE (CGVAE) can be directly constructed by concatenating the condition vector C to each node latent representation Z_i to get the updated node latent representation $\hat{Z}_i$. Thus, the distribution of the individual edge $A_{i,j}$ is assumed as a Bernoulli distribution, which is parameterized by the value $\hat{A}_{i,j}$ and is calculated as $\hat{A}_{i,j} = \text{Sigmoid}(f(\hat{Z}_i)^T f(\hat{Z}_j))$, where $f(\cdot)$ is constructed by a few fully connected layers. Li et al. [79] propose a conditional deep graph generative model that adds the semantic context information into the initialized latent representations Z_i at the beginning of the decoding process.

Li et al. [80] add the context information C into the message passing module in its MPNN-based decoding process. Specifically, they parameterize the decoding process as a Markov process and generate the graph by iteratively refining and updating from the initialized graph. At each step t, an action is conducted based on the current node hidden states $H^t = \{h_1^t, ..., h_N^t\}$. To calculate $h_i^t \in \mathbb{R}^L$ (L denotes the length of the representation) for node v_i in the intermediate graph G_t after each updating of the graph, they utilize message passing network with node message propagation. Thus the context information $C \in \mathbb{R}^K$ is added to the operation of the MPNN layer as follows:

$$h_i^t = W h_i^{t-1} + \Phi \sum\nolimits_{v_j \in N(v_j)} h_j^{t-1} + \Theta C, \tag{35}$$

where $W \in R^{L \times L}$, $\Theta \in R^{L \times L}$ and $\Phi \in R^{K \times L}$ are all learnable weights vectors and K denotes the length of the semantic context vector.

Jonas [60] adds the semantic context as one of the inputs for calculating the conditional distribution parameter at each step during the sequential generating process. The aim is to solve the molecule inverse problem, namely inferring the chemical structure conditioning on the formula and spectra of a molecule, which provide a distinguishable fingerprint of its bond structure. The problem is framed as an MDP and molecules are constructed incrementally one bond at a time based on a deep neural network, where they learn to imitate a “subisomorphic oracle" that knows whether the generated bonds are correct. The context information (i.e. spectra) works in two places. Specifically, they begin with an empty edge set $\mathcal{E}_0$ and sequentially update the edge set to $\mathcal{E}_k$ at each step k by adding an edge sampling from $p(e_{i,j}|\mathcal{E}_{k-1}, \mathcal{V}, C)$. $\mathcal{V}$ denotes the node set that is defined in the given molecule formula. The edge set keeps updating until the existing edges satisfy all the valence constraints of a molecule. The resulting edge set $\mathcal{E}_K$ serves as the candidate graph. For a given spectra C, the process is repeated T times, generating T (potentially different) candidate structures, $\{\mathcal{E}_K^{(i)}\}_{i=1}^T$. Then based on a spectra prediction function $f(\cdot)$, the quality of these candidate structures are evaluated by measuring how close their predicted spectra are to the condition spectra C and the optimal generated graph is finally selected according to $\underset{i}{\text{argmin}} \parallel f(\mathcal{E}_K^{(i)}) - C \parallel_2$.

5 EVALUATION METRICS FOR DEEP GRAPH GENERATION

Evaluating the generated graphs as well as the learnt distribution of graphs are challenging and critical tasks for deep generative models in graph generation problem due to two major reasons: 1) Different from conventional prediction problems where merely deterministic predictions need to be evaluated, deep graph generation requires the evaluation of the performance of graph generation,

which involves stochastic outputs and additional aspects. 2) Graph structured data is much more difficult to evaluate than simple data with matrix/vector structures or semantic data such as images and texts. Thus, we summarize the typical evaluation metrics in evaluating deep generative models for graph generation as follows. We first summarize the general metrics that can be used for both unconditional and conditional deep graph generation, and then introduce the additional metrics that are specially designed for conditional deep graph generation.

5.1 General Evaluation for Deep Graph Generation

To evaluate the quality of the generated graphs, existing literature covers three categories of evaluation metrics, namely statistics-based, classifier-based, and self-quality-based evaluations. The first two evaluation categories require comparison between the generated graph set and test set, while the self-quality evaluation directly evaluates the generated graph set's properties.

5.1.1 Statistics-based. In statistics-based evaluation, the quality of the generated graphs is accessed by computing the distance between the graph statistic distribution of the graphs in the test set and graphs that are generated. We first introduce seven typical graph statistics that measure different properties of graphs and, thereafter introduce the metrics that measure the distance between two distributions regarding different graph statistics.

There are seven typical graph statistics that are used in existing literature, which are summarized as follows: (1) *Node degree distribution*: the empirical node degree distribution of a graph, which could encode its local connectivity patterns. (2) *Clustering coefficient distribution*: the empirical clustering coefficient distribution of a graph. Intuitively, the clustering coefficient of a node is calculated as the ratio of the potential number of triangles the node could be part of to the actual number of triangles the node is part of. (3) *Orbit count distribution*; the distribution of the counts of node 4-orbits of a graph. Intuitively, an orbit count specifies how many of these 4-orbits substructures the node is part of. This measure is useful in understanding if the model is capable of matching higher-order graph statistics, as opposed to node degree and clustering coefficient, which represent measures of local (or close to local) proximity. (4) *Largest connected component*: the size of the largest connected component of the graphs. (5) *Triangle count*: the number of triangles counted in the graph. (6) *Characteristic path length*: the average number of steps along the shortest paths for all node pairs in the graph. (7) *Assortativity*: the Pearson correlation of degrees of connected nodes in the graph.

The first three graph statistics are about distributions of each graph and are always represented as a vector, while the last four graph statistics are represented as scalar values of each graph. Therefore, to evaluate the distance between two sets of graphs in terms of the above distribution statistics, two major metrics are usually utilized in existing literature, which are introduced as follows.

Average Kullback-Leibler Divergence. Considering that each graph set has a set of distributions in terms of a graph property x, we first need to calculate the average distribution of the whole set. To get the average distribution of a graph set, the vectors of counts of the property x of all the graphs in the set are first concatenated. Then the probability densities of the graph property x is calculated based on this concatenated vector for the average node degree distribution of the whole set. Fianlly, the Kullback-Leibler divergence (KL-D [72]) between the average node degree distribution of the generated graph set $P_{\text{ave}}(x)$ and that of the real graph set $Q_{\text{ave}}(x)$ is calculated as:

$$KL - D(P_{\text{ave}}, Q_{\text{ave}}) = -\sum\nolimits_{x \sim P_{\text{ave}}} P_{\text{ave}}(x) log(Q_{\text{ave}}(x)/P_{\text{ave}}(x)). \tag{36}$$

Maximum Mean Discrepancy (MMD) [46]. First, the squared MMD between the graph statistics distribution of the generated graph set P and that of the real graph set Q can be derived as:

$$MMD(P, Q) = \mathbb{E}_{x,y \sim P}[k(x, y)] + \mathbb{E}_{x,y \sim Q}[k(x, y)] - 2\mathbb{E}_{x \sim P, y \sim Q}[k(x, y)], \tag{37}$$

where x, y refer to the graph statistics that are sampled from the two distributions. The kernel $k(*)$ is designed as follows:

$$k(x, y) = exp(W(x, y)/2\sigma^2), \tag{38}$$

where σ refers to the standard deviation of P or Q. Considering the sampled graph statistics x, y are also two distributions; thus, $W(x, y)$ is defined as the Wasserstein distances (WD):

$$W(x, y) = inf_{\gamma \in \prod(x,y)} \mathbb{E}_{(i,j)\sim\gamma}[\| i - j \|], \tag{39}$$

where $\prod(x, y)$ is the set of all measures whose marginals are x and y respectively.

Distance metrics for scalar-valued statistics. The calculation of distance between two sets of graphs in terms of the scalar-valued statistic is much easier than that of distribution statistics. There are two major ways: (1) calculating the difference between the averaged value of the scalar-valued statistic of the generated graph set and that of the real graph set; (2) calculating the distance between the distribution of the scalar-valued statistic of the generated graph set and that of the real graph set. Many distance metrics can be used, such as KL-D, Jensen-Shannon distances (JS), the Hellinger distance (HD), and WD.

5.1.2 Classifier-based. Classifier-based evaluation typically utilizes a graph classifier to evaluate whether the generated graphs follows the same distribution as the real graphs without explicitly defining the graph statistics. Typically, a classifier is trained on the set of real graphs and is tested on the set of generated graphs. It only could be utilized when multiple graph generative models are trained for generating multiple types of graphs, respectively. Here we introduce two existing classifier-based evaluations [83] that are based on graph isomorphism network (GIN) [136]:

Accuracy-based. First, a GIN is pre-trained on the training set consisting of multiple types of graphs previously used for training the generative model. Then for each type of generated graph, the classification accuracy of classifying this type of generated graphs based on the trained GIN is the final evaluation metric.

FrÃľchet Inception Distance (FID)-based. FID computes the distance in the embedding space between two multivariate Gaussian distributions fitted to a generated set and a test set. A lower FID value indicates better generation quality and diversity. For each type of graph, first, the generated and real graphs in the testing set are inputted into the pre-trained GIN to get the graph embeddings. Then the means μ_{G} and covariance matrices $\sum_{\mathrm{G}}$ of the embeddings of the generated graph set, and the means μ_{R} and covariance matrices $\sum_{\mathrm{R}}$ of real graphs in the testing set are estimated. Finally, the FID metric for this type of graphs is computed as follows:

$$FID = \| \mu_{\mathrm{G}} - \mu_{\mathrm{R}} \|_2^2 + \mathrm{Tr}(\sum\nolimits_{\mathrm{G}} + \sum\nolimits_{\mathrm{R}} - 2(\sum\nolimits_{\mathrm{G}} \sum\nolimits_{\mathrm{R}})^{\frac{1}{2}}), \tag{40}$$

where $\mathrm{Tr}(\cdot)$ refers to the trance of a matrix.

5.1.3 Self-quality-based. Besides the evaluation by measuring the similarity between the real and generated graphs, there are three additional metrics that directly evaluate the quality of the generated graphs: the validity, uniqueness and novelty of the generated graphs.

Validity. Since sometimes the generated graphs are required to preserve some properties, it is straightforward to evaluate them by judging whether they satisfy such requirements, such as the following: (1) Cycles graphs/Tree graphs: Cycles and trees are graphs that have obvious structural properties and the validity is calculated as what percentage of generated graphs are actually cycles or trees [79]. (2) Molecule graphs: Validity for molecule generation is the percentage of chemically valid molecules based on some domain specific rules [107].

Uniqueness. Ideally, high-quality generated graphs should be diverse and similar, but not identical. Thus, uniqueness is utilized to capture the diversity of generated graphs [6, 44, 79, 92, 107]. To calculate the uniqueness of a generated graph, the generated graphs that are sub-graph

isomorphic to some other generated graphs are first removed. The percentage of graphs remaining after this operation is defined as Uniqueness. For example, if the model generates 100 graphs, all of which are identical, the uniqueness is 1/100 = 1%.

Novelty. Novelty measures the percentage of generated graphs that are not sub-graphs of the training graphs and vice versa [6, 44, 92]. Note that identical graphs are defined as graphs that are sub-graph isomorphic to each other. In other words, novelty checks if the model has learned to generalize unseen graphs.

5.2 Evaluation for Conditional Deep Graph Generation

In addition to the above general evaluation metrics for graph generation, for conditional deep generative models for graph generation, some additional evaluation metrics can be involved, including: graph-property-based and mapping-relationship-based evaluations.

5.2.1 Graph-property-based. Considering that each of generated graph can have its associated real graph as label in the conditional graph generation task, we can directly compare each generated graph to its label graph by measuring their similarity or distance based on some graph properties or kernels, such as the following: (1) random-walk kernel similarity by using the random-walk based graph kernel [63]; (2) combination of Hamming and Ipsen-Mikhailov distances(HIM) [61]; (3) spectral entropies of the density matrices; (4) eigenvector centrality distance [12]; (5) closeness centrality distance [37]; (6) Weisfeiler Lehman kernel similarity [116]; (7) Neighborhood Sub-graph Pairwise Distance Kernel [44] by matching pairs of subgraphs with different radii and distances.

5.2.2 Mapping-relationship-based. Mapping-relationship-based evaluation measures whether the learned relationship between the conditions and the generated graphs is consistent with the true relationship between the conditions and the real graphs.

Explicit mapping relationship. Considering the situation where the true relationship between the input conditions and the generated graphs is known in advance, the evaluation can be conducted as follows: (1) When the condition is the category label of the graph, we can examine whether the generated graph falls into the conditional category by utilizing a graph classifier to classify a generated graph to a category [34, 124]. Specifically, the real graphs are used to train a classifier and the classifier is used to classify the generated graphs. Then the accuracy is calculated as the percentage of the predicted categories that are the same as the input conditional category. (2) When the condition is a graph in the source domain, where the task is to change some properties of the input graph, we can quantitatively compare the property scores of the generated and input graphs to see if the change indeed meets the requirement. For example, one can compute the improvement of logP scores from the input molecule to the optimized molecule in molecule optimization task [140].

Implicit mapping relationship. Regarding the deep graph translation problem, which is introduced in Section 4.1, sometimes, the underlying patterns of the mapping from the input graphs to the real target graphs are implicit and complex to define and measure. Thus, a classifier-based evaluation metric can be utilized [51]. By regarding the input and target graphs as two classes, it assumes that a classifier that is capable of distinguishing the generated target graphs would also succeed in distinguishing the real target graphs from the input graphs. Specifically, a graph classifier is first trained based on the input and generated target graphs. Then this trained graph classifier is tested to classify the input graph and real target graphs, and the results will be used as the evaluation metrics.

6 APPLICATIONS

Deep generative models for graph generation is a very promising domain that has a continously increasing number of applications including molecule optimization and generation in molecular chemistry, semantic Parsing in NLP, code modeling, and pseudo-industrial SAT instance generation.

6.1 Applications in Molecular Chemistry

There are numerous works applying deep generative models for graph generation in the domain of molecular chemistry, especially in the aspect of molecule generation and optimization.

Molecule generation is a fundamental problem in drug discovery and material science; its aim is to design novel molecules under a range of chemical properties. This is a highly challenging mathematical and computational problem with combinatorial nature. Any small perturbation in the chemical structure may result in a large variation in the desired molecular property. Besides, the space of valid molecules quickly becomes prohibitively huge and complex as the number of combinatorial permutations of atoms and bonds grows. Currently, most drugs are hand-crafted by human experts in chemistry and pharmacology. The recent advances of deep generative models for graph generation has opened a new research direction by treating the molecule as a graph with atoms as nodes and bonds as edges, with the potential to learn these molecular' generative representation for novel molecule generation without hand-crafting them. Early works on deep graph generative models for molecule generation [22, 73] instantiated the decoder with syntactic and semantic constraints of SMILES string by context free and attribute grammars, but these grammars do not fully capture chemical validity. Very recently, Simonovsky and Komodakis [119] propose the generation of molecular graphs by predicting their adjacency matrices, and Li et al. [79] generated molecules node by node. Then more and more deep graph generative models for molecule generation are proposed in an attempt to ensure chemical validity and efficiency [24, 57, 82, 107, 114, 144].

Molecule Optimization means optimizing the drug-like molecules to adhere to the desired activity profile, the physicochemical, and pharmacokinetic properties. The task is challenging since the chemical space is vast and difficult to navigate. Based on the recent development of deep graph generation, two categories of methods based on deep graph generative models are proposed for molecule optimization. One prevalent strategy is to build a graph generative model, that maps a particular molecule structure down to low-dimensional latent space, and then performs search or optimization in the latent space to find the optimized molecules [22, 57, 73, 79, 114]. These approaches based on this strategy can only indirectly optimize molecular properties in the learned latent embedding space before decoding to a molecule. Another strategy is to formalize the molecule optimization as a *graph-to-graph translation*. Given a corpus of molecule pairs, the goal is to learn to translate the input molecule graphs into graphs with the properties closer to the desired ones. Thus, some works [58, 59, 93, 140, 148] propose graph translation models through supervised learning by exploring the relationships between the input and output molecule graphs.

6.2 Protein Structure Modeling

Proteins are massive molecules that can be characterized as one of the multiple long chains of amino acids. Analyzing the structure and function of proteins is a key part of understanding biological properties at the molecular and cellular level. Protein structure modeling is highly important in two critical problems of biological molecules including protein structure prediction and protein design. Current computational modeling methods for protein design are slow and often require human oversight and intervention, which are often biased and incomplete. Inspired by recent momentum in deep graph generative models, some works [3, 42, 50, 87] demonstrate the potential of deep graph generative modeling for fast generation of new, viable protein structures. Specifically, in these methods, the contact map/distance matrix of a protein molecule is treated as a graph, while each amino acid molecule in the protein is regarded as a node and their pairwise contacts or distance are edge weights. A graph generative model is then utilized to model and generate the contacts/distances between each pair of amino acids. Finally, 3D structure recovery is utilized to recover the protein structures from the generated pairwise distance contact maps. Deep generative

models for graph generation are deemed to explore the latent distribution of the protein structures in an efficient way that is invariant to rotational and translational symmetry.

6.3 Semantic Parsing

Semantic parsing problem is about mapping the natural language information to its logical forms, namely abstract meaning representation (AMR), which is a graph-based formalism used for capturing sentence-level semantics. Traditional semantic parsers are usually based on compositionally and manually designed grammar to create the structure of AMR, and used lexicons for semantic grounding, which is time-consuming and heuristic. Recent years have witnessed a surge of interests in developing neural semantic parsers with sequence-to-sequence models [29, 56]. However, these methods only consider the word sequence information and ignore other rich syntactic information such as parse tree and knowledge graph. Because AMR are naturally structured objects (e.g. tree structures), semantic AMR parsing methods based on deep graph generative models are deemed as promising [19, 35, 88, 130, 145]. These methods represent the semantics of a sentence as a semantic graph (i.e., a sub-graph of a knowledge base) and treat semantic parsing as a semantic graph matching/generation process. These end-to-end deep graph generation techniques for semantic parsing show powerful ability in automatically capturing semantic information.

6.4 Code Modeling

Code modelling considers both hard syntactic and semantic constraints in generating natural programming code, which can make the development of source code easier, faster, and less error-prone. Early works in this area have shown that approaches from natural language processing can be applied successfully to the source code, whereas the programming languages community has had successes in focusing exclusively on formal semantics. However, though these methods are successful at generating programs that satisfy some formal specifications, they cannot generate realistic-looking and valid programs. Since *program graphs* have been shown to have the ability to encode semantically meaningful representations of programs, deep graph generative models have shown promising capability in modeling small but semantic programs generation [17, 22, 90, 100]. Specifically, the code is represented as an abstract syntax trees (AST). Then, an AST is generated by expanding one node at a time using production rules from the underlying programming language grammar. This simplifies the code generation task to a sequence of sampling problems, in which an appropriate production rule has to be chosen based on the partial AST generated so far.

6.5 Pseudo-industrial SAT Instance Generation

The problem of pseudo-industrial Boolean Satisfiability (SAT) instance generation is about generating artificial SAT problems that display the same characteristics as their real-world counterparts. Generating large amounts of SAT instances is important in developing and evaluating practical SAT solvers, which historically relies on extensive empirical testing on a large amount of SAT instances. Prior works addressing this problem relied on hand-crafted algorithms, focusing on capturing one or two of the graph statistics exhibited by real-world SAT formulas, but they are heavily hand-crafted and have difficult in simultaneously capturing a wide range of characteristics exhibited by real-world SAT instances [41, 99]. Since deep generative models for graphs demonstrated their ability to capture many of the essential features of real-world graphs, it is promising to represent SAT formulas as graphs, thus recasting the original problem as a deep graph generation task. Existing works [134, 141] developed a graph generative model by representing the SAT formula as a Literal-Clause Graph (LCG) or Literal-Incidence Graph (LIG). In LCG, a node refers to each literal and each clause, with an edge denoting the occurrence of a literal in a clause. In LCG, a node refers to each literal and two literals have an edge if they co-occur in a clause. The

generated SAT formulas could closely resemble given real-world SAT instances as well as be used to improve SAT solver performance on real-world benchmarks.

7 CONCLUSION AND OPPORTUNITIES

To the best of our knowledge, our work provides the first systematic review of deep generative models for graph generation. We present a taxonomy of deep graph generative models based on problem settings and techniques details, followed by a detailed introduction, comparison, and discussion about them. We also conduct a systematic review of the evaluation measures of deep graph generative models, including the general evaluation metrics for both unconditional and conditional graph generation. After that, we summarized popular applications in this domain.

As a fast-developing, promising domain, there are still many open challenges in the domain of deep generative models for graph generation. Thus, we would like to highlight a number of open challenges for further research in scalability, validity constrain, interpretability, graph editing, and special graphs such as temporal graphs.

Scalability. Existing deep generative models typically have super-linear time complexity to the number of nodes and cannot scale well to large networks. Only few methods have linear time complexity of $O(N)$ [5, 44, 83, 117, 142] and $O(M)$ [114], where N is the number of nodes and M is the number of edges. Consequentially, most existing works merely focus on small graphs, typically with dozens to thousands of nodes [24, 34, 47, 79, 113, 119, 140]. However, many real-world networks are large, with millions to billions of nodes [44], such as the Internet, biological neuronal networks, and social networks. It is important for any generative model to scale to large graph.

Validity Constraint. Many real-world networks are constrained by specific validity requirements [89]. For example, in molecular graphs, the number of bonding-electron pairs cannot exceed the valency of an atom. In protein interaction networks, two proteins may be connected only when they belong to the same or correlated gene ontology terms. Many real-world geographical networks (e.g., river networks and transportation networks) are planar graphs whose topology follows Euler formula [31, 32]. Graph-topological constraints are challenging to enforce during the model training process. Intuitive ways include designing heuristic and customized algorithms to ensure the validity of generated graphs. For example, Kusner et al. [73] propose the use of SMILES grammar to generate a parse tree, which in turn is flattened as a SMILES string. This approach guarantees the syntactic validity of the output, but semantic validity is still questionable. Dai et al. [22] further apply attribute grammar as a constraint in the parse-tree generation, a step toward semantic validity. Jin et al. [57] exploits the fact that molecular graphs may be turned into a tree by treating the rings as super nodes. However, they are restricted to domain-specific applications. Some recent works started to construct a more generic framework under constrained optimization scenario, which minimizes training loss under graph validity constraints [89]. However, as such constraints are typically discrete and non-differentiable, they need to be approximated with a smooth relaxation which introduces errors and cannot preclude all the invalid topologies.

Interpretability. When we learn the underlying distribution of complex structured data, i.e. graphs, learning interpretable representations of data that expose semantic meaning is very important [74]. For example, it is highly beneficial if we could identify which latent variable(s) control(s) which specific properties (e.g., molecule mass) of the generated graphs (e.g., molecules). It is also useful to disentangle local generative dependencies among different subgraphs. However, existing works on this topic only focus on graph embedding (which is a new topic by itself) but not generation [14, 102]. For example, Stoehr et al. [122] demonstrates the potential of latent variable disentanglement in graph deep learning for unsupervised discovery of generative parameters of random and real-world graphs. Investigations on graph decoding and generation are still open problems without existing works except one very recently published one [53].

Beyond Training Data. Deep generative models are data-driven models based on training data. The novelty of the generated graphs are highly desired yet usually restricted by training data and model properties (e.g., mode collapse of generative adversarial nets). To address such issues, attempts in the domain of images modified the attribute of a generated image by adding a learned vector on its latent code [108] or by combining the latent code of two images [64]. Additional works have been developed for inserting extra control in the image generation [108] with additional labels corresponding to key factors such as object size and facial expression. However, works on graph generation that could require very different technique sets than image generation are lacking.

Dynamic Graphs. Existing deep graph generative models typically focus on static graphs but many graphs in the real-world are dynamic, and their node attributes and topology can evolve over time, such as social network, mobility network, and protein folding. Representation learning for dynamic graphs is a hot domain, but it only focuses on graph embedding instead of generation. Modeling and understanding the generation of dynamic graphs have not been explored. Therefore, additional problems such as jointly modeling temporal and graph patterns and temporal validity constraints need to be addressed.

REFERENCES

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

[2] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. 2019. Learning to solve circuit-SAT: An unsupervised differentiable approach. *ICLR'2019*.

[3] Namrata Anand and Possu Huang. 2018. Generative modeling for protein structures. In *NeurIPS'2018*. 7494–7505.

[4] Rim Assouel, Mohamed Ahmed, and Marwin H Segler et al. 2018. Defactor: Differentiable edge factorization-based probabilistic graph generation. *arXiv preprint arXiv:1811.09766* (2018).

[5] Davide Bacciu, Alessio Micheli, and Marco Podda. 2019. Graph generation by sequential edge prediction. In *ESANN'2019*. 95–100.

[6] Davide Bacciu, Alessio Micheli, and Marco Podda. 2020. Edge-based sequential graph generation with recurrent neural networks. *Neurocomputing* (2020).

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *ICLR'2015*.

[8] Albert-László Barabási et al. 2016. *Network science*. Cambridge university press.

[9] Jens Behrmann, Will Grathwohl, and Ricky TQ Chen et al. 2019. Invertible residual networks. In *ICML'2019*. 573–582.

[10] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.

[11] Aleksandar Bojchevski, Oleksandr Shchur, and Daniel Zügner et al. 2018. NetGAN: generating graphs via random walks. In *ICML'2018*. 609–618.

[12] Phillip Bonacich. 1987. Power and centrality: A family of measures. *American journal of sociology* 92, 5 (1987), 1170–1182.

[13] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr. 2020. Graph Generators: State of the art and open challenges. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–30.

[14] Diane Bouchacourt, Ryota Tomioka, and Sebastian Nowozin. 2018. Multi-level variational autoencoder: Learning disentangled representations from grouped observations. In *AAAI'2018*. 2095–2102.

[15] Samuel Bowman, Luke Vilnis, and Oriol Vinyals et al. 2016. Generating sentences from a continuous space. In *CoNLL'2016*. 10–21.

[16] Xavier Bresson and Thomas Laurent. 2019. A two-Step graph convolutional decoder for molecule generation. *NeurIPS'2019 Workshop* (2019).

[17] Marc Brockschmidt, Miltiadis Allamanis, and Alexander L Gaunt et al. 2019. Generative code modeling with graphs. *ICLR'2019*.

[18] Vinicius Caridá, Amir Jalilifard, and Alex Mansano et al. 2019. Can NetGAN be improved on short random walks?. In *The 8th Brazilian Conference on Intelligent Systems (BRACIS'2019)*. 663–668.

[19] Bo Chen, Le Sun, and Xianpei Han. 2018. Sequence-to-action: End-to-end semantic graph generation for semantic parsing. In *ACL'2018*. 766–777.

[20] Lele Chen, Sudhanshu Srivastava, and Zhiyao Duan et al. 2017. Deep cross-modal audio-visual generation. In *Proceedings of the on Thematic Workshops of ACM Multimedia*. 349–357.

[21] Kyunghyun Cho, Bart van Merriënboer, and Caglar Gulcehre et al. 2014. Learning phrase representations using RNN Encoder–Decoder for statistical machine translation. In *EMNLP'2014*. 1724–1734.

[22] Hanjun Dai, Yingtao Tian, and Bo Dai et al. 2018. Syntax-directed variational autoencoder for structured data. *ICLR'2018*.

[23] Laura D'Arcy, Padraig Corcoran, and Alun Preece. 2019. Deep Q-Learning for directed acyclic graph generation. *ICML'2019 Workshop* (2019).

[24] Nicola De Cao and Thomas Kipf. 2018. MolGAN: An implicit generative model for small molecular graphs. *ICML'2018 Workshop* (2018).

[25] Jörg Degen, Christof Wegscheid-Gerlach, and Andrea Zaliani et al. 2008. On the art of compiling and using'Drug-Like'Chemical fragment spaces. *ChemMedChem: Chemistry Enabling Drug Discovery* 3, 10 (2008), 1503–1507.

[26] Emily L Denton, Soumith Chintala, and Rob Fergus et al. 2015. Deep generative image models using aï£ij laplacian pyramid of adversarial networks. In *NeurIPS'2015*. 1486–1494.

[27] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. 2017. Density estimation using real NVP. In *ICLR'2017*.

[28] Kien Do, Truyen Tran, and Svetha Venkatesh. 2019. Graph transformation policy network for chemical reaction prediction. In *KDD'2019*. 750–760.

[29] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *ACL'2016*, Vol. 1. 33–43.

[30] Chris Dyer, Miguel Ballesteros, and Wang Ling et al. 2015. Transition-based dependency parsing with stack long short-term memory. In *ACL'2015*. 334–343.

[31] Jason A Dykes. 1997. Exploring spatial data representation with dynamic graphics. *Computers & Geosciences* 23, 4 (1997), 345–370.

[32] David Eppstein and Michael T Goodrich. 2008. Studying geometric graph properties of road networks through an algorithmic lens. *SIGSPATIAL'2008*, 345–370.

[33] Paul Erdös, Alfréd Rényi, et al. 1959. On random graphs. *Publicationes mathematicae* 6, 26 (1959), 290–297.

[34] Shuangfei Fan and Bert Huang. 2019. Labeled graph generative adversarial networks. *arXiv preprint arXiv:1906.03220* (2019).

[35] Federico Fancellu, Sorcha Gilroy, and Adam Lopez et al. 2019. Semantic graph parsing with recurrent neural network DAG grammars. In *EMNLP-IJCNLP'2019*. 2769–2778.

[36] Daniel Flam-Shepherd, Tony Wu, and Alan Aspuru-Guzik. 2020. Graph deconvolutional generation. *arXiv preprint arXiv:2002.07087* (2020).

[37] Linton C Freeman. 1978. Centrality in social networks conceptual clarification. *Social networks* 1, 3 (1978), 215–239.

[38] Anuththari Gamage, Eli Chien, and Jianhao Peng et al. 2020. Multi-MotifGAN (MMGAN): Motif-Targeted graph generation and prediction. In *ICASSP'2020*. 4182–4186.

[39] Yuyang Gao, Xiaojie Guo, and Liang Zhao. 2018. Local event forecasting and synthesis using unpaired deep graph translations. In *SIGSPATIAL'2018 Workshop*. 1–8.

[40] Justin Gilmer, Samuel S Schoenholz, and Patrick F Riley et al. 2017. Neural message passing for quantum chemistry. In *ICML'2017*. 1263–1272.

[41] Jesús Giráldez-Cru and Jordi Levy. 2015. A modularity-based random SAT instances generator. In *IJCAI'2015*. 1952–1958.

[42] Vladimir Golkov, Marcin J Skwark, and Antonij Golkov et al. 2016. Protein contact prediction from amino acid co-evolution using convolutional networks for graph-valued images. In *NeurIPS'2016*. 4222–4230.

[43] Ian Goodfellow, Jean Pouget-Abadie, and Mehdi Mirza et al. 2014. Generative adversarial nets. In *NeurIPS'2014*. 2672–2680.

[44] Nikhil Goyal, Harsh Vardhan Jain, and Sayan Ranu. 2020. GraphGen: a scalable approach to domain-agnostic labeled graph generation. In *WWW'20*. 1253–1263.

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

[46] Arthur Gretton, Karsten M Borgwardt, and Malte J Rasch et al. 2012. A kernel two-sample test. *Journal of Machine Learning Research* 13, Mar (2012), 723–773.

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

[48] Xiuye Gu. 2019. Explore deep graph generation. (2019).

[49] Michael Guarino, Alexander Shah, and Pablo Rivas. 2017. DiPol-GAN: generating molecular graphs adversarially with relational differentiable pooling. (2017).

[50] Xiaojie Guo, Sivani Tadepalli, and Liang Zhao et al. 2020. Generating tertiary protein structures via an interpretative variational autoencoder. *arXiv preprint arXiv:2004.07119* (2020).

[51] Xiaojie Guo, Lingfei Wu, and Liang Zhao. 2018. Deep graph translation. *arXiv preprint arXiv:1805.09980* (2018).

[52] Xiaojie Guo, Liang Zhao, and Cameron Nowzari et al. 2019. Deep multi-attributed graph translation with node-Edge Co-evolution. In *ICDM'2019*. 250–259.

[53] Xiaojie Guo, Liang Zhao, and Zhao Qin et al. 2020. Interpretable Deep Graph Generation with Node-Edge Co-Disentanglement. *KDD'2020*.

[54] Shion Honda, Hirotaka Akita, and Katsuhiko Ishiguro et al. 2019. Graph residual flow for molecular graph generation. *arXiv preprint arXiv:1909.13521* (2019).

[55] Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparametrization with gumble-softmax. In *ICLR'2017*.

[56] Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *ACL'2016*, Vol. 1. 12–22.

[57] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. 2018. Junction tree variational autoencoder for molecular graph generation. In *ICML'2018*. 2323–2332.

[58] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. 2020. Composing Molecules with Multiple Property Constraints. *arXiv preprint arXiv:2002.03244* (2020).

[59] Wengong Jin, Kevin Yang, and Regina Barzilay et al. 2018. Learning multimodal graph-to-graph translation for molecular optimization. *ICLR'2018*.

[60] Eric Jonas. 2019. Deep imitation learning for molecular inverse problems. In *NeurIPS'2019*. 4991–5001.

[61] Giuseppe Jurman, Roberto Visintainer, and Michele Filosi et al. 2015. The HIM glocal metric and kernel for network comparison and classification. In *DSAA'2015*. 1–10.

[62] M Clara De Paolis Kaluza, Saeed Amizadeh, and Rose Yu. 2018. A neural framework for learning DAG to DAG translation. *NeurIPS'2018 Workshop* (2018).

[63] U Kang, Hanghang Tong, and Jimeng Sun. 2012. Fast random walk graph kernel. In *SDM'2012*. 828–838.

[64] Tero Karras, Samuli Laine, and Timo Aila. 2019. A style-based generator architecture for generative adversarial networks. In *CVPR'2019*. 4401–4410.

[65] Jeremy Kawahara, Colin J Brown, and Steven P Miller et al. 2017. BrainNetCNN: Convolutional neural networks for brain networks; towards predicting neurodevelopment. *NeuroImage* 146 (2017), 1038–1049.

[66] Steven Kearnes, Li Li, and Patrick Riley. 2019. Decoding molecular graph embeddings with reinforcement learning. *ICML'2019 Workshop* (2019).

[67] Mahdi Khodayar, Ying Zhang, and Jianhui Wang. 2019. Deep generative graph distribution learning for synthetic power grids. *IEEE Power Engineering Letters* (2019).

[68] Diederik P Kingma and Max Welling. 2014. Auto-encoding variational bayes. *ICLR'2014*.

[69] Thomas N Kipf and Max Welling. 2016. Variational graph auto-encoders. *NeurIPS'2016 Workshop* (2016).

[70] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. *ICLR'2017*.

[71] Lauwerens Kuipers and Harald Niederreiter. 2012. *Uniform distribution of sequences*. Courier Corporation.

[72] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.

[73] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. 2017. Grammar variational autoencoder. In *ICML'2017*. 1945–1954.

[74] Brenden M Lake, Tomer D Ullman, and Joshua B Tenenbaum et al. 2017. Building machines that learn and think like people. *Behavioral and brain sciences* 40 (2017), e253.

[75] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *ICML'2014*. 1188–1196.

[76] Tao Lei, Yu Zhang, and Sida I Wang et al. 2018. Simple recurrent units for highly parallelizable recurrence. In *EMNLP'2018*. 4470–4481.

[77] Jure Leskovec, Deepayan Chakrabarti, and Jon Kleinberg et al. 2010. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research* 11, Feb (2010), 985–1042.

[78] Jiwei Li, Will Monroe, and Alan Ritter et al. 2016. Deep reinforcement learning for dialogue generation. In *EMNLP'2016*. 1192–1202.

[79] Yujia Li, Oriol Vinyals, and Chris Dyer et al. 2018. Learning deep generative models of graphs. *ICLR'2018 Workshop* (2018).

[80] Yibo Li, Liangren Zhang, and Zhenming Liu. 2018. Multi-objective de novo drug design with conditional graph generative model. *Journal of cheminformatics* 10, 1 (2018), 33.

[81] Renjie Liao, Yujia Li, and Yang Song et al. 2019. Efficient graph generation with graph recurrent attention networks. In *NeurIPS'2019*. 4257–4267.

[82] Jaechang Lim, Sang-Yeon Hwang, and Kim et al. 2019. Scaffold-based Molecular Design Using Graph Generative Model. In *The 2019 Asia-Pacific Association of Theoretical and Computational Chemists*.

[83] Chia-Cheng Liu, Harris Chan, and Kevin. Luk et al. 2019. Auto-regressive graph generation modeling with improved evaluation methods. *NeurIPS'2019 Workshop* (2019).

[84] Jing Liu, Yaxiong Chi, and Chen Zhu. 2015. A dynamic multiagent genetic algorithm for gene regulatory network reconstruction based on fuzzy cognitive maps. *IEEE Transactions on Fuzzy Systems* 24, 2 (2015), 419–431.

[85] Jenny Liu, Aviral Kumar, and Jimmy Ba et al. 2019. Graph normalizing flows. In *NeurIPS'2019*. 13556–13566.
[86] Qi Liu, Miltiadis Allamanis, and Marc Brockschmidt et al. 2018. Constrained graph variational autoencoders for molecule design. In *NeurIPS'2018*. 7795–7804.
[87] Lorenzo Livi, Enrico Maiorino, and Alessandro Giuliani et al. 2016. A generative model for protein contact networks. *Journal of Biomolecular Structure and Dynamics* 34, 7 (2016), 1441–1454.
[88] Chunchuan Lyu and Ivan Titov. 2018. AMR parsing as graph prediction with latent alignment. In *ACL'2018*. 397–407.
[89] Tengfei Ma, Jie Chen, and Cao Xiao. 2018. Constrained generation of semantically valid graphs via regularizing variational autoencoders. In *NeurIPS'2018*. 7113–7124.
[90] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *ICML'2014*. 649–657.
[91] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The concrete distribution: A continuous relaxation of discrete random variables. In *ICLR'2017*.
[92] Kaushalya Madhawa, Katushiko Ishiguro, and Kosuke Nakago et al. 2019. Graphnvp: An invertible flow model for generating molecular graphs. *arXiv preprint arXiv:1905.11600* (2019).
[93] Łukasz Maziarka, Agnieszka Pocha, and Jan Kaczmarczyk et al. 2020. Mol-CycleGAN: a generative model for molecular optimization. *Journal of Cheminformatics* 12, 1 (2020), 1–18.
[94] Tomáš Mikolov, Martin Karafiát, and Lukáš Burget et al. 2010. Recurrent neural network based language model. In *INTERSPEECH'2010*. 1045–148.
[95] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *NeurIPS'2014 Workshop* (2014).
[96] Volodymyr Mnih, Koray Kavukcuoglu, and David Silver et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
[97] Seonghyeon Nam, Yunji Kim, and Seon Joo Kim. 2018. Text-adaptive generative adversarial networks: manipulating images with natural language. In *NeurIPS'2018*. 42–51.
[98] Mark Newman. 2018. *Networks*. Oxford university press.
[99] Zack Newsham, Vijay Ganesh, and Sebastian Fischmeister et al. 2014. Impact of community structure on SAT solver performance. In *SAT'2014*. 252–268.
[100] Anh Tuan Nguyen and Tien N Nguyen. 2015. Graph-based statistical language model for code. In *ICSE'2015*, Vol. 1. 858–868.
[101] Chenhao Niu, Yang Song, and Jiaming Song et al. 2020. Permutation invariant graph generation via score-Based generative modeling. *AISTATS'2020*.
[102] Emmanuel Noutahi, Dominique Beani, and Julien Horwood et al. 2019. Towards interpretable sparse graph representation learning with laplacian pooling. *arXiv preprint arXiv:1905.11577* (2019).
[103] Achraf Oussidi and Azeddine Elhassouny. 2018. Deep generative models: Survey. In *ISCV'2018*. 1–8.
[104] George Papamakarios, Theo Pavlakou, and Iain Murray. 2017. Masked autoregressive flow for density estimation. In *NeurIPS'17*. 2338–2347.
[105] Marco Podda, Davide Bacciu, and Alessio Micheli. 2020. A deep generative model for fragment-based molecule generation. *AISTATS'2020*.
[106] Sebastian Pölsterl and Christian Wachinger. 2019. Likelihood-free inference and generation of molecular graphs. *arXiv preprint arXiv:1905.10310* (2019).
[107] Mariya Popova, Mykhailo Shvets, and Junier Oliva et al. 2019. MolecularRNN: generating realistic molecular graphs with optimized properties. *arXiv preprint arXiv:1905.13372* (2019).
[108] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised representation learning with deep convolutional generative adversarial networks. *ICLR'2016*.
[109] Sayan Ranu and Ambuj K Singh. 2009. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE'2009*. 844–855.
[110] Matthias Rarey and J Scott Dixon. 1998. Feature trees: a new molecular similarity measure based on tree matching. *Journal of computer-aided molecular design* 12, 5 (1998), 471–490.
[111] Garry Robins, Tom Snijders, and Peng Wang et al. 2007. Recent developments in exponential random graph (p*) models for social networks. *Social networks* 29, 2 (2007), 192–215.
[112] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI'2015*. 234–241.
[113] Guillaume Salha, Stratis Limnios, and Romain Hennequin et al. 2019. Gravity-inspired graph autoencoders for directed link prediction. In *CIKM'2019*. 589–598.
[114] Bidisha Samanta, DE Abir, and Gourhari Jana et al. 2019. Nevae: A deep generative model for molecular graphs. In *AAAI'2019*, Vol. 33. 1110–1117.
[115] Franco Scarselli, Marco Gori, and Ah Chung Tsoi et al. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

[116] Nino Shervashidze, Pascal Schweitzer, and Erik Jan Van Leeuwen et al. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, 77 (2011), 2539–2561.
[117] Chence Shi, Minkai Xu, and Zhaocheng Zhu et al. 2020. GraphAF: a flow-based autoregressive model for molecular graph generation. *ICLR'2020*.
[118] David Silver, Richard S Sutton, and Martin Müller. 2007. Reinforcement learning of local shape in the game of go.. In *IJCAI'2007*, Vol. 7. 1053–1058.
[119] Martin Simonovsky and Nikos Komodakis. 2018. GraphVAE: Towards generation of small graphs using variational autoencoders. In *ICANN'2018*. 412–422.
[120] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. 2015. Learning structured output representation using deep conditional generative models. In *NeurIPS'2015*. 3483–3491.
[121] Yang Song and Stefano Ermon. 2019. Generative modeling by estimating gradients of the data distribution. In *NeurIPS'2019*. 11895–11907.
[122] Niklas Stoehr, Marc Brockschmidt, and Jan Stuehmer et al. 2019. Disentangling interpretable generative parameters of random and real-World Graphs. *NeurIPS'2019 Workshop* (2019).
[123] Shih-Yang Su, Hossein Hajimirsadeghi, and Greg Mori. 2019. Graph generation with variational recurrent neural network. *NeurIPS'2019 Workshop* (2019).
[124] Mingming Sun and Ping Li. 2019. Graph to graph: a topology aware approach for graph structures learning and generation. In *AISTATS'2019*. 2946–2955.
[125] BRUCE W SUTER. 1990. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks* 1, 4 (1990), 291.
[126] Ilya Sutskever, James Martens, and Geoffrey E Hinton. 2011. Generating text with recurrent neural networks. In *ICML'2011*. 1017–1024.
[127] Richard S Sutton, Andrew G Barto, et al. 1998. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.
[128] Cong Tran, Won-Yong Shin, and Andreas Spitz et al. 2019. DeepNC: Deep generative network completion. *arXiv preprint arXiv:1907.07381* (2019).
[129] Hongwei Wang, Jia Wang, and Jialin Wang et al. 2018. Graphgan: Graph representation learning with generative adversarial nets. In *AAAI'2018*. 2508–2515.
[130] Yuxuan Wang, Wanxiang Che, and Jiang Guo et al. 2018. A neural transition-based approach for semantic dependency graph parsing. In *AAAI'2018*. 5561–5568.
[131] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.
[132] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of âĂŸsmall-worldâĂŹnetworks. *nature* 393, 6684 (1998), 440.
[133] David Weininger. 1988. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. *Journal of chemical information and computer sciences* 28, 1 (1988), 31–36.
[134] Haoze Wu and Raghuram Ramanujan. 2019. Learning to generate industrial sat instances. In *SoCS'2019*. 206–207.
[135] Zonghan Wu, Shirui Pan, and Fengwen Chen et al. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020), 1–21.
[136] Keyulu Xu, Weihua Hu, and Jure Leskovec et al. 2019. How powerful are graph neural networks? *ICLR'2019*.
[137] Xifeng Yan and Jiawei Han. 2002. gspan: Graph-based substructure pattern mining. In *ICDM'2002*. 721–724.
[138] Carl Yang, Peiye Zhuang, and Wenhan Shi et al. 2019. Conditional structure generation through graph variational generative adversarial nets. In *NeurIPS'2019*. 1338–1349.
[139] Shanchao Yang, Jing Liu, and Kai Wu et al. 2020. Learn to generate time series conditioned graphs with generative adversarial nets. *arXiv preprint arXiv:2003.01436* (2020).
[140] Jiaxuan You, Bowen Liu, and Zhitao Ying et al. 2018. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS'2018*. 6410–6421.
[141] Jiaxuan You, Haoze Wu, and Clark Barrett et al. 2019. G2SAT: Learning to generate SAT formulas. In *NeurIPS'2019*. 10552–10563.
[142] Jiaxuan You, Rex Ying, and Xiang Ren et al. 2018. GraphRNN: generating realistic graphs with deep auto-regressive Models. In *ICML'2018*. 5708–5717.
[143] Liming Zhang. 2019. STGGAN: Spatial-temporal graph generation. In *SIGSPATIAL'19*. 608–609.
[144] Muhan Zhang, Shali Jiang, and Zhicheng Cui et al. 2019. D-VAE: A variational autoencoder for directed acyclic graphs. In *NeurIPS'2019*. 1586–1598.
[145] Sheng Zhang, Xutai Ma, and Kevin Duh et al. 2019. AMR parsing as sequence-to-graph transduction. In *ACL'2019*. 80–94.
[146] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[147] Dawei Zhou, Lecheng Zheng, and Jiejun Xu et al. 2019. Misc-GAN: A multi-Scale generative model for Graphs. *Frontiers in Big Data* 2 (2019), 3.
[148] Zhenpeng Zhou, Steven Kearnes, and Li Li et al. 2019. Optimization of molecules via deep reinforcement learning. *Scientific reports* 9, 1 (2019), 1–10.
[149] Jun-Yan Zhu, Taesung Park, and Phillip Isola et al. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV'2017*. 2223–2232.
[150] Dongmian Zou and Gilad Lerman. 2019. Encoding robust representation for graph generation. In *IJCNN'2019*. 1–9.