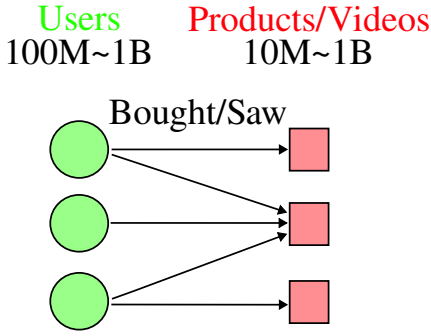
Graph Generation

APM_5DS30_TP, Machine Learning with Graphs

Jhony H. Giraldo (Assistant Professor).
jhony.giraldo@telecom-paris.fr
November 28, 2025

- Information Explosion in the Era of the Internet
  - 10K+ movies in Netflix
  - 12M products in Amazon (350M on Marketplace)
  - 70M+ music tracks on Spotify
  - 10B+ videos on YouTube
  - 200B+ pins (images) on Pinterest
- **Personalized recommendation (*i.e.*, suggesting a small number of interesting items for each user)** is critical for users to effectively explore the content of their interest



- We cannot evaluate $f(u, v)$ for every user u – item v pair (*e.g.* $f(u, v) = \langle \mathbf{z}_u, \mathbf{z}_v \rangle$)
- Solution, 2-stage process: (i) Candidate generation (cheap, fast), and (ii) ranking (slow, accurate)

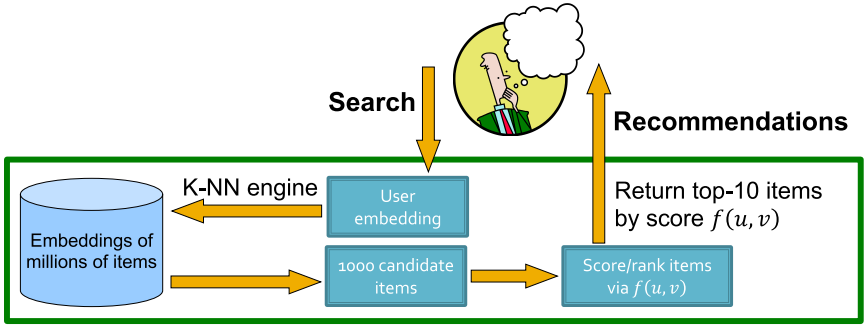
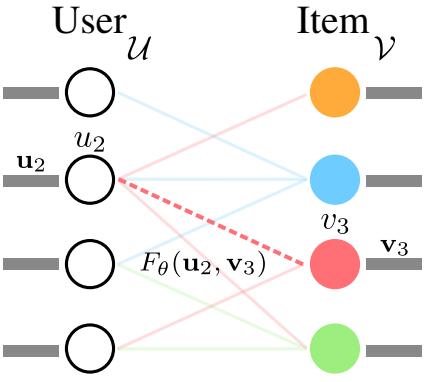


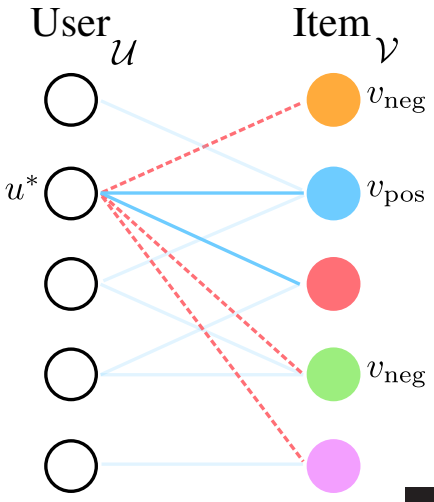
Figure: **Source**: CS224W, Machine Learning with Graphs, Stanford

- We consider embedding-based models for scoring user-item interactions
- For each user $u \in \mathcal{U}$, let $\mathbf{u} \in \mathbb{R}^D$ be its D-dimensional embedding
- For each item $v \in \mathcal{V}$, let $\mathbf{v} \in \mathbb{R}^D$ be its D-dimensional embedding
- Let $F_\theta(\cdot, \cdot) : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be a parametrized function
- Then, $score(u, v) := F_\theta(\mathbf{u}, \mathbf{v})$

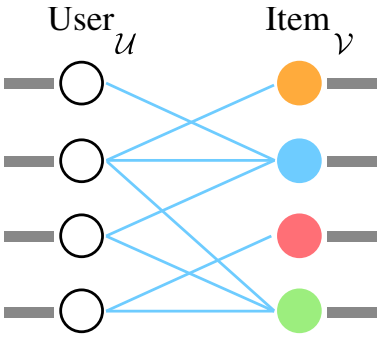


- Mini-batch training for the BPR loss:
  - In each mini-batch, we sample a subset of users $\mathcal{U}_{\text{mini}} \subset \mathcal{U}$
  - For each user $u^* \in \mathcal{U}_{\text{mini}}$, we sample one positive item v_{pos} and a set of sampled negative items $\mathcal{V}_{\text{neg}} = \{v_{\text{neg}}\}$
- The mini-batch loss is computed as:

$$\frac{1}{|\mathcal{U}_{\text{mini}}|} \sum_{u^* \in \mathcal{U}_{\text{mini}}} \frac{1}{|\mathcal{V}_{\text{neg}}|} \sum_{v_{\text{neg}} \in \mathcal{V}_{\text{neg}}} - \log \Big(\sigma \big(F_\theta(u^*, v_{\text{pos}}) - F_\theta(u^*, v_{\text{neg}}) \big) \Big)$$

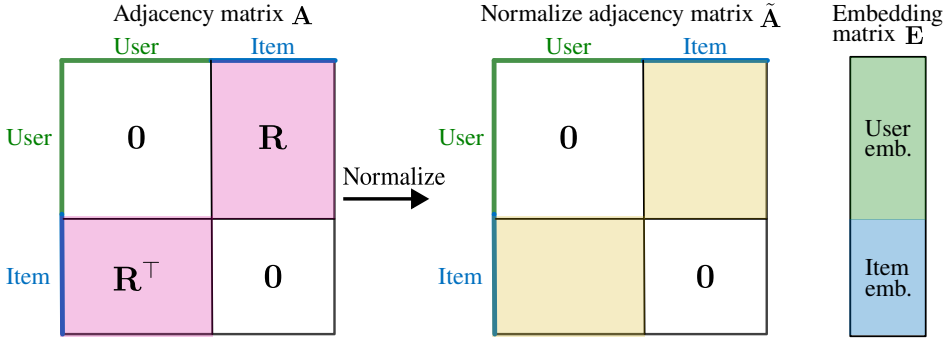


- Given the user-item bipartite graph:
  - Prepare shallow learnable embedding for each node
  - Use multi-layer GNNs to propagate embeddings along the bipartite graph
  - High-order graph structure is captured
  - Final embeddings are explicitly graph-aware
- Two kinds of learnable parameters are jointly learned:
  - Shallow user/item embeddings
  - GNN's parameters



Learnable shallow user/item embeddings

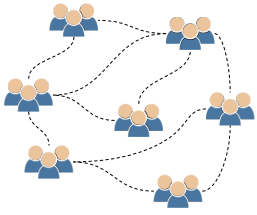
- Given:
  - The adjacency matrix $\mathbf{A}$
  - The initial *learnable* embedding matrix $\mathbf{E}$



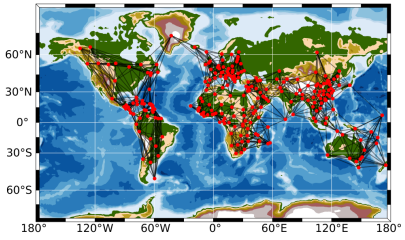
Motivation of Graph Generation

- So far, we have been learning from graphs
- We assume the graphs are given

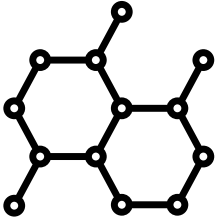
Social networks



Sensor networks



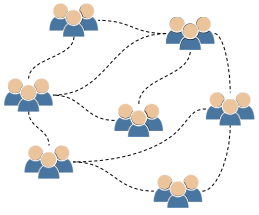
Molecules



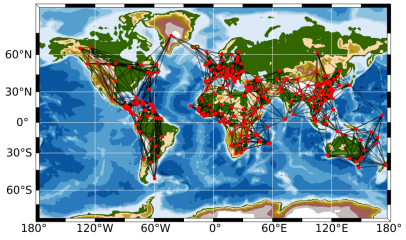
Motivation of Graph Generation

- So far, we have been learning from graphs
- We assume the graphs are given

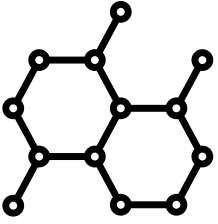
Social networks



Sensor networks



Molecules

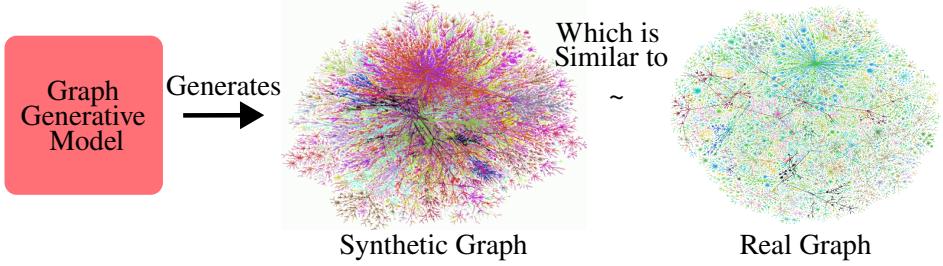


- **But how are these graphs generated?**

* Some images in these slides are taken directly or modified from Leskovec's lectures at Standford

The Problem of Graph Generation

- We want to generate realistic graphs, using **graph generative models**



Why Graph Generation?

- Please discuss with your classmate next to you about possible applications of graph generation
- Please write down the one or two most important applications you see
- We'll have a small discussion about your thoughts

You have 5 minutes

Why Graph Generation?

- Please discuss with your classmate next to you about possible applications of graph generation
- Please write down the one or two most important applications you see
- We'll have a small discussion about your thoughts

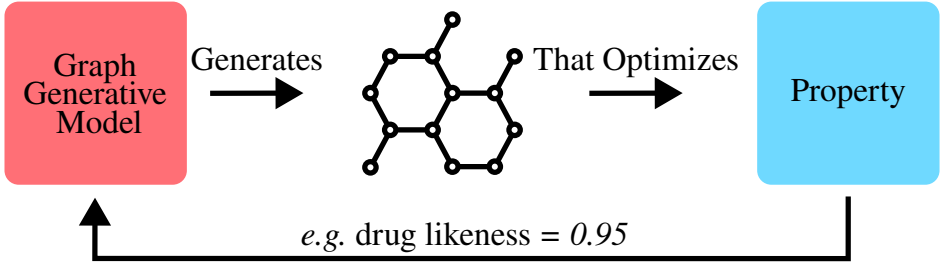
You have 5 minutes

- **Applications**:
  - Drug discovery, material design
  - Social network modeling

Why Graph Generation?

- **To get insights**: We can understand the formulation of graphs
- **Predictions**: We can predict how will the graph further evolve
- **Simulations**: We can use the same process to generate novel graph instances
- **Anomaly detection**: We can decide if a graph is normal/abnormal

Can we learn a model that can generate **valid** and **realistic** molecules with **optimized** property scores?



Roadmap of this Lecture

- **Step 1**: Properties of real-world graphs
  - A successful graph generative model should fit these properties

Roadmap of this Lecture

- **Step 1**: Properties of real-world graphs
  - A successful graph generative model should fit these properties
- **Step 2**: Traditional graph generative models
  - It comes with different assumptions on the graph formulation process

Roadmap of this Lecture

- **Step 1**: Properties of real-world graphs
  - A successful graph generative model should fit these properties
- **Step 2**: Traditional graph generative models
  - It comes with different assumptions on the graph formulation process
- **Step 3**: Deep graph generative models
  - Learn the graph formulation process from the data

Outline

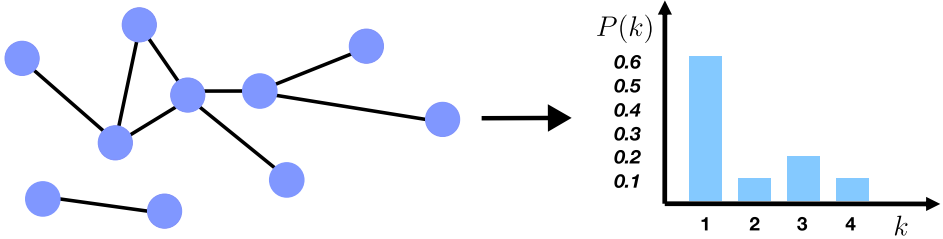
Properties of Real-world Graphs

We will characterize graphs by:

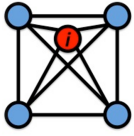
- Degree distribution $P(k)$
- Clustering coefficient C
- Connected components s
- Path length h

- **Degree distribution** $P(k)$: Probability that a randomly chosen node has degree k
- We compute the degree distribution as the normalized histogram $P(k) = N_k/N$, where N_k is the number of nodes with degree k

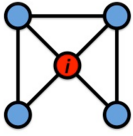
- **Degree distribution** $P(k)$: Probability that a randomly chosen node has degree k
- We compute the degree distribution as the normalized histogram $P(k) = N_k/N$, where N_k is the number of nodes with degree k



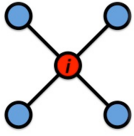
- **Clustering coefficient:**
- How connected are i's neighbors to each other?
- Node i with degree k_i
- $C_i = \frac{2e_i}{k_i(k_i-1)}$, where e_i is the number of edges between the neighbors of node i



$C_i = 1$



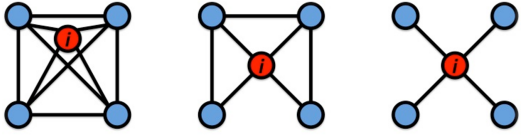
$C_i = 1/2$



$C_i = 0$

$C_i \in [0, 1]$

- **Clustering coefficient:**
- How connected are i's neighbors to each other?
- Node i with degree k_i
- $C_i = \frac{2e_i}{k_i(k_i-1)}$, where e_i is the number of edges between the neighbors of node i

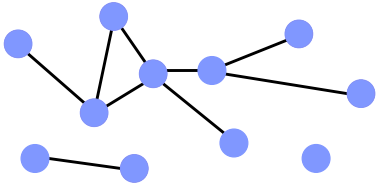


$C_i = 1$ $C_i = 1/2$ $C_i = 0$ $C_i \in [0, 1]$

- Graph clustering coefficient: Take average over all the nodes $C = \frac{1}{N}\sum_i^N C_i$

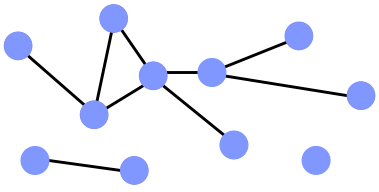
- Size of the largest connected component
  - Largest set where any two vertices can be joined by a path
- Largest component = Giant component

- Size of the largest connected component
  - Largest set where any two vertices can be joined by a path
- Largest component = Giant component



- How to find connected components?

- Size of the largest connected component
  - Largest set where any two vertices can be joined by a path
- Largest component = Giant component



- How to find connected components?
- Start from random node and perform Breadth First Search (BFS)
- Label the nodes that BFS visits
- If all nodes are visited, the network is connected
- Otherwise, find an unvisited node and repeat BFS

- **Diameter**: The maximum (shortest path) distance between any pair of nodes in a graph
  - **Issue**: We can have a large string somewhere in the graph, and this will significantly increase the diameter of the graph

- **Diameter**: The maximum (shortest path) distance between any pair of nodes in a graph
  - **Issue**: We can have a large string somewhere in the graph, and this will significantly increase the diameter of the graph
- The average shortest path length for a connected graph is $\bar{h} = \frac{1}{E_{max}} \sum_{i,j \neq i} h_{i,j}$, where $h_{i,j}$ is the **distance** from node i to node j and $E_{max} = N(N-1)$ is the **max number of edges** (without self-loops)
- Many times we compute the average only over the connected pairs of nodes (that is, we ignore "infinite" length paths)

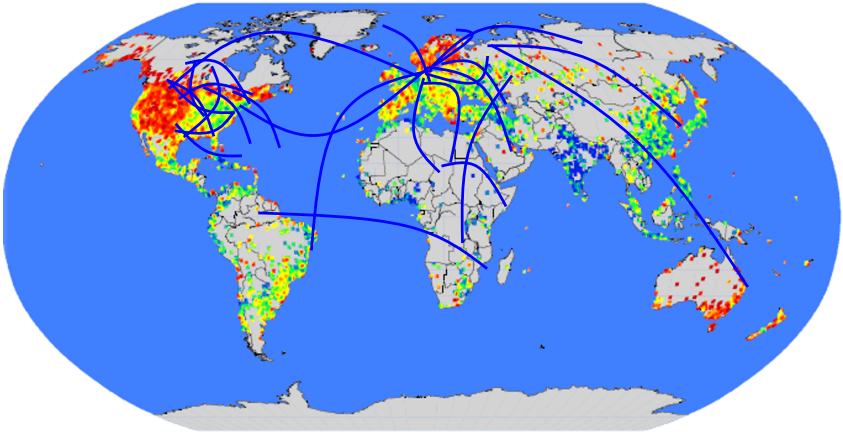
Properties of Real-world Graphs

Example: MSN Graph

- MSN Messenger
- 1 month of activity
- 245 million users logged in
- 180 million users engaged in conversations
- More than 30 billion conversations
- More than 255 billion exchanged messages

Properties of Real-world Graphs

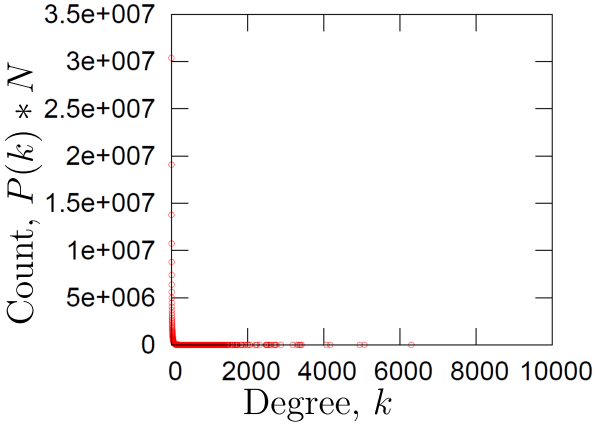
Communication Network



Network: 180M people, 1.3B edges (we connect two people if they exchange at least one message)

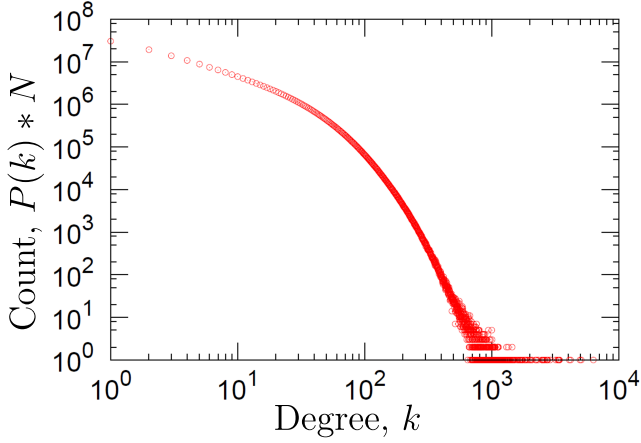
Properties of Real-world Graphs

MSN: Degree Distribution



Properties of Real-world Graphs

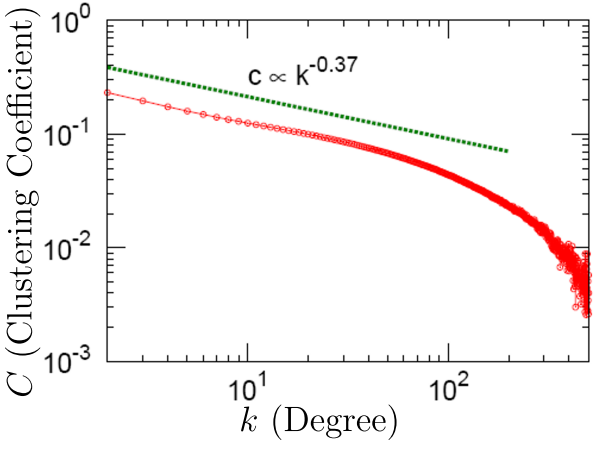
MSN: Log-Log Degree Distribution



Note: We plotted the same data as on the previous slide; just the axes are now in logarithmic scale

Properties of Real-world Graphs

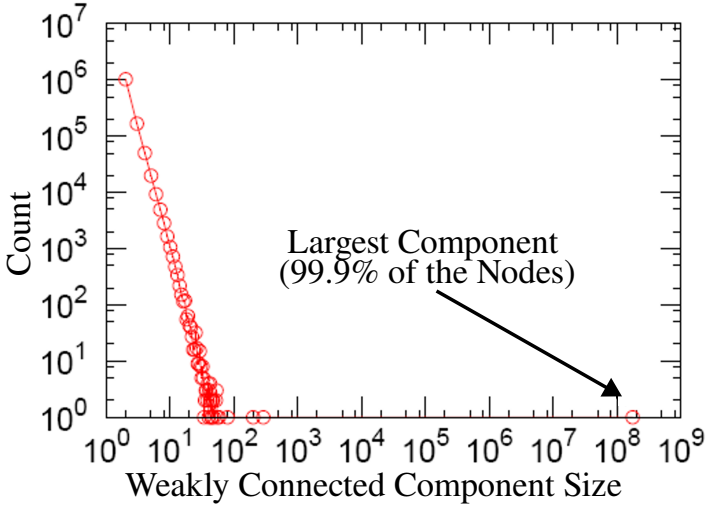
MSN: Clustering Coefficient



Avgerage clustering coefficient of the MSN: $C = 0.1140$

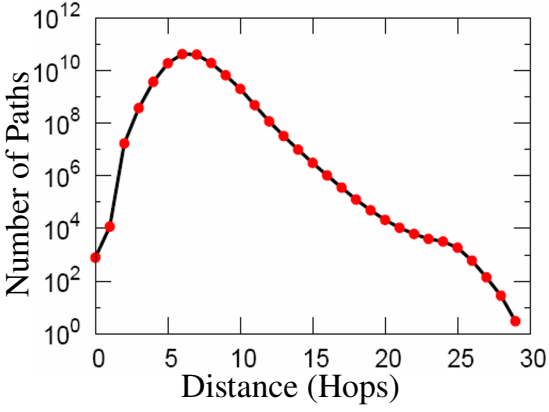
Properties of Real-world Graphs

MSN: Connected Components



Properties of Real-world Graphs

MSN: Path Length



Number of links between pairs of nodes in the largest connected component

Avgerage path length 6.6. 90% of the nodes can be reached in less than 8 hops

- Degree distribution: Heavily skewed, average degree $= 14.4$
- Clustering coefficient: 0.11
- Connectivity: giant component
- Path length: 6.6

Are these values "expected"? Are they "surprising"?
We need a model to answer this.

Erdös-Rényi Random Graphs

Erdös-Renyi Random Graphs

Simplest Model of Graphs

- Erdös-Renyi random graphs [Erdős and Rényi(1959)] are useful for analyzing graph machine learning models (expected performance, computational complexity, running time, scaling, stability, etc.)
- Two variants:
  - G_{np}: undirected graph on N nodes where each edge (i, j) appears *i.i.d.* with probability p
  - G_{nm}: undirected graph with N nodes, and m edges picked uniformly at random

Erdös-Renyi Random Graphs

Simplest Model of Graphs

- Erdös-Renyi random graphs [Erdős and Rényi(1959)] are useful for analyzing graph machine learning models (expected performance, computational complexity, running time, scaling, stability, etc.)
- Two variants:
  - G_{np}: undirected graph on N nodes where each edge (i, j) appears *i.i.d.* with probability p
  - G_{nm}: undirected graph with N nodes, and m edges picked uniformly at random

What kind of networks do such models produce?

Erdös-Renyi Random Graphs

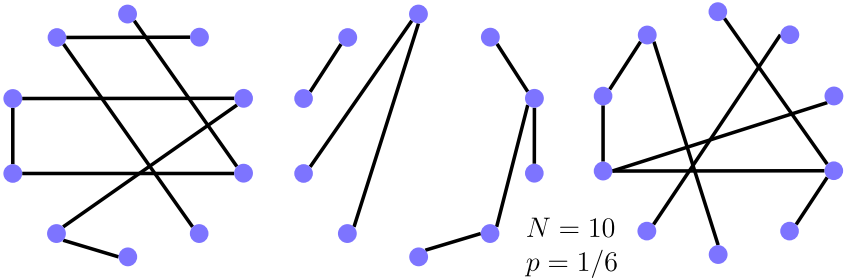
Random Graph Model G_{np}

- n and p do not uniquely determine the graph
- The graph is a result of a random process
- We can have many different realizations given the same n and p

Erdös-Renyi Random Graphs

Random Graph Model G_{np}

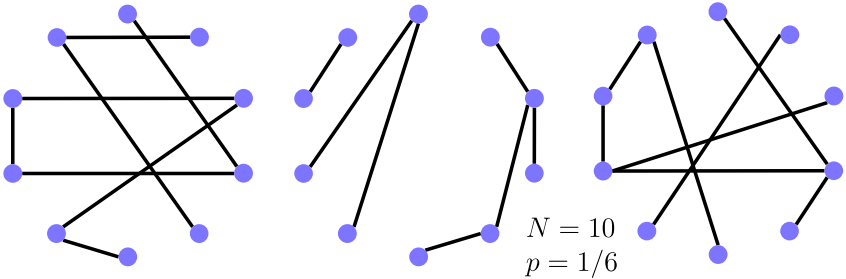
- n and p do not uniquely determine the graph
- The graph is a result of a random process
- We can have many different realizations given the same n and p



Erdös-Renyi Random Graphs

Random Graph Model G_{np}

- n and p do not uniquely determine the graph
- The graph is a result of a random process
- We can have many different realizations given the same n and p



- Let's analyze the properties for G_{np}

Erdös-Renyi Random Graphs

Degree Distribution of G_{np}

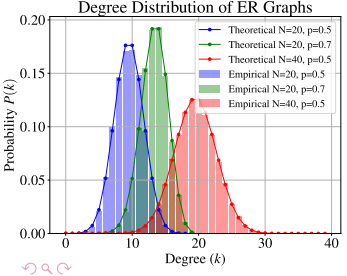
- The degree distribution of G_{np} is binomial
- Let $P(k)$ denote the fraction of nodes with degree k:

$$P(k) = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$

Select k nodes out of N-1

Probability of having k edges

Probability of missing the rest of the N-1-k edges

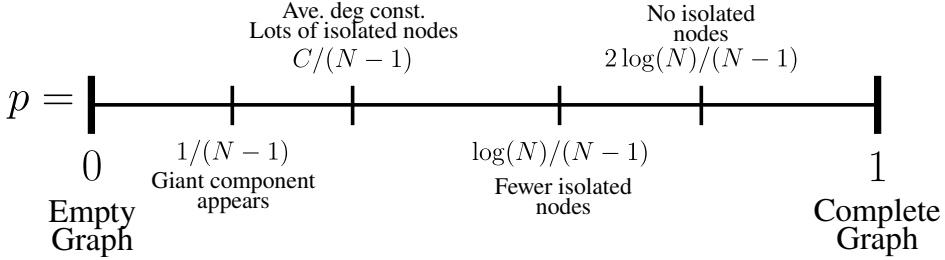


Mean, variance of a binomial distribution

$$\bar{k} = p(N-1)$$
$$\sigma^2 = p(1-p)(N-1)$$

- Remember: $C_i = \frac{2e_i}{k_i(k_i-1)}$
- Since edges in G_{np} appear *i.i.d.* with probability p we have $\mathbb{E}[e_i] = p\frac{k_i(k_i-1)}{2}$, where each pair of nodes is connected with probability p, and $\frac{k_i(k_i-1)}{2}$ is the number of distinct pairs of neighbors of node i of degree k_i

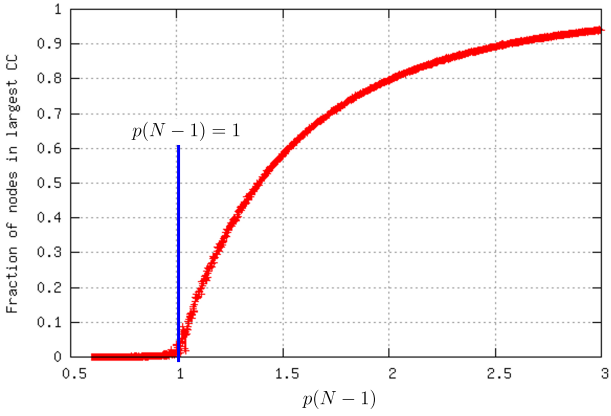
- Remember: $C_i = \frac{2e_i}{k_i(k_i-1)}$
- Since edges in G_{np} appear *i.i.d.* with probability p we have $\mathbb{E}[e_i] = p\frac{k_i(k_i-1)}{2}$, where each pair of nodes is connected with probability p, and $\frac{k_i(k_i-1)}{2}$ is the number of distinct pairs of neighbors of node i of degree k_i
- Then $\mathbb{E}[C_i] = \frac{pk_i(k_i-1)}{k_i(k_i-1)} = p = \frac{\bar{k}}{N-1} \approx \frac{\bar{k}}{N}$
- The clustering coefficient of a random graph is small. If we generate bigger and bigger graphs with fixed average degree k (*i.e.*, we set $p = k/N$), then C decreases with the graph size N

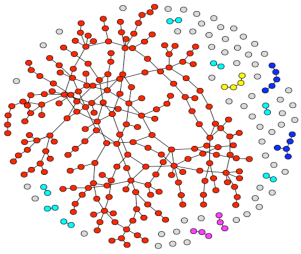


- We have the emergence of a giant component when the average degree is equal to 1, *i.e.*, when $p = 1/(N-1)$
- Each node has at least one edge in expectation

Erdös-Renyi Random Graphs

G_{np} Simulation Experiment





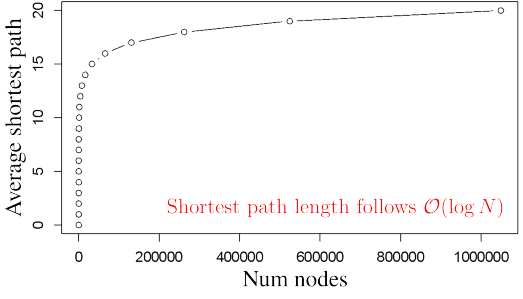
Fraction of nodes in the largest component

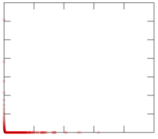
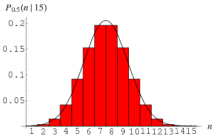
$$G_{np}, N = 100,000, k = p(N-1) = 0.5, \ldots, 3$$

Erdös-Renyi Random Graphs

Path Length of G_{np}

- I'm not going into the math details but for G_{np} the diameter is given by $\mathcal{O}(\log(N)/\log(pN))$
- In random graphs, it takes a logarithmic number of steps for BFS to visit all nodes
- Erdös-Renyi random graph can grow very large, but nodes will be a few hops apart (in the fig., we control the average degree to be constant)



	MSN	G_{np} $N = 180M$	
Degree distribution			✗
Avg. path length:	*6.6*	$\mathcal{O}(\log N)$ ≈ 8.2	✓
Avg. clustering coef.:	*0.11*	$\bar{k}/N$ $\approx 8 * 10^{-8}$	✗
Largest conn. comp.:	*99%*	GCC exists when $k = 1$	✓

Erdös-Renyi Random Graphs

Comparison MSN and Erdös-Renyi Graph

- Are real networks like random graphs?
  - Giant connected component: ✓
  - Average path length: ✓
  - Clustering Coefficient: ✗
  - Degree Distribution: ✗

- Are real networks like random graphs?
  - Giant connected component: ✓
  - Average path length: ✓
  - Clustering Coefficient: ✗
  - Degree Distribution: ✗
- Problems with the random networks model:
  - Degree distribution differs from that of real networks
  - The giant component in most real networks does NOT emerge through a phase transition
  - No local structure – clustering coefficient is too low

Erdös-Renyi Random Graphs

Comparison MSN and Erdös-Renyi Graph

- Are real networks like random graphs?
  - Giant connected component: ✓
  - Average path length: ✓
  - Clustering Coefficient: ✗
  - Degree Distribution: ✗
- Problems with the random networks model:
  - Degree distribution differs from that of real networks
  - The giant component in most real networks does NOT emerge through a phase transition
  - No local structure – clustering coefficient is too low
- Are real networks random? NO!

- Are real networks like random graphs?
  - Giant connected component: ✓
  - Average path length: ✓
  - Clustering Coefficient: ✗
  - Degree Distribution: ✗
- Problems with the random networks model:
  - Degree distribution differs from that of real networks
  - The giant component in most real networks does NOT emerge through a phase transition
  - No local structure – clustering coefficient is too low
- Are real networks random? NO!
- The Erdös-Renyi model is still very important because it serves as a first reference point whenever you want to generate graphs

Other Classical Graph Generation Methodologies

- We're not further exploring other classical graph generation models for the sake of time
- However, there are other models like the **small-world model** that offer a better clustering coefficient than the Erdös-Renyi graph
- There's another important methodology called the **Kronecker graph model** that's also very useful to approximate real graphs

Other Classical Graph Generation Methodologies

- We're not further exploring other classical graph generation models for the sake of time
- However, there are other models like the **small-world model** that offer a better clustering coefficient than the Erdös-Renyi graph
- There's another important methodology called the **Kronecker graph model** that's also very useful to approximate real graphs
- We're gonna focus the rest of this lecture on deep generative models, so these other important models are left for self-study from your side

- Ideas?

Other Classical Graph Generation Methodologies

Why Classical Models Are Not Enough?

- Ideas?
- Traditional models (*e.g.* Erdös-Renyi, Small-World, Kronecker) rely on **strong assumptions** about how graphs form, limiting flexibility
- Classical models use **predefined rules** rather than learning from data, making them less effective in capturing unseen or evolving network patterns
- They often fail to generalize well across different domains, requiring manual tuning for each specific application

Other Classical Graph Generation Methodologies

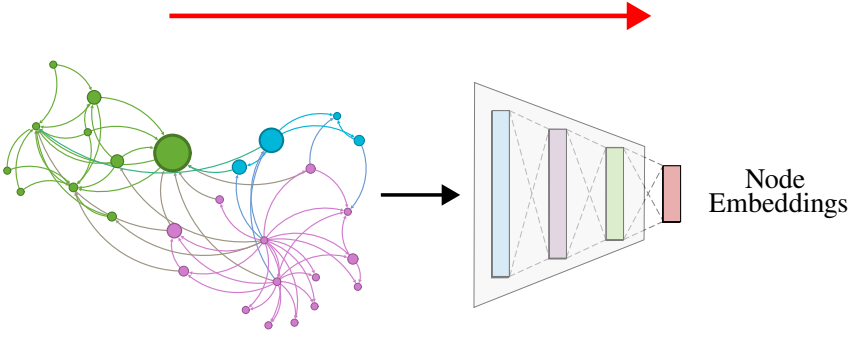
Why Classical Models Are Not Enough?

- Ideas?
- Traditional models (*e.g.* Erdös-Renyi, Small-World, Kronecker) rely on **strong assumptions** about how graphs form, limiting flexibility
- Classical models use **predefined rules** rather than learning from data, making them less effective in capturing unseen or evolving network patterns
- They often fail to generalize well across different domains, requiring manual tuning for each specific application
- Deep models can automatically **learn** generation rules without relying on handcrafted assumptions
- These models capture complex dependencies, generating graphs that resemble real-world networks more accurately

Deep Generative Models

Deep Generative Models

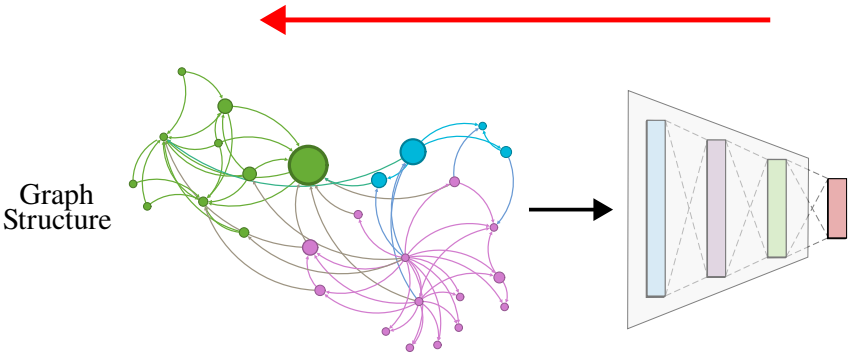
So Far We've Been Doing



The outputs are typically node embeddings

Deep Generative Models

Today

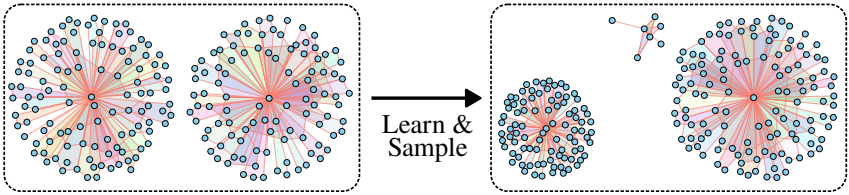


The output is a graph structure

- Task 1: Realistic graph generation
  - Generate graphs that are similar to a given set of graphs

- Task 1: Realistic graph generation
  - Generate graphs that are similar to a given set of graphs
- Another task: Goal-directed (conditional) graph generation
  - Generate graphs that optimize given objectives/constraints, *e.g.* drug molecule generation/optimization (**not covered today**)

- **Given:** Graphs sampled from $p_{data}(G)$ (real distribution, we don't know it)
- **Goal:**
  - Learn the distribution $p_{model}(G)$
  - Sample from $p_{model}(G)$



Setup:

- Assume we want to learn a generative model from a set of data points (*i.e.*, graphs) $\{\mathbf{x}_i\}$
  - $p_{data}(\mathbf{x})$ is the **data distribution**, which is never known to us, but we have sampled $\mathbf{x}_i \sim p_{data}(\mathbf{x})$
  - $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ is the **model**, parametrized by $\boldsymbol{\theta}$, that we use to approximate $p_{data}(\mathbf{x})$

Setup:

- Assume we want to learn a generative model from a set of data points (*i.e.*, graphs) $\{\mathbf{x}_i\}$
  - $p_{data}(\mathbf{x})$ is the **data distribution**, which is never known to us, but we have sampled $\mathbf{x}_i \sim p_{data}(\mathbf{x})$
  - $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ is the **model**, parametrized by $\boldsymbol{\theta}$, that we use to approximate $p_{data}(\mathbf{x})$

Goal:

1. Make $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ close to $p_{data}(\mathbf{x})$ (density estimation)
2. Make sure we can sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ (sampling)
   - To generate new graphs, we sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$

1. Make $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ close to $p_{data}(\mathbf{x})$
   - Key principle: Maximum likelihood
   - Fundamental approach to modeling distributions

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim p_{data}} \log p_{model}(\mathbf{x}|\boldsymbol{\theta})$$

1. Make $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ close to $p_{data}(\mathbf{x})$
   - Key principle: Maximum likelihood
   - Fundamental approach to modeling distributions

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim p_{data}} \log p_{model}(\mathbf{x}|\boldsymbol{\theta})$$

   - Find parameters $\boldsymbol{\theta}^*$, such that for observed data points $\mathbf{x}_i \sim p_{data}$ the $\sum_i \log p_{model}(\mathbf{x}_i, \boldsymbol{\theta}^*)$ has the highest value, among all possible choices of $\boldsymbol{\theta}$

1. Make $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ close to $p_{data}(\mathbf{x})$
   - Key principle: Maximum likelihood
   - Fundamental approach to modeling distributions

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim p_{data}} \log p_{model}(\mathbf{x}|\boldsymbol{\theta})$$

   - Find parameters $\boldsymbol{\theta}^*$, such that for observed data points $\mathbf{x}_i \sim p_{data}$ the $\sum_i \log p_{model}(\mathbf{x}_i, \boldsymbol{\theta}^*)$ has the highest value, among all possible choices of $\boldsymbol{\theta}$
   - In other words, we try to find the model that is most likely to have generated the observed data $\mathbf{x}$

2. Make sure we can sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$
   - **Goal:** Sample from a complex distribution
   - The most common approach:
     1. Sample from a simple noise distribution $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$
     2. Transform the noise $\mathbf{z}_i$ via some function $f(\cdot)$, $\mathbf{x}_i = f(\mathbf{z}_i, \boldsymbol{\theta})$

2. Make sure we can sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$
   - **Goal:** Sample from a complex distribution
   - The most common approach:
     1. Sample from a simple noise distribution $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$
     2. Transform the noise $\mathbf{z}_i$ via some function $f(\cdot)$, $\mathbf{x}_i = f(\mathbf{z}_i, \boldsymbol{\theta})$
   - Then $\mathbf{x}_i$ follows a complex distribution

2. Make sure we can sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$
   - **Goal:** Sample from a complex distribution
   - The most common approach:
     1. Sample from a simple noise distribution $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$
     2. Transform the noise $\mathbf{z}_i$ via some function $f(\cdot)$, $\mathbf{x}_i = f(\mathbf{z}_i, \boldsymbol{\theta})$
   - Then $\mathbf{x}_i$ follows a complex distribution
   - How to design $f(\cdot)$?

2. Make sure we can sample from $p_{model}(\mathbf{x}, \boldsymbol{\theta})$
   - **Goal:** Sample from a complex distribution
   - The most common approach:
     1. Sample from a simple noise distribution $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$
     2. Transform the noise $\mathbf{z}_i$ via some function $f(\cdot)$, $\mathbf{x}_i = f(\mathbf{z}_i, \boldsymbol{\theta})$
   - Then $\mathbf{x}_i$ follows a complex distribution
   - How to design $f(\cdot)$?
   - Use **deep neural networks**, and train it using the data we have!

Auto-regressive models:

- $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ is used for both **density estimation** and **sampling**
- Other models like Variational Auto Encoders (VAEs), Generative Adversarial Networks (GANs) have 2 or more models, each playing one of the roles

Auto-regressive models:

- $p_{model}(\mathbf{x}, \boldsymbol{\theta})$ is used for both **density estimation** and **sampling**
- Other models like Variational Auto Encoders (VAEs), Generative Adversarial Networks (GANs) have 2 or more models, each playing one of the roles
- **Idea:** Chain rule. Joint distribution is a product of conditional distributions:

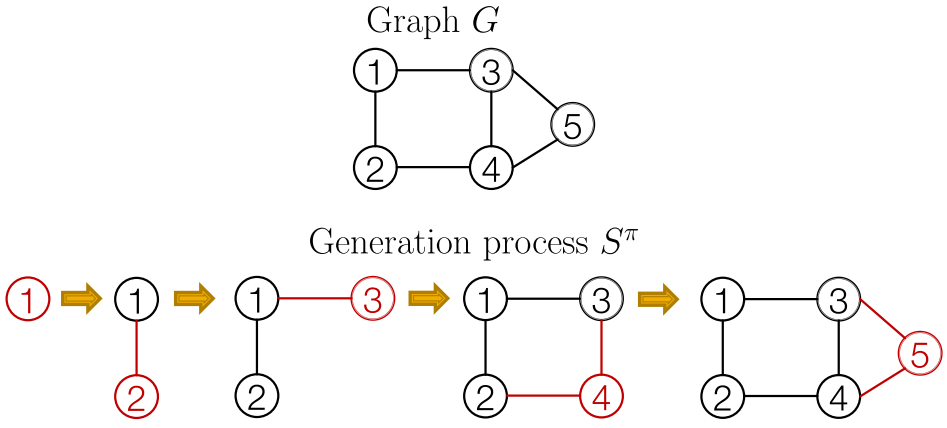
$$p_{model}(\mathbf{x}, \boldsymbol{\theta}) = \prod_{t=1}^{N} p_{model}(\mathbf{x}_t | \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}; \boldsymbol{\theta}), \tag{1}$$

  where $\mathbf{x}_t$ is the t-th dimension of $\mathbf{x}$, *e.g.* $\mathbf{x}$ is a sentence, $\mathbf{x}_t$ is the t-th word
- In this case $\mathbf{x}_t$ will be the t-th action (add node, add edge)

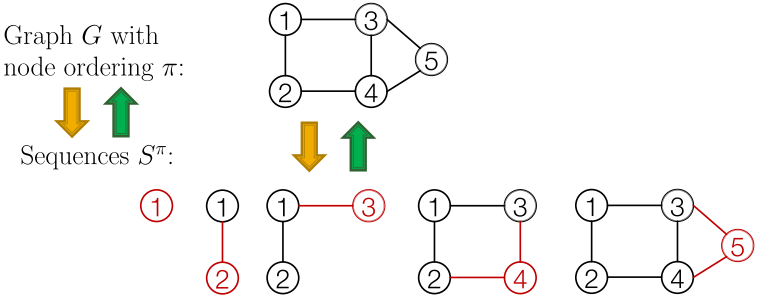
- Today, we'll see two models:
- GraphRNN [You et al.(2018)], one relevant methodology in auto-regressive models
- "Efficient and scalable graph generation through iterative local expansion" [Bergmeister et al.(2024)], a more recent method with similar ideas that uses diffusion models

GraphRNN: Generating Realistic Graphs

Generating graphs via sequentially adding nodes and edges



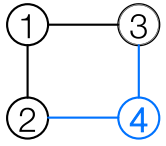
Graph G with node ordering π can be uniquely mapped into a sequence of node and edge additions S^π



$$S^\pi = (\ S_1^\pi\ ,\ S_2^\pi\ ,\ S_3^\pi\ ,\ S_4^\pi\ ,\ S_5^\pi\ )$$

The sequences S^π has two levels:

- Each node-level step is an edge-level sequence
- Edge-level: At each step, add a new edge



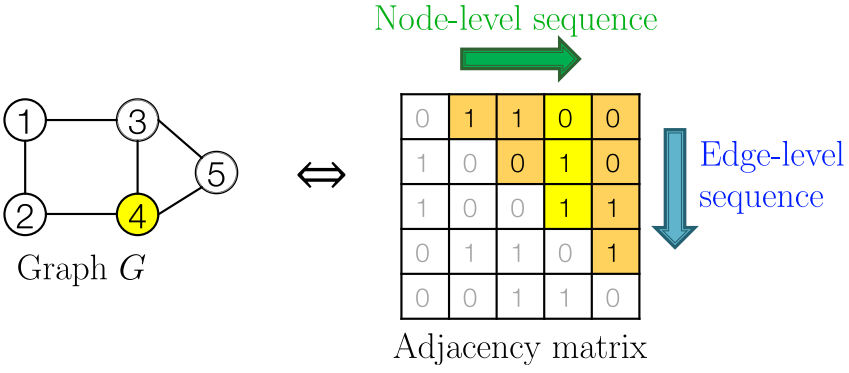
S_4^π

$$S_4^\pi = (\ S_{4,1}^\pi \quad , \quad S_{4,2}^\pi \quad , \quad S_{4,3}^\pi \ )$$

Not connect *4, 1* | Connect *4, 2* | Connect *4, 3*

0 | 1 | 1

- Summary: A graph plus a node ordering = A sequence of sequences
- Node ordering is randomly selected

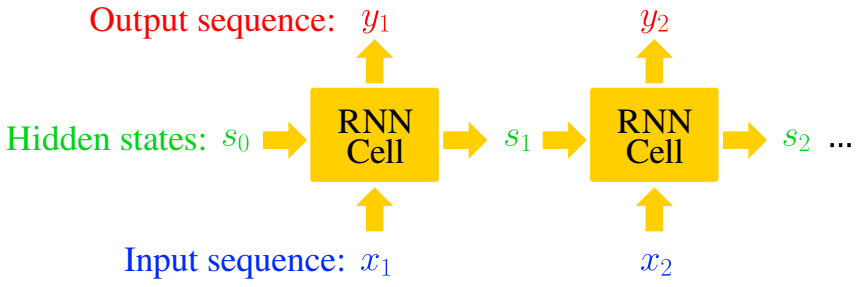


- We have transformed graph generation problem into a sequence generation problem

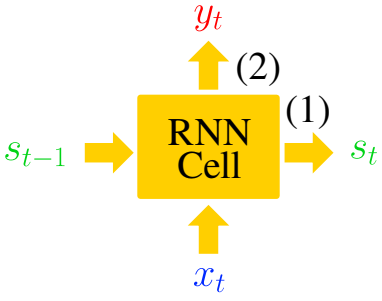
- We have transformed graph generation problem into a sequence generation problem
- Need to model two processes:
  1. Generate a state for a new node (node-level sequence)
  2. Generate edges for the new node based on its state (edge-level sequence)

- We have transformed graph generation problem into a sequence generation problem
- Need to model two processes:
  1. Generate a state for a new node (node-level sequence)
  2. Generate edges for the new node based on its state (edge-level sequence)
- **Approach on GraphRNN:** Use Recurrent Neural Networks (RNNs) to model these processes.
- Same principles apply to Transformers

- RNNs are designed for **sequential data**
- RNN sequentially takes **input sequence** to update its **hidden states**
- The **hidden states** summarize all the information input to RNN
- The **output** is computed via **RNN cells**



- s_t: **State** of the RNN after step t
- x_t: **Input** of the RNN at step t
- y_t: **Output** of the RNN at step t
- RNN cell $(\mathbf{W}, \mathbf{U}, \mathbf{V})$: Trainable parameters



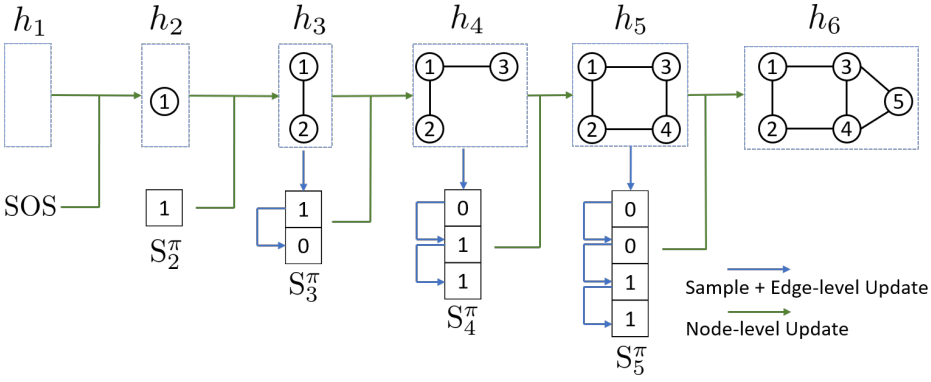
The RNN cell:

1. Update hidden state:
   $s_t = \sigma\left(x_t \mathbf{W} + s_{t-1} \mathbf{U}\right)$
2. Output prediction:
   $y_t = s_t \mathbf{V}$

In our case s_t, x_t and y_t will be scalars (edge probabilities)

- GraphRNN has a node-level RNN and an edge-level RNN
- Relationships between two RNNs:
- Node-level RNN generates the initial state for edge-level RNN
- Edge-level RNN sequentially predicts if the new node will connect to each of the previous nodes

The node-level RNN generates the initial state for the edge-level RNN



The edge-level RNN sequentially predicts if the new node will connect to each of the previous node

- How to use RNN to generate sequences?

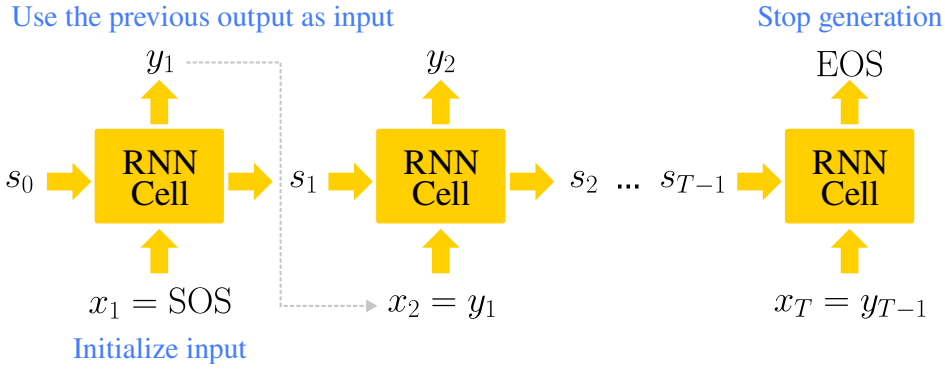
- How to use RNN to generate sequences?
- $x_{t+1} = y_t$ (use the previous output as input)

- How to use RNN to generate sequences?
- $x_{t+1} = y_t$ (use the previous output as input)
- How to initialize the input sequence?

- How to use RNN to generate sequences?
- $x_{t+1} = y_t$ (use the previous output as input)
- How to initialize the input sequence?
- Use start of sequence token (SOS) as the initial input
  - SOS is usually a vector with all zero/ones

- How to use RNN to generate sequences?
- $x_{t+1} = y_t$ (use the previous output as input)
- How to initialize the input sequence?
- Use start of sequence token (SOS) as the initial input
  - SOS is usually a vector with all zero/ones
- When to stop generation?

- How to use RNN to generate sequences?
- $x_{t+1} = y_t$ (use the previous output as input)
- How to initialize the input sequence?
- Use start of sequence token (SOS) as the initial input
  - SOS is usually a vector with all zero/ones
- When to stop generation?
- Use end of sequence token (EOS) as an extra RNN output



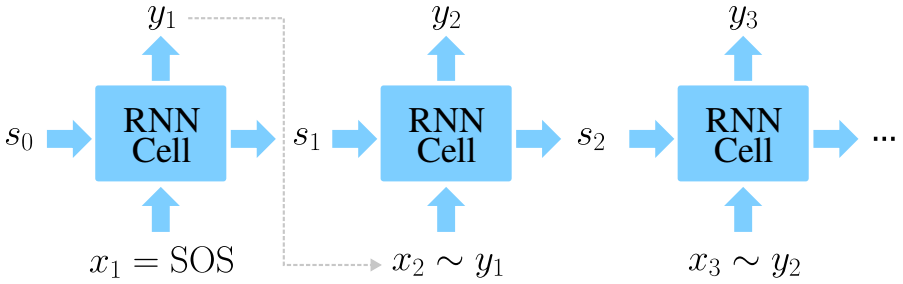
This model is **deterministic**, but we need a stochastic model

Consider the Edge-level RNN for now

- Our goal: Model $\prod_{t=1}^{N} p_{model}(x_t|x_1, \ldots, x_{t-1}; \boldsymbol{\theta})$
- Let $y_t = p_{model}(x_{t+1}|x_1, \ldots, x_t; \boldsymbol{\theta})$
- Then we need to sample x_{t+1} from y_t : $x_{t+1} \sim y_t$

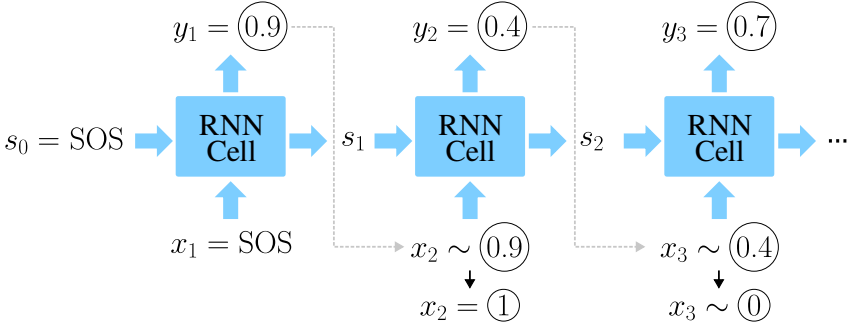
Consider the Edge-level RNN for now

- Our goal: Model $\prod_{t=1}^{N} p_{model}(x_t|x_1, \ldots, x_{t-1}; \boldsymbol{\theta})$
- Let $y_t = p_{model}(x_{t+1}|x_1, \ldots, x_t; \boldsymbol{\theta})$
- Then we need to sample x_{t+1} from y_t : $x_{t+1} \sim y_t$
  - Each step of RNN outputs a probability of a single edge
  - We then sample from the distribution and feed the sample to the next step



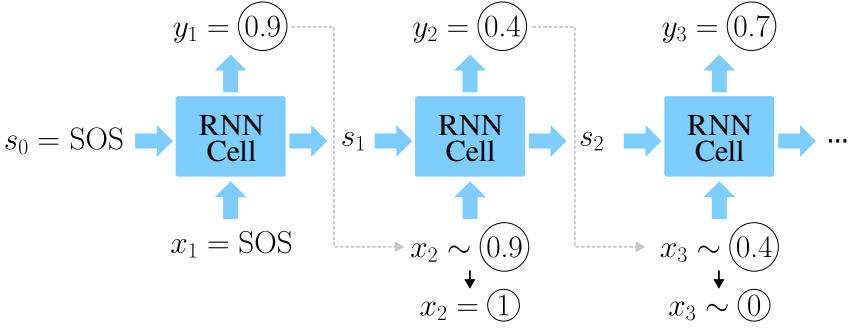
Suppose we already have trained the edge-level RNN

- y_t is a **scalar**, following a **Bernoulli distribution**
- $\textcircled{p}$ means the value **1** has probability p, while the value **0** has probability $1-p$



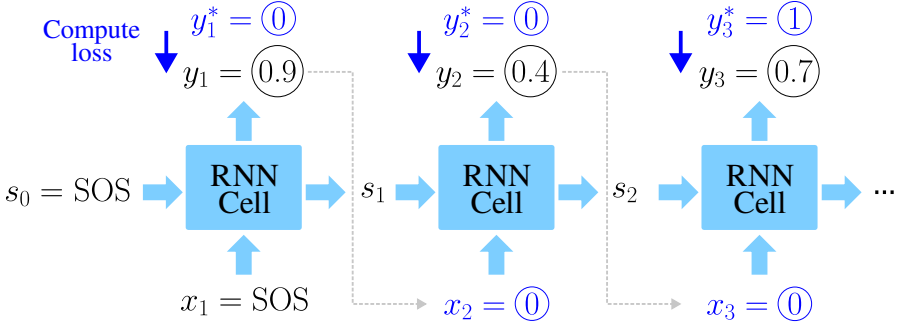
Suppose we already have trained the edge-level RNN

- y_t is a **scalar**, following a **Bernoulli distribution**
- $\textcircled{p}$ means the value **1** has probability p, while the value **0** has probability $1-p$



Now, how do we train the model?

- During training, we observe a sequence y^* of edges representing the connection or not connection with specific nodes $[0, 0, 1, \dots]$
- The model is trained with the **teacher forcing** principle, meaning that we replace input and output by the real sequence and compute the loss function



- Loss $\mathcal{L}$: Binary cross entropy

$$\mathcal{L} = -\left(y_1^* \log(y_1) + (1 - y_1^*) \log(1 - y_1)\right) \tag{2}$$

Compute loss

$$y_1^* = 0$$
$$y_1 = 0.9$$

- Loss $\mathcal{L}$: Binary cross entropy

$$\mathcal{L} = -\left(y_1^* \log(y_1) + (1 - y_1^*) \log(1 - y_1)\right) \tag{2}$$

Compute loss

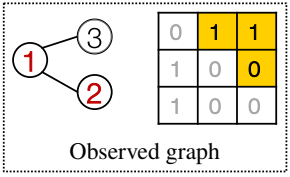
$y_1^* = 0$

$y_1 = 0.9$

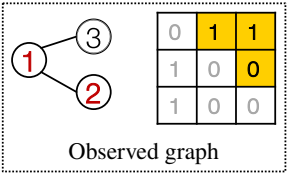
- If $y_1^* = 1$, we minimize $-\log(y_1)$, making y_1 higher
- If $y_1^* = 0$, we minize $-\log(1 - y_1)$, making y_1 lower
- Therefore, y_1 is fitting the data samples y_1^*

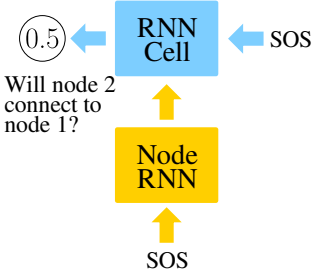
1. Add a new node: We run the node-level RNN for a step and use its output to initialize the edge-level RNN
2. Add new edges for the new node: We run the edge-level RNN to predict if the new node will connect to each of the previous nodes
3. Add new nodes: We use the last hidden state of the edge-level RNN to run the node-level RNN for another step
4. Stop graph generation: If we generate an isolated node, we stop the generation (EOS), *i.e.*, the edge-level RNN outputs all zeros.

We assume **Node 1** is in the graph
Now we add **Node 2**

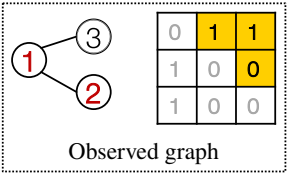


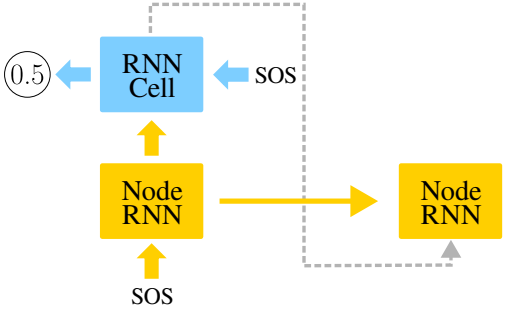
The edge-level RNN predicst how **Node 2** connects to **Node 1**





Update the node RNN using the edge RNN's hidden state

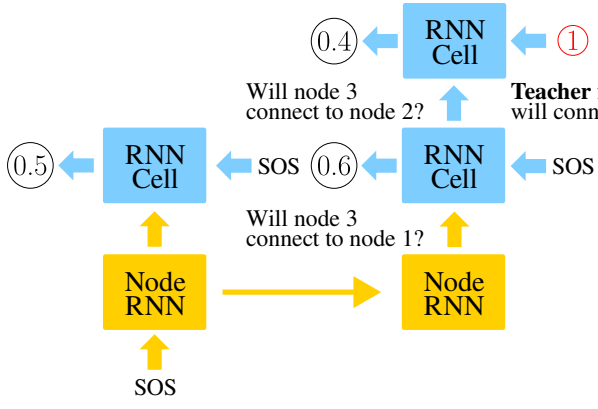


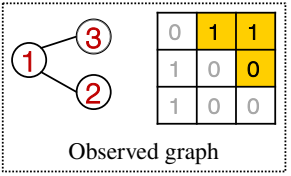


GraphRNN: Generating Realistic Graphs

Training

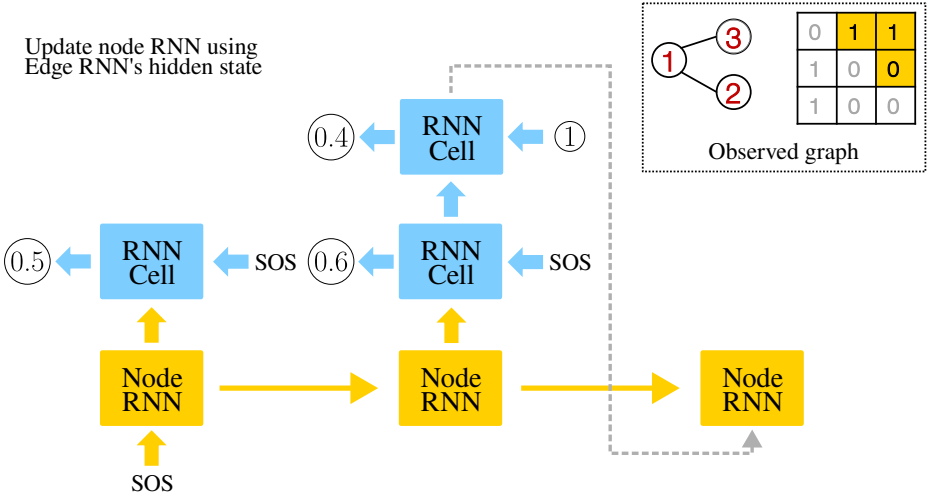
Edge RNN predicts how **Node 3** tries to connect to **Nodes 1, 2**





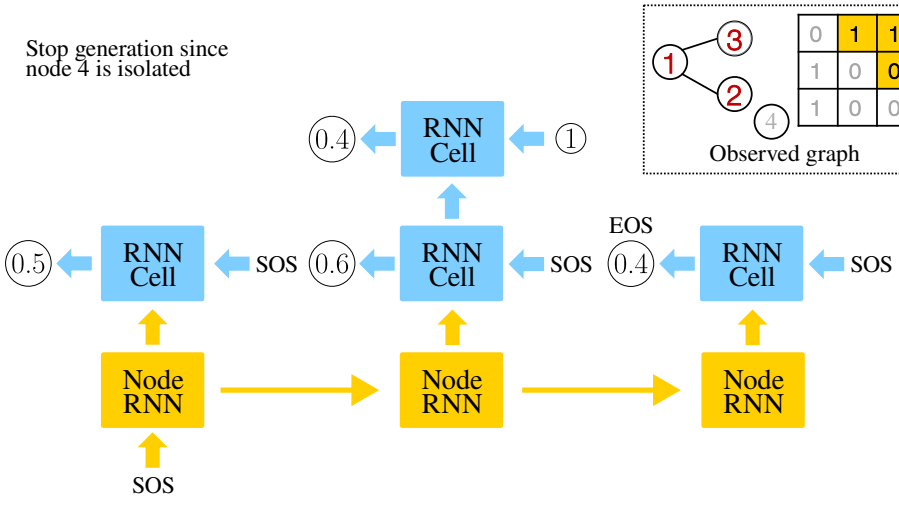
GraphRNN: Generating Realistic Graphs

Training



GraphRNN: Generating Realistic Graphs

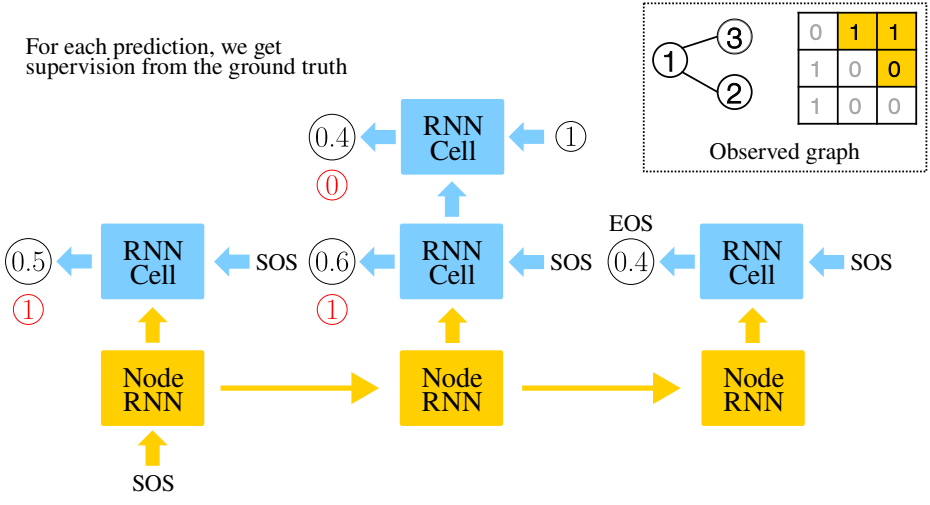
Training



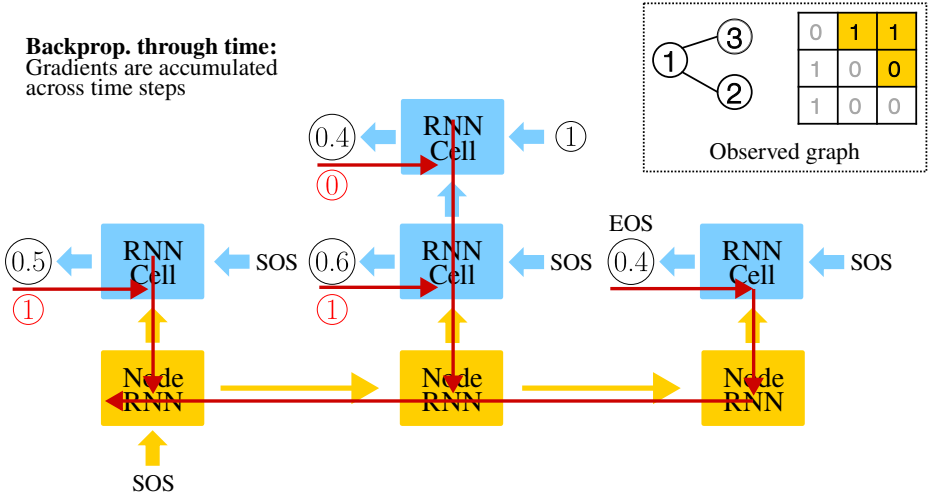
GraphRNN: Generating Realistic Graphs

Training

For each prediction, we get supervision from the ground truth



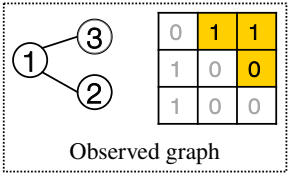
Backprop. through time:
Gradients are accumulated across time steps

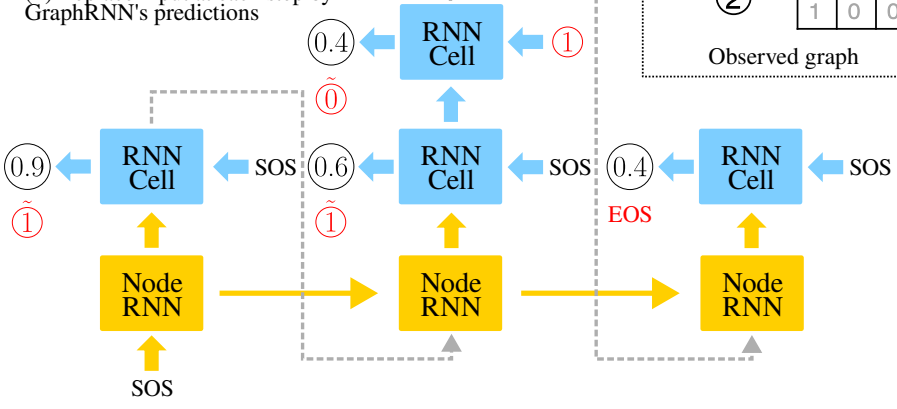


GraphRNN: Generating Realistic Graphs

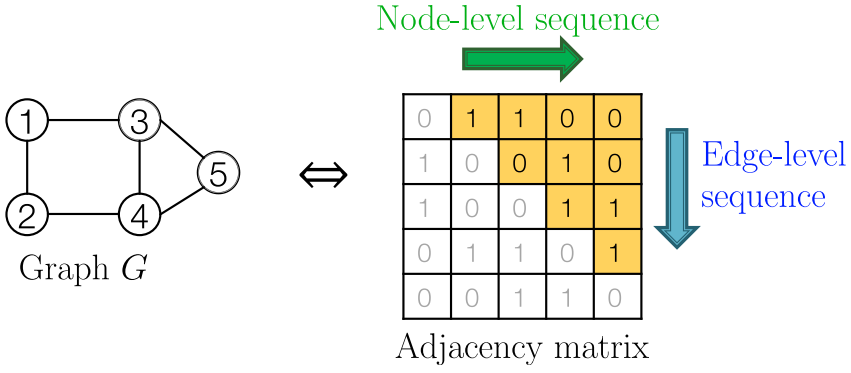
Testing

During test: (i) Sample edge connectivity based on predicted distribution.
(ii) Replace input at each step by GraphRNN's predictions





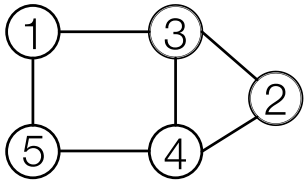
- Generate a graph by generating a two-level sequence
- Use RNN to generate the sequences



GraphRNN: Generating Realistic Graphs

What's the Problem with GraphRNN?

- Any node can connect to any prior node
- Too many steps for edge generation
- Complex for long edge depedencies



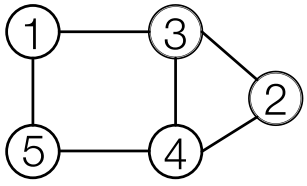
How do we generate this graph?

- Add node 1
- Add node 2
- Add node 3
- Connect 3 with 2 and 1
- Add node 4
- ...

GraphRNN: Generating Realistic Graphs

What's the Problem with GraphRNN?

- Any node can connect to any prior node
- Too many steps for edge generation
- Complex for long edge depedencies



How do we generate this graph?

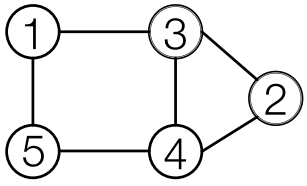
- Add node 1
- Add node 2
- Add node 3
- Connect 3 with 2 and 1
- Add node 4
- ...

- The authors of GraphRNN addressed the problem of long edge dependencies with a BFS ordering, but do you see another problem?

GraphRNN: Generating Realistic Graphs

What's the Problem with GraphRNN?

- Any node can connect to any prior node
- Too many steps for edge generation
- Complex for long edge depedencies



How do we generate this graph?

- Add node 1
- Add node 2
- Add node 3
- Connect 3 with 2 and 1
- Add node 4
- ...

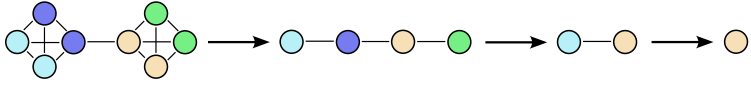
- The authors of GraphRNN addressed the problem of long edge dependencies with a BFS ordering, but do you see another problem?
- The complexity to generate the graph is $\mathcal{O}\left(\sum_{i=1}^{N} i\right) = \mathcal{O}\left(\frac{N(N+1)}{2}\right)$

Efficient Graph Generation

Efficient Graph Generation

Idea

- Based on [Bergmeister et al.(2024)]
- In GraphRNN, we were relying on the ordering of the nodes for the generation, while here we rely on coarsening

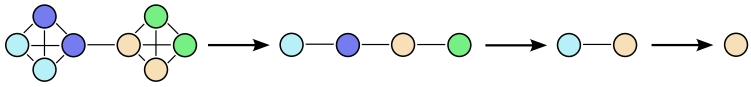


Reduce the graph by merging pairs of adjacent nodes

Efficient Graph Generation

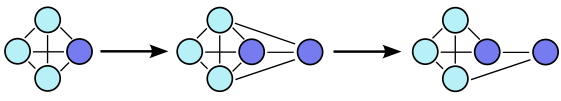
Idea

- Based on [Bergmeister et al.(2024)]
- In GraphRNN, we were relying on the ordering of the nodes for the generation, while here we rely on coarsening

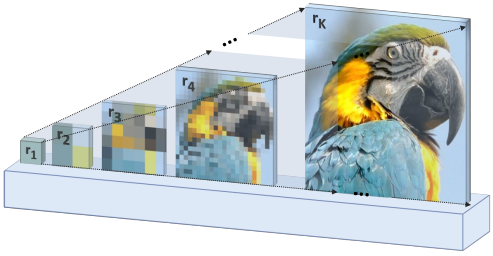


Reduce the graph by merging pairs of adjacent nodes

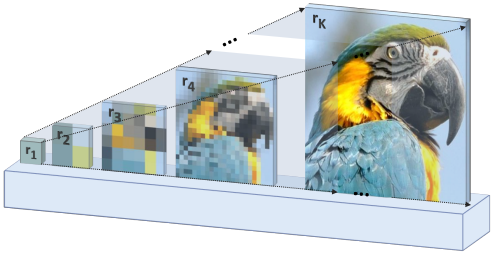
- The idea is to have a sequence of coarsened graphs, and the generative model should learn to reconstruct the graph



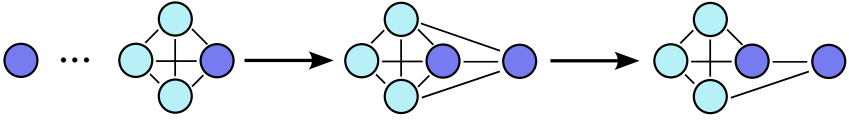
Learn to reconstruct the graph



Generating images autoregresively. **Source:** [Tian et al.(2024)]



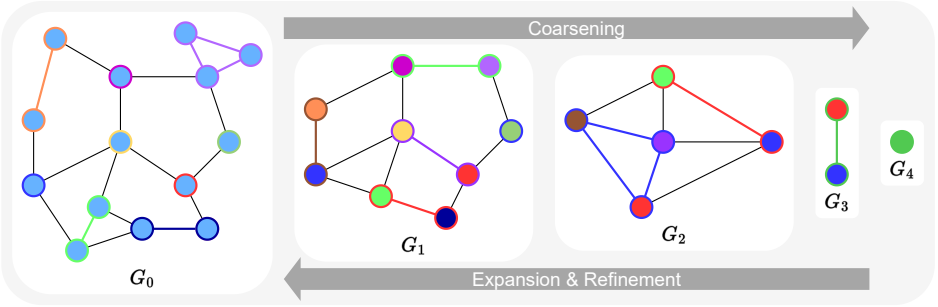
Generating images autoregresively. **Source:** [Tian et al.(2024)]



Generating the graph from one side to the other

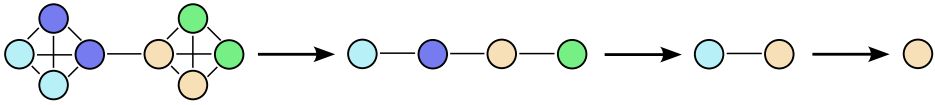
- What do you think are the advantages of this approach regarding GraphRNN?

- What do you think are the advantages of this approach regarding GraphRNN?
- Avoids the ordering problem
- Starts from a coarsen view (single node) and iteratively adds details to each region of the graph
- For each new node, we don't need to infer the connection with all previous nodes, so complexity is significantly reduced

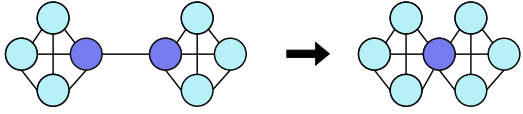


Example of a 4-level coarsening sequence. Colors indicate the node contraction sets. The generation process aims at reversing with expansions and refinements the T steps of this sequence from G_T to G_0. **Source:** [Bergmeister et al.(2024)]

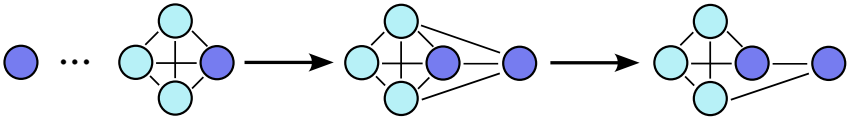
- We start from a graph from the dataset
- We iteratively merge pairs of adjacent nodes until only a single node remains
- Each merging is selected such that the spectral properties of the Laplacian of the graph are preserved [Loukas(2019)]
- This is just a formalization of “keeping the information of the graph as much as possible" (which is contained in the spectral properties of the Laplacian)
- We're not going into the math details here



Ideal coarsening that preserves the aspect



Coarsening losing properties

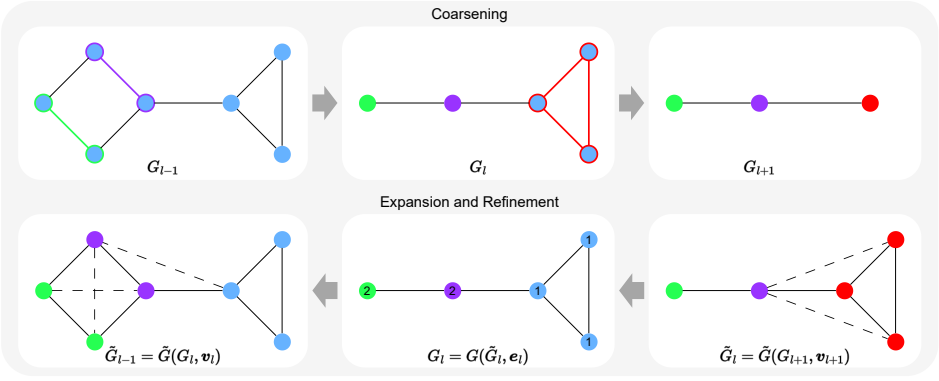


First, the model identifies which nodes need to be duplicated (here only the purple one), then the graph is expanded, and a second model decides which edges to keep

In practice, we use the diffusion framework. The ground truth for expansion is known, we noise it, and then we train a model to denoise it. The generation starts with Gaussian noise and the model denoise towards a coherent choice for expansion

Efficient Graph Generation

Method



Single step schematic representation of the efficient model. The upper row delineates two sequential coarsening steps, using color differentiation to denote the contraction sets. Colors distinguish membership within expansion sets while dashed lines indicate edges to be removed. **Source:** [Bergmeister et al.(2024)]

- This model generates graphs based on iterative local expansion, where a graph is progressively built by adding nodes and edges in a localized manner
- Unlike traditional methods that struggle with large graphs due to quadratic complexity, this approach achieves subquadratic runtime
- The model first establishes a global structure and then refines local details through a stepwise expansion process, avoiding the need to model the entire joint distribution over all node pairs
- The denoising is performed with a GNN

Activity: Pitch Your Graph Machine Learning Startup

Please form teams of 3 or 4 students

- Each team prepares a 45–60 second pitch for your graph machine learning startup. Focus on technical feasibility - convince me your idea can actually work
- You can use any concept/model you learned during the course
- You may want to address:
  - What real-world problem are you solving?
  - What graph properties matter most?
  - Which graph machine learning model will you use, and why?
  - What is the biggest technical challenge, and how will you overcome it?
- You have 10 minutes to prepare your pitch
- If you want to present your pitch, you can write on a piece of paper your names and give it to me before leaving (participation). Otherwise, some teams will be randomly selected to present

Main Takeaways

- Many real-world problems (*e.g.* drug discovery, social networks) require generating new graphs
- A good generative model should produce graphs that resemble real-world networks in key structural properties
- Erdös-Rényi graphs assume edges form randomly but fail to capture real-world degree distributions and clustering
- Deep generative models learn graph structures directly from data rather than relying on predefined rules

Main Takeaways

- GraphRNN models graphs as sequences of node and edge additions, using RNNs to predict connections
- GraphRNN is computationally expensive
- Newer models avoid the sequence problem by generating graphs iteratively via local expansion
- Coarsening reduces graphs to a simple structure, then expands them back while preserving key properties

Time to Evaluate your Professor: Wooclap

Thank you!

Jhony H. Giraldo: jhony.giraldo@telecom-paris.fr
https://jhonygiraldo.github.io/

References I


Andreas Bergmeister et al.
Efficient and scalable graph generation through iterative local expansion.
In *International Conference on Learning Representations*, 2024.


P. Erdős and A. Rényi.
On random graphs i.
Publ. math. debrecen, 6(290-297):18, 1959.


Andreas Loukas.
Graph reduction with spectral and cut guarantees.
Journal of Machine Learning Research, 20(116):1–42, 2019.

References II


Keyu Tian et al.
Visual autoregressive modeling: Scalable image generation via next-scale prediction.
Advances in Neural Information Processing Systems, 2024.


Jiaxuan You et al.
GraphRNN: Generating realistic graphs with deep auto-regressive models.
In *International Conference on Machine Learning*, 2018.