Chapter 11
Graph Neural Networks: Graph Generation

Renjie Liao

Abstract In this chapter, we first review a few classic probabilistic models for graph generation including the Erdős–Rényi model and the stochastic block model. Then we introduce several representative modern graph generative models that leverage deep learning techniques like graph neural networks, variational auto-encoders, deep auto-regressive models, and generative adversarial networks. At last, we conclude the chapter with a discussion on potential future directions.

11.1 Introduction

The study of graph generation revolves around building probabilistic models over *graphs* which are also called *networks* in many scientific disciplines. This problem has its roots in a branch of mathematics, called *random graph theory* (Bollobás, 2013), which largely lies at the intersection between the probability theory and the graph theory. It is also at the core of a new academic field, called *network science* (Barabási, 2013). Historically, researchers in these fields are often interested in building random graph models (*i.e.*, constructing distributions of graphs using certain parametric families of distributions) and proving the mathematical properties of such models. Albeit being an extremely fruitful and successful research direction that spawns numerous outcomes, these classic models suffer from being too simplistic to capture the complex phenomenon (e.g., highly-clustered, well-connected, scale-free) that appeared in the real-world graphs.

With the advent of powerful deep learning techniques like *graph neural networks*, we can build more expressive probabilistic models of graphs, *i.e.*, the so-called *deep graph generative models*. Such deep models can better capture the complex dependencies within the graph data to generate more realistic graphs and further build accurate predictive models. However, the downside is that these models

Renjie Liao
University of Toronto, e-mail: rjliao@cs.toronto.edu

are often so complicated that we can rarely analyze their properties in a precise manner. The recent practices of these models have demonstrated impressive performances in modeling real-world graphs/networks, e.g., social networks, citation networks, and molecule graphs.

In the following, we first introduce the classic graph generative models in Section 11.2 and then the modern ones that leverage the deep learning techniques in Section 11.3. At last, we conclude the chapter and discuss some promising future directions.

11.2 Classic Graph Generative Models

In this section, we review two popular variants of the classic graph generative models: the Erdős–Rényi model (Erdős and Rényi, 1960) and the stochastic block model (Holland et al, 1983). They are often used as handy baselines in many applications since we have already gained deep understandings of their properties. There are many other graph generative models like the Watts–Strogatz small-world model (Watts and Strogatz, 1998) and the Barabási–Albert (BA) preferential attachment model (Barabási and Albert, 1999). Barabási (2013) provides a thorough survey on these models and other aspects of network science. In the context of machine learning, there are also quite a few non-deep-learning graph generative models like Kronecker graphs (Leskovec et al, 2010). We do not cover these models due to the space limit.

11.2.1 Erdős–Rényi Model

We first explain one of the most well known random graph models, *i.e.*, Erdős–Rényi model (Erdős and Rényi, 1960), named after two Hungarian mathematicians Paul Erdős and Alfréd Rényi. Note that this model has been independently proposed at around the same time by Edgar Gilbert in (Gilbert, 1959). In the following, we first describe the model along with its properties and then discuss its limitations.

11.2.1.1 Model

The Erdős–Rényi model has two closely variants, namely, $G(n,p)$ and $G(n,m)$.

G(n,p) Model In the $G(n,p)$ model, we are given n labeled nodes and generate a graph by randomly connecting an edge linking one node to the other with the probability p, independently from every other edge. In other words, all $\binom{n}{2}$ possible edges have the equal probability p to be included. Therefore, the probability of generating a graph with m edges under this model is as below,

$$p(\text{a graph with } n \text{ nodes and } m \text{ edges}) = p^m(1-p)^{\binom{n}{2}-m}. \tag{11.1}$$

The parameter p controls the "density" of the graph, *i.e.*, a larger value of p makes the graph become more likely to contain more edges. When $p = \frac{1}{2}$, the above probability becomes $\frac{1}{2}^{\binom{n}{2}}$, *i.e.*, all possible $2^{\binom{n}{2}}$ graphs are chosen with equal probability.

Due to the independence of the edges in $G(n,p)$, we can easily derive a few properties from this model.

- The expected number of edges is $\binom{n}{2}p$.
- The degree distribution of any node v is binomial:

$$p(\text{degree}(v) = k) = \binom{n}{k} p^k (1-p)^{n-1-k} \tag{11.2}$$

- If Np is a constant and $n \to \infty$, the degree distribution of any node v is Poisson:

$$p(\text{degree}(v) = k) = \frac{(np)^k e^{-np}}{k!} \tag{11.3}$$

There is an enormous number of more involved properties of this model that has been proved (e.g., by Erdős and Rényi in the original paper). We list a few others as below.

- If $p > \frac{(1+\varepsilon)\ln n}{n}$, then a graph will almost surely be connected.
- If $p < \frac{(1+\varepsilon)\ln n}{n}$, then a graph will almost surely contain isolated vertices, and thus be disconnected.
- If $Np < 1$, then a graph will almost surely have no connected components of size larger than $O(\log(n))$.

Here almost surely means the probability of the event happens with probability 1 (*i.e.*, the set of possible exceptions has zero measure).

G(n,m) Model In the $G(n,m)$ model, we are given n labeled nodes and generate a graph by uniformly randomly choosing a graph from the set of all graphs with n nodes and m edges, *i.e.*, the probability of choosing each graph is $\binom{\binom{n}{2}}{m}^{-1}$. There are also many important properties associated with the $G(n,m)$ model. In particular, it is interchangeable with the $G(n,p)$ model provided that m is close to $\binom{n}{2}p$ in most investigations. Chapter 2 of (Bollobás and Béla, 2001) provides a comprehensive discussion on the relationship between these two models. The $G(n,p)$ model is more commonly used in practice than the $G(n,m)$ model, partly due to the ease of analysis brought by the independence of the edges.

11.2.1.2 Discussion

As a seminal work in the random graph theory, the Erdős–Rényi model inspires much subsequent work to study and generalize this model. However, the assumptions of this model, e.g., edges are independent and each edge is equally likely to be generated, are too strong to capture the properties of the real-world graphs. For example, the degree distribution of the Erdős–Rényi model has an exponential tail

which means we rarely see node degrees span a broad range, e.g., several orders of magnitude. Meanwhile, real-world graphs/networks like the World Wide Web (WWW) are believed to possess a degree distribution that follows a power law, *i.e.*, $p(d) \propto d^{-\gamma}$ where d is the degree and the exponent γ is typically between 2 and 3. Essentially, this means that there are many nodes that have small node degrees, whereas there are a few nodes which have extremely large node degrees (, hubs) in the real-world graphs like WWW. Therefore, many improved models like the scale-free networks (Barabási and Albert, 1999) were later proposed, which fit better to the degree distribution of the real-world graphs.

11.2.2 Stochastic Block Model

Stochastic block models (SBM) are a family of random graphs with clusters of nodes and are often employed as a canonical model for tasks like community detection and clustering. It is proposed independently in a few scientific communities, e.g., machine learning and statistics (Holland et al, 1983), theoretical computer science (Bui et al, 1987), and mathematics (Bollobás et al, 2007). It is arguably the simplest model of a graph with communities/clusters. As a generative model, SBM could provide ground-truth cluster memberships, which in turn could help benchmark and understand different clustering/community detection algorithms. In the following, we first introduce the basics of the model and then discuss its advantages as well as limitations.

11.2.2.1 Model

We start the introduction by denoting the total number of nodes as n and the number of communities/clusters as k. A prior probability vector $\mathbf{p}$ over the k clusters and a $k \times k$ matrix W with entries in $[0,1]$ are also given. We generate a random graph following the procedure below:

1. For each node, we generate its community label (an integer from $\{1, \cdots, k\}$) by independently sampling from $\mathbf{p}$.
2. For each pair of nodes, denoting their community labels as i and j, we generate an edge by independently sampling with probability $W_{i,j}$.

Basically, the community assignments of a pair of nodes determine the specific entry of W to be used, which in turn indicates how likely we connect this pair of nodes. We denote such a model as $\mathrm{SBM}(n, \mathbf{p}, W)$. Note that, if we set $W_{i,j} = q$ for all communities (i, j), then the corresponding SBM degenerates to the Erdős–Rényi model $G(n, q)$.

In the context of community detection, people are often interested in recovering the community label given a random graph drawn from the SBM model. Denoting the recovered and the ground-truth community labels as $X \in \mathbb{R}^{n \times 1}$ and $Y \in \mathbb{R}^{n \times 1}$,

we can define the agreement R between two community labels as,

$$R(X,Y) = \max_{P \in \Pi} \quad \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\left[X_i = (PY)_i\right], \tag{11.4}$$

where P is a permutation matrix and Π is the set of all permutation matrices. X_i and $(PY)_i$ are the i-th element of X and PY respectively. In short, the agreement considers the best possible reshuffle between two sequences of labels. Depending on the requirement, we could examine the community detection algorithms in the sense of exact recovery (*i.e.*, cluster assignments are exactly recovered almost surely, $p(R(X,Y) = 1) = 1$) or partial recovery (*i.e.*, at most $1 - \varepsilon$ fraction of nodes are mislabeled almost surely, $p(R(X,Y) \geq \varepsilon) = 1$). Researchers have established various conditions under which a particular type of recovery is possible for SBM graphs. For example, for SBMs with $W = \frac{\log(n)Q}{n}$, where Q is a matrix with positive entries and the same size as W, Abbe and Sandon (2015) shows that the exact recovery is possible if and only if the minimum Chernoff-Hellinger divergence between any two columns of $\text{diag}(\mathbf{p})Q$ is no less than 1, where $\text{diag}(\mathbf{p})$ is a diagonal matrix with diagonal entries as $\mathbf{p}$.

11.2.2.2 Discussion

Abbe (2017) provides an up-to-date and comprehensive survey on the SBM and the fundamental limits (from both information-theoretic and computational perspectives) for community detection in the SBM. SBM is a more realistic random graph model for describing graphs with community structures compared to the Erdős–Rényi model. It also spawns many subsequent variants of block models like the mixed membership SBM (Airoldi et al, 2008). However, the estimation of SBMs on real-world graphs is hard since the number of communities is often unknown in advance and some graphs may not exhibit clear community structures.

11.3 Deep Graph Generative Models

In this section, we review several representative deep graph generative models which aim at building probabilistic models of graphs using deep neural networks. Based on the type of deep learning techniques being used, we can roughly divide the current literature into three categories: variational autoencoder (VAEs) (Kingma and Welling, 2014) based methods, deep auto-regressive (Van Oord et al, 2016) methods, and generative adversarial networks (GANs) (Goodfellow et al, 2014b) based methods. We introduce all three model classes in the subsequent sections.

11.3.1 Representing Graphs

We first introduce how a graph is represented in the context of deep graph generative models. Suppose we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes/vertices and $\mathcal{E}$ is the set of edges. Conditioning on a specific node ordering π, we can represent the graph $\mathcal{G}$ as an adjacency matrix A_π where $A_\pi \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, where $|\mathcal{V}|$ is the size of set $\mathcal{V}$ (*i.e.*, the number of nodes). The adjacency matrix not only provides a convenient representation of graphs on computers but also offers a natural way to mathematically define a probability distribution over graphs. Here we explicitly write the node ordering π in the subscript to emphasize that the rows and columns of A are arranged according to the π. If we change the node ordering from π to π', the adjacency matrix will be permuted (shuffling rows and columns) accordingly, *i.e.*, $A_{\pi'} = PA_\pi P^\top$, where the permutation matrix P is constructed based on the pair of node orderings (π, π'). In other words, A_π and $A_{\pi'}$ represent the same graph $\mathcal{G}$. Therefore, a graph $\mathcal{G}$ with an adjacency matrix A_π can be equivalently represented as a set of adjacency matrices $\{PA_\pi P^\top | P \in \Pi\}$ where Π is the set of all permutation matrices with size $|\mathcal{V}| \times |\mathcal{V}|$. Note that, depending on the symmetric structures of A_π, there may exist two permutation matrices $P_1, P_2 \in \Pi$ so that $P_1 A_\pi P_1^\top = P_2 A_\pi P_2^\top$. Therefore, we remove such redundancies and keep those uniquely permutated adjacency matrices, denoted as $\mathcal{A} = \{PA_\pi P^\top | P \in \Pi_\mathcal{G}\}$. More precisely, $\Pi_\mathcal{G}$ is the maximal subset of Π so that $P_1 A_\pi P_1^\top \neq P_2 A_\pi P_2^\top$ holds for any $P_1, P_2 \in \Pi_\mathcal{G}$. We add the subscript $\mathcal{G}$ to emphasize that $\Pi_\mathcal{G}$ depends on the given graph $\mathcal{G}$. Note that there exists a surjective mapping between Π and $\Pi_\mathcal{G}$. For the ease of notations, we will drop the subscript of the node ordering and use $\mathcal{G} \equiv \mathcal{A} = \{PAP^\top | P \in \Pi_\mathcal{G}\}$ to represent a graph from now on.

When considering the node features/attributes X, we can denote the graph structured data as $\mathcal{G} \equiv \{(PAP^\top, PX) | P \in \Pi_\mathcal{G}\}$[1]. Note that the rows of X are shuffled according to P since each row of X corresponds to a node. In our context, we can assume the maximum number of nodes of all graphs is n. If a graph has fewer nodes than n, we can add dummy nodes (e.g., with all-zero features) which are isolated to other nodes to make the size equal n. Therefore, $X \in \mathbb{R}^{n\times d_X}$ and $A \in \mathbb{R}^{n\times n}$ where d_X is the feature dimension. To simplify the explanation, we do not include the edge feature. But it is straightforward to modify the following models accordingly to incorporate edge features.

11.3.2 Variational Auto-Encoder Methods

Due to the great success of VAEs in image generation (Kingma and Welling, 2014; Rezende et al, 2014), it is natural to extend this framework to graph generation. This

[1] Technically, there may exist two permutation matrices $P_1, P_2 \in \Pi$ so that $P_1 A P_1^\top = P_2 A P_2^\top$ and $P_1 X \neq P_2 X$. It thus seems to be necessary to define $\mathcal{G} \equiv \{(PAP^\top, PX) | P \in \Pi\}$. However, as seen later, we are always interested in distributions of node features that are exchangeable over nodes, *i.e.*, $p(P_1 X) = p(P_2 X)$. Therefore, restricting ourselves to $\Pi_\mathcal{G}$ is sufficient for our exposition.

idea has been explored from different aspects (Kipf and Welling, 2016; Jin et al, 2018a; Simonovsky and Komodakis, 2018; Liu et al, 2018d; Ma et al, 2018; Grover et al, 2019; Liu et al, 2019b) and is often collectively named as *GraphVAE*. In the following, we first highlight the common framework shared by all these methods and then discuss some important variants.

11.3.2.1 The GraphVAE Family

Similar to vanilla VAEs, every model instance within the GraphVAE family consists of an encoder (*i.e.*, a variational distribution $q_\phi(Z|A,X)$ parameterized by ϕ), a decoder (*i.e.*, a conditional distribution $p_\theta(\mathscr{G}|Z)$ parameterized by θ), and a prior distribution (*i.e.*, a distribution $p(Z)$ typically with fixed parameters). Before introducing individual components, we first describe what the latent variables Z are. In the context of graph generation, we typically assume that each node is associated with a latent vector. Denoting the latent vector of the i-th node as $\mathbf{z}_i$, then $Z \in \mathbb{R}^{n\times d_Z}$ is obtained by stacking $\{\mathbf{z}_i\}$ as row vectors. Such latent vectors should summarize the information of the local subgraphs associated with individual nodes so that we can decode/generate edges based on them. In other words, any pair of latent vectors $(\mathbf{z}_i,\mathbf{z}_j)$ is supposed to be informative to determine whether nodes (i,j) should be connected. We could further introduce edge latent variables $\{\mathbf{z}_{ij}\}$ to enrich the model. Again, we do not consider such an option for simplicity since the underlying modeling technique is roughly the same.

Encoder We first explain how to construct the encoder using a deep neural network. Recall that the input to the encoder is the graph data (A,X). The natural candidate to deal with such data is a graph neural network, e.g., a graph convolutional network (GCN) (Kipf and Welling, 2017b). For example, let us consider a two-layer GCN as below,

$$H = \tilde{A}\sigma(\tilde{A}XW_1)W_2, \tag{11.5}$$

where $H \in \mathbb{R}^{n\times d_H}$ are the node representations (each node is associated with a size-d_H row vector). $\tilde{A} = D^{-\frac{1}{2}}(A+\mathrm{I})D^{-\frac{1}{2}}$ where D is the degree matrix (*i.e.*, a diagonal matrix of which the entries are the row sum of $A+\mathrm{I}$). I is the identity matrix. σ is the nonlinearity which is often chosen to be the rectified linear unit (ReLU) (Nair and Hinton, 2010). $\{W_1,W_2\}$ are the learnable parameters. We can pad a constant to the input feature dimension so that the bias term is absorbed into the weight matrix. We adopt this convention for ease of notation.

Relying on the learned node representations H, we can construct the variational distribution as below,

$$q_\phi(Z|A,X) = \prod_{i=1}^{n} q(\mathbf{z}_i|A,X) \tag{11.6}$$

$$q(\mathbf{z}_i|A,X) = \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i \mathrm{I}) \tag{11.7}$$

$$\boldsymbol{\mu} = \mathrm{MLP}_{\boldsymbol{\mu}}(H) \tag{11.8}$$

$$\log \boldsymbol{\sigma} = \mathrm{MLP}_{\boldsymbol{\sigma}}(H). \tag{11.9}$$

Here we typically assume that the variational distribution $q(Z|A,X)$ is conditionally node-wise independent for the tractability consideration. $\boldsymbol{\mu}_i$ and $\boldsymbol{\sigma}_i$ are the i-th rows of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ respectively. The learnable parameters ϕ consist of all parameters of the two multi-layer perceptrons (MLPs) and the above GCN. Although the approximated variational distribution defined in Eq. (11.6) is simple, it possesses a few great properties. First, the probability distribution is invariant w.r.t. the permutation of nodes. Mathematically, it means that given two different permutation matrices $P_1, P_2 \in \Pi$, we have

$$q(P_1 Z|P_1 A P_1^\top, P_1 X) = q(P_2 Z|P_2 A P_2^\top, P_2 X) \tag{11.10}$$

This can be easily verified from the exchangeability of the product of probabilities and the equivariance property of graph neural networks. Second, the neural networks underlying each Gaussian (*i.e.*, "GNN + MLP") are very powerful so that the conditional distributions are expressive in capturing the uncertainty of latent variables. Third, this encoder is computationally cheaper than those which consider the dependencies among different $\{\mathbf{z}_i\}$ (e.g., an autoregressive encoder). It thus provides a solid baseline for investigating whether a more powerful encoder is needed in a given problem.

Prior Similar to most VAEs, GraphVAEs often adopt a prior that is fixed during the learning. For example, a common choice is an node-independent Gaussian as below,

$$p(Z) = \prod_{i=1}^{n} p(\mathbf{z}_i) \tag{11.11}$$

$$p(\mathbf{z}_i) = \mathcal{N}(0, \mathrm{I}). \tag{11.12}$$

Again, we could replace this fixed prior with more powerful ones like an autoregressive model at the cost of more computation and/or a time-consuming pre-training stage. But this prior serves as a good starting point to benchmark more complicated alternatives, e.g., the normalizing flow based one in (Liu et al, 2019b).

Decoder The aim of a decoder in graph generative models is to construct a probability distribution over the graph and its feature/attributes conditioned on the latent variables, *i.e.*, $p(\mathcal{G}|Z)$. However, as we discussed previously, we need to consider all possible node orderings (each corresponds to a permuted adjacency matrix) which leaves the graph unchanged, *i.e.*,

$$p(\mathcal{G}|Z) = \sum_{P \in \Pi_{\mathcal{G}}} p(PAP^\top, PX|Z). \tag{11.13}$$

Recall that $\Pi_{\mathcal{G}}$ is the maximal subset of the set of all possible permutation matrices Π so that $P_1 A_\pi P_1^\top \neq P_2 A_\pi P_2^\top$ holds for any $P_1, P_2 \in \Pi_{\mathcal{G}}$. To build such a decoder, we first construct a probability distribution over adjacency matrix and node feature matrix. For example, we show a popular and simple construction (Kipf and Welling, 2016) as below,

$$p(A, X|Z) = \prod_{i,j} p(A_{ij}|Z) \prod_{i=1}^{n} p(\mathbf{x}_i|Z) \tag{11.14}$$

$$p(A_{ij}|Z) = \text{Bernoulli}(\Theta_{ij}) \tag{11.15}$$

$$p(\mathbf{x}_i|Z) = \mathcal{N}(\tilde{\boldsymbol{\mu}}_i, \tilde{\boldsymbol{\sigma}}_i) \tag{11.16}$$

$$\Theta_{ij} = \text{MLP}_\Theta([\mathbf{z}_i \| \mathbf{z}_j]) \tag{11.17}$$

$$\tilde{\boldsymbol{\mu}}_i = \text{MLP}_{\tilde{\boldsymbol{\mu}}}(\mathbf{z}_i) \tag{11.18}$$

$$\tilde{\boldsymbol{\sigma}}_i = \text{MLP}_{\tilde{\boldsymbol{\sigma}}}(\mathbf{z}_i), \tag{11.19}$$

where we adopt an edge-independent Bernoulli distribution over edges and node-wise independent Gaussian distribution over node features. $[\mathbf{z}_i \| \mathbf{z}_j]$ means concatenating $\mathbf{z}_i$ and $\mathbf{z}_j$. $\mathbf{x}_i$ is the i-th row of node feature matrix X. The first product term in Eq. (11.14) sums over all n^2 possible edges. The learnable parameters consist of those of three MLPs. This decoder is simple yet powerful. However, given the latent variables Z, the decoder is not permutation invariant in general, *i.e.*, for any two different permutation matrices P_1 and P_2,

$$p(P_1 A P_1^\top, P_1 X|Z) \neq p(P_2 A P_2^\top, P_2 X|Z). \tag{11.20}$$

Note that there are corner cases so that $p(P_1 A P_1^\top, P_1 X|Z) = p(P_2 A P_2^\top, P_2 X|Z)$ holds. For example, if an adjacency matrix A has certain symmetries, there could exist a pair of (P_1, P_2) so that $P_1 A P_1^\top = P_2 A P_2^\top$. But this does not hold for all pairs of (P_1, P_2). As a second example, if all Θ_{ij} are the same for all (i, j), all $\tilde{\boldsymbol{\mu}}_i$ are the same for all i, and all $\tilde{\boldsymbol{\sigma}}_i$ are the same for all i, then for any two permutation matrices (P_1, P_2), we have $p(P_1 A P_1^\top, P_1 X|Z) = p(P_2 A P_2^\top, P_2 X|Z)$. Nevertheless, these two cases happen rarely in practice.

Equipped with the distribution in Eq. (11.14), we can evaluate the terms on the right hand side of Eq. (11.13). However, the number of permutation matrices in $\Pi_{\mathcal{G}}$ can be as large as $n!$ which makes the exact evaluation computationally prohibitive. There are a few ways in the literature to approximate it. For example, we can just use the maximum term as below,

$$p(\mathcal{G}|Z) = \sum_{P \in \Pi_{\mathcal{G}}} p(PAP^\top, PX|Z) \approx \max_{P \in \Pi_{\mathcal{G}}} \quad p(PAP^\top, PX|Z). \tag{11.21}$$

Unfortunately, this maximization problem can be interpreted as an integer quadratic programming which is itself a hard optimization problem. To approximately solve the matching problem, Simonovsky and Komodakis (2018) exploit a relaxed max-pooling matching solver (Cho et al, 2014b). On the other hand, there are some canonical node orderings in certain applications. For example, the simplified molecular-

input line-entry system (SMILES) string (Weininger, 1988) provides a sequential ordering of atoms (nodes) of molecule graphs in chemistry. Based on the canonical node ordering, we can construct the corresponding permutation $\tilde{P}$ and simply approximate the conditional probability as,

$$p(\mathcal{G}|Z) = \sum_{P \in \Pi_{\mathcal{G}}} p(PAP^\top, PX|Z) \approx p(\tilde{P}A\tilde{P}^\top, \tilde{P}X|Z). \tag{11.22}$$

Objective The training objective of GraphVAE is similar to regular VAEs, *i.e.*, the evidence lower bound (ELBO),

$$\max_{\theta,\phi} \quad \mathbb{E}_{q_\phi(Z|A,X)}\left[\log p_\theta(\mathcal{G}|Z)\right] - \mathrm{KL}(q_\phi(Z|A,X)\|p(Z)) \tag{11.23}$$

To learn the encoder and the decoder, we need to sample from the encoder to approximate the expectation in Eq. (11.23) and leverage the reparameterization trick (Kingma and Welling, 2014) to back-propagate the gradient.

11.3.2.2 Hierarchical and Constrained GraphVAEs

There are many variants derived from the GraphVAE family mentioned above. We now briefly introduce two important types of variants, *i.e.*, hierarchical GraphVAE (Jin et al, 2018a) and Constrained GraphVAE (Liu et al, 2018d; Ma et al, 2018).

Hierarchical GraphVAEs One representative work of hierarchical GraphVAEs is *Junction Tree VAEs* (Jin et al, 2018a) which aim at modeling the molecule graphs. The key idea is to build a GraphVAE relying on the hierarchical graph representations of molecules. In particular, we first apply the tree decomposition to obtain a junction tree $\mathcal{T}$ from the original molecule graph $\mathcal{G}$. A *junction tree* is a cluster tree (each node is a set of one or more variables of the original graph) with the running intersection property (Barber, 2004). It provides a coarsened representation of the original graph since one node in a junction tree may correspond to a subgraph with several nodes in the original graph. As shown in Figure 11.1, there are two graphs corresponding to two levels, *i.e.*, the original molecule graph $\mathcal{G}$ (1st level) and the decomposed junction tree $\mathcal{T}$ (2nd level). Since we can efficiently perform tree decomposition to obtain the junction tree, the tree itself is not a latent variable. Jin et al (2018a) propose to use Gated Graph Neural Networks (GGNNs) (Li et al, 2016b) as encoders (one for each level) and construct variational posteriors $q(Z_{\mathcal{G}}|\mathcal{G})$ and $q(Z_{\mathcal{T}}|\mathcal{T})$ as Gaussians. To decode the molecule graph, we need to perform a two-level generation process conditioned on the sampled latent variables $Z_{\mathcal{T}}$ and $Z_{\mathcal{G}}$. A junction tree is first generated by a autoregressive decoder which is again based on GGNNs. Conditioned on the generated tree, Jin et al (2018a) resort to maximum-a-posterior (MAP) formulation to generate the final molecule graph, *i.e.*, finding the compatible subgraphs at each node of the tree so that the overall score (log-likelihood) of the resultant graph (*i.e.*, replacing each node in the tree with the chosen subgraph) is maximized. The whole model can be learned similarly to other

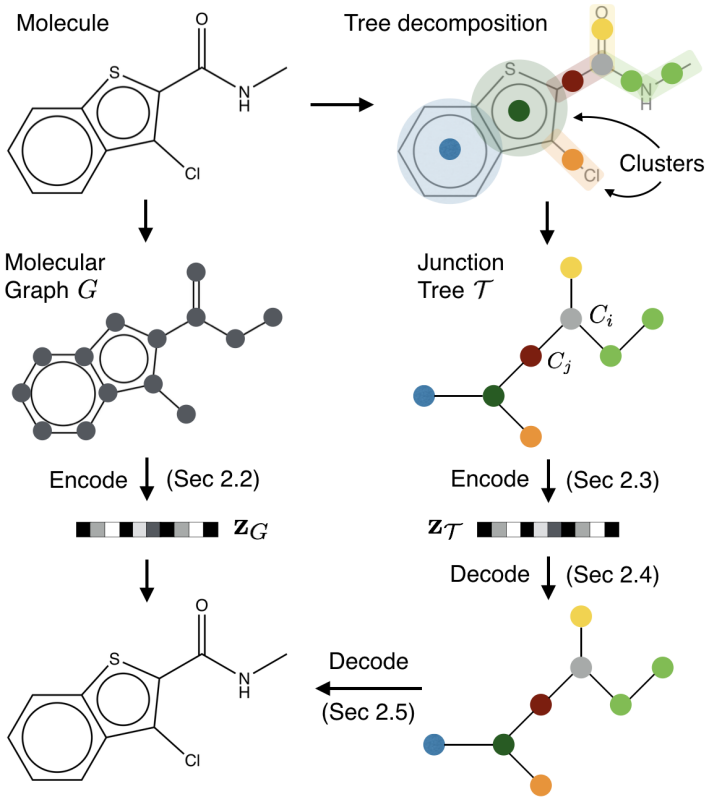

Fig. 11.1: Junction Tree VAEs. The junction tree corresponding to the molecule graph is obtained via the tree decomposition as shown in the top-right. A node/cluster in the junction tree (color-shaded) may correspond to a subgraph in the original molecule graph. Two GNN-based encoders are applied to the molecular graph and junction tree respectively to construct the variational posterior distributions over latent variables $Z_\mathcal{G}$ and $Z_\mathcal{T}$. During the generation, we first generate the junction tree using an autoregressive decoder and then obtains the final molecule graph via approximately solving a maximum-a-posterior problem. Adapted from Figure 3 of (Jin et al, 2018a).

GraphVAEs. This model provides an interesting extension of GraphVAEs to hierarchical graph generation and demonstrates strong empirical performances. There are other important application-dependent details which greatly improve efficiency. For example, we can build a dictionary of chemically valid subgraphs so that each generation step in the 2nd level decoding generates a subgraph rather than a single node. Nevertheless, the model design largely relies on the efficiency of the chosen junction tree algorithm and certain application-dependent properties. It is unclear how well this model performs on general graphs other than molecules.

Constrained GraphVAEs In many applications of deep graph generative models, certain constraints on the generated graphs are preferred. For example, while generating molecule graphs, the configuration of chemical bonds (edges) must meet the valence criteria of the atoms (nodes). How to ensure the generated graphs satisfy such constraints is a challenging problem. There are generally two types of approaches to overcome it in the context of GraphVAEs. The first type is to design a decoder so that all generated graphs satisfy the constraints by construction. For example, an autoregressive decoder is often adopted as in (Liu et al, 2018d; Dai et al, 2018b). At each step, conditioned on the currently generated graph, the model generates a new node, a new edge, and the node/edge attributes following certain rules, *i.e.*, ruling out invalid options (those would violate the constraints) like what GrammarVAEs (Kusner et al, 2017) do. The other type of approach is to treat the constraints softly. Similar to how constrained optimization problems are converted to unconstrained ones by adding Lagrangians, Ma et al (2018) propose Lagrangian-based regularizers to incorporate constraints like valence constraint for molecule graphs, connectivity constraint, and node compatibility. The benefits of such methods are that the generation could be much simpler and more efficient since we do not need a slow autoregressive decoder. Also, the regularization is only applied during learning and does not bring any overhead in the generation. Of course, the downside is that the generated graph my not exactly satisfy all constraints since the regularization only acts softly in the optimization.

11.3.3 Deep Autoregressive Methods

Deep autoregressive models like PixelRNNs (Van Oord et al, 2016) and PixelCNNs (Oord et al, 2016) have achieved tremendous successes in image modeling. Therefore, it is natural to generalize this type of method to graphs. The shared underlying idea of these autoregressive models is to characterize the graph generation process as a sequential decision-making process and make a new decision at each step conditioning on all previously made decisions. For example, as shown in Figure 11.2, we can first decide whether to add a new node, then decide whether to add a new edge, so on and so forth. If node/edge labels are considered, we can further sample from a categorical distribution at each step to specify such labels. The key question of this class of methods is how to build a probabilistic model so that our current decision depends on all previous historical choices.

11.3.3.1 GNN-based Autoregressive Model

The first GNN-based autoregressive model was proposed in (Li et al, 2018d) of which the high-level idea is exactly the same as shown in Figure 11.2. Suppose at time step $t-1$, we already generated a partial graph denoted as $\mathscr{G}^{t-1} = (\mathscr{V}^{t-1}, \mathscr{E}^{t-1})$. The corresponding adjacency matrix and node feature matrix are de-

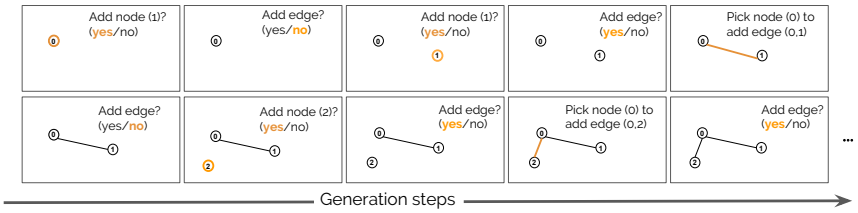

Fig. 11.2: The overview of the deep graph generative model in (Li et al, 2018d). The graph generation is formulated as a sequential decision-making process. At each step of the generation, the model needs to decide: 1) whether add a new node or stop the whole generation; 2) whether add a new edge (one end connected to the new node) or not; 3) which existing node to connect for the new edge. Adapted from Figure 1 of (Li et al, 2018d).

noted as (A^{t-1}, X^{t-1}). At time step t, the model needs to decide: 1) whether we add a new node or we stop the generation (denoting the probability as p_{AddNode}); 2) whether we add an edge that links any existing node to the newly added node (denoting the probability as p_{AddEdge}); 3) choose a existing node to link to the newly added node (denoting the probability as p_{Nodes}). For simplicity, we define p_{AddNode} to be a Bernoulli distribution. We can extend it to a categorical one if node labels/-types are considered. p_{AddEdge} is yet another Bernoulli distribution whereas p_{Nodes} is a categorical distribution with size $|\mathcal{V}^{t-1}|$ (*i.e.*, its size will change as the generation goes on).

Message Passing Graph Neural Networks To construct the above probabilities of decisions, we first build a message passing graph neural network (Scarselli et al, 2008; Li et al, 2016b; Gilmer et al, 2017) to learn node representations. The input to the GNN at time step $t-1$ is (A^{t-1}, H^{t-1}) where H^{t-1} is the node representation (one row corresponds to a node). Note that at time 0, since the graph is empty, we need to generate a new node to start. The generation probability p_{AddNode} will be output by the model based on some randomly initialized hidden state. If we model the node labels/types or node features, we can also use them as additional node representations, e.g., concatenating them with rows of H^{t-1}.

The one-step message passing is shown as below,

$$\mathbf{m}_{ij} = f_{\text{Msg}}(\mathbf{h}_i^{t-1}, \mathbf{h}_j^{t-1}) \qquad \forall (i,j) \in \mathcal{E} \tag{11.24}$$

$$\bar{\mathbf{m}}_i = f_{\text{Agg}}(\{\mathbf{m}_{ij} | \forall j \in \Omega_i\}) \qquad \forall i \in \mathcal{V} \tag{11.25}$$

$$\tilde{\mathbf{h}}_i^{t-1} = f_{\text{Update}}(\mathbf{h}_i^{t-1}, \bar{\mathbf{m}}_i) \qquad \forall i \in \mathcal{V}, \tag{11.26}$$

where f_{Msg}, f_{Agg}, and f_{Update} are the message function, the aggregation function, and the node update function respectively. For the message function, we often instantiate f_{Msg} as an MLP. Note that if edge features are considered, one can incorporate them as input to f_{Msg}. f_{Agg} could simply be an average or summation operator. Typical examples of f_{Update} include gated recurrent units (GRUs) (Cho et al, 2014a)

and long-short term memory (LSTM) (Hochreiter and Schmidhuber, 1997). $\mathbf{h}_i^{t-1}$ is the input node representation at time step $t-1$. Ω_i denotes the set of neighboring nodes of the node i. $\tilde{\mathbf{h}}_i^{t-1}$ is the updated node representation which serves as the input node representation for the next message passing step. The above message passing process is typically executed for a fixed number of steps, which is tuned as a hyperparameter. Note that the generation step t is different from the message passing step (we deliberately omit its notation to avoid confusion).

Output Probabilities After the message passing process is done, we obtain the new node representations H^t. Now we can construct the aforementioned output probabilities as follows,

$$\mathbf{h}_{\mathscr{G}^{t-1}} = f_{\text{ReadOut}}(H^t) \tag{11.27}$$

$$p_{\text{AddNode}} = \text{Bernoulli}(\sigma(\text{MLP}_{\text{AddNode}}(\mathbf{h}_{\mathscr{G}^{t-1}}))) \tag{11.28}$$

$$p_{\text{AddEdge}} = \text{Bernoulli}(\sigma(\text{MLP}_{\text{AddEdge}}(\mathbf{h}_{\mathscr{G}^{t-1}}, \mathbf{h}_v))) \tag{11.29}$$

$$s_{uv} = \text{MLP}_{\text{Nodes}}(\mathbf{h}_u^t, \mathbf{h}_v) \qquad \forall u \in \mathscr{V}^{t-1} \tag{11.30}$$

$$p_{\text{Nodes}} = \text{Categorical}(\text{softmax}(\mathbf{s})). \tag{11.31}$$

Here we first summarize the graph representation $\mathbf{h}_{\mathscr{G}^{t-1}}$ (a vector) by reading out from the node representation H^t via f_{ReadOut}, which could be an average operator or an attention-based one. Based on $\mathbf{h}_{\mathscr{G}^{t-1}}$, we predict the probability of adding a new node p_{AddNode} where σ is the sigmoid function. If we decide to add a new node by sampling 1 from the Bernoulli distribution p_{AddNode}, we denote the new node as v. We can initialize its representation $\mathbf{h}_v$ as random features by sampling either from $\mathscr{N}(0,\text{I})$ or learned distribution over node type/label if provided. Then we compute similarity scores between every existing node u in $\mathscr{G}^{t-1}$ and v as s_{uv}. $\mathbf{s}$ is the concatenated vector of all similarity scores. Finally, we normalize the scores using softmax to form the categorical distribution from which we sample an existing node to obtain the new edge. By sampling from all these probabilities, we could either stop the generation or obtain a new graph with a new node and/or a new edge. We repeat this procedure by carrying on the node representations along with the generated graphs until the model generates a stop signal from p_{AddNode}.

Training To train the model, we need to maximize the likelihood of the observed graphs. Recall that we need to consider the permutations that leave the graph unchanged as discussed in Section 11.3.2.1. For simplicity, we focus on the adjacency matrix alone following (Li et al, 2018d), *i.e.*, $\mathscr{G} \equiv \{PAP^\top | P \in \Pi_{\mathscr{G}}\}$, where $\Pi_{\mathscr{G}}$ is the maximal subset of Π so that $P_1 A P_1^\top \neq P_2 A P_2^\top$ holds for any $P_1, P_2 \in \Pi_{\mathscr{G}}$. The ideal objective is to maximize the following,

$$\max \quad \log p(\mathscr{G}) \Leftrightarrow \max \quad \log \left(\sum_{P \in \Pi_{\mathscr{G}}} p(PAP^\top) \right). \tag{11.32}$$

Here we omit the variables being optimized, *i.e.*, parameters of models defined in Eq. (11.24) and Eq. (11.27). Note that given a node ordering (corresponding to one specific permutation matrix P), we have a bijection between a sequence of cor-

rect decisions and an adjacency matrix. In other words, we can equivalently write $p(PAP^\top)$ as a product of probabilities that are explained in Eq. (11.27). However, the marginalization inside the logarithmic function on the right hand side is intractable due to the nearly factorial size of $\Pi_{\mathcal{G}}$ in practice. Li et al (2018d) propose to randomly sample a few different node orderings as $\tilde{\Pi}_{\mathcal{G}}$ and train the model with following approximated objective,

$$\max \quad \log \left(\sum_{P \in \tilde{\Pi}_{\mathcal{G}}} p(PAP^\top) \right). \tag{11.33}$$

Note that this objective is a strict lower bound of the one in Eq. (11.32). If canonical node orderings like the SMILES ordering for molecule graphs are available, we can also use that to compute the above objective.

Discussion This model formulates the graph generation as a sequential decision-making process and provides a GNN-based autoregressive model to construct probabilities of possible decisions at each step. The overall model design is well-motivated. It also achieves good empirical performances in generating small graphs like molecules (e.g., less than 40 nodes). However, since the model only generates at most one new node and one new edge per step, the total number of generation steps scales with the number of nodes quadratically for dense graphs. It is thus inefficient to generate moderately large graphs (e.g., with a few hundreds of nodes).

11.3.3.2 Graph Recurrent Neural Networks (GraphRNN)

Graph Recurrent Neural Networks (GraphRNN) (You et al, 2018b) is another deep autoregressive model which has a similar sequential decision-making formulation and leverages RNNs to construct the conditional probabilities. We again rely on the adjacency matrix representation of a graph, *i.e.*, $\mathcal{G} \equiv \{PAP^\top | P \in \Pi_{\mathcal{G}}\}$. Before dealing with the permutations, let us assume the node ordering is given so that $P = \mathrm{I}$.

A Simple Variant of GraphRNN GraphRNN starts with an autoregressive decomposition of the probability of an adjacency matrix as follows,

$$p(A) = \prod_{t=1}^{n} p(A_t | A_{<t}), \tag{11.34}$$

where A_t is the t-th column of the adjacency matrix A and $A_{<t}$ is a matrix formed by columns $A_1, A_2, \cdots, A_{t-1}$. n is the maximum number of nodes. If a graph has less than n nodes, we pad dummy nodes similarly as discussed in Section 11.3.1. Then we can construct the conditional probability as an edge-independent Bernoulli distribution,

$$p(A_t|A_{<t}) = \text{Bernoulli}(\Theta_t) = \prod_{i=1}^{n} \Theta_{t,i}^{\mathbf{1}[A_{i,t}=1]} (1-\Theta_{t,i})^{\mathbf{1}[A_{i,t}=0]} \tag{11.35}$$

$$\Theta_t = f_{\text{out}}(\mathbf{h}_t) \tag{11.36}$$

$$\mathbf{h}_t = f_{\text{trans}}(\mathbf{h}_{t-1}, A_{t-1}), \tag{11.37}$$

where Θ_t is a size-n vector of Bernoulli parameters. $\Theta_{t,i}$ denotes its i-th element. $A_{i,t}$ denotes the i-th element of the column vector A_t. f_{out} could be an MLP which takes the hidden state $\mathbf{h}_t$ as input and outputs Θ_t. f_{trans} is the RNN cell function which takes the $(t-1)$-th column of the adjacency matrix A_{t-1} and the hidden state $\mathbf{h}_{t-1}$ as input and outputs the current hidden state $\mathbf{h}_t$. We can use an LSTM or GRU as the RNN cell function. Note that the conditioning on $A_{<t}$ is implemented via the recurrent use of the hidden state in an RNN. The hidden state can be initialized as zeros or randomly sampled from a standard normal distribution. This model variant is very simple and can be easily implemented since it only consists of a few common neural network modules, *i.e.*, an RNN and an MLP.

Full Version of GraphRNN To further improve the model, You et al (2018b) propose a full version of GraphRNN. The idea is to build a hierarchical RNN so that the conditional distribution in Eq. (11.34) becomes more expressive. Specifically, instead of using an edge-independent Bernoulli distribution, we leverage another autoregressive construction to model the dependencies among entries within one column of the adjacency matrix as below,

$$p(A_t|A_{<t}) = \prod_{i=1}^{n} p(A_{i,t}|A_{<i,<t}) \tag{11.38}$$

$$p(A_{i,t}|A_{<i,<t}) = \text{sigmoid}(g_{\text{out}}(\tilde{\mathbf{h}}_{i,t})) \tag{11.39}$$

$$\tilde{\mathbf{h}}_{i,t} = g_{\text{trans}}(\tilde{\mathbf{h}}_{i-1,t}, A_{<i,t}) \tag{11.40}$$

$$\tilde{\mathbf{h}}_{0,t} = \mathbf{h}_t \tag{11.41}$$

$$\mathbf{h}_t = f_{\text{trans}}(\mathbf{h}_{t-1}, A_{t-1}). \tag{11.42}$$

Here the bottom RNN cell function f_{trans} still recurrently updates the hidden state to get $\mathbf{h}_t$, thus implementing the conditioning on all previous $t-1$ columns of the adjacency matrix A. To generate individual entries of the t-th column, the top RNN cell function g_{trans} takes its own hidden state $\tilde{\mathbf{h}}_{i-1,t}$ and the already generated t-th column A as input and updates the hidden state as $\tilde{\mathbf{h}}_{i,t}$. The output distribution is a Bernoulli parameterized by the output of an MLP g_{out} which takes $\tilde{\mathbf{h}}_{i,t}$ as input. Note that the initial hidden state $\tilde{\mathbf{h}}_{0,t}$ of the top RNN is set to the hidden state $\mathbf{h}_t$ returned by the bottom RNN.

Objective To train the GraphRNN, we can again resort to the maximum log likelihood similarly to Section 11.3.3.1. We also need to deal with permutations of nodes that leave the graph unchanged. Instead of randomly sampling a few orderings like (Li et al, 2018d), You et al (2018b) propose to use a random-breadth-first-search ordering. The idea is to first randomly sample a node ordering and then pick the first node in this ordering as the root. A breadth-first-search (BFS) algorithm is applied

starting from this root node to generate the final node ordering. Let us denote the corresponding permutation matrix as P_{BFS}. The final objective is,

$$\max \quad \log\left(p(P_{\text{BFS}}AP_{\text{BFS}}^{\top})\right), \tag{11.43}$$

which is again a strict lower bound of the true log likelihood. Empirical results in (You et al, 2018b) suggest that this random-BFS ordering provides good performances on a few benchmarks.

Discussion The design of the GraphRNN is simple yet effective. The implementation is straightforward since most of the modules are standard. The simple variant is more efficient than the previous GNN-based model (Li et al, 2018d) since it generates multiple edges (corresponding to one column of the adjacency matrix) per step. Moreover, the simple variant performs comparably with the full version in the experiments. Nevertheless, GraphRNN still has certain limitations. For example, RNN highly depends on the node ordering since different node orderings would result in very different hidden states. The sequential ordering could make two nearby (even neighboring) nodes far away in the generation sequence (*i.e.*, far away in the generation time step). Typically, hidden states of an RNN that are far away regarding the generation time step tend to be quite different, thus making it hard for the model to learn that these nearby nodes should be connected. We call this phenomenon the *sequential ordering bias*.

11.3.3.3 Graph Recurrent Attention Networks (GRAN)

Following the line of the work (Li et al, 2018d; You et al, 2018b), Liao et al (2019a) propose the graph recurrent attention networks (GRAN). It is a GNN-based autoregressive model, which greatly improves the previous GNN-based model (Li et al, 2018d) in terms of capacity and efficiency. Furthermore, it alleviates the *sequential ordering bias* of GraphRNN (You et al, 2018b). In the following, we introduce the details of the model.

Model We start with the adjacency matrix representation of graphs, *i.e.*, $\mathcal{G} \equiv \{PAP^{\top}|P \in \Pi_{\mathcal{G}}\}$. GRAN aims at directly building a probabilistic model over the adjacency matrix similarly to GraphRNN. Again, node/edge features are not of primary interests but can be incorporated without much modification to the model. In particular, from the perspective of modeling the adjacency matrix, the GNN-based autoregressive model in (Li et al, 2018d) generates one entry of the adjacency matrix at a step, whereas GraphRNN (You et al, 2018b) generates one column of entries at a step. GRAN takes a step further along this line by generating a block of columns/rows[2] of the adjacency matrix at a step, which greatly improves the generation speed. Denoting the submatrix with first k rows of the adjacency matrix A as $A_{1:k,:}$, we have the following autoregressive decomposition of the probability,

[2] Since we are mainly interested in simple graphs, *i.e.*, unweighted, undirected graphs containing no self-loops or multiple edges, modeling columns or rows makes no difference. We adopt the row-wise notations to follow the original paper.

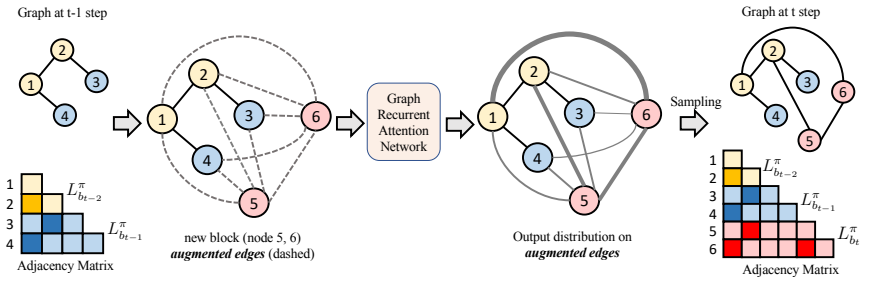

Fig. 11.3: The overview of the graph recurrent attention networks (GRAN). At each step, given an already generated graph, we add a new block of nodes (block size is 2 and color indicates the membership of individual group in the visualization) and augmented edges (dashed lines). Then we apply GRAN to this graph to obtain the output distribution over augmented edges (we show an edge-independent Bernoulli where the line width indicates the probability of generating individual augmented edges). Finally, we sample from the output distribution to obtain a new graph. Adapted from Figure 1 of (Liao et al, 2019a).

$$p(A) = \prod_{t=1}^{\lceil n/k \rceil} p(A_{(t-1)k:tk,:} | A_{:(t-1)k,:}), \tag{11.44}$$

where $A_{:(t-1)k,:}$ indicates the adjacency matrix that has been generated before the t-th step (*i.e.*, $t-1$ blocks with block size k). We use $A_{(t-1)k:tk,:}$ to denote the to-be-generated block at t-th time step. Note that this part is a straightforward generalization to the autoregressive model of GraphRNNs in Eq. (11.34).

To build the condition probability $p(A_{(t-1)k:tk,:}|A_{:(t-1)k,:})$, GRAN leverages a message passing graph neural network. Specifically, denoting the already generated graph before step t (corresponding to $A_{:(t-1)k,:}$) as $\mathcal{G}^{t-1} = (\mathcal{V}^{t-1}, \mathcal{E}^{t-1})$, we first initialize every node representation vector with its corresponding row of the adjacency matrix, *i.e.*, $\mathbf{h}_v = A_{v,:}$ for all $v \leq (t-1)k$. Since we assume the maximum number of nodes is n and pad dummy nodes for graphs with a smaller size, $\mathbf{h}_v$ is of size n. At time step t, we are interested in generating a new block of nodes (corresponding to $A_{(t-1)k:tk,:}$) and their associated edges. For the k new nodes in the t-th block, since their corresponding rows in the adjacency matrix are initially all zeros, we give them an arbitrary ordering from 1 to k and use the one-hot-encoding of the order index as an additional representation to distinguish them, denoting as x_u. We first form a new graph $\tilde{\mathcal{G}}^t = (\mathcal{V}^t, \tilde{\mathcal{E}}^t)$ by connecting the k new nodes to themselves (excluding self-loops) and every other nodes in $\mathcal{G}^{t-1}$. We call such edges as the augmented edges, which are shown as the dashed edges in Figure 11.3. In other words, $\mathcal{V}^t$ is the union of $\mathcal{V}^{t-1}$ and k new nodes whereas $\tilde{\mathcal{E}}^t$ is the union of $\mathcal{E}^{t-1}$ and augmented edges. The core part of GRAN is to construct a probability distribution over such augmented edges from which we can sample a new graph $\mathcal{G}^t$. Note that $\mathcal{G}^t$ has the same set of nodes but potentially fewer edges compared to $\tilde{\mathcal{G}}^t$. To construct the

probability, we use a GNN with the following one-step message passing process,

$$\mathbf{m}_{ij} = f_{\text{msg}}(\mathbf{h}_i - \mathbf{h}_j), \qquad \forall (i,j) \in \tilde{\mathscr{E}}^t \tag{11.45}$$

$$\tilde{\mathbf{h}}_i = [\mathbf{h}_i \| \mathbf{x}_i], \qquad \forall i \in \mathscr{V}^t \tag{11.46}$$

$$a_{ij} = \text{sigmoid}\left(g_{\text{att}}(\tilde{\mathbf{h}}_i - \tilde{\mathbf{h}}_j)\right), \qquad \forall (i,j) \in \tilde{\mathscr{E}}^t \tag{11.47}$$

$$\mathbf{h}'_i = \text{GRU}(\mathbf{h}_i, \textstyle\sum_{j \in \Omega(i)} a_{ij}\mathbf{m}_{ij}), \qquad \forall i \in \mathscr{V}^t \tag{11.48}$$

where $\mathbf{m}_{ij}$ is the again the message over edge (i,j) and Ω_i is the set of neighboring nodes of node i. The message function f_{msg} and the attention head g_{att} could be MLPs. Note that we set $\mathbf{x}_u$ to zeros for any node u that is in the already generated graph $\mathscr{G}^{t-1}$ since the one-hot-encoding is only used to distinguish those newly added nodes. $[a\|b]$ means concatenating two vectors a and b. The updated node representation $\mathbf{h}'_i$ would serve as the input to the next message passing step. We typically unroll this message passing for a fixed number of steps, which is set as a hyperparameter. Note that the message passing step is independent of the generation step. The attention weights a_{ij} depends on the one-hot-encoding $\mathbf{x}_i$ so that messages on augmented edges could be weighted differently compared to those on edges belonging to $\mathscr{E}^{t-1}$. Based on the final node representations returned by the message passing, we can construct the output distribution is as follows,

$$p(A_{(t-1)k:tk,:}|A_{:(t-1)k,:}) = \sum_{c=1}^{C} \alpha_c \prod_{i=(t-1)k+1}^{tK} \prod_{j=1}^{n} \Theta_{c,i,j} \tag{11.49}$$

$$\alpha = \text{softmax}\left(\sum_{i=(t-1)k+1}^{tK} \sum_{j=1}^{n} \text{MLP}_\alpha(\mathbf{h}_i^R - \mathbf{h}_j^R)\right) \tag{11.50}$$

$$\Theta_{c,i,j} = \text{sigmoid}\left(\text{MLP}_\Theta(\mathbf{h}_i^R - \mathbf{h}_j^R)\right). \tag{11.51}$$

Here we use a mixture of Bernoulli distributions where the mixture coefficients are $\boldsymbol{\alpha} = \{\alpha_1, \cdots, \alpha_C\}$ and the parameters are $\{\Theta_{c,i,j}\}$. Compared to the edge-independent Bernoulli distribution used in the simple variant of GraphRNN, this output distribution can capture dependencies among multiple generated edges. Furthermore, it is more efficient to sample compared to the autoregressive distribution used in the full version of GraphRNN.

Objective To train the model, we also need to deal with permutations in order to maximize the log likelihood. Similar to the strategy used in (Li et al, 2018d; You et al, 2018b), Liao et al (2019a) propose to use a set of canonical orderings, *i.e.*, breadth-first-search (BFS), depth-first-search (DFS), node-degree-descending, node-degree-ascending, and the k-core ordering. In particular, the BFS and the DFS ordering start from the node with the largest node degree. The k-core graph decomposition has been shown to be very useful for modeling cohesive groups in social networks (Seidman, 1983). The k-core of a graph $\mathscr{G}$ is a maximal subgraph that contains nodes of degree k or more. Cores are nested, *i.e.*, i-core belongs to j-core if $i > j$, but they are not necessarily connected subgraphs. Most importantly, the core decomposition, *i.e.*, all cores ranked based on their orders, can be found in lin-

ear time (w.r.t. the number of edges) (Batagelj and Zaversnik, 2003). Based on the largest core number per node, we can uniquely determine a partition of all nodes, *i.e.*, disjoint sets of nodes which share the same largest core number. We then assign the core number of each disjoint set by the largest core number of its nodes. Starting from the set with the largest core number, we rank all nodes within the set in node degree descending order. Then we move to the second largest core and so on to obtain the final ordering of all nodes. We call this core descending ordering as *k-core node ordering*.

Our final training objective is,

$$\max \quad \log \left(\sum_{P \in \tilde{\Pi}_{\mathcal{G}}} p(PAP^{\top}) \right). \tag{11.52}$$

where $\tilde{\Pi}_{\mathcal{G}}$ is the set of permutation matrices corresponding to the above node orderings. This is again a strict lower bound of the true log likelihood.

Discussion GRAN improves the previous GNN-based autoregressive model (Li et al, 2018d) and GraphRNN (You et al, 2018b) in the following ways. First, it generates a block of rows of the adjacency matrix per step, which is more efficient than generating an entry per step and than generating a row per step. Second, GRAN uses a GNN to construct the conditional probability. This helps alleviate the sequential ordering bias in GraphRNN since GNN is permutation equivariant, *i.e.*, the node ordering would not affect the conditional probability per step. Third, the output distribution in GRAN is more expressive and more efficient for sampling. GRAN outperform previous deep graph generative models in terms of empirical performances and the sizes of graphs that can be generated (e.g., GRAN can generate graphs up to 5K nodes). Nevertheless, GRAN still suffers from the fact that the overall model depends on the particular choices of node orderings. It may be hard to find good orderings in certain applications. How to build an order-invariant deep graph generative model would be an interesting open question.

11.3.4 Generative Adversarial Methods

In this part, we review a few methods (De Cao and Kipf, 2018; Bojchevski et al, 2018; You et al, 2018a) that apply the idea of generative adversarial networks (GAN) (Goodfellow et al, 2014b) in the context of graph generation. Based on how a graph is represented during training, we roughly divide them into two categories: adjacency matrix based and random walks based methods. In the following, we explain these two types of methods in detail.

11.3.4.1 Adjacency Matrix Based GAN

MolGAN (De Cao and Kipf, 2018) and graph convolutional policy network (GCPN) (You et al, 2018a) propose a similar GAN-based framework to generate molecule graphs that satisfy certain chemical properties. Here the graph data is represented slightly different from previous sections since one needs to specify both node types (*i.e.*, atom types) and edge types (*i.e.*, chemical bond types). We denote the adjacency matrix[3] as $A \in \mathbb{R}^{N\times N\times Y}$ where Y is the number of chemical bond types. Basically, one slice along the 3rd dimension of A gives an adjacency matrix that characterizes the connectivities among atoms under a specific chemical bond type. We denote the node type as $X \in \mathbb{R}^{N\times T}$ where T is the number of atom types. The goal is to generate (A,X) so that it is similar to observed molecule graphs and possesses certain desirable properties.

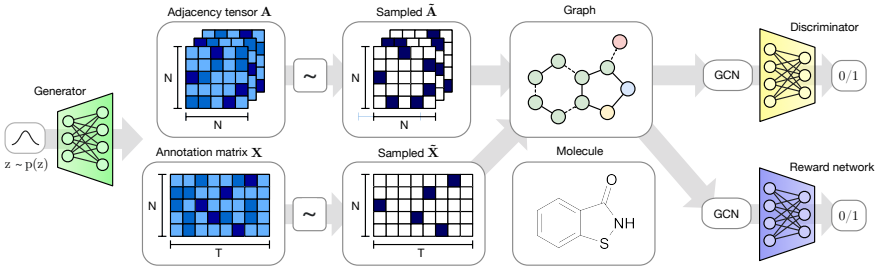

Fig. 11.4: The overview of the MolGAN. We first draw a latent variable $Z \sim p(Z)$ and feed it to a generator which produces a probabilistic (continuous) adjacency matrix A and a probabilistic (continuous) node type matrix X. Then we draw a discrete adjacency matrix $\tilde{A} \sim A$ and a discrete node type matrix $\tilde{X} \sim X$, which together specify a molecule graph. During training, we simultaneously feed the generated graph to a discriminator and a reward network to obtain the adversarial loss (measuring how similar the generated and the observed graphs are) and the negative reward (measuring how likely the generated graphs satisfy the certain chemical constraints). Adapted from Figure 2 of (De Cao and Kipf, 2018).

Model We now explain the details of MolGAN and then highlight the difference between GCPN and MolGAN. Similar to regular GANs, MolGAN consists of a generator $\bar{\mathscr{G}}_\theta(Z)$ and a discriminator $\mathscr{D}_\phi(A,X)$. To ensure the generated samples satisfy desirable chemical properties, MolGAN adopts an additional reward network $\mathscr{R}_\psi(A,X)$. The overall pipeline of MolGAN is illustrated in Figure 11.4.

To generate a molecule graph, we first sample a latent variable $Z \in \mathbb{R}^d$ from some prior, e.g., $Z \sim \mathscr{N}(0,\mathrm{I})$. Then we use an MLP to directly map the sampled Z to a continuous adjacency matrix A and a continuous node type matrix X. The continuous version of the graph data has a natural probabilistic interpretation, *i.e.*, $A_{i,j,c}$

[3] Note that A is actually a tensor. We slightly abuse the terminology here to ease the exposition.

means the probability of connecting the atom i and the atom j using the chemical bond type c, whereas $X_{i,t}$ means the probability of assigning the t-th atom type to the i-atom. One can sample a discrete graph data $(\tilde{A}, \tilde{X})$ from the continuous version, *i.e.*, $\tilde{A} \sim A$ and $\tilde{X} \sim X$. This sampling procedure can be implemented using the Gumbel softmax (Jang et al, 2017; Maddison et al, 2017). The discrete adjacency matrix $\tilde{A}$ along with the discrete node type $\tilde{X}$ specify a molecule graph and complete the generation process.

To evaluate how similar the generated graphs and the observed graphs are, we need to build a discriminator. Since we are dealing with graphs, the natural candidate for a discriminator is a graph neural network, e.g., a graph convolutional network (GCN) (Kipf and Welling, 2017b). In particular, we use a variant of GCN (Schlichtkrull et al, 2018) to incorporate multiple edge types. One such graph convolutional layer is shown as below,

$$\mathbf{h}_i' = \tanh\left(f_s(\mathbf{h}_i, \mathbf{x}_i) + \sum_{j=1}^{N}\sum_{y=1}^{Y} \frac{\tilde{A}_{i,j,y}}{|\Omega_i|} f_y(\mathbf{h}_j, \mathbf{x}_i)\right), \tag{11.53}$$

where $\mathbf{h}_i$ and $\mathbf{h}_i'$ are the input and the output node representations of the graph convolutional layer. Ω_i is the set of neighboring nodes of the node i. $\mathbf{x}_i$ is the i-th row of X, *i.e.*, the node type vector of the node i. f_s and f_y are linear transformation functions that are to be learned. After stacking this type of graph convolution for multiple layers, we can readout the graph representation using the following attention-weighted aggregation,

$$\mathbf{h}_{\mathscr{G}} = \tanh\left(\sum_{v \in \mathscr{V}} \text{sigmoid}\left(\text{MLP}_{\text{att}}(\mathbf{h}_v, \mathbf{x}_v)\right) \odot \tanh\left(\text{MLP}(\mathbf{h}_v, \mathbf{x}_v)\right)\right), \tag{11.54}$$

where $\mathbf{h}_v$ is the node representation returned by the top graph convolutional layer. Note that MLP_{att} and MLP are two different instances of MLPs. $\odot$ means element-wise product. We can use the graph representation vector $\mathbf{h}_{\mathscr{G}}$ to compute the discriminator score $\mathscr{D}_\phi(A, X)$, *i.e.*, the probability of classifying a graph as positive (*i.e.*, coming from the data distribution).

Objective Originally, GANs learn the model by performing the minimax optimization as below,

$$\min_\theta \max_\phi \quad \mathbb{E}_{A,X \sim p_{\text{data}}(A,X)}[\log \mathscr{D}_\phi(A, X)] + \mathbb{E}_{Z \sim p(Z)}[\log\left(1 - \mathscr{D}_\phi(\bar{\mathscr{G}}_\theta(Z))\right)], \tag{11.55}$$

where the generator aims at fooling the discriminator and the discriminator aims at correctly classifying the generated samples and the observed samples. To address certain issues in training GANs such as the mode collapse and the instability, Wasserstein GAN (WGAN) (Arjovsky et al, 2017) and its improved version (Gulrajani et al, 2017) have been proposed. MolGAN follows the improved WGAN and uses the following objective to train the discriminator $\mathscr{D}_\phi(A, X)$,

$$\max_{\phi} \quad \sum_{i=1}^{B} -\mathscr{D}_{\phi}(A^{(i)},X^{(i)}) + \mathscr{D}_{\phi}(\bar{\mathscr{G}}_{\theta}(Z^{(i)})) + \alpha \left(\|\nabla_{\hat{A}^{(i)},\hat{X}^{(i)}} \mathscr{D}_{\phi}(\hat{A}^{(i)},\hat{X}^{(i)})\| - 1 \right)^2, \tag{11.56}$$

where B is the mini-batch size, $Z^{(i)}$ is the i-th sample drawn from the prior, $A^{(i)},X^{(i)}$ are the i-th graph data drawn from the data distribution, and $\hat{A}^{(i)},\hat{X}^{(i)}$ are their linear combinations, *i.e.*, $(\hat{A}^{(i)},\hat{X}^{(i)}) = \varepsilon(A^{(i)},X^{(i)}) + (1-\varepsilon)\bar{\mathscr{G}}_{\theta}(Z^{(i)})$, $\varepsilon \sim \mathscr{U}(0,1)$. The squared term on the right-hand side penalizes the gradient of the discriminator so that the training becomes more stable. α is a weighting term to balance the regularization and the objective. Moreover, fixing the discriminator, we train the generator $\bar{\mathscr{G}}_{\theta}(A,X)$ by adding the additional constraint-dependent reward,

$$\min_{\theta} \quad \sum_{i=1}^{B} \lambda \mathscr{D}_{\phi}(\bar{\mathscr{G}}_{\theta}(Z^{(i)})) + (1-\lambda)\mathscr{L}_{\mathrm{RL}}(\bar{\mathscr{G}}_{\theta}(Z^{(i)})), \tag{11.57}$$

where $\mathscr{L}_{\mathrm{RL}}$ is the negative reward returned by the reward network $\mathscr{R}_{\psi}$ and λ is the weighting hyperparameter to regulate the trade-off between two losses. The reward could be some non-differentiable quantities that characterize the chemical properties of the generated molecules, e.g., how likely the generated molecule is to be soluble in water. To learn the model with the non-differentiable reward, the deep deterministic policy gradient (DDPG) (Lillicrap et al, 2015) is used. The architecture of the reward network is the same as the discriminator, *i.e.*, a GCN. It is pre-trained by minimizing the squared error between the predicted reward given by $\mathscr{R}_{\psi}$ and an external software which produces a property score per molecule. The pre-training is necessary since the external software is typically slow and could significantly delay the training if it is included in the whole training framework.

Discussion MolGAN demonstrates strong empirical performances on a large chemical database called QM9 (Ramakrishnan et al, 2014). Similar to other GANs, the model is likelihood-free and can thus enjoy more flexible and powerful generators. More importantly, although the generator still depends on the node ordering, the discriminator and the reward networks are order (permutation) invariant since they are built from GNNs. Interestingly enough, graph convolutional policy network (GCPN) (You et al, 2018a) solves the same problem using a similar approach. GCPN has a similar GAN-type of objective and some additional domain-specific rewards that capture the chemical properties of the molecules. It also learns both a generator and a discriminator. However, they do not use a reward network to speed up the reward computation. To deal with the learning of non-differentiable reward, GCPN leverages the proximal policy optimization (PPO) (Schulman et al, 2017) method, which empirically performs better than the vanilla policy gradient method. Another important difference is that GCPN generates the adjacency matrix in an entry-by-entry autoregressive fashion so that the dependencies among multiple generated edges are captured whereas MolGAN generates all entries of the adjacency matrix in parallel conditioned on the latent variable. GCPN also achieves impressive empirical results on another large chemical database called ZINC (Irwin et al, 2012). Nevertheless, there are still limitations with the above models. The discrete

gradient estimators (e.g., the policy gradient type of methods) could have large variances, which may slow down the training. Since the domain-specific rewards are non-differentiable and may be time-consuming to obtain, learning a neural network based approximated reward function like what MolGAN does is appealing. However, as reported in MolGAN, pre-training seems to be crucial to make the whole training successful. More exploration along the line of learning a reward function would be beneficial to simplify the whole training pipeline. On the other hand, both methods use some variant of GCNs as the discriminator, which is shown to be insufficient in distinguishing certain graphs[4] (**?**). Therefore, exploring more powerful discriminators like the Lanczos network (Liao et al, 2019b) that exploits the spectrum of the graph Laplacian as the input feature would be promising to further improve the performance of the above methods.

11.3.4.2 Random Walk Based GAN

In contrast to previous methods, NetGAN (Bojchevski et al, 2018) resorts to the random walk based representations of graphs. The key idea is to map a graph to a set of random walks and learn a generator and a discriminator in the space of random walks. The generator should generate random walks that are similar to those sampled from the observed graphs, whereas the discriminator should correctly distinguish whether a random walk comes from the data distribution or the implicit distribution corresponding to the generator.

Model We start by sampling a set of random walks with fixed length T from the given graph $\mathcal{G}$ using the biased second order random walk sampling strategy described in (Grover and Leskovec, 2016). We denote a random walk as a sequence $(v_1, \cdots, v_T)$ where v_i represents one node in $\mathcal{G}$. Note that a random walk may contain duplicate nodes since it could revisit one node multiple times during the sampling. We again assume the maximum number of nodes for any graph is N. For any node v_i, we use the one-hot-encoding vector as its node feature. In other words, we can view a random walk with a sequence along with its features. Therefore, similar to language models, it is natural to use an RNN as the generator for generating such random walks. NetGAN exploits an LSTM as the generator of which the initial hidden state $\mathbf{h}_0$ and the memory c_0 are computed by feeding a randomly sampled latent vector (drawn from $\mathcal{N}(0, \mathrm{I})$) to two separate MLPs. Then the LSTM generator predicts a categorical distribution over all possible nodes and then samples a node. The one-hot-encoding of the node index is treated as the node representation and fed to the LSTM generator as the input for the next step. We unroll this LSTM for T steps to obtain the final length-T random walk. For the discriminator, we can use another LSTM, which takes a random walk as input and predicts the probability that a given random walk is sampled from the data distribution. The model is trained with the same objective as the improved WGAN (Gulrajani et al, 2017).

[4] For example, a GCN can not distinguish two triangles versus a six node circle (both have the same number of nodes and every node has exactly two neighbors) assuming all individual node features are identical.

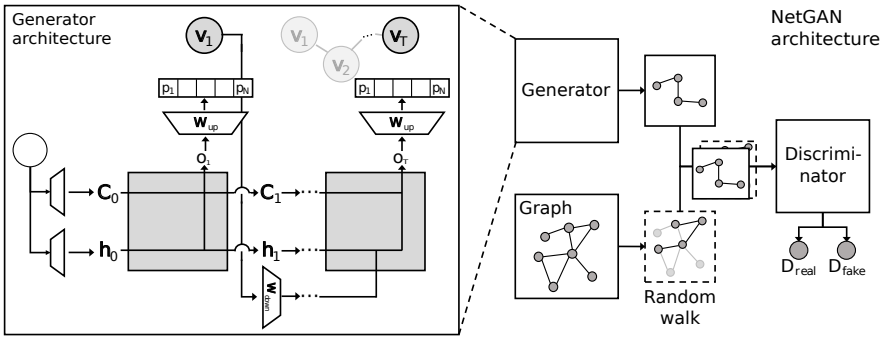

Fig. 11.5: The overview of the NetGAN. We first draw a random vector from a fixed prior $\mathcal{N}(0,\mathrm{I})$ and initialized the memory c_0 and the hidden state $\mathbf{h}_0$ of an LSTM. Then the LSTM generator generates which node to visit per step and is unrolled for a fixed number of steps T. The one-hot-encoding of node index is fed to the LSTM as the input for the next step. The discriminator is another LSTM which performs a binary classification to predict if a given random walk is sampled from a data distribution. Adapted from Figure 2 of (Bojchevski et al, 2018).

After training the LSTM generator, we are capable of generating random walks. However, we need an additional step to construct a graph from a set of generated random walks. The strategy used by NetGAN is as follows. First, we count the edges that appeared in the set of random walks to obtain a scoring matrix S, which has the same size as the adjacency matrix. The (i,j)-th entry of the score matrix $S_{i,j}$ indicates how many times edge (i,j) appears in the set of generated random walks. Second, for each node i, we sample a neighbor according to the probability $\frac{S_{i,j}}{\sum_v S_{i,v}}$. We repeat the sampling until node i has at least one neighbor connected and skip if the edge has already been generated. At last, for any edge (i,j), we perform sampling without replacement according to the probability $\frac{S_{i,j}}{\sum_{u,v} S_{u,v}}$ until the maximum number of edges is reached.

Discussion The random walk based representations for graphs are novel in the context of deep graph generative models. Moreover, they could be more scalable than the adjacency matrix representation since we are not bound by the quadratic (w.r.t. the number of nodes) complexity. The core modules of the NetGAN are LSTMs which are efficient in handling sequences and easy to be implemented. Nevertheless, the graph construction from a set of generated random walks seems to be ad-hoc. There is no theoretical guarantee on how accurate the proposed construction method is. It may require a large number of sampled random walks in order to generate a graph with good qualities.

11.4 Summary

In this chapter, we review a few classic graph generative models and some modern ones which are constructed based on deep neural networks. From the perspectives of the model capacity and the empirical performances, e.g., how good the model can fit observed data, deep graph generative models significantly outperform their classic counterparts. For example, they could generate molecule graphs which are both chemically valid and similar to observed ones in terms of certain graph statistics.

Although we have already made impressive progress in recent years, deep generative models are still in the early stage. Moving forward, there are at least two main challenges. First, how can we scale these models so that they can handle real-world graphs like large scale social networks and WWW? It requires not only more computational resources but also more algorithmic improvements. For example, building a hierarchical graph generative model would be one promising direction to boost efficiency and scale. Second, how can we effectively add domain-specific constraints and/or conditioning on some input information? This question is important since many real-world applications require the graph generation to be conditioned on some inputs (e.g., scene graph generations conditioned on input images). Many graphs in practice come with certain constraints (e.g., chemical validity in the molecule generation).

Editor's Notes: Deep learning-based graph generation can be considered as a downstream task of graph representation learning, where the learned representations are usually enforced to follow some probabilistic assumptions. Hence the techniques in this topic widely enjoy the relevant properties and theories introduced in the previous chapters, such as scalability (Chapter 6), expressiveness power (Chapter 5), and robustness (Chapter 8). Graph generation also further motivates its downstream tasks in various interesting, important, yet usually challenging areas such as drug discovery (see Chapter 24), protein analysis (see Chapter 25), and program synthesis (see Chapter 22).