GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models

Jiaxuan You [*1] **Rex Ying** [*1] **Xiang Ren** [2] **William L. Hamilton** [1] **Jure Leskovec** [1]

Abstract

Modeling and generating graphs is fundamental for studying networks in biology, engineering, and social sciences. However, modeling complex distributions over graphs and then efficiently sampling from these distributions is challenging due to the non-unique, high-dimensional nature of graphs and the complex, non-local dependencies that exist between edges in a given graph. Here we propose GraphRNN, a deep autoregressive model that addresses the above challenges and approximates any distribution of graphs with minimal assumptions about their structure. GraphRNN learns to generate graphs by training on a representative set of graphs and decomposes the graph generation process into a sequence of node and edge formations, conditioned on the graph structure generated so far. In order to quantitatively evaluate the performance of GraphRNN, we introduce a benchmark suite of datasets, baselines and novel evaluation metrics based on Maximum Mean Discrepancy, which measure distances between sets of graphs. Our experiments show that GraphRNN significantly outperforms all baselines, learning to generate diverse graphs that match the structural characteristics of a target set, while also scaling to graphs $50\times$ larger than previous deep models.

1. Introduction and Related Work

Generative models for real-world graphs have important applications in many domains, including modeling physical and social interactions, discovering new chemical and molecular structures, and constructing knowledge graphs. Development of generative graph models has a rich history, and many methods have been proposed that can generate graphs based on a priori structural assumptions (Newman, 2010). However, a key open challenge in this area 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 paves a way for new kinds of applications, such as discovering new graph structures and completing evolving graphs.

In contrast, traditional generative models for graphs (*e.g.*, Barabási-Albert model, Kronecker graphs, exponential random graphs, and stochastic block models) (Erdős & Rényi, 1959; Leskovec et al., 2010; Albert & Barabási, 2002; Airoldi et al., 2008; Leskovec et al., 2007; Robins et al., 2007) are hand-engineered to model a particular family of graphs, and thus do not have the capacity to directly learn the generative model from observed data. For example, the Barabási-Albert model is carefully designed to capture the scale-free nature of empirical degree distributions, but fails to capture many other aspects of real-world graphs, such as community structure.

Recent advances in deep generative models, such as variational autoencoders (VAE) (Kingma & Welling, 2014) and generative adversarial networks (GAN) (Goodfellow et al., 2014), have made important progress towards 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 (Kipf & Welling, 2016; Grover et al., 2017; Simonovsky & Komodakis, 2018; Li et al., 2018). For example, Simonovsky & Komodakis 2018 propose a VAE-based approach, while Li et al. 2018 propose a framework based upon graph neural networks. However, these recently proposed deep models are either limited to learning from a single graph (Kipf & Welling, 2016; Grover et al., 2017) or generating small graphs with 40 or fewer nodes (Li et al., 2018; Simonovsky & Komodakis, 2018)—limitations that stem from three fundamental challenges in the graph generation problem:

Limitations of current approach

- **Large and variable output spaces:** To generate a graph with n nodes the generative model has to output n^2 values to fully specify its structure. Also, the number of nodes n and edges m varies between different graphs and a generative model needs to accommodate such complexity and variability in the output space.

[*]Equal contribution [1]Department of Computer Science, Stanford University, Stanford, CA, 94305 [2]Department of Computer Science, University of Southern California, Los Angeles, CA, 90007. Correspondence to: Jiaxuan You <jiaxuan@stanford.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

- **Non-unique representations:** In the general graph generation problem studied here, we want distributions over 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/numbering. Such high representation complexity is challenging to model and makes it expensive to compute and then optimize objective functions, like reconstruction error, during training. For example, GraphVAE (Simonovsky & Komodakis, 2018) uses approximate graph matching to address this issue, requiring $O(n^4)$ operations in the worst case (Cho et al., 2014).

- **Complex dependencies:** Edge formation in graphs involves complex structural dependencies. For example, in many real-world graphs two nodes are more likely to be connected if they share common neighbors (Newman, 2010). Therefore, edges cannot be modeled as a sequence of independent events, but rather need to be generated jointly, where each next edge depends on the previously generated edges. Li et al. 2018 address this problem using graph neural networks to perform a form of "message passing"; however, while expressive, this approach takes $O(mn^2\text{diam}(G))$ operations to generate a graph with m edges, n nodes and diameter $\text{diam}(G)$.

Present work. Here we address the above challenges and present *Graph Recurrent Neural Networks* (**GraphRNN**), a scalable framework for learning generative models of graphs. GraphRNN models a graph in an autoregressive (or recurrent) manner—as a sequence of additions of new nodes and edges—to capture the complex joint probability of all nodes and edges in the graph. In particular, GraphRNN can be viewed as a hierarchical model, where a *graph-level RNN* maintains the state of the graph and generates new nodes, while an *edge-level RNN* generates the edges for each newly generated node. Due to its autoregressive structure, GraphRNN can naturally accommodate variable-sized graphs, and we introduce a breadth-first-search (BFS) node-ordering scheme to drastically improve scalability. This BFS approach alleviates the fact that graphs have non-unique representations—by collapsing distinct representations to unique BFS trees—and the tree-structure induced by BFS allows us to limit the number of edge predictions made for each node during training. Our approach requires $O(n^2)$ operations on worst-case (*i.e.*, complete) graphs, but we prove that our BFS ordering scheme permits sub-quadratic complexity in many cases.

In addition to the novel GraphRNN framework, we also introduce a comprehensive suite of benchmark tasks and baselines for the graph generation problem, with all code made publicly available[1]. A key challenge for the graph generation problem is quantitative evaluation of the quality of generated graphs. Whereas prior studies have mainly relied on visual inspection or first-order moment statistics for evaluation, we provide a comprehensive evaluation setup by comparing graph statistics such as the degree distribution, clustering coefficient distribution and motif counts for two sets of graphs based on variants of the Maximum Mean Discrepancy (MMD) (Gretton et al., 2012). This quantitative evaluation approach can compare higher order moments of graph-statistic distributions and provides a more rigorous evaluation than simply comparing mean values.

Extensive experiments on synthetic and real-world graphs of varying size demonstrate the significant improvement GraphRNN provides over baseline approaches, including the most recent deep graph generative models as well as traditional models. Compared to traditional baselines (*e.g.*, stochastic block models), GraphRNN is able to generate high-quality graphs on all benchmark datasets, while the traditional models are only able to achieve good performance on specific datasets that exhibit special structures. Compared to other state-of-the-art deep graph generative models, GraphRNN is able to achieve superior quantitative performance—in terms of the MMD distance between the generated and test set graphs—while also scaling to graphs that are $50\times$ larger than what these previous approaches can handle. Overall, GraphRNN reduces MMD by 80%-90% over the baselines on average across all datasets and effectively generalizes, achieving comparatively high log-likelihood scores on held-out data.

2. Proposed Approach

We first describe the background and notation for building generative models of graphs, and then describe our autoregressive framework, GraphRNN.

2.1. Notations and Problem Definition

An undirected graph[2] $G = (V, E)$ is defined by its node set $V = \{v_1, ..., v_n\}$ and edge set $E = \{(v_i, v_j)|v_i, v_j \in V\}$. One common way to represent a graph is using an adjacency matrix, which requires a node ordering π that maps nodes to rows/columns of the adjacency matrix. More precisely, π is a permutation function over V (*i.e.*, $(\pi(v_1), ..., \pi(v_n))$ is a permutation of $(v_1, ..., v_n)$). We define Π as the set of all $n!$ possible node permutations. Under a node ordering π, a graph G can then be represented by the adjacency matrix $A^\pi \in \mathbb{R}^{n\times n}$, where $A^\pi_{i,j} = \mathbb{1}[(\pi(v_i), \pi(v_j)) \in E]$.

[1]The code is available in https://github.com/snap-stanford/GraphRNN, the appendix is available in https://arxiv.org/abs/1802.08773.

[2]We focus on undirected graphs. Extensions to directed graphs and graphs with features are discussed in the Appendix.

Note that elements in the set of adjacency matrices $A^{\Pi} = \{A^{\pi} | \pi \in \Pi\}$ all correspond to the same underlying graph.

The goal of *learning generative models of graphs* is to learn a distribution $p_{model}(G)$ over graphs, based on a set of observed graphs $\mathbb{G} = \{G_1, ..., G_s\}$ sampled from data distribution $p(G)$, where each graph G_i may have a different number of nodes and edges. When representing $G \in \mathbb{G}$, we further assume that we may observe any node ordering π with equal probability, *i.e.*, $p(\pi) = \frac{1}{n!}, \forall \pi \in \Pi$. Thus, the generative model needs to be capable of generating graphs where each graph could have exponentially many representations, which is distinct from previous generative models for images, text, and time series.

Finally, note that traditional graph generative models (surveyed in the introduction) usually assume a single input training graph. Our approach is more general and can be applied to a single as well as multiple input training graphs.

2.2. A Brief Survey of Possible Approaches

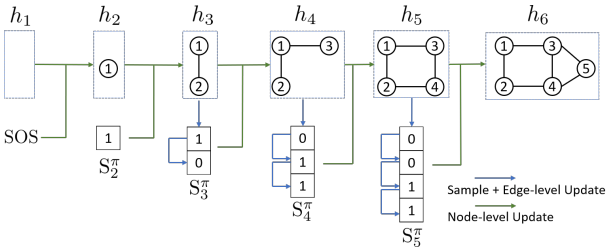
We start by surveying some general alternative approaches for modeling $p(G)$, in order to highlight the limitations of existing non-autoregressive approaches and motivate our proposed autoregressive architecture.

Vector-representation based models. One naïve approach would be to represent G by flattening A^{π} into a vector in $\mathbb{R}^{n^2}$, which is then used as input to any off-the-shelf generative model, such as a VAE or GAN. However, this approach suffers from serious drawbacks: it cannot naturally generalize to graphs of varying size, and requires training on all possible node permutations or specifying a canonical permutation, both of which require $O(n!)$ time in general.

Node-embedding based models. There have been recent successes in encoding a graph's structural properties into node embeddings (Hamilton et al., 2017), and one approach to graph generation could be to define a generative model that decodes edge probabilities based on pairwise relationships between learned node embeddings (as in Kipf & Welling 2016). However, this approach is only well-defined when given a fixed-set of nodes, limiting its utility for the general graph generation problem, and approaches based on this idea are limited to learning from a single input graph (Kipf & Welling, 2016; Grover et al., 2017).

2.3. GraphRNN: Deep Generative Models for Graphs

The key idea of our approach is to represent graphs under different node orderings as sequences, and then to build an autoregressive generative model on these sequences. As we will show, this approach does not suffer from the drawbacks common to other general approaches (*c.f.*, Section 2.2), allowing us to model graphs of varying size with complex



This makes sense because, Adjacency matrix for undirected graphs are symmetric.

Figure 1. GraphRNN at inference time. Green arrows denote the graph-level RNN that encodes the "graph state" vector h_i in its hidden state, updated by the predicted adjacency vector S_i^{π} for node $\pi(v_i)$. Blue arrows represent the edge-level RNN, whose hidden state is initialized by the graph-level RNN, that is used to predict the adjacency vector S_i^{π} for node $\pi(v_i)$.

edge dependencies, and we introduce a BFS node ordering scheme to drastically reduce the complexity of learning over all possible node sequences (Section 2.3.4). In this autoregressive framework, the model complexity is greatly reduced by weight sharing with recurrent neural networks (RNNs). Figure 1 illustrates our GraphRNN approach, where the main idea is that we decompose graph generation into a process that generates a sequence of nodes (via a *graph-level RNN*), and another process that then generates a sequence of edges for each newly added node (via an *edge-level RNN*).

2.3.1. Modeling Graphs as Sequences

We first define a mapping f_S from graphs to sequences, where for a graph $G \sim p(G)$ with n nodes under node ordering π, we have

$$S^{\pi} = f_S(G, \pi) = (S_1^{\pi}, ..., S_n^{\pi}), \tag{1}$$

where each element $S_i^{\pi} \in \{0,1\}^{i-1}, i \in \{1, ..., n\}$ is an adjacency vector representing the edges between node $\pi(v_i)$ and the previous nodes $\pi(v_j), j \in \{1, ..., i-1\}$ already in the graph:[3]

$$S_i^{\pi} = (A_{1,i}^{\pi}, ..., A_{i-1,i}^{\pi})^T, \forall i \in \{2, ..., n\}. \tag{2}$$

For undirected graphs, S^{π} determines a unique graph G, and we write the mapping as $f_G(\cdot)$ where $f_G(S^{\pi}) = G$.

Thus, instead of learning $p(G)$, whose sample space cannot be easily characterized, we sample the auxiliary π to get the observations of S^{π} and learn $p(S^{\pi})$, which can be modeled autoregressively due to the sequential nature of S^{π}. At inference time, we can sample G without explicitly computing $p(G)$ by sampling S^{π}, which maps to G via f_G.

Given the above definitions, we can write $p(G)$ as the marginal distribution of the joint distribution $p(G, S^{\pi})$:

$$p(G) = \sum_{S^{\pi}} p(S^{\pi}) \, \mathbb{1}[f_G(S^{\pi}) = G], \tag{3}$$

[3] We prohibit self-loops and S_1^{π} is defined as an empty vector.

Algorithm 1 GraphRNN inference algorithm

Input: RNN-based transition module f_{trans}, output module f_{out}, probability distribution $\mathcal{P}_{\theta_i}$ parameterized by θ_i, start token SOS, end token EOS, empty graph state h'
Output: Graph sequence S^π
$S_1^\pi = \text{SOS}$, $h_1 = h'$, $i = 1$
repeat
  $i = i + 1$
  $h_i = f_{\text{trans}}(h_{i-1}, S_{i-1}^\pi)$ {update graph state}
  $\theta_i = f_{\text{out}}(h_i)$
  $S_i^\pi \sim \mathcal{P}_{\theta_i}$ {sample node i's edge connections}
until S_i^π is EOS
Return $S^\pi = (S_1^\pi, ..., S_i^\pi)$

where $p(S^\pi)$ is the distribution that we want to learn using a generative model. Due to the sequential nature of S^π, we further decompose $p(S^\pi)$ as the product of conditional distributions over the elements:

$$p(S^\pi) = \prod_{i=1}^{n+1} p(S_i^\pi | S_1^\pi, ..., S_{i-1}^\pi) \tag{4}$$

where we set S_{n+1}^π as the end of sequence token EOS, to represent sequences with variable lengths. We simplify $p(S_i^\pi | S_1^\pi, ..., S_{i-1}^\pi)$ as $p(S_i^\pi | S_{<i}^\pi)$ in further discussions.

2.3.2. The GraphRNN framework

So far we have transformed the modeling of $p(G)$ to modeling $p(S^\pi)$, which we further decomposed into the product of conditional probabilities $p(S_i^\pi | S_{<i}^\pi)$. Note that $p(S_i^\pi | S_{<i}^\pi)$ is highly complex as it has to capture how node $\pi(v_i)$ links to previous nodes based on how previous nodes are interconnected among each other. Here we propose to parameterize $p(S_i^\pi | S_{<i}^\pi)$ using expressive neural networks to model the complex distribution. To achieve scalable modeling, we let the neural networks share weights across all time steps i. In particular, we use an RNN that consists of a *state-transition* function and an *output* function:

$$h_i = f_{\text{trans}}(h_{i-1}, S_{i-1}^\pi), \tag{5}$$
$$\theta_i = f_{\text{out}}(h_i), \tag{6}$$

where $h_i \in \mathbb{R}^d$ is a vector that encodes the state of the graph generated so far, S_{i-1}^π is the adjacency vector for the most recently generated node $i-1$, and θ_i specifies the distribution of next node's adjacency vector (*i.e.*, $S_i^\pi \sim \mathcal{P}_{\theta_i}$). In general, f_{trans} and f_{out} can be arbitrary neural networks, and $\mathcal{P}_{\theta_i}$ can be an arbitrary distribution over binary vectors. This general framework is summarized in Algorithm 1.

Note that the proposed problem formulation is fully general; we discuss and present some specific variants with implementation details in the next section. Note also that RNNs require fixed-size input vectors, while we previously defined S_i^π as having varying dimensions depending on i; we describe an efficient and flexible scheme to address this issue in Section 2.3.4.

Issue with varying dimension

2.3.3. GraphRNN variants

Different variants of the GraphRNN model correspond to different assumptions about $p(S_i^\pi | S_{<i}^\pi)$. Recall that each dimension of S_i^π is a binary value that models existence of an edge between the new node $\pi(v_i)$ and a previous node $\pi(v_j), j \in \{1, ..., i-1\}$. We propose two variants of GraphRNN, both of which implement the transition function f_{trans} (*i.e.*, the graph-level RNN) as a Gated Recurrent Unit (GRU) (Chung et al., 2014) but differ in the implementation of f_{out} (*i.e.*, the edge-level model). Both variants are trained using stochastic gradient descent with a maximum likelihood loss over S^π — *i.e.*, we optimize the parameters of the neural networks to optimize $\prod p_{model}(S^\pi)$ over all observed graph sequences.

Multivariate Bernoulli. First we present a simple baseline variant of our GraphRNN approach, which we term GraphRNN-S ("S" for "simplified"). In this variant, we model $p(S_i^\pi | S_{<i}^\pi)$ as a multivariate Bernoulli distribution, parameterized by the $\theta_i \in \mathbb{R}^{i-1}$ vector that is output by f_{out}. In particular, we implement f_{out} as single layer multilayer perceptron (MLP) with sigmoid activation function, that shares weights across all time steps. The output of f_{out} is a vector θ_i, whose element $\theta_i[j]$ can be interpreted as a probability of edge (i, j). We then sample edges in S_i^π *independently* according to a multivariate Bernoulli distribution parametrized by θ_i.

Dependent Bernoulli sequence. To fully capture complex edge dependencies, in the full GraphRNN model we further decompose $p(S_i^\pi | S_{<i}^\pi)$ into a product of conditionals,

$$p(S_i^\pi | S_{<i}^\pi) = \prod_{j=1}^{i-1} p(S_{i,j}^\pi | S_{i,<j}^\pi, S_{<i}^\pi), \tag{7}$$

where $S_{i,j}^\pi$ denotes a binary scalar that is 1 if node $\pi(v_{i+1})$ is connected to node $\pi(v_j)$ (under ordering π). In this variant, each distribution in the product is approximated by an another RNN. Conceptually, we have a hierarchical RNN, where the first (*i.e.*, the graph-level) RNN generates the nodes and maintains the state of the graph, while the second (*i.e.*, the edge-level) RNN generates the edges of a given node (as illustrated in Figure 1). In our implementation, the edge-level RNN is a GRU model, where the hidden state is initialized via the graph-level hidden state h_i and where the output at each step is mapped by a MLP to a scalar indicating the probability of having an edge. $S_{i,j}^\pi$ is sampled from this distribution specified by the jth output of the ith edge-level RNN, and is fed into the $j+1$th input of the same RNN. All edge-level RNNs share the same parameters.

2.3.4. TRACTABILITY VIA BREADTH-FIRST SEARCH

A crucial insight in our approach is that rather than learning to generate graphs under any possible node permutation, we learn to generate graphs using breadth-first-search (BFS) node orderings, without a loss of generality. Formally, we modify Equation (1) to

$$S^{\pi} = f_S(G, \text{BFS}(G, \pi)), \tag{8}$$

where BFS$(\cdot)$ denotes the deterministic BFS function. In particular, this BFS function takes a random permutation π as input, picks $\pi(v_1)$ as the starting node and appends the neighbors of a node into the BFS queue in the order defined by π. Note that the BFS function is many-to-one, *i.e.*, multiple permutations can map to the same ordering after applying the BFS function.

Using BFS to specify the node ordering during generation has two essential benefits. The first is that we only need to train on all possible BFS orderings, rather than all possible node permutations, *i.e.*, multiple node permutations map to the same BFS ordering, providing a reduction in the overall number of sequences we need to consider.[4] The second is that the BFS ordering makes learning easier by reducing the number of edge predictions we need to make in the edge-level RNN; in particular, when we are adding a new node under a BFS ordering, the only possible edges for this new node are those connecting to nodes that are in the "frontier" of the BFS (*i.e.*, nodes that are still in the BFS queue)—a notion formalized by Proposition 1 (proof in the Appendix):

Proposition 1. *Suppose $v_1, \ldots, v_n$ is a BFS ordering of n nodes in graph G, and $(v_i, v_{j-1}) \in E$ but $(v_i, v_j) \notin E$ for some $i < j \leq n$, then $(v_{i'}, v_{j'}) \notin E$, $\forall 1 \leq i' \leq i$ and $j \leq j' < n$.*

Importantly, this insight allows us to redefine the variable size S_i^{π} vector as a fixed M-dimensional vector, representing the connectivity between node $\pi(v_i)$ and nodes in the current BFS queue with maximum size M:

$$S_i^{\pi} = (A^{\pi}_{\max(1,i-M),i}, ..., A^{\pi}_{i-1,i})^T, i \in \{2, ..., n\}. \tag{9}$$

As a consequence of Proposition 1, we can bound M as follows:

Corollary 1. *With a BFS ordering the maximum number of entries that GraphRNN model needs to predict for S_i^{π}, $\forall 1 \leq i \leq n$ is $O\left(\max_{d=1}^{\text{diam}(G)} |\{v_i | \text{dist}(v_i, v_1) = d\}|\right)$, where* dist *denotes the shortest-path-distance between vertices.*

The overall time complexity of GraphRNN is thus $O(Mn)$. In practice, we estimate an empirical upper bound for M (see the Appendix for details).

[4] In the worst case (*e.g.*, star graphs), the number of BFS orderings is $n!$, but we observe substantial reductions on many real-world graphs.

3. GraphRNN Model Capacity

In this section we analyze the representational capacity of GraphRNN, illustrating how it is able to capture complex edge dependencies. In particular, we discuss two very different cases on how GraphRNN can learn to generate graphs with a *global community structure* as well as graphs with a very *regular geometric structure*. For simplicity, we assume that h_i (the hidden state of the graph-level RNN) can exactly encode $S^{\pi}_{<i}$, and that the edge-level RNN can encode $S^{\pi}_{i,<j}$. That is, we assume that our RNNs can maintain memory of the decisions they make and elucidate the models capacity in this ideal case. We similarly rely on the universal approximation theorem of neural networks (Hornik, 1991).

Graphs with community structure. GraphRNN can model structures that are specified by a given probabilistic model. This is because the posterior of a new edge probability can be expressed as a function of the outcomes of previous nodes. For instance, suppose that the training set contains graphs generated from the following distribution $p_{com}(G)$: half of the nodes are in community A, and half of the nodes are in community B (in expectation), and nodes are connected with probability p_s within each community and probability p_d between communities. Given such a model, we have the following key (inductive) observation:

Observation 1. *Assume there exists a parameter setting for GraphRNN such that it can generate $S^{\pi}_{<i}$ and $S^{\pi}_{i,<j}$ according to the distribution over S^{π} implied by $p_{com}(G)$, then there also exists a parameter setting for GraphRNN such that it can output $p(S^{\pi}_{i,j} | S^{\pi}_{i,<j}, S^{\pi}_{<i})$ according to $p_{com}(G)$.*

This observation follows from three facts: First, we know that $p(S^{\pi}_{i,j} | S^{\pi}_{i,<j}, S^{\pi}_{<i})$ can be expressed as a function of p_s, p_d, and $p(\pi(v_j) \in A), p(\pi(v_j) \in B)$ $\forall 1 \leq j \leq i$ (which holds by p_{com}'s definition). Second, by our earlier assumptions on the RNN memory, $S^{\pi}_{<i}$ can be encoded into the initial state of the edge-level RNN, and the edge-level RNN can also encode the outcomes of $S^{\pi}_{i,<j}$. Third, we know that $p(\pi(v_i) \in A)$ is computable from $S^{\pi}_{<i}$ and $S^{\pi}_{i,1}$ (by Bayes' rule and p_{com}'s definition, with an analogous result for $p(\pi(v_i) \in B)$). Finally, GraphRNN can handle the base case of the induction in Observation 1, *i.e.*, $S_{i,1}$, simply by sampling according to $0.5p_s + 0.5p_d$ at the first step of the edge-level RNN (*i.e.*, 0.5 probability i is in same community as node $\pi(v_1)$).

Graphs with regular structure. GraphRNN can also naturally learn to generate regular structures, due to its ability to learn functions that only activate for $S^{\pi}_{i,j}$ where v_j has specific degree. For example, suppose that the training set consists of ladder graphs (Noy & Ribó, 2004). To generate a ladder graph, the edge-level RNN must handle three key cases: if $\sum_{k=1}^{j} S^{\pi}_{i,j} = 0$, then the new node should only connect to the degree 1 node or else any degree 2 node; if

$\sum_{k=1}^{j} S^{\pi}_{i,j} = 1$, then the new node should only connect to the degree 2 node that is exactly two hops away; and finally, if $\sum_{k=1}^{j} S^{\pi}_{i,j} = 2$ then the new node should make no further connections. And note that all of the statistics needed above are computable from $S^{\pi}_{<i}$ and $S^{\pi}_{i,<j}$. The appendix contains visual illustrations and further discussions on this example.

4. Experiments

We compare GraphRNN to state-of-the-art baselines, demonstrating its robustness and ability to generate high-quality graphs in diverse settings.

4.1. Datasets

We perform experiments on both synthetic and real datasets, with drastically varying sizes and characteristics. The sizes of graphs vary from $|V| = 10$ to $|V| = 2025$.

Community. 500 two-community graphs with $60 \leq |V| \leq 160$. Each community is generated by the Erdős-Rényi model (E-R) (Erdős & Rényi, 1959) with $n = |V|/2$ nodes and $p = 0.3$. We then add $0.05|V|$ inter-community edges with uniform probability.

Grid. 100 standard 2D grid graphs with $100 \leq |V| \leq 400$. We also run our models on 100 standard 2D grid graphs with $1296 \leq |V| \leq 2025$, and achieve comparable results.

B-A. 500 graphs with $100 \leq |V| \leq 200$ that are generated using the Barabási-Albert model. During generation, each node is connected to 4 existing nodes.

Protein. 918 protein graphs (Dobson & Doig, 2003) with $100 \leq |V| \leq 500$. Each protein is represented by a graph, where nodes are amino acids and two nodes are connected if they are less than 6 Angstroms apart.

Ego. 757 3-hop ego networks extracted from the Citeseer network (Sen et al., 2008) with $50 \leq |V| \leq 399$. Nodes represent documents and edges represent citation relationships.

4.2. Experimental Setup

We compare the performance of our model against various traditional generative models for graphs, as well as some recent deep graph generative models.

Traditional baselines. Following Li et al. 2018 we compare against the Erdős-Rényi model (E-R) (Erdős & Rényi, 1959) and the Barabási-Albert (B-A) model (Albert & Barabási, 2002). In addition, we compare against popular generative models that include learnable parameters: Kronecker graph models (Leskovec et al., 2010) and mixed-membership stochastic block models (MMSB) (Airoldi et al., 2008).

Deep learning baselines. We compare against the recent methods of Simonovsky & Komodakis 2018 (GraphVAE) and Li et al. 2018 (DeepGMG). We provide reference implementations for these methods (which do not currently have associated public code), and we adapt GraphVAE to our problem setting by using one-hot indicator vectors as node features for the graph convolutional network encoder.[5]

Experiment settings. We use 80% of the graphs in each dataset for training and test on the rest. We set the hyperparameters for baseline methods based on recommendations made in their respective papers. The hyperparameter settings for GraphRNN were fixed after development tests on data that was not used in follow-up evaluations (further details in the Appendix). Note that all the traditional methods are only designed to learn from a single graph, therefore we train a separate model for each training graph in order to compare with these methods. In addition, both deep learning baselines suffer from aforementioned scalability issues, so we only compare to these baselines on a small version of the community dataset with $12 \leq |V| \leq 20$ (Community-small) and 200 ego graphs with $4 \leq |V| \leq 18$ (Ego-small).

4.3. Evaluating the Generated Graphs

Evaluating the sample quality of generative models is a challenging task in general (Theis et al., 2016), and in our case, this evaluation requires a comparison between two sets of graphs (the generated graphs and the test sets). Whereas previous works relied on qualitative visual inspection (Simonovsky & Komodakis, 2018) or simple comparisons of average statistics between the two sets (Leskovec et al., 2010), we propose novel evaluation metrics that compare all moments of their empirical distributions.

Our proposed metrics are based on Maximum Mean Discrepancy (MMD) measures. Suppose that a unit ball in a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ is used as its function class $\mathcal{F}$, and k is the associated kernel, the squared MMD between two sets of samples from distributions p and q can be derived as (Gretton et al., 2012)

$$\begin{aligned} \text{MMD}^2(p||q) = \mathbb{E}_{x,y\sim p}[k(x,y)] + \mathbb{E}_{x,y\sim q}[k(x,y)] \\ - 2\mathbb{E}_{x\sim p, y\sim q}[k(x,y)]. \end{aligned} \tag{10}$$

Proper distance metrics over graphs are in general computationally intractable (Lin, 1994). Thus, we compute MMD using a set of graph statistics $\mathbb{M} = \{M_1, ..., M_k\}$, where each $M_i(G)$ is a univariate distribution over $\mathbb{R}$, such as the degree distribution or clustering coefficient distribution. We then use the first Wasserstein distance as an efficient distance metric between two distributions p and q:

$$W(p,q) = \inf_{\gamma \in \Pi(p,q)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||], \tag{11}$$

where $\Pi(p,q)$ is the set of all distributions whose marginals

[5] We also attempted using degree and clustering coefficients as features for nodes, but did not achieve better performance.

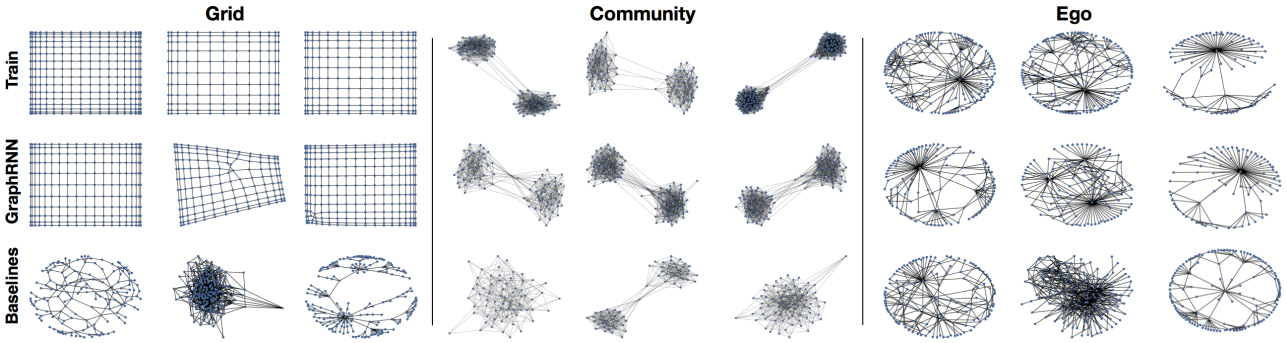


Figure 2. Visualization of graphs from grid dataset (Left group), community dataset (Middle group) and Ego dataset (Right group). Within each group, graphs from training set (First row), graphs generated by GraphRNN(Second row) and graphs generated by Kronecker, MMSB and B-A baselines respectively (Third row) are shown. Different visualization layouts are used for different datasets.

Table 1. Comparison of GraphRNN to traditional graph generative models using MMD. $(\max(|V|), \max(|E|))$ of each dataset is shown.

	Community (160,1945)			Ego (399,1071)			Grid (361,684)			Protein (500,1575)		
	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit
E-R	0.021	1.243	0.049	0.508	1.288	0.232	1.011	0.018	0.900	0.145	1.779	1.135
B-A	0.268	0.322	0.047	0.275	0.973	0.095	1.860	0	0.720	1.401	1.706	0.920
Kronecker	0.259	1.685	0.069	0.108	0.975	0.052	1.074	0.008	0.080	0.084	0.441	0.288
MMSB	0.166	1.59	0.054	0.304	0.245	0.048	1.881	0.131	1.239	0.236	0.495	0.775
GraphRNN-S	0.055	0.016	0.041	0.090	**0.006**	0.043	0.029	10^{-5}	0.011	0.057	**0.102**	**0.037**
GraphRNN	**0.014**	**0.002**	**0.039**	**0.077**	0.316	**0.030**	$\mathbf{10^{-5}}$	**0**	$\mathbf{10^{-4}}$	**0.034**	0.935	0.217

are p and q respectively, and γ is a valid transport plan. To capture high-order moments, we use the following kernel, whose Taylor expansion is a linear combination of all moments (proof in the Appendix):

Proposition 2. *The kernel function defined by* $k_W(p,q) = \exp \frac{W(p,q)}{2\sigma^2}$ *induces a unique RKHS.*

In experiments, we show this derived MMD score for degree and clustering coefficient distributions, as well as average orbit counts statistics, *i.e.*, the number of occurrences of all orbits with 4 nodes (to capture higher-level motifs) (Hočevar & Demšar, 2014). We use the RBF kernel to compute distances between count vectors.

4.4. Generating High Quality Graphs

Our experiments demonstrate that GraphRNN can generate graphs that match the characteristics of the ground truth graphs in a variety of metrics.

Graph visualization. Figure 2 visualizes the graphs generated by GraphRNN and various baselines, showing that GraphRNN can capture the structure of datasets with vastly differing characteristics—being able to effectively learn regular structures like grids as well as more natural structures like ego networks. Specifically, we found that grids generated by GraphRNN do not appear in the training set, *i.e.*, it learns to generalize to unseen grid widths/heights.

Evaluation with graph statistics. We use three graph statistics—based on degrees, clustering coefficients and orbit counts—to further quantitatively evaluate the generated graphs. Figure 3 shows the average graph statistics in the test vs. generated graphs, which demonstrates that even from hundreds of graphs with diverse sizes, GraphRNN can still learn to capture the underlying graph statistics very well, with the generated average statistics closely matching the overall test set distribution.

Tables 1 and 2 summarize MMD evaluations on the full datasets and small versions, respectively. Note that we train all the models with a fixed number of steps, and report the test set performance at the step with the lowest training error.[6] GraphRNN variants achieve the best performance on all datasets, with 80% decrease of MMD on average compared with traditional baselines, and 90% decrease of MMD compared with deep learning baselines. Interestingly, on the protein dataset, our simpler GraphRNN-S model performs very well, which is likely due to the fact that the protein dataset is a nearest neighbor graph over Euclidean space and thus does not involve highly complex edge dependencies. Note that even though some baseline models perform well on specific datasets (*e.g.*, MMSB on the community dataset), they fail to generalize across other types of input graphs.

[6]Using the training set or a validation set to evaluate MMD gave analogous results, so we used the train set for early stopping.

Table 2. GraphRNN compared to state-of-the-art deep graph generative models on small graph datasets using MMD and negative log-likelihood (NLL). $(\max(|V|), \max(|E|))$ of each dataset is shown. (DeepVAE and GraphVAE cannot scale to the graphs in Table 1.)

	Community-small (20,83)					Ego-small (18,69)				
	Degree	Clustering	Orbit	Train NLL	Test NLL	Degree	Clustering	Orbit	Train NLL	Test NLL
GraphVAE	0.35	0.98	0.54	13.55	25.48	0.13	0.17	0.05	12.45	14.28
DeepGMG	0.22	0.95	0.40	106.09	112.19	0.04	0.10	0.02	21.17	22.40
GraphRNN-S	**0.02**	0.15	**0.01**	31.24	35.94	0.002	**0.05**	**0.0009**	8.51	9.88
GraphRNN	0.03	**0.03**	**0.01**	28.95	35.10	**0.0003**	**0.05**	**0.0009**	9.05	10.61

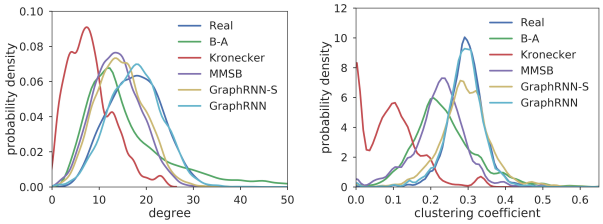


Figure 3. Average degree (Left) and clustering coefficient (Right) distributions of graphs from test set and graphs generated by GraphRNN and baseline models.

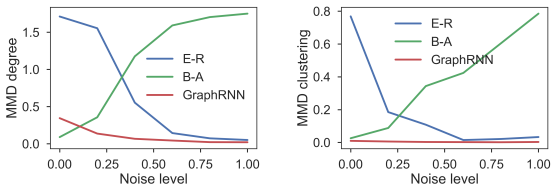


Figure 4. MMD performance of different approaches on degree (Left) and clustering coefficient (Right) under different noise level.

Generalization ability. Table 2 also shows negative log-likelihoods (NLLs) on the training and test sets. We report the average $p(S^\pi)$ in our model, and report the likelihood in baseline methods as defined in their papers. A model with good generalization ability should have small NLL gap between training and test graphs. We found that our model can generalize well, with 22% smaller average NLL gap.[7]

4.5. Robustness

Finally, we also investigate the robustness of our model by interpolating between Barabási-Albert (B-A) and Erdős-Rényi (E-R) graphs. We randomly perturb $[0\%, 20\%, ..., 100\%]$ edges of B-A graphs with 100 nodes. With 0% edges perturbed, the graphs are E-R graphs; with 100% edges perturbed, the graphs are B-A graphs. Figure 4 shows the MMD scores for degree and clustering coefficient distributions for the 6 sets of graphs. Both B-A and E-R perform well when graphs are generated from their respective distributions, but their performance degrades significantly once noise is introduced. In contrast, GraphRNN maintains strong performance as we interpolate between these structures, indicating high robustness and versatility.

5. Further Related Work

In addition to the deep graph generative approaches and traditional graph generation approaches surveyed previously, our framework also builds off a variety of other methods.

Molecule and parse-tree generation. There has been related domain-specific work on generating candidate molecules and parse trees in natural language processing. Most previous work on discovering molecule structures make use of a expert-crafted sequence representations of molecular graph structures (SMILES) (Olivecrona et al., 2017; Segler et al., 2017; Gómez-Bombarelli et al., 2016). Most recently, SD-VAE (Dai et al., 2018) introduced a grammar-based approach to generate structured data, including molecules and parse trees. In contrast to these works, we consider the fully general graph generation setting without assuming features or special structures of graphs.

Deep autoregressive models. Deep autoregressive models decompose joint probability distributions as a product of conditionals, a general idea that has achieved striking successes in the image (Oord et al., 2016b) and audio (Oord et al., 2016a) domains. Our approach extends these successes to the domain of generating graphs. Note that the DeepGMG algorithm (Li et al., 2018) and the related prior work of Johnson 2017 can also be viewed as deep autoregressive models of graphs. However, unlike these methods, we focus on providing a scalable (*i.e.*, $O(n^2)$) algorithm that can generate general graphs.

6. Conclusion and Future Work

We proposed GraphRNN, an autoregressive generative model for graph-structured data, along with a comprehensive evaluation suite for the graph generation problem, which we used to show that GraphRNN achieves significantly better performance compared to previous state-of-the-art models, while being scalable and robust to noise. However, significant challenges remain in this space, such as scaling to even larger graphs and developing models that are capable of doing efficient conditional graph generation.

[7] The average likelihood is ill-defined for the traditional models.

Acknowledgements

The authors thank Ethan Steinberg, Bowen Liu, Marinka Zitnik and Srijan Kumar for their helpful discussions and comments on the paper. This research has been supported in part by DARPA SIMPLEX, ARO MURI, Stanford Data Science Initiative, Huawei, JD, and Chan Zuckerberg Biohub. W.L.H. was also supported by the SAP Stanford Graduate Fellowship and an NSERC PGS-D grant.

References

Airoldi, E., Blei, D., Fienberg, S., and Xing, E. Mixed membership stochastic blockmodels. *JMLR*, 2008.

Albert, R. and Barabási, L. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002.

Cho, M., Sun, J., Duchenne, O., and Ponce, J. Finding matches in a haystack: A max-pooling strategy for graph matching in the presence of outliers. In *CVPR*, 2014.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *NIPS Workshop on Deep Learning*, 2014.

Dai, H., Tian, Y., Dai, B., Skiena, S., and Song, L. Syntax-directed variational autoencoder for structured data. *ICLR*, 2018.

Dobson, P. and Doig, A. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, 330(4):771–783, 2003.

Erdős, P. and Rényi, A. On random graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

Gómez-Bombarelli, R., Wei, J., Duvenaud, D., Hernández-Lobato, J. M., Sánchez-Lengeling, B., Sheberla, D., Aguilera-Iparraguirre, J., Hirzel, T. D., Adams, R. P., and Aspuru-Guzik, A. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 2016.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *NIPS*, 2014.

Gretton, A., Borgwardt, K. M., Rasch, M. J., Schölkopf, B., and Smola, A. A kernel two-sample test. *JMLR*, 2012.

Grover, A., Zweig, A., and Ermon, S. Graphite: Iterative generative modeling of graphs. In *NIPS Bayesian Deep Learning Workshop*, 2017.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, 2017.

Hočevar, T. and Demšar, J. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Johnson, D. D. Learning graphical state transitions. In *ICLR*, 2017.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. In *ICLR*, 2014.

Kipf, T. N. and Welling, M. Variational graph auto-encoders. In *NIPS Bayesian Deep Learning Workshop*, 2016.

Leskovec, J., Kleinberg, J., and Faloutsos, C. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1): 2, 2007.

Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., and Ghahramani, Z. Kronecker graphs: An approach to modeling networks. *JMRL*, 2010.

Li, Y., Vinyals, O., Dyer, C., Pascanu, R., and Battaglia, P. Learning deep generative models of graphs, 2018. URL https://openreview.net/forum?id=Hy1d-ebAb.

Lin, C. Hardness of approximating graph transformation problem. In *International Symposium on Algorithms and Computation*, 1994.

Newman, M. *Networks: an introduction*. Oxford university press, 2010.

Noy, M. and Ribó, A. Recursively constructible families of graphs. *Advances in Applied Mathematics*, 32(1-2): 350–363, 2004.

Olivecrona, M., Blaschke, T., Engkvist, O., and Chen, H. Molecular de novo design through deep reinforcement learning. *Journal of Cheminformatics*, 9(1):48, 2017.

Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016a.

Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. In *ICML*, 2016b.

Robins, G., Pattison, P., Kalish, Y., and Lusher, D. An introduction to exponential random graph (p*) models for social networks. *Social Networks*, 29(2):173–191, 2007.

Segler, M., Kogej, T., Tyrchan, C., and Waller, M. Generating focussed molecule libraries for drug discovery with recurrent neural networks. *ACS Central Science*, 2017.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI Magazine*, 29(3):93, 2008.

Simonovsky, M. and Komodakis, N. GraphVAE: Towards generation of small graphs using variational autoencoders, 2018. URL https://openreview.net/forum?id=SJlhPMWAW.

Theis, L., van den Oord, A., and Bethge, M. A note on the evaluation of generative models. In *ICLR*, 2016.

A. Appendix

A.1. Implementation Details of GraphRNN

In this section we detail parameter setting, data preparation and training strategies for GraphRNN.

We use two sets of model parameters for GraphRNN. One larger model is used to train and test on the larger datasets that are used to compare with traditional methods. One smaller model is used to train and test on datasets with nodes up to 20. This model is only used to compare with the two most recent preliminary deep generative models for graphs proposed in (Li et al., 2018; Simonovsky & Komodakis, 2018).

For GraphRNN, the graph-level RNN uses 4 layers of GRU cells, with 128 dimensional hidden state for the larger model, and 64 dimensional hidden state for the smaller model in each layer. The edge-level RNN uses 4 layers of GRU cells, with 16 dimensional hidden state for both the larger model and the smaller model. To output the adjacency vector prediction, the edge-level RNN first maps the highest layer of the 16 dimensional hidden state to a 8 dimensional vector through a MLP with ReLU activation, then another MLP maps the vector to a scalar with sigmoid activation. The edge-level RNN is initialized by the output of the graph-level RNN at the start of generating S_i^π, $\forall 1 \leq i \leq n$. Specifically, the highest layer hidden state of the graph-level RNN is used to initialize the lowest layer of edge-level RNN, with a liner layer to match the dimensionality. During training time, teacher forcing is used for both graph-level and edge-level RNNs, i.e., we use the groud truth rather than the model's own prediction during training. At inference time, the model uses its own preditions at each time steps to generate a graph.

For the simple version GraphRNN-S, a two-layer MLP with ReLU and sigmoid activations respectively is used to generate S_i^π, with 64 dimensional hidden state for the larger model, and 32 dimensional hidden state for the smaller model. In practice, we find that the performance of the model is relatively stable with respect to these hyperparameters.

We generate the graph sequences used for training the model following the procedure in Section 2.3.4. Specifically, we first randomly sample a graph from the training set, then randomly permute the node ordering of the graph. We then do the deterministic BFS discussed in Section 2.3.4 over the graph with random node ordering, resulting a graph with BFS node ordering. An exception is in the robustness section, where we use the node ordering that generates B-A graphs to get graph sequences, in order to see if GraphRNN can capture the underlying preferential attachment properties of B-A graphs.

With the proposed BFS node ordering, we can reduce the maximum dimension M of S_i^π, illustrated in Figure 5. To set the maximum dimension M of S_i^π, we use the following empirical procedure. We randomly ran 100000 times the above data pre-processing procedure to get graph with BFS node orderings. We remove the all consecutive zeros in all resulting S_i^π, to find the empirical distribution of the dimensionality of S_i^π. We set M to be roughly the 99.9 percentile, to account for the majority dimensionality of S_i^π. In principle, we find that graphs with regular structures tend to have smaller M, while random graphs or community graphs tend to have larger M. Specifically, for community dataset, we set $M = 100$; for grid dataset, we set $M = 40$; for B-A dataset, we set $M = 130$; for protein dataset, we set $M = 230$; for ego dataset, we set $M = 250$; for all small graph datasets, we set $M = 20$.

The Adam Optimizer is used for minibatch training. Each minibatch contains 32 graph sequences. We train the model for 96000 batchs in all experiments. We set the learning rate to be 0.001, which is decayed by 0.3 at step 12800 and 32000 in all experiments.

A.2. Running Time of GraphRNN

Training is performed on only 1 Titan X GPU. For the protein dataset that consists of about 1000 graphs, each containing about 500 nodes, training converges at around 64000 iterations. The runtime is around 12 to 24 hours. This also includes pre-processing, batching and BFS, which are currently implemented using CPU without multi-threading. The less expressive GraphRNN-S variant is about twice faster. At inference time, for the above dataset, generating a graph using the trained model only takes about 1 second.

A.3. More Details on GraphRNN's Expressiveness

We illustrate the intuition underlying the good performance of GraphRNNon graphs with regular structures, such as grid and ladder networks. Figure 6 (a) shows the generation process of a ladder graph at an intermediate step. At this time step, the ground truth data (under BFS node ordering) specifies that the new node added to the graph should make an edge to the node with degree 1. Note that node degree is

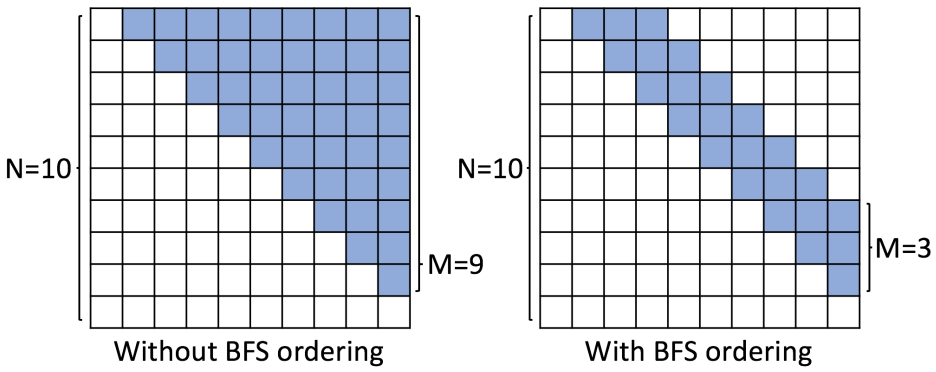


Figure 5. Illustrative example of reducing the maximum dimension M of S_i^π through the BFS node ordering. Here we show the adjacency matrix of a graph with $N = 10$ nodes. Without the BFS node ordering (Left), we have to set $M = N - 1$ to encode all the necessary connection information (shown in dark square). With the BFS node ordering, we could set M to be a constant smaller than N (we show $M = 3$ in the figure).

a function of $S^\pi_{<i}$, thus could be approximated by a neural network.

Once the first edge has been generated, the new node should make an edge with another node of degree 2. However, there are multiple ways to do so, but only one of them gives a valid grid structure, *i.e.* one that forms a 4-cycle with the new edge. GraphRNN crucially relies on the edge-level RNN and the knowledge of the previously added edge, in order to distinguish between the correct and incorrect connections in Figure 6 (c) and (d).

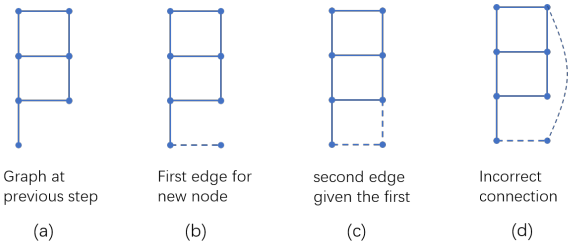


Figure 6. Illustration that generation of ladder networks relies on dependencies modeled by GraphRNN.

A.4. Code Overview

In the code repository, `main.py` consists of the main training pipeline, which loads datasets and performs training and inference. It also consists of the `Args` class, which stores the hyper-parameter settings of the model. `model.py` consists of the RNN, MLP and loss function modules that are use to build GraphRNN. `data.py` contains the minibatch sampler, which samples a random BFS ordering of a batch of randomly selected graphs. `evaluate.py` contains the code for evaluating the generated graphs using the MMD metric introduced in Sec. 4.3.

Baselines including the Erdős-Rényi model, Barabási-Albert model, MMSB, and rge very recent deep generative models (GraphVAE, DeepGMG) are also implemented in the `baselines` folders. We adopt the C++ Kronecker graph model implementation in the SNAP package [8].

A.5. Proofs

A.5.1. Proof of Proposition 1

We use the following observation:

Observation 2. *By definition of BFS, if $i < k$, then the children of v_i in the BFS ordering come before the children of v_k that do not connect to $v_{i'}$, $\forall 1 \leq i' \leq i$.*

By definition of BFS, all neighbors of a node v_i include the parent of v_i in the BFS tree, all children of v_i which have consecutive indices, and some children of $v_{i'}$ which connect to both $v_{i'}$ and v_i, for some $1 \leq i' \leq i$. Hence if $(v_i, v_{j-1}) \in E$ but $(v_i, v_j) \notin E$, v_{j-1} is the last children of v_i in the BFS ordering. Hence $(v_i, v_{j'}) \notin E, \forall j \leq j' \leq n$.

For all $i' \in [i]$, supposed that $(v_{i'}, v_{j'-1}) \in E$ but $(v_{i'}, v_{j'}) \notin E$. By Observation 2, $j' < j$. By conclusion in the previous paragraph, $(v_{i'}, v_{j''}) \notin E, \forall j' \leq j'' \leq n$. Specifically, $(v_{i'}, v_{j''}) \notin E, \forall j \leq j'' \leq n$. This is true for all $i' \in [i]$. Hence we prove that $(v_{i'}, v_{j'}) \notin E, \forall 1 \leq i' \leq i$ and $j \leq j' < n$.

A.5.2. Proof of Proposition 2

As proven in **?**, this Wasserstein distance based kernel is a positive definite (p.d.) kernel. By properties that linear combinations, product and limit (if exists) of p.d. kernels

[8] The SNAP package is available at `http://snap.stanford.edu/snap/index.html`.

are p.d. kernels, $k_W(p, q)$ is also a p.d. kernel.[9] By the Moore-Aronszajn theorem, a symmetric p.d. kernel induces a unique RKHS. Therefore Equation (9) holds if we set k to be k_W.

A.6. Extension to Graphs with Node and Edge Features

Our GraphRNN model can also be applied to graphs where nodes and edges have feature vectors associated with them. In this extended setting, under node ordering π, a graph G is associated with its node feature matrix $X^\pi \in \mathbb{R}^{n\times m}$ and edge feature matrix $F^\pi \in \mathbb{R}^{n\times k}$, where m and k are the feature dimensions for node and edge respectively. In this case, we can extend the definition of S^π to include feature vectors of corresponding nodes as well as edges $S_i^\pi = (X_i^\pi, F_i^\pi)$. We can adapt the f_{out} module, by using a MLP to generate X_i^π and an edge-level RNN to genearte F_i^π respectively. Note also that directed graphs can be viewed as an undirected graphs with two edge types, which is a special case under the above extension.

A.7. Extension to Graphs with Four Communities

To further show the ability of GraphRNN to learn from community graphs, we further conduct experiments on a four-community synthetic graph dataset. Specifically, the data set consists of 500 four community graphs with $48 \leq |V| \leq 68$. Each community is generated by the Erdős-Rényi model (E-R) (Erdős & Rényi, 1959) with $n \in [|V|/4 - 2, |V|/4+2]$ nodes and $p = 0.7$. We then add $0.01|V|^2$ inter-community edges with uniform probability. FIgure 7 shows the comparison of visualization of generated graph using GraphRNN and other baselines. We observe that in contrast to baselines, GraphRNN consistently generate 4-community graphs and each community has similar structure to that in the training set.

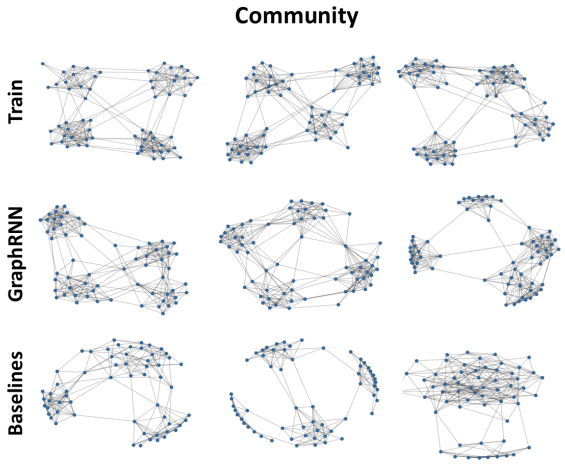


Figure 7. Visualization of graph dataset with four communities. Graphs from training set (First row), graphs generated by GraphRNN(Second row) and graphs generated by Kronecker, MMSB and B-A baselines respectively (Third row) are shown.

[9] This can be seen by expressing the kernel function using Taylor expansion.