Revisit the Scalability of Deep Auto-Regressive Models for Graph Generation

Shuai Yang
North Carolina State University
Raleigh, USA
syang16@ncsu.edu

Xipeng Shen
North Carolina State University
Raleigh, USA
xshen5@ncsu.edu

Seung-Hwan Lim
Oak Ridge National Laboratory
Oak Ridge, USA
lims1@ornl.gov

***Abstract*—As a new promising approach to graph generations, deep auto-regressive graph generation has drawn increasing attention. It however has been commonly deemed as hard to scale up to work with large graphs. In existing studies, it is perceived that the consideration of the full non-local graph dependences is indispensable for this approach to work, which entails the needs for keeping the entire graph's info in memory and hence the perceived "inherent" scalability limitation of the approach. This paper revisits the common perception. It proposes three ways to relax the dependences and conducts a series of empirical measurements. It concludes that the perceived "inherent" scalability limitation is a misperception; with the right design and implementation, deep auto-regressive graph generation can be applied to graphs much larger than the device memory. The rectified perception removes a fundamental barrier for this approach to meet practical needs.**

I. Introduction

Graphs are omnipresent in our lives and they are the natural representations of many phenomena such as Social networks, Road networks, Gene Regulatory networks and Bitcoin Transaction Graphs. Graph analysis for uncovering the underlying patterns and dynamism, as a fundamental research direction, has been attracting a surge of interests. Graph generative model is the basis of graph analysis and aims to generate synthetic graphs that match the properties of target graphs.

There are many applications of graph generative models. For example, a corporation would like to share its user network with its partner for development, but due to the information security and privacy, the real graph can not be shared; so a similar but synthetic graph would be a good replacement. Also, drug design needs to build a number of similar molecular graphs to test on; graph generative models can help to generate the candidates. Besides that, generative models have achieved excellent performance in many other tasks, such as: image generation [1], speech generation [2] and natural language generation [3]. Compared with those tasks, graph generative models are more challenging due to the essence of complex local and global dependencies among nodes and edges as well as various graph properties, e.g., degree distribution, clustering coefficient, occurrences of all orbits of graphlets [4].

The graph generative models have a long history. Dating back to 1959, Paul Erdős and Alfréd Rényi have proposed the first graph generative model Erdős–Rényi model [5] to generate random graphs. Beyond generating random graphs, later studies tried to model some properties of target graphs. Barabási–Albert model [6] aims to generate scale-free networks whose degree distribution follows a power law, which is very common in human-made networks such as: the internet, world wide web, social network and so on. Exponential Family Random Graph model [7] further extends the random graph model by postulating an exponential family over the space of graphs and incorporating graph features as sufficient statistics. Small-world network [8] delineates the small diameter in real-world graphs which grows slowly and proportionally to the logarithm of the number of nodes in the graph. Kronecker graph [9] iteratively applies Kronecker product to generate a sequence of graphs from a small base graph which meet the power law distribution of eigenvalues and eigenvectors of adjacency matrix.

All the above models are considered as traditional graph generative models since they all have a priori assumption about the structures of the target graphs. However, for complicated real-world graphs, studies [9], [10] have found that these assumptions may not accurately describe the underlying structural information or only consider part of properties. Therefore, traditional graph generative models only perform well on some particular family of graphs. To overcome this problem, Deep Neural Network-based models have been proposed. Instead of setting a priori assumption about the properties of target graphs, deep models learn the properties directly from a set of target graphs and are hence much more flexible than traditional models. Deep models can generate a much larger spectrum of graphs, achieving pronounced improvements on the fidelity of generated graphs [10], [11].

Most deep graph generative models can be categorized into three broad groups. The first category is Variational Autoencoders (VAE)-based [12], [13]. One paradigm of this category is GraphVAE [12] which firstly encodes training graphs into opaque vectors and then decodes those vectors back to graphs. As there is no natural order of nodes in graphs, VAE-based models usually involve an expensive graph matching procedure. The second category is based on Generative Adversarial Networks (GAN) [11], [14]. NetGAN [11] is a notable example in which the generator learns to generate sequences of random walks while discriminator tries to distinguish them from real random walks of training graphs. The graph is synthesized by generating a large number of random walks to get probabilities of edges and then do sampling of edges. GAN-based approaches do not encourage diversities in output and it could limit the variability of outputs [14].

To alleviate the limitations of the first two categories of methods, researchers have recently proposed a third approach, deep auto-regressive graph generative models [10], [15]–[18]. In this approach, new nodes and edges are conditioned on previously self-generated nodes and edges. GraphRNN [10] is a typical example. It consists of two levels of RNNs: graph level and edge level. GraphRNN proposes to represent graphs as sequences. New nodes and edges are generated sequentially through RNNs. Although the new approach shows new promise over prior methods, it has been commonly deemed as hard to scale up to work with large graphs. Experiments done in previous studies have used only small graphs (hundreds of nodes or even less) [10], [16], [18]. In a most recent study [15], the largest graph is up to 5,000 nodes, which is still substantially smaller than the scale of many real-world graphs. Multiple studies have stated the limitation [10], [16]

The reason for the scalability limitation stems from the following common perception:

> *The full non-local graph dependences are indispensable for deep auto regressive graph generative models. Since there are strong interactions among graph nodes and edges: the newly generated nodes are based on previously generated nodes and, on the other side, newly generated nodes affect the degrees and connectivities of previously generated nodes.*

The design of the approach, hence, tries to capture and reproduce the dependences of nodes and edges. As a result, all existing designs keep the entire graph info in memory for forward and backward propagation to function. On Graph Processing Units (GPU), the typical device these models use, memory is often limited to a number of GBs. In GraphRNN, for instance, long sequences of large entire graphs as well as corresponding large underlying computational graphs are all kept in the GPU memory, which confines its usage on limited size ($\leqslant$2000 nodes) graphs.

This paper revisits the commonly perceived scalability issues of deep auto-regressive graph generative models, trying to answer the following questions: (1) Is the common perception correct? Although the perception is intuitive at the first glance, its truthfulness has never been systematically examined. (2) Is it possible to relax the assumed constraint to both address the scalability issue and maintain enough quality of the generated graphs? (3) What are the possible ways to do that? How well do they each work empirically, in terms of result quality, execution speed, and scalability?

In this work, based on GraphRNN, we investigated the three questions by exploring three different ways to free auto-regressive graph generative models from the scalability limitation. (1) Split-Merge methods: large graphs are split into smaller subgraphs which are trained separately. The trained models then generate subgraphs which are finally merged into large graphs; (2) Merge-split method: nodes in large graphs are merged to reduce the graph size. The information about merging is kept on nodes; learning happens on the reduced graphs. The nodes in the generated graphs are finally split such that the graphs expand to the original size; and (3) Truncated-training method: large graphs correspond to long sequences which are cut into several segments. Backpropagation is performed after each segment and all the GPU memory occupied by the data and computational graph of the segment are then released. The gradients are accumulated and the model is updated once all segments are processed. The inference stage of truncated-training works in the same manner as the original model does.

Our experiments elucidate that all three methods can effectively reduce the GPU memory consumption compared with the original GraphRNN. Among them, the Merge-split method achieves the fastest training speed. The Split-merge method attains higher quality results than the Merge-split method but inferior to that of GraphRNN. The Truncated-training approach manifests comparable performance to GraphRNN while reducing the GPU memory usage by a magnitude. We further show that the Truncated-training approach maintains consistently good performance on large graphs (up to 16000 nodes) which exceed the capability of GraphRNN. And this is yet to be the upper limit. we demonstrate the truncated-training approach's potential to be applied on even larger graphs.

We organize the rest of this paper as follows: Section 2 briefly introduces key ideas in GraphRNN; Section 3 describes all the scalable methods in detail; Section 4 presents the experimental results and comparison with GraphRNN; the last section summarizes the work.

II. Background on GraphRNN

Given a set of observed graphs, deep graph generative models are supposed to generate synthetic graphs that match the properties of observed graphs. All the observed graphs are considered as samples from a common underlying distribution $p(G)$. The objective of deep graph generative models is to learn $p(G)$ over observed graphs. Due to non-unique orders of nodes in a graph, the same graph with different orders lead to different adjacency matrices. Let's denote the order as π and S^π as the adjacency information under the order π. S^π can be expanded as $(S_1^\pi, S_2^\pi, ..., S_n^\pi)$ where n is the number of nodes in the graph and S_i^π is the adjacency vector of node i. S_i^π represents the connection between nodes i to all its previous nodes $j, (j < i)$. Therefore, $p(G)$ can be written as:

$$p(G) = \sum_{\pi \in \Pi} p(S^\pi) \tag{1}$$

where Π is the set of all orders of nodes in the graph G. Formula 1 is satisfied only when all possible orders of nodes are enumerated, which requires $O(n!)$ time and is unaffordable for large graphs. In practice, models usually approximate $p(G)$ through sampling a large number of different orders. Owing to the high dependences between different nodes, $p(S^\pi)$ is further decomposed as a sequence of conditional probabilities:

$$\begin{aligned} p(S^\pi) &= p(S_1^\pi, S_2^\pi, ..., S_n^\pi) \\ &= p(S_1^\pi) \cdot p(S_2^\pi | S_1^\pi) \cdot ... \cdot p(S_n^\pi | S_1^\pi, S_2^\pi, ..., S_{n-1}^\pi) \\ &= \prod_{i=1}^{n} p(S_i^\pi | S_{<i}^\pi) \end{aligned} \tag{2}$$

where $p(S_i^\pi|S_{<i}^\pi) = p(S_i^\pi|S_1^\pi, S_2^\pi, ..., S_{i-1}^\pi)$. Combining Formula 1 and 2, modeling of $p(G)$ has been converted to modeling of $p(S_i^\pi|S_{<i}^\pi)$. Considering that edges from one node are interdependent, $p(S_i^\pi|S_{<i}^\pi)$ can be expressed as:

$$\begin{aligned} p(S_i^\pi|S_{<i}^\pi) &= p(S_{i,1}^\pi, S_{i,2}^\pi, ..., S_{i,i-1}^\pi | S_{<i}^\pi) \\ &= p(S_{i,1}^\pi|S_{<i}^\pi) \cdot ... \cdot p(S_{i,i-1}^\pi | S_{i,1}^\pi, ..., S_{i,i-2}^\pi, S_{<i}^\pi) \\ &= \prod_{j=1}^{i-1} p(S_{i,j}^\pi | S_{i,<j}^\pi, S_{<i}^\pi) \end{aligned} \tag{3}$$

where $p(S_{i,j}^\pi|S_{i,<j}^\pi, S_{<i}^\pi) = p(S_{i,j}^\pi|S_{i,1}^\pi, S_{i,2}^\pi, ..., S_{i,j-1}^\pi, S_{<i}^\pi)$. In GraphRNN, it parameterizes $p(S_i^\pi|S_{<i}^\pi)$ with a state-transition function f_{trans} and an output function f_{out}:

$$h_i = f_{trans}(h_{i-1}, S_i^\pi) \tag{4}$$

$$S_{i+1}^\pi = f_{out}(h_i) \tag{5}$$

where h_i is the hidden state of the subgraph with the first i nodes. Both f_{trans} and f_{out} are implemented by Gated Recurrent Units(GRU), a popular gating mechanism in RNN. f_{trans} and f_{out} correspond to the graph-level RNN and edge-level RNN, respectively. As shown in Formula 4 and 5, the graph-level RNN takes the hidden state of the subgraph with first $i-1$ nodes h_{i-1} and adjacency vector of the ith node to generate the hidden state of the subgraph with the first i nodes. The edge-level RNN is initialized with h_i and outputs S_{i+1}^π by generating the edges from the $i+1$th node to all previous nodes (node 1 to i) sequentially, which reflects $p(S_{i,j}^\pi|S_{i,<j}^\pi, S_{<i}^\pi)$.

One example of the whole training process of GraphRNN is presented in Figure 1. After generating all adjacent vector S_i^π's, loss calculations are between S^π (target) and $S^{\pi\prime}$ (prediction). In the reference stage, as shown in Figure 2, $S^{\pi\prime}$ takes place of S^π. Each edge is generated along with a probability and final graphs are sampled according to the probabilities.

III. Scalable Models

To make deep auto-regressive graph generative models scalable, we want to reduce the sizes of training graphs. There are three widely used strategies to achieve this goal: (1) split-merge method, (2) merge-split method, (3) truncate-training method. We have explored the feasibility and performance of all three methods on the deep auto-regressive graph generative model. In this section, we introduce how we apply each of the strategies on graphRNN to enhance its scalability.

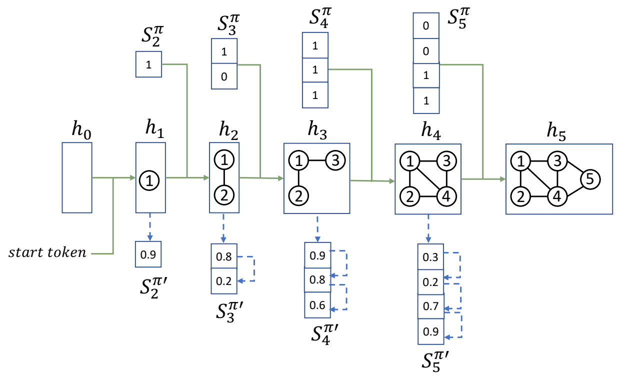

Fig. 1. GraphRNN training stage. The graphs in the middle boxes are corresponding graphs of h_i. The green solid line corresponds to the graph-level RNN which takes h_{i-1} and S_i^π to generate h_i while the blue dot line represents the edge-level RNN which is based on h_i and predicts probabilities of edges to previous nodes sequentially.

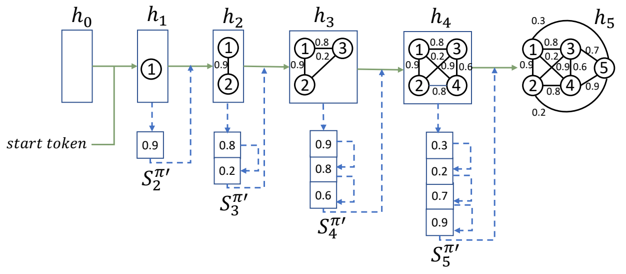

Fig. 2. GraphRNN inference stage.

A. Split-merge method

One natural thought about dealing with large graphs which can not be processed directly is to split them into small ones. This is also known as graph partitioning. The crux of graph partitioning is to balance the sizes of the split graphs. However, balanced graph partitioning is an NP-hard problem [19], so, in practice, many approximate approaches are proposed [20], [21]. In the setting of graph generative models, we first split large graphs using Fennel [20], one of the most widely used graph partitioning algorithms and train subgraphs separately using graphRNN. At the inference stage, we collect generated subgraphs from each trained graphRNN model and merge them together.

There exists an obvious problem that when a large graph is split into several subgraphs, the edges between different subgraphs are cut which are not included in the subsequent training process. So, when we merge generated subgraphs, we do not have information about how subgraphs are connected. One naive way to deal with this issue is to randomly sample some nodes from different subgraphs and connect them. This would work when the number of inter-subgraph edges are way less than that of intra-subgraph edges. Even with the most recent graph partitioning approaches, dramatic increase of the ratio of inter-subgraph edges to all edges have been observed when the number of subgraphs increases (the ratio exceeds 50% with 8 or more subgraphs) [20], which makes the naive approach not applicable.

Therefore, to avoid the non-negligible "loss" of inter-subgraph edges, we propose SnapButton, a new Split-merge strategy. The gist of SnapButton is to create an overlapping zone between two subgraphs. Two overlapping nodes from different subgraphs will be like the two discs of a snap button that helps us to align the two generated graphs during the merging stage.

The splitting process of SnapButton is as shown in Figure 3. As the GraphRNN applies the Breadth-first Search(BFS) order of nodes, our work maintains this setting. With the BFS order, one straightforward idea is to split nodes according to their positions in that order. First half of nodes form the first subgraph and the second half of nodes form the second one. But what are the overlapping nodes? During the process of BFS, we maintain a queue. Every time we pick one node from the head of the queue and push all its unvisited neighbor nodes to the tail of the queue. The nodes in the queue are visited but have not been processed (adding neighbor nodes). They form the frontier of current BFS as the green nodes in Figure 3(b). One appealing property of BFS is that nodes inside the frontier (processed nodes) are never connected with nodes outside the frontier (unvisited nodes). Because all processed nodes' neighbor nodes are all visited and appended to the queue. Therefore, we choose this frontier as the overlapping zone since it's the boundary of two parts. Now the graph can be divided into two subgraphs: the induced graphs of processed nodes + overlapping nodes and unvisited nodes + overlapping nodes. It's worth noting that GraphRNN requires the connected graphs. Given a connected graph, after the splitting process, whereas the first subgraph is guaranteed to be connected, the second subgraph could be disconnected. We hence add edges between all pairs of consecutive nodes in the overlapping zone as shown in Figure 3(d).

Through the splitting process, training graphs are divided into two sets of subgraphs, which are fed into GraphRNN separately. The merging stage is simple. After we collected two sets of generated subgraphs from two GraphRNN models, we firstly remove all edges between consecutive nodes in the overlapping zone in the second subgraphs. But what if there are some edges between consecutive overlapping nodes originally existing? We don't need to worry about that, since if one edge between two consecutive overlapping nodes exists in the original graph, it must be included in the first subgraph as the edge (8,9) in Figure 3(c). After merging, it will present in the final merged graph. Let's assume the size of the overlapping zone is n_o. Then we just need to merge the overlapping nodes (the last n_o nodes from the first subgraph and the first n_o nodes from the second subgraph) from two subgraphs to get the complete generated graph. Note that for simplicity, we have described the case with two subgraphs, but, in fact, Snapbutton can recursively work on subgraphs to further reduce the graph scale.

B. Merge-split method

In contrast with Split-Merge methods, Merge-split methods reduce the scale of original graphs through merging neighbor nodes into single nodes. This course is called graph coarsening which has been used for creating multi-scale representations of graphs [22]–[24]. One of the most widely used strategies for graph coarsening is to find a maximal matching of the original graph and merge two end nodes of edges in the maximal matching.

Definition 1: A matching of a graph $G(N,E)$ is a subset E_m of E such that no two edges in E_m share an endpoint. N and E are the node set and edge set of G.

Definition 2: A maximal matching of a graph $G(N,E)$ is a matching E_m to which no more edges can be added and remain a matching.

Figure 4(a) gives an example of maximal matching. While we get a smaller graph by merging two end nodes of edges in the maximal matching, coarsening entails information loss. We want the process to be lossless and reversible so that coarsened graphs can be restored to original scale. Therefore, we maintain an extra term for each node, which we call the *signature*.

Definition 3: The signature of a node is a vector that records the merging history of that node. The format of signature is:

$$s_i^t=\begin{cases}(s^{t-1}_{i,[0:n_i^{t-1}-1]},s^{t-1}_{j,[0:n_j^{t-1}-1]},p_i,p_e,\displaystyle\sum_{k\in\{i,j,e\}}(w_k^{t-1})),t\geqslant 1\\ (w_i^0),t=0\end{cases} \tag{6}$$

where s_i^t is the signature of node i after t^{th} coarsening and $[0:n_i^{t-1}-1]$ represents indices from 0 to $n_i^{t-1}-2$. n_i^{t-1} and n_j^{t-1} are the lengths of s_i^{t-1} and s_j^{t-1}. $p_i=w_i^{t-1}/(w_i^{t-1}+w_j^{t-1})$, $p_e=w_e^{t-1}/(w_i^{t-1}+w_j^{t-1}+w_e^{t-1})$. w_i^{t-1},w_j^{t-1} and w_e^{t-1} are the weight of node i, node j and edge e(the edge between node i and node j) after $t-1^{th}$ coarsening. Weights of nodes are initialized as their degrees and weights of edges are initialized as 1.

s_i^t is a vector. The last element of the vector is the merged node's weight which is compose of the weights of its two end nodes and the weight of edge between them; p_e shows the proportion of weight contribution from the edge; p_i is the proportion of end nodes' weight contributed by the head node(inside a pair the one with smaller vector value are appointed as head node, if vector values are equal, the one with smaller index is the head node). $s^{t-1}_{i,[0:n_i^{t-1}-1]}$ and $s^{t-1}_{j,[0:n_j^{t-1}-1]}$ are the previous merging history of node i and j. Since node j is merged into node i in the t^{th} coarsening, so we combine $s^{t-1}_{i,[0:n_i^{t-1}-1]}$ and $s^{t-1}_{j,[0:n_j^{t-1}-1]}$ and add the record of the t^{th} coarsening (the last three elements) to get the new signature. One special case is that the node is not merged with others in the t^{th} iteration, we hence replaces $s^{t-1}_{j,[0:n_j^{t-1}-1]}$ with all 0s and w_j^{t-1}, w_e^{t-1} are also 0.

When refining, given the signature of a merged node, it's easy to recover the signatures of two end nodes merged into

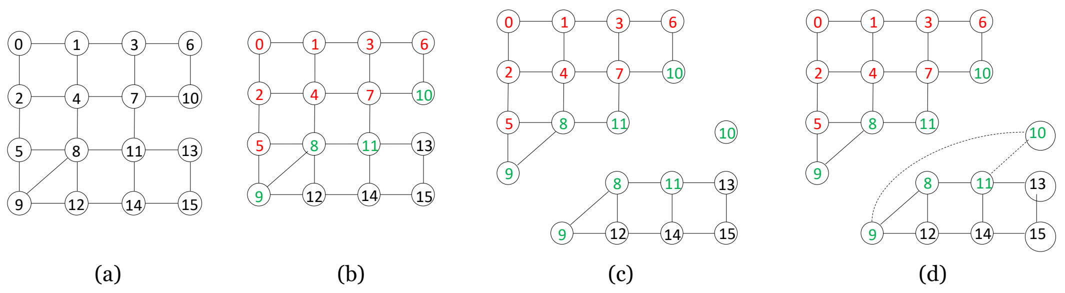
Fig. 3. Example of SnapButton splitting process: (a)The first step is to get the BFS order of nodes. The number in each node shows its position in the BFS order. (b) Red nodes are the first half of nodes; Green nodes are the overlapping nodes; Black nodes are the remaining ones. (c) The first subgraph is composed of red nodes and overlapping nodes while the second subgraph is composed of overlapping nodes and black nodes. (d) If not connected, add edges between all pairs of consecutive overlapping nodes (edges in dot line) in the second subgraph to ensure the graph is connected.

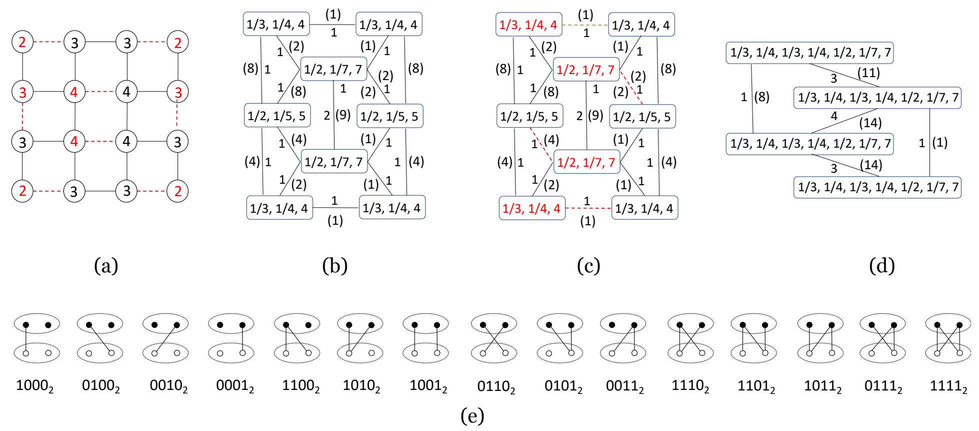
Fig. 4. Example of 2-level graph coarsening: (a) We find a maximal matching of the graph (red dot edges are in maximal matching; red nodes are the head nodes. (b) One-level coarsened graph, numbers on the nodes are signatures while numbers on the edges are weights and edge types (in parenthesis). (c) Repeat the process to find the maximal matching. (d) Two-level coarsened graph. (e) All types of connections that are merged into single edges

it:

$$s_i^{t-1}=(s_{i,[0:(n_i^t-3)/2]}^t, w_i^t*(1-p_e)*p_i)$$
$$s_j^{t-1}=(s_{i,[(n_i^t-3)/2:n_i^t-3]}^t, w_i^t*(1-p_e)*(1-p_i)), (p_e\neq0) \quad (7)$$

If $p_e = 0$, then s_i^t is directly generated by s_i^{t-1}, so we do not need to recover s_j^{t-1}. All nodes in the coarsened graph have its merging history in their signatures while a single edge in the coarsen graph could correspond to 15 different cases in the previous graph as shown in 4(e). The ovals represent merged nodes in the coarsen graph and small dots inside ovals are the corresponding nodes in the previous graph. The numbers under them are their type codes which are shown in binary numbers where each digit corresponds to one edge. Due to the strong dependences of edges, we train another standard GRU model which takes the previous 8 edges as input to predict the type of the current edge.

The Figure 4 shows an example of doing two-level coarsening. Finally, the signatures of nodes are concatenated to their adjacency vectors to feed into graphRNN. The refining of generated graphs is straightforward: we split nodes according to Formula 7 and use trained RNN to predict the edge type so as to split edges.

C. Truncate-training method

The previous two methods boost the scalability from the perspective of graphs. Our third method looks into the problem from the perspective of models. GraphRNN has a hierarchical RNN structure which iteratively feeds adjacency vectors into the recurrent unit. If we unfold the RNN model, recurrent units can be considered as layers of a standard feed-forward network. With a long sequence of data, the number of layers increases accordingly. All the internal states are stored, which entails the intensive GPU memory usage. For some typical applications with RNN, like: time-series prediction and machine translation, Truncated Backpropagation Through Time(TBPTT) has been used to circumvent these difficulties [25]–[27].

The process of TBPTT can be described as every k_1-step forward propagation is followed by a k_2-step backpropagation. The most common setting is $k_1 = k_2$. Therefore, A long sequence is divided into several subsequences with length k_1. Every time when RNN models have processed a subsequence, we do backpropagation for the subsequence. But we do not update the model immediately, instead the gradients are accumulated and the update will be done only when the whole sequence is processed. The last hidden state of

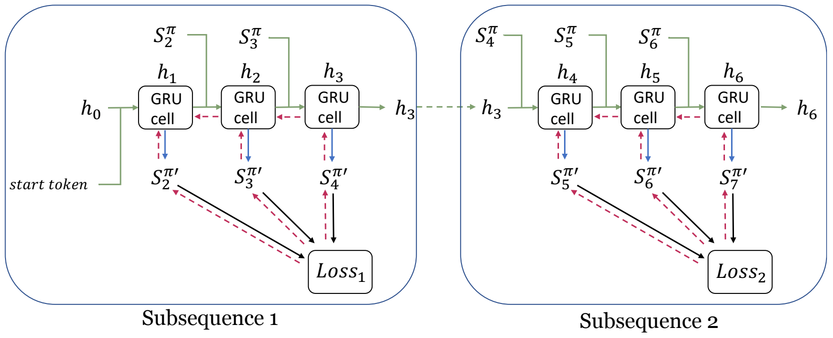

Fig. 5. Truncate-training: The sequence of adjacency vectors is divided into fix-length subsequences. Backpropagation (red dot line) is performed after each subsequence and the last hidden state h_3 is passed on to the next subsequence as initial state. Gradients are accumulated by each subsequence and the model is updated when all sequences are processed.

the subsequence is passed on as the initial state to the next subsequence. Then the RNN works on the next subsequence. The forward propagation of TBPTT is equivalent to continuous forward propagation on the whole sequence. The difference is in the backpropagation. The gradients do not flow back from the next subsequence to the previous one. That's why it is called truncated backpropagation through time.

Due to the truncated mechanism, the GPU memory saving of TBPTT is pronounced because all the internal variables and the computational graph are only maintained for the current subsequence and can be immediately released after backpropagation. In theory, TBPTT focuses primarily on intra-subsequence dependencies rather than inter-subsequence dependencies due to this truncation. However, since all subsequences contribute to the gradients of parameters of the RNN model, the loss of latter subsequences still affect the output of prior ones implicitly. In practice, TBPTT has shown promising performance in different fields [28]–[30]. However, due to the common perception that there are strong interactions among graph nodes and edges, all previous deep auto-regressive graph generative models adopt full graph dependences which store the entire sequence in the GPU memory. Is this really necessary? To answer this question, we propose our truncate-training method which integrates the idea of TBPTT with GraphRNN as shown Figure 5.

IV. Experiments

We implement the methods on GraphRNN, the representative open-source framework for deep auto-regressive graph generation. We conduct a series of experiments on both synthetic and real-world graph datasets. The graphs are chosen to be diverse in both properties and the intended usage:(1) Grid: standard 2D grid graphs which are the most widely used graphs in all experiments of generative models; (2) p2p-Ego: ego networks extracted from Gnutella peer-to-peer network [31]. Nodes represent hosts in the Gnutella network and edges represent connections between Gnutella hosts; (3) DD: protein structure dataset [32] where each protein is represented as a graph and each node is an amino acid. Two nodes are connected if they are less than 5 Angstroms apart. In our experiments, we compare different methods in terms of result quality, execution speed, and scalability: (1) Quality: we adopt maximum mean discrepancy (MMD) score as the evaluation metric over distributions of degrees, clustering coefficient, and the number of occurrence of all orbits with 4 nodes. The MMD score has served as the de facto standard for quality evaluation of graph generative models [10], [15], [16]; (2) Speed: we report the training time of every epoch and the inference takes a small portion of time when compared with training; (3) Scalability: we measure the maximum GPU memory allocation during the whole training process as the indicator of the scalability. All the experiments are conducted in the following experimental environment: CPU:Xeon E5-2630 Main memory:128GB, GPU:Titan Xp 12GB, OS:Ubuntu 16.04.

A. *Quality Evaluation*

We have tested the quality of different methods on two sets of graphs with different scales. The small ones are chosen to be the scales that can be easily handled by graphRNN, while the mid ones are the scales that graphRNN can process but with intensive GPU memory usage. In our experiments, GraphRNN applies its original settings besides the batch size is set to 8. As for scalable methods, we adopt the setting of 4 subsequences for Truncate-training, one-level graph partition for Split-merge method and one-level coarsening for Merge-split method. The results are shown in Table I. The smaller MMD score indicates higher fidelity of generated graphs and the best results are highlighted in bold. Taken together, GraphRNN shows the best performance. The truncate-training method has achieved comparable results on small-scale graphs and close performance on mid-scale graphs to GraphRNN, which infers that cross-subsequence long dependences, which truncate-training does not directly capture, only marginally affects the performance. This observation meets our expectation that one node is mostly determined by its local structure (intra-subsequence dependences). Split-merge method and Merge-split method have a performance gap to the previous two methods. To our understanding, these two methods all use GraphRNN as one of their components, so their performance is subject to the performance of GraphRNN while extra steps for enhancing scalability may introduce extra quality loss. The

TABLE I
COMPARISON OF MMD SCORES OF DIFFERENT METHODS ON SMALL AND MEDIUM SCALE GRAPHS. (MAX($|V|$), MAX($|E|$), NUMBER OF GRAPHS) OF EACH DATASET IS SHOWN.

	Ego-small (200,241,25)			Grid-small (800, 1540, 16)			DD-small (298,978, 25)			Ego-mid (1193, 2673, 25)			Grid-mid (2400,4700, 16)			DD-mid (779, 2600, 14)		
	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit
GraphRNN	**0.198**	**0.006**	**0.572**	0.109	**$2.2e^{-4}$**	0.099	0.149	1.725	1.568	**0.525**	0.021	**1.078**	**0.122**	$1.21e^{-5}$	**0.056**	**0.161**	**1.805**	0.808
Trunc. train.	0.261	0.064	1.491	**0.039**	$2.65e^{-4}$	**0.027**	**0.085**	1.584	**0.535**	0.762	0.022	1.25	0.153	$6.81e^{-5}$	0.057	0.315	1.815	**0.23**
Split-merge	0.939	0.158	1.243	0.257	0.063	0.063	0.419	1.497	1.474	1.798	**0.012**	1.25	0.318	0.086	0.122	0.541	1.897	1.646
Merge-split	0.713	0.147	0.776	0.973	1.963	0.161	1.392	**1.261**	0.87	1.945	1.999	1.25	1.066	**0**	0.647	1.712	1.904	0.946

design of SnapButton is lossless only when the two subgraphs follow the same order. However, in practice, since we train on a set of graphs with sampled BFS orders, the learned "average" orders from two sets of subgraphs may not be matched very well which introduces extra quality loss. The Merge-split method is lossless for node splitting according to signatures while edge splitting is based on another GRU model which also brings extra quality loss.

B. *Efficiency and GPU memory footprint*

Besides the quality, we measured the efficiency and GPU memory footprint of all the methods on different datasets. As shown in Table II, Truncate-training consistently outperforms all other methods on all datasets and gains up to 11X (on avg. 6.2X) GPU memory savings. The Split-merge method and the Merge-split method achieves 2.14X and 1.63X savings, respectively. Theoretically, we can achieve more memory savings by recursively applying Split-merge and Merge-split method, or truncating the whole graph sequence into more subsequences for truncate-training.

As for the efficiency, the difference is small, if we use the speed of GraphRNN as the baseline, then the speed of Truncate-training, Split-merge and Merge-split are 0.93X, 0.91X and 1.3X, respectively. The Merge-split method is the fastest attributing to the fact that it has only one small coarsened graph to train. The other two methods' speed is close to that of graphRNN. Taken with the evaluations collected in our experiments, the Truncate-training method shows comparable quality and speed to GraphRNN but with significant memory savings. Therefore, we further explore its performance on large graphs which exceeds the capacity of GraphRNN.

C. *Scalability*

Scalability is the main issue this paper targets on. Table III reports the results on four datasets of large graphs. We have adapted the number of subsequences so that large graphs can fit into the GPU memory. For Ego-large I and II, we set the number of subsequences to be 8 while for Grid-large I and II, it's chosen to be 2 and 4. Since these graphs exceed the scale that can be processed by GraphRNN, we can not directly compare the results with GraphRNN's. However, we still get insights about the performance by comparing them with results of small-scale and mid-scale graphs. Larger scale lowers the quality of generated graphs due to more complicated structures and dependences. This trend has been observed in all methods based on previous experimental results. In experiments on these large graphs, whereas this trend exists, the quality reduction is slow compared with the quick increase of scales. This indicates the consistent good performance of truncate-training on large graphs. Beyond that, Grid-large II (16000, 31720, 16) is much larger than the scales that have been used in previous studies [10], [15]–[18], we further investigated the potential of the Truncate-training method in our experiment setting and found that Truncate-training can process up to 30000 nodes (Grid (30000, 59650, 16)). After that, bottleneck becomes the main memory rather than GPU memory. This problem can be mitigated by avoiding to load all training graphs into the main memory and process graphs in a streaming style, but this is not the focus of this paper. In theory, by adapting the number of subsequences, the Truncate-training method can fit into any fixed memory budget.

V. CONCLUSION

This paper revisits the scalability issue of deep auto-regressive graph generative models. Common perception that the full graph dependences must be always maintained for high quality graph generation entails prohibitive GPU memory usage. We take GraphRNN as a representative of auto-regressive graph generative models. From three different aspects, we propose three methods to relax the constraints on dependences. Experiments show that all methods enhance scalability. Among all three methods, Truncate-training method attains comparable high-quality results and similar training speed to GraphRNN while at the same time reducing memory usage by up to 11X. Furthermore, it has been applied to much larger graphs (one order of magnitude larger than the scale GraphRNN can process) and achieve consistent good performance. This finding breaks that long-time perception and confirms that full graph dependences are not indispensable for graph generation. Without this constraint, deep auto-regressive graph generative models can tap into their full potentials on much larger graphs and benefit more real-world applications.

Acknowledgement This material is based upon work supported by the National Science Foundation (NSF) under Grants CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

TABLE II
COMPARISON OF RUNNING TIME (SECOND) OF ONE EPOCH AND MEMORY FOOTPRINT (MB) OF DIFFERENT METHODS

	Ego-small		Ego-mid		Grid-small		Grid-mid		DD-small		DD-mid	
	Mem.	Time	Mem	Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem	Time
GraphRNN	293.8	2.1	9623.9	36.12	438.3	3.12	2192.9	10.62	347.1	**1.68**	1413.6	**4.44**
Trunc. train.	**36.6**	2.46	**874.8**	18.54	**112.8**	3.96	**541.2**	12.18	**67.1**	3.12	**265.6**	7.86
Split-merge	160.5	3.72	3628.4	49.62	222.0	**2.76**	1004.2	**7.14**	176.0	2.64	634.7	4.74
Merge-split	328.6	**1.5**	5972.7	**12.66**	235.1	2.88	1142.3	10.14	235.1	3.0	690.1	4.62

TABLE III
PERFORMANCE OF TRUNCATE-TRAINING ON LARGE GRAPHS, (MAX($|V|$), MAX($|E|$), NUMBER OF GRAPHS) OF EACH DATASET IS SHOWN.

	Deg.	Clus.	Orbit	Mem(MB).	Time(s)
Ego-large I (5499, 23205, 23)	0.458	0.003	1.234	4708.2	182.2
Ego-large II (6981, 30891, 24)	1.067	0.003	1.479	7832.0	333.6
Grid-large I (8000, 15760, 16)	0.121	$5.0e^{-4}$	0.04	3614.8	70.1
Grid-large II (16000, 31720, 16)	0.161	$4.38e^{-4}$	0.064	6611.3	279.3

REFERENCES

[1] D. Zou and G. Lerman, “Encoding robust representation for graph generation,” in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–9.

[2] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “Wavenet: A generative model for raw audio,” *arXiv preprint arXiv:1609.03499*, 2016.

[3] J. Tang, Y. Yang, S. Carton, M. Zhang, and Q. Mei, “Context-aware natural language generation with recurrent neural networks,” *arXiv preprint arXiv:1611.09900*, 2016.

[4] I. Melckenbeeck, P. Audenaert, D. Colle, and M. Pickavet, “Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations,” *Bioinformatics*, vol. 34, no. 8, pp. 1372–1380, 2018.

[5] P. Erdős and A. Rényi, “On the evolution of random graphs,” *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[6] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[7] P. W. Holland and S. Leinhardt, “An exponential family of probability distributions for directed graphs,” *Journal of the american Statistical association*, vol. 76, no. 373, pp. 33–50, 1981.

[8] D. J. Watts and S. H. Strogatz, “Collective dynamics of ‘small-world’networks,” *nature*, vol. 393, no. 6684, p. 440, 1998.

[9] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, “Kronecker graphs: An approach to modeling networks,” *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 985–1042, 2010.

[10] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, “Graphrnn: Generating realistic graphs with deep auto-regressive models,” *arXiv preprint arXiv:1802.08773*, 2018.

[11] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, “Netgan: Generating graphs via random walks,” *arXiv preprint arXiv:1803.00816*, 2018.

[12] M. Simonovsky and N. Komodakis, “Graphvae: Towards generation of small graphs using variational autoencoders,” in *International Conference on Artificial Neural Networks*. Springer, 2018, pp. 412–422.

[13] Q. Liu, M. Allamanis, M. Brockschmidt, and A. Gaunt, “Constrained graph variational autoencoders for molecule design,” in *Advances in neural information processing systems*, 2018, pp. 7795–7804.

[14] N. De Cao and T. Kipf, “Molgan: An implicit generative model for small molecular graphs,” *arXiv preprint arXiv:1805.11973*, 2018.

[15] R. Liao, Y. Li, Y. Song, S. Wang, W. Hamilton, D. K. Duvenaud, R. Urtasun, and R. Zemel, “Efficient graph generation with graph recurrent attention networks,” in *Advances in Neural Information Processing Systems*, 2019, pp. 4257–4267.

[16] W. Kawai, Y. Mukuta, and T. Harada, “Gram: Scalable generative models for graphs with graph attention mechanism,” *arXiv preprint arXiv:1906.01861*, 2019.

[17] D. D. Johnson, “Learning graphical state transitions,” 2016.

[18] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, “Learning deep generative models of graphs,” *arXiv preprint arXiv:1803.03324*, 2018.

[19] M. R. Garey, D. S. Johnson, and L. Stockmeyer, “Some simplified np-complete problems,” in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 47–63.

[20] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, “Fennel: Streaming graph partitioning for massive scale graphs,” in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 333–342.

[21] G. Karypis and V. Kumar, “Parallel multilevel graph partitioning,” in *Proceedings of International Conference on Parallel Processing*. IEEE, 1996, pp. 314–319.

[22] S. Lafon and A. B. Lee, “Diffusion maps and coarse-graining: A unified framework for dimensionality reduction, graph partitioning, and data set parameterization,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 9, pp. 1393–1403, 2006.

[23] D. I. Shuman, M. J. Faraji, and P. Vandergheynst, “A multiscale pyramid transform for graph signals,” *IEEE Transactions on Signal Processing*, vol. 64, no. 8, pp. 2119–2134, 2015.

[24] I. Safro, P. Sanders, and C. Schulz, “Advanced coarsening schemes for graph partitioning,” *Journal of Experimental Algorithmics (JEA)*, vol. 19, pp. 1–24, 2015.

[25] I. Sutskever, *Training recurrent neural networks*. University of Toronto Toronto, Ontario, Canada, 2013.

[26] J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[27] R. J. Williams and J. Peng, “An efficient gradient-based algorithm for on-line training of recurrent network trajectories,” *Neural computation*, vol. 2, no. 4, pp. 490–501, 1990.

[28] J. Henriques, P. Gil, A. Dourado, and H. Duarte-Ramos, “Application of a recurrent neural network in on-line modeling of real-time systems,” *Informatics engineering Department UC p'oloII*, vol. 3030, 2004.

[29] A. Shaban, C. Cheng, N. Hatch, and B. Boots, “Truncated back-propagation for bilevel optimization,” *arXiv:1810.10667*, 2018.

[30] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, “Recurrent neural network based language model,” in *Eleventh annual conference of the international speech communication association*, 2010.

[31] M. Ripeanu, I. Foster, and A. Iamnitchi, “Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design,” *arXiv preprint cs/0209028*, 2002.

[32] P. D. Dobson and A. J. Doig, “Distinguishing enzyme structures from non-enzymes without alignments,” *Journal of molecular biology*, vol. 330, no. 4, pp. 771–783, 2003.