Efficient Graph Generation with Graph Recurrent Attention Networks

In **NeurIPS 2019**

Seonguk Seo

2018-20721

Seoul National University

Existing Work

- **Graph RNN** [You18]
  - It models graph generation as a sequential process, which accommodate complex dependencies between generated edges.
  - $O(N^2)$ for the best model (not scalable).
  - It has significant bottlenecks in handling long-term dependencies, and the results depend on node orderings.

[You18] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. **Graphrnn: Generating realistic graphs with deep auto-regressive models.** In ICML 2018

Contributions

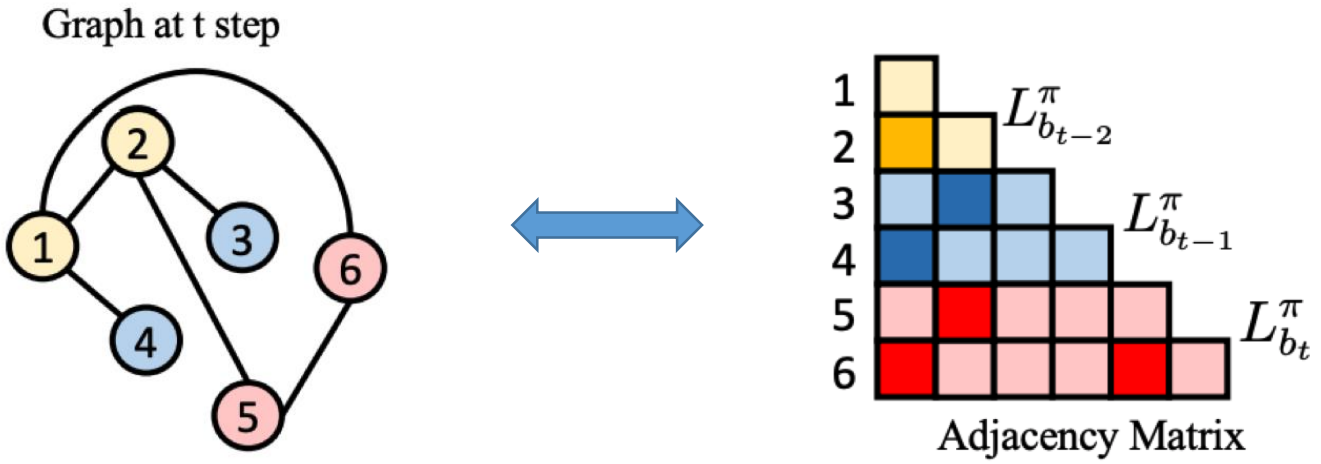
- Their model consists of **O(N)** auto-regressive generation steps.
- Compared to RNN-based model, they propose an attention-based GNN that better utilizes the topology of the already generated graphs.
- They approximate the likelihood by marginalizing over a family of canonical node orderings.

Representation of Graphs

- Model the distribution of undirected graph $G = (V, E)$:

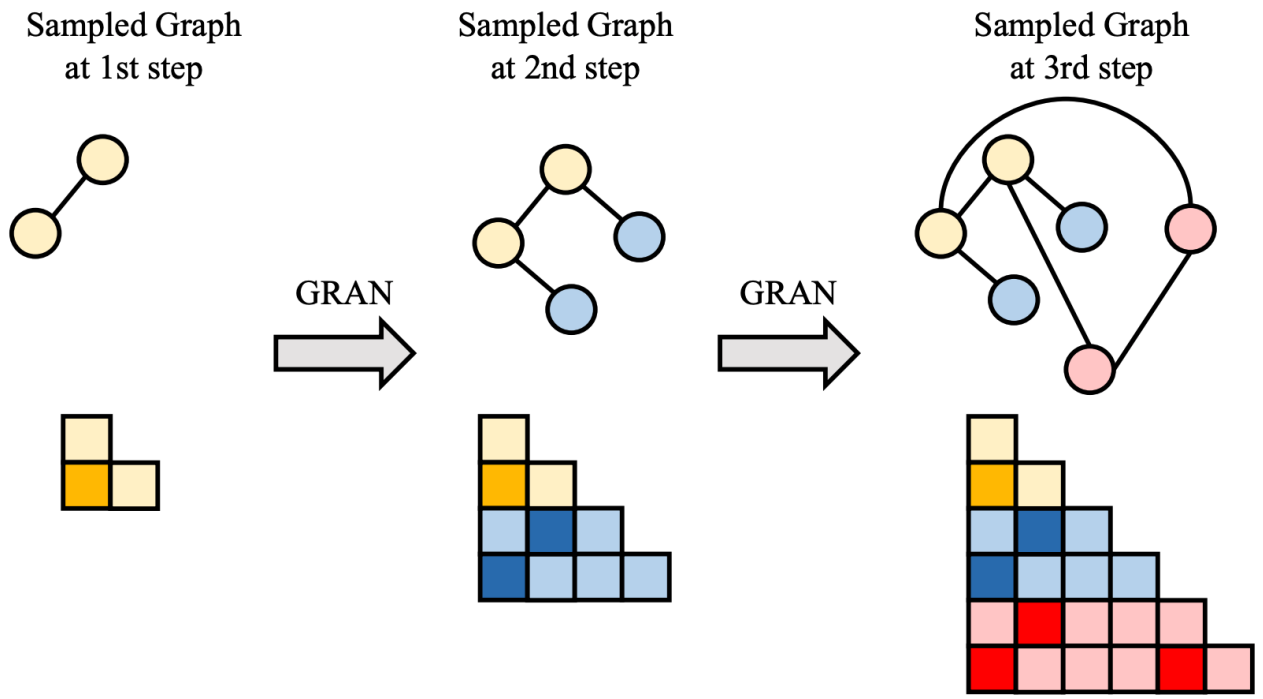
$$P(G) = \sum_{\pi} P(G, \pi) = \sum_{\pi} P(A_{\pi}) = \sum_{\pi} P(L_{\pi})$$

where π denotes a node ordering, A_{π} is an adjacency matrix and L_{π} denotes a lower triangular part of A_{π}.



Representation of Generation Process

- They generate one block of B rows of L_π for each time step in one pass conditioned on the already generated graphs.
- By increasing the block size and the stride of generation, they can trade-off model expressiveness for speed.



Graph Recurrent Attention Networks (GRAN)

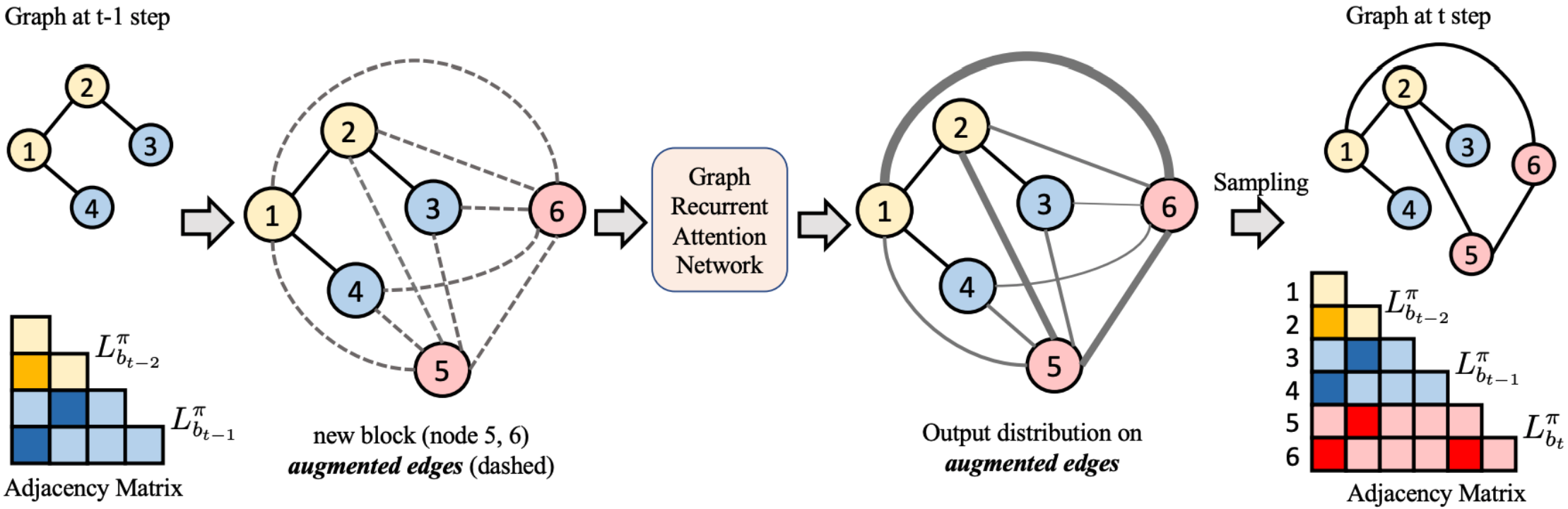


Figure 1: Overview of our model. Dashed lines are *augmented edges*. Nodes with the same color belong to the same block (block size = 2). In the middle right, for simplicity, we visualize the output

Graph Recurrent Attention Networks (GRAN)

$$p(L^{\pi}) = \prod_{t=1}^{T} p(L^{\pi}_{\boldsymbol{b_t}} | L^{\pi}_{\boldsymbol{b_1}}, \cdots, L^{\pi}_{\boldsymbol{b_{t-1}}}).$$

- RNNs are standard neural networks for handling this sequential dependency structures.
  - It may cause long-term bottleneck on graph topology.
  - Thus they use GNNs to make generation decisions depend on the graph structure.

- They do not carry hidden states of GNNs from one generation step to the next, which enables parallel training.
  - This is more efficient than typical auto-regressive models.

Graph Recurrent Attention Networks (GRAN)

- Initial Node Representation

$$h_{\boldsymbol{b_i}}^0 = W L_{\boldsymbol{b_i}}^{\pi} + b, \qquad \forall i < t.$$

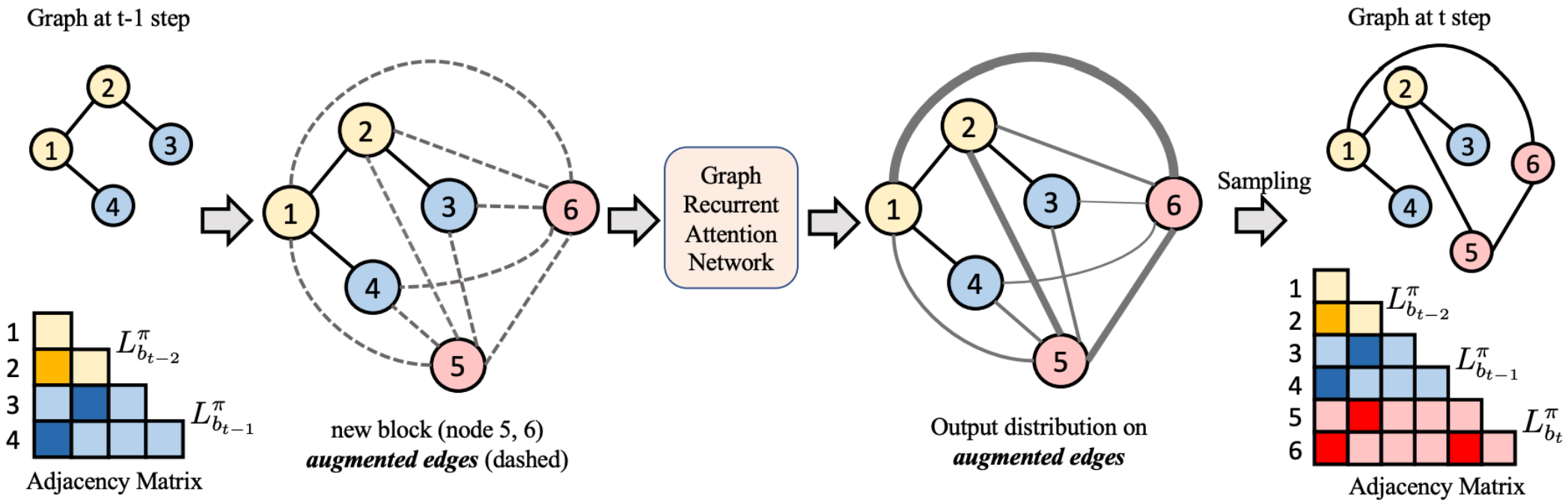
- Message Passing

$$m_{ij}^r = f(h_i^r - h_j^r), \qquad a_{ij}^r = \text{Sigmoid}\left(g(\tilde{h}_i^r - \tilde{h}_j^r)\right),$$
$$\tilde{h}_i^r = [h_i^r, x_i], \qquad h_i^{r+1} = \text{GRU}(h_i^r, \sum\nolimits_{j \in \mathcal{N}(i)} a_{ij}^r m_{ij}^r).$$

- Output Distribution

$$p(L_{\boldsymbol{b_t}}^{\pi} | L_{\boldsymbol{b_1}}^{\pi}, ..., L_{\boldsymbol{b_{t-1}}}^{\pi}) = \sum_{k=1}^{K} \alpha_k \prod_{i \in \boldsymbol{b_t}} \prod_{1 \le j \le i} \theta_{k,i,j},$$
$$\alpha_1, \ldots, \alpha_K = \text{Softmax}\left(\sum\nolimits_{i \in \boldsymbol{b_t}, 1 \le j \le i} \text{MLP}_{\alpha}(h_i^R - h_j^R)\right),$$
$$\theta_{1,i,j}, \ldots, \theta_{K,i,j} = \text{Sigmoid}\left(\text{MLP}_{\theta}(h_i^R - h_j^R)\right)$$

Graph Recurrent Attention Networks (GRAN)



Node Representation

- They first compute the initial node representations of the already-generated graph via a linear mapping

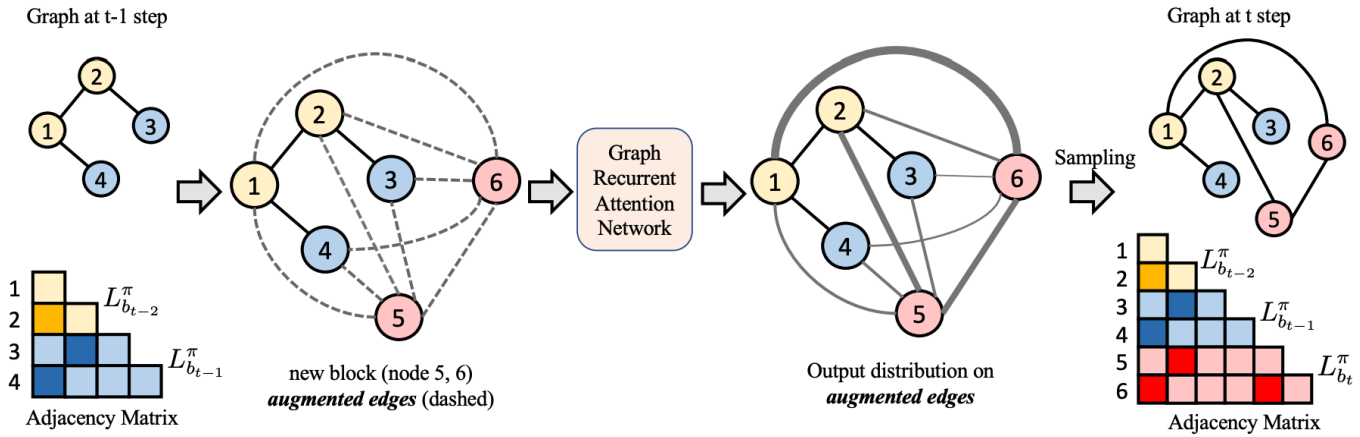
$$h^0_{\boldsymbol{b_i}} = W L^{\pi}_{\boldsymbol{b_i}} + b, \qquad \forall i < t.$$

- For current block, $h^0_{\boldsymbol{b_t}} = \mathbf{0}$
- In practice, computing $h^0_{\boldsymbol{b_{t-1}}}$ alone at t[th] generation step is enough, because $\{h^0_{\boldsymbol{b_i}} | i < t-1\}$ can be cached from previous steps, which reduces computation.

Graph Neural Network with Attentive Message Passing

- From the initial node representation, all edges associated with the current block are generated using a GNN.
- The r[th] round of message passing is

$$m_{ij}^r = f(h_i^r - h_j^r), \qquad a_{ij}^r = \text{Sigmoid}\left(g(\tilde{h}_i^r - \tilde{h}_j^r)\right),$$
$$\tilde{h}_i^r = [h_i^r, x_i], \qquad h_i^{r+1} = \text{GRU}(h_i^r, \sum_{j \in \mathcal{N}(i)} a_{ij}^r m_{ij}^r).$$



Output Distribution

- After R round message passing, they obtain the final node representation vector, and then model the probability of generating edges in the current block via a mixture of Bernoulli distributions.

$$p(L^{\pi}_{\boldsymbol{b_t}} | L^{\pi}_{\boldsymbol{b_1}}, ..., L^{\pi}_{\boldsymbol{b_{t-1}}}) = \sum_{k=1}^{K} \alpha_k \prod_{i \in \boldsymbol{b_t}} \prod_{1 \le j \le i} \theta_{k,i,j},$$

$$\alpha_1, \ldots, \alpha_K = \text{Softmax}\left(\sum\nolimits_{i \in \boldsymbol{b_t}, 1 \le j \le i} \text{MLP}_{\alpha}(h_i^R - h_j^R) \right),$$

$$\theta_{1,i,j}, \ldots, \theta_{K,i,j} = \text{Sigmoid}\left(\text{MLP}_{\theta}(h_i^R - h_j^R) \right)$$

Approximated Likelihood

$$P(G) = \sum_{\pi} P(G, \pi) = \sum_{\pi} P(A_\pi) = \sum_{\pi} P(L_\pi)$$

- They aim to maximize a lower bound:

$$\log p(G) \geq \log \sum_{\pi \in \mathcal{Q}} p(G, \pi)$$

  where the size Q achieve a tradeoff between tightness of the bound (usually correlated with better model quality) and computational cost.

- Variational Interpretation

$$\log p(G) \geq \mathbb{E}_{q(\pi|G)}[\log p(G, \pi)] + \mathcal{H}(q(\pi|G)), \quad q^*(\pi|G) = p(G, \pi) / \left(\sum_{\pi \in \mathcal{Q}} p(G, \pi) \right)$$

- They adopts DFS, BFS, k-core and degree descent ordering.

Experiments

- Dataset
  - Grid : 100 standard 2D grid graphs with $100 < |V| < 400$
  - Protein : 918 protein graphs with $100 < |V| < 500$
  - Point Cloud : 41 simulated 3D point clouds with $|V|_{avg} >= 1k$
- 64% train split, 16% valid split, 20% test split

Evaluation Metrics

- Compare the distributions of graph statistics between the generated and ground-truth graphs, by computing the maximum mean discrepancy (**MMD**) over the following 4 statistics.

1. Degree distributions
2. Clustering coefficient distributions
3. The number of occurrences of all orbits with 4 nodes
4. Spectra of the graphs (the eigenvalues of normalized graph Laplacian)

Results

- For all metrics, lower value is preferred.

	Grid				Protein				3D Point Cloud			
	$\lvert V\rvert_{\max} = 361$, $\lvert E\rvert_{\max} = 684$ $\lvert V\rvert_{\text{avg}} \approx 210$, $\lvert E\rvert_{\text{avg}} \approx 392$				$\lvert V\rvert_{\max} = 500$, $\lvert E\rvert_{\max} = 1575$ $\lvert V\rvert_{\text{avg}} \approx 258$, $\lvert E\rvert_{\text{avg}} \approx 646$				$\lvert V\rvert_{\max} = 5037$, $\lvert E\rvert_{\max} = 10886$ $\lvert V\rvert_{\text{avg}} \approx 1377$, $\lvert E\rvert_{\text{avg}} \approx 3074$			
	Deg.	Clus.	Orbit	Spec.	Deg.	Clus.	Orbit	Spec.	Deg.	Clus.	Orbit	Spec.
Erdos-Renyi	0.79	2.00	1.08	0.68	$5.64e^{-2}$	1.00	1.54	$9.13e^{-2}$	0.31	1.22	1.27	$4.26e^{-2}$
GraphVAE*	$7.07e^{-2}$	$7.33e^{-2}$	0.12	$1.44e^{-2}$	0.48	$7.14e^{-2}$	0.74	0.11	-	-	-	-
GraphRNN-S	0.13	$3.73e^{-2}$	0.18	0.19	$4.02e^{-2}$	$\mathbf{4.79e^{-2}}$	0.23	0.21	-	-	-	-
GraphRNN	$1.12e^{-2}$	$\mathbf{7.73e^{-5}}$	$\mathbf{1.03e^{-3}}$	$\mathbf{1.18e^{-2}}$	$1.06e^{-2}$	0.14	0.88	$1.88e^{-2}$	-	-	-	-
GRAN	$\mathbf{8.23e^{-4}}$	$3.79e^{-3}$	$1.59e^{-3}$	$1.62e^{-2}$	$\mathbf{1.98e^{-3}}$	$4.86e^{-2}$	**0.13**	$\mathbf{5.13e^{-3}}$	$\mathbf{1.75e^{-2}}$	**0.51**	**0.21**	$\mathbf{7.45e^{-3}}$

Table 1: Comparison with other graph generative models. For all MMD metrics, the smaller the better. *: our own implementation, -: not applicable due to memory issue, Deg.: degree distribution, Clus.: clustering coefficients, Orbit: the number of 4-node orbits, Spec.: spectrum of graph Laplacian.

Visualization

- Sample graphs generated by GRAN with its comparison.

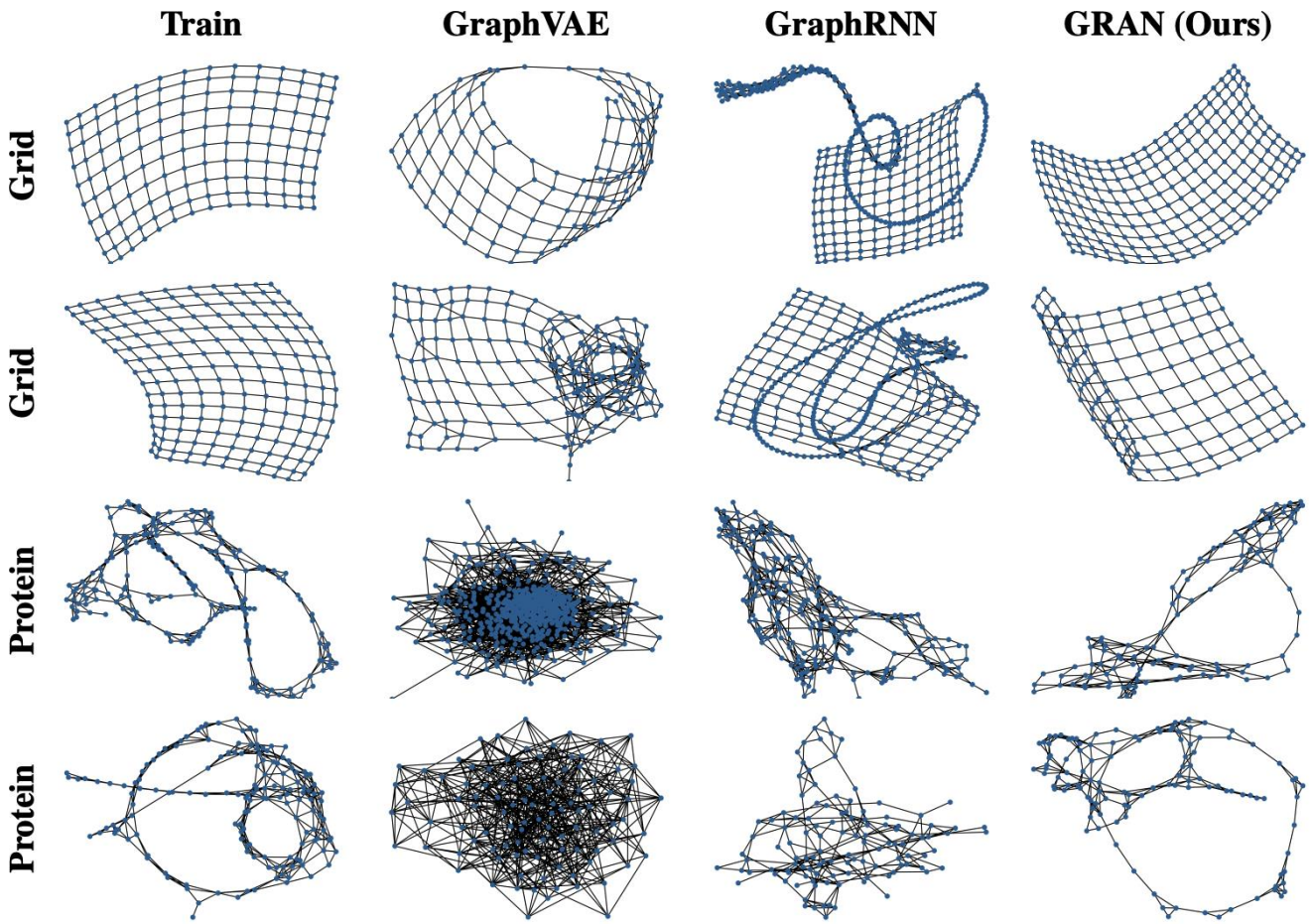
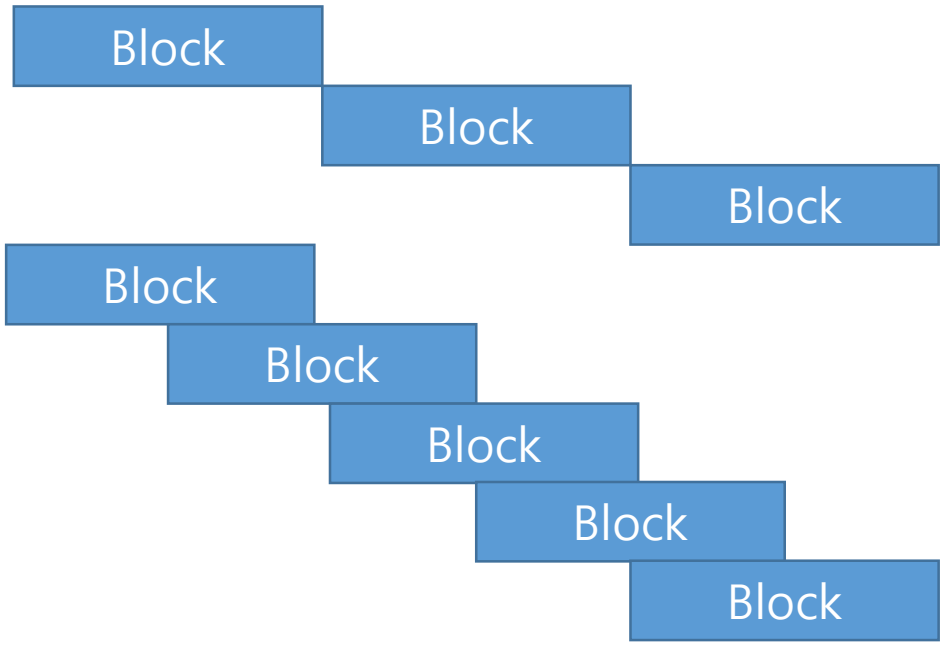
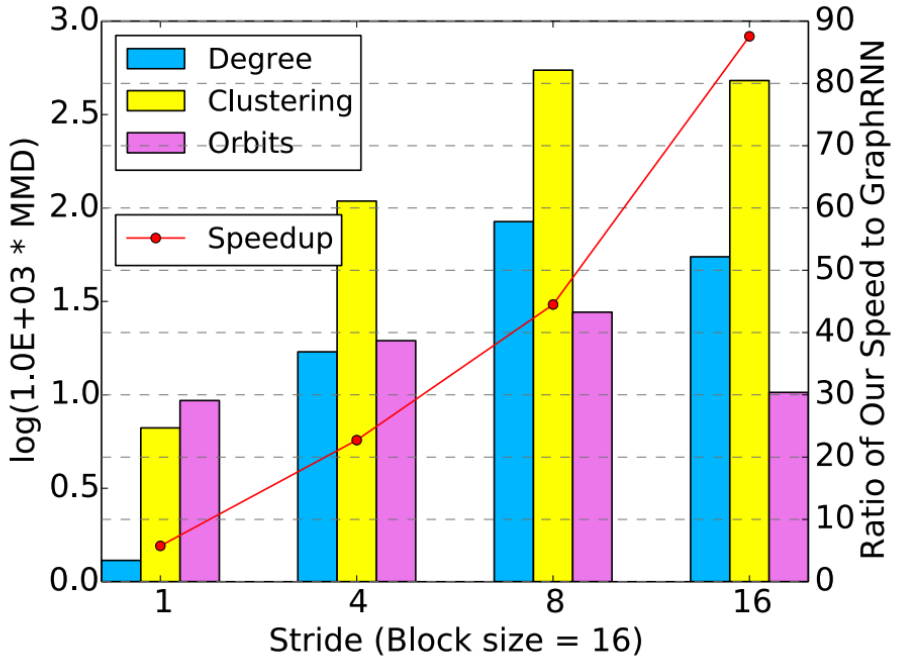


Figure 2: Visualization of sample graphs generated by different models.

Ablation: Efficiency vs Sample Quality

- The main limiting factor for graph generation speed is the number of generation steps T, which is related to the block size B.
- If B grows, which can improve speed, the model quality may suffer.
- They propose “stride sampling”, where neighboring blocks have an overlap





Code Review

- B : batch size, C : number of orderings, K : block size, E : number of edges
  N : number of rows (nodes) of adjacency matrix
- A : adjacency matrix (B * C * N * N)
- att_idx : indicator of newly generated nodes (N * 1)
- edges : list of edges [incoming / outgoing node index] (E * 2)
- Repository link : https://github.com/skynbe/GRAN

Code Review – Node Embedding

Caching node embeddings

From ii to ii+K : node orderings of current block

self.decoder_input :

$$h_{\boldsymbol{b_i}}^0 = WL_{\boldsymbol{b_i}}^{\pi} + b, \qquad \forall i < t.$$

Set node embeddings of current block to 0

```
node_state = torch.zeros(B, N_pad, dim_input).to(self.device)

for ii in range(0, N_pad, S):
  # for ii in range(0, 3530, S):
  jj = ii + K
  if jj > N_pad:
    break

  # reset to discard overlap generation
  A[:, ii:, :] = .0
  A = torch.tril(A, diagonal=-1)

  if ii >= K:
    if self.dimension_reduce:
      node_state[:, ii - K:ii, :] = self.decoder_input(A[:, ii - K:ii, :N])
    else:
      node_state[:, ii - K:ii, :] = A[:, ii - S:ii, :N]
  else:
    if self.dimension_reduce:
      node_state[:, :ii, :] = self.decoder_input(A[:, :ii, :N])
    else:
      node_state[:, :ii, :] = A[:, ii - S:ii, :N]

  node_state_in = F.pad(
      node_state[:, :ii, :], (0, 0, 0, K), 'constant', value=.0)
```

Code Review – Attentive Message Passing

Translating adjacency matrix to list of edges (for computation)

att_idx : indicator x

$$\tilde{h}_i^r = [h_i^r, x_i],$$

self.decoder : message passing GNN

```
adj = F.pad(
    A[:, :ii, :ii], (0, K, 0, K), 'constant', value=1.0)  # B X jj X jj
adj = torch.tril(adj, diagonal=-1)
adj = adj + adj.transpose(1, 2)
edges = [
    adj[bb].to_sparse().coalesce().indices() + bb * adj.shape[1]
    for bb in range(B)
]
edges = torch.cat(edges, dim=1).t()

att_idx = torch.cat([torch.zeros(ii).long(),
                     torch.arange(1, K + 1)]).to(self.device)
att_idx = att_idx.view(1, -1).expand(B, -1).contiguous().view(-1, 1)

# create one-hot feature
att_edge_feat = torch.zeros(edges.shape[0],
                            2 * self.att_edge_dim).to(self.device)
att_edge_feat = att_edge_feat.scatter(1, att_idx[[edges[:, 0]]], 1)
att_edge_feat = att_edge_feat.scatter(
    1, att_idx[[edges[:, 1]]] + self.att_edge_dim, 1)

node_state_out = self.decoder(
    node_state_in.view(-1, H), edges, edge_feat=att_edge_feat)
node_state_out = node_state_out.view(B, jj, -1)
```

Code Review – Attentive Message Passing

```
node_state_out = self.decoder(
    node_state_in.view(-1, H), edges, edge_feat=att_edge_feat)
```

Compute message

$$m_{ij}^r = f(h_i^r - h_j^r),$$

Compute attention

$$a_{ij}^r = \text{Sigmoid}\left(g(\tilde{h}_i^r - \tilde{h}_j^r)\right),$$

Aggregate message

State update by GRU

$$h_i^{r+1} = \text{GRU}(h_i^r, \sum_{j \in \mathcal{N}(i)} a_{ij}^r m_{ij}^r).$$

```
def _prop(self, state, edge, edge_feat, layer_idx=0):

    state_diff = state[edge[:, 0], :] - state[edge[:, 1], :]
    if self.edge_feat_dim > 0:
      edge_input = torch.cat([state_diff, edge_feat], dim=1)
    else:
      edge_input = state_diff

    msg = self.msg_func[layer_idx](edge_input)

    if self.has_attention:
      att_weight = self.att_head[layer_idx](edge_input)
      msg = msg * att_weight

    state_msg = torch.zeros(state.shape[0], msg.shape[1]).to(state.device)
    scatter_idx = edge[:, [1]].expand(-1, msg.shape[1])
    state_msg = state_msg.scatter_add(0, scatter_idx, msg)

    state = self.update_func[layer_idx](state_msg, state)
    return state
```

Code Review – Output Distribution

Get output alpha and theta

$$\alpha_1, \ldots, \alpha_K = \text{Softmax}\left(\sum_{i \in \boldsymbol{b_t}, 1 \le j \le i} \text{MLP}_\alpha(h_i^R - h_j^R)\right),$$
$$\theta_{1,i,j}, \ldots, \theta_{K,i,j} = \text{Sigmoid}\left(\text{MLP}_\theta(h_i^R - h_j^R)\right)$$

Get mixture of Bernoulli distribution

$$p(L_{\boldsymbol{b_t}}^\pi | L_{\boldsymbol{b_1}}^\pi, ..., L_{\boldsymbol{b_{t-1}}}^\pi) = \sum_{k=1}^{K} \alpha_k \prod_{i \in \boldsymbol{b_t}} \prod_{1 < j < i} \theta_{k,i,j},$$

```
node_state_out = self.decoder(
    node_state_in.view(-1, H), edges, edge_feat=att_edge_feat)
node_state_out = node_state_out.view(B, jj, -1)

idx_row, idx_col = np.meshgrid(np.arange(ii, jj), np.arange(jj))
idx_row = torch.from_numpy(idx_row.reshape(-1)).long().to(self.device)
idx_col = torch.from_numpy(idx_col.reshape(-1)).long().to(self.device)

diff = node_state_out[:,idx_row, :] - node_state_out[:,idx_col, :]  # B X (ii+K)K X H
diff = diff.view(-1, node_state.shape[2])
log_theta = self.output_theta(diff)
log_alpha = self.output_alpha(diff)

log_theta = log_theta.view(B, -1, K, self.num_mix_component)  # B X K X (ii+K) X L
log_theta = log_theta.transpose(1, 2)  # B X (ii+K) X K X L

log_alpha = log_alpha.view(B, -1, self.num_mix_component)  # B X K X (ii+K)
prob_alpha = F.softmax(log_alpha.mean(dim=1), -1)
alpha = torch.multinomial(prob_alpha, 1).squeeze(dim=1).long()

prob = []
for bb in range(B):
  prob += [torch.sigmoid(log_theta[bb, :, :, alpha[bb]])]

prob = torch.stack(prob, dim=0)
A[:, ii:jj, :jj] = torch.bernoulli(prob[:, :jj - ii, :])
```

Reproducing Results

- Trained, validated and tested on Grid Dataset

- Validation MMD scores :
  - degree 1.73e-3, clustering 3.51e-4, orbits 1.92e-3, spectral 2.41e-2
- Test MMD scores :
  - degree 9.36e-3, clustering 3.50e-4, orbits 1.63e-3, spectral 1.33e-2

- Github link : https://github.com/lrjconan/GRAN