

Deep Graph Generation

Motivation for Graph Generation

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



Image credit: [Medium](#)

Social Networks

Economic Networks

Communication Networks

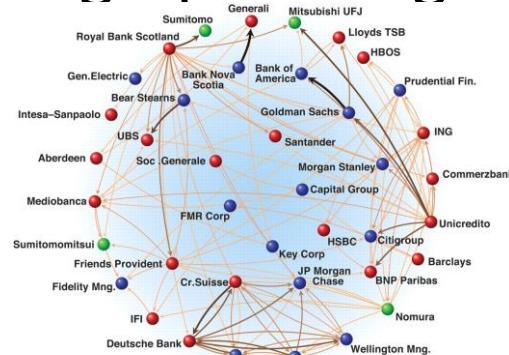


Image credit: [Science](#)



Image credit: [Lumen Learning](#)

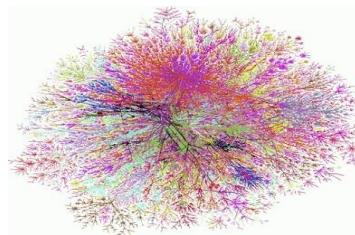
- But how are these graphs generated?

The Problem: Graph Generation

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

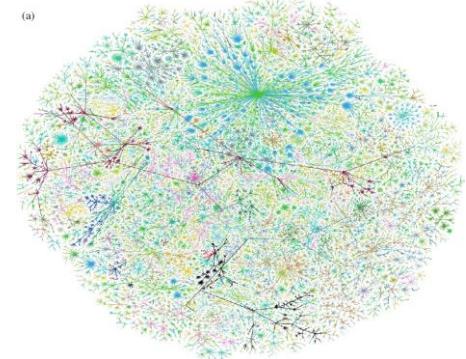
Graph
Generative
Model

Generate



Synthetic graph

which is
similar to



Real graph

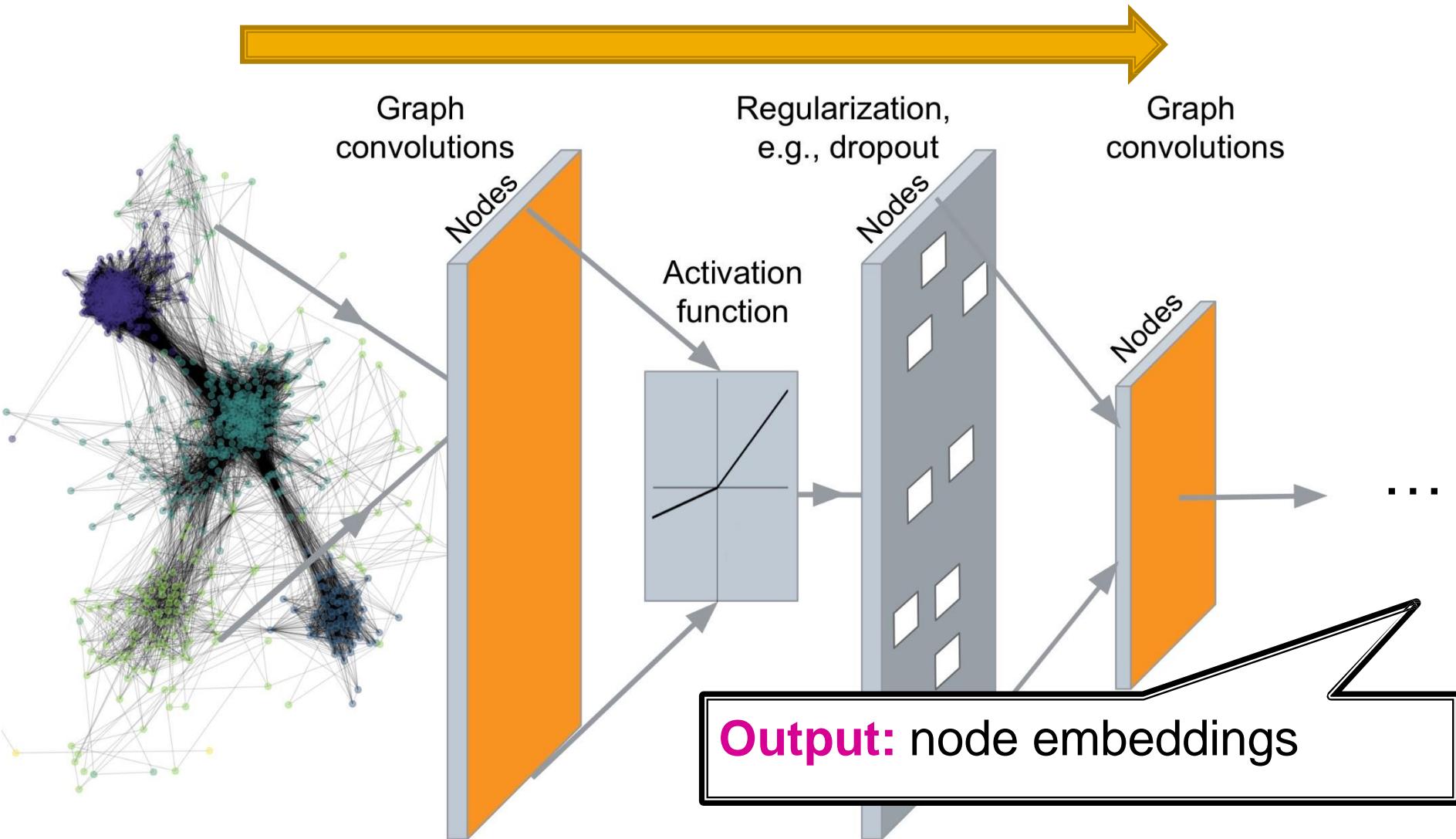
- **Applications:**

- Drug discovery, material design
- Social network modeling

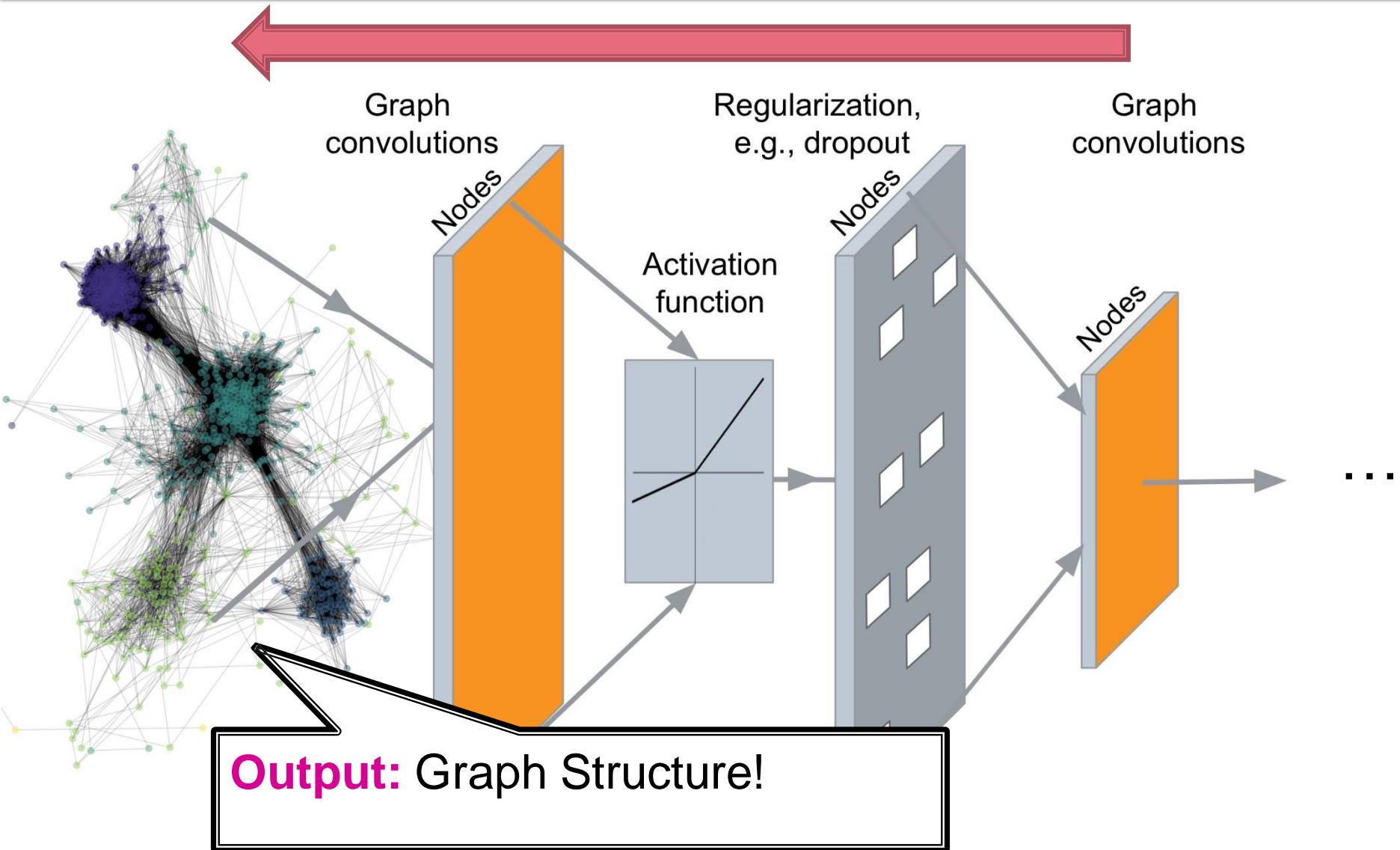
Why Do We Study Graph Generation

- **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

So far: Deep Graph Encoders



Today: Deep Graph Decoders



Graph Generation Tasks

Task 1: Realistic graph generation

- Generate graphs that are similar to a given set of graphs [Focus of this lecture]

Task 2: Goal-directed graph generation

- Generate graphs that optimize given objectives/constraints
 - E.g., Drug molecule generation/optimization

Graph Generative Models

- Given: Graphs sampled from $p_{data}(G)$
- Goal:
 - Learn the distribution $p_{model}(G)$
 - Sample from $p_{model}(G)$

$p_{data}(G)$



Learn &
Sample



$p_{model}(G)$



Generative Models Basics

- 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 $\mathbf{x}_i \sim p_{data}(\mathbf{x})$
 - $p_{model}(\mathbf{x}; \theta)$ is the **model**, parametrized by θ , that we use to approximate $p_{data}(\mathbf{x})$
- **Goal:**
 - **(1) Make $p_{model}(\mathbf{x}; \theta)$ close to $p_{data}(\mathbf{x})$ (Density estimation)**
 - **(2) Make sure we can sample from $p_{model}(\mathbf{x}; \theta)$ (Sampling)**
 - We need to generate examples (graphs) from $p_{model}(\mathbf{x}; \theta)$

Generative Models Basics

(1) Make $p_{model}(x; \theta)$ close to $p_{data}(x)$

- Key Principle: **Maximum Likelihood**
- Fundamental approach to modeling distributions

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

- Find parameters θ^* , such that for observed data points $x_i \sim p_{data}$ the $\sum_i \log p_{model}(x_i; \theta^*)$ has the highest value, among all possible choices of θ
 - That is, find the model that is most likely to have generated the observed data x

Generative Models Basics

(2) Sample from $p_{model}(x; \theta)$

- **Goal:** Sample from a complex distribution
- The most common approach:
 - (1) Sample from a simple noise distribution
$$\mathbf{z}_i \sim N(0, 1)$$
 - (2) Transform the noise \mathbf{z}_i via $f(\cdot)$
$$x_i = f(\mathbf{z}_i; \theta)$$
- Then x_i follows a complex distribution
- **Q: How to design $f(\cdot)$?**
- **A: Use Deep Neural Networks,** and train it using the data we have!

Deep Generative Models

Auto-regressive models:

- $p_{model}(x; \theta)$ is used for **both density estimation and sampling** (remember our two goals)

- Other models like Variational Auto Encoders (VAEs), Generative Adversarial Nets (GANs) have 2 or more models, each playing one of the roles
- **Idea: Chain rule.** Joint distribution is a product of conditional distributions:

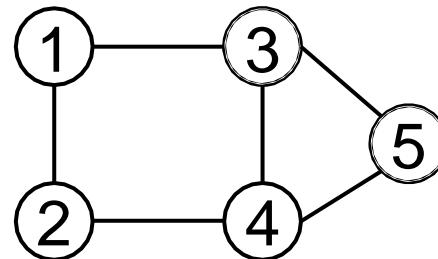
$$p_{model}(x; \theta) = \prod_{t=1}^n p_{model}(x_t | x_1, \dots, x_{t-1}; \theta)$$

- E.g., x is a vector, x_t is the t -th dimension;
 x is a sentence, x_t is the t -th word.
- **In our case:** x_t will be the t -th action (add node, add edge)

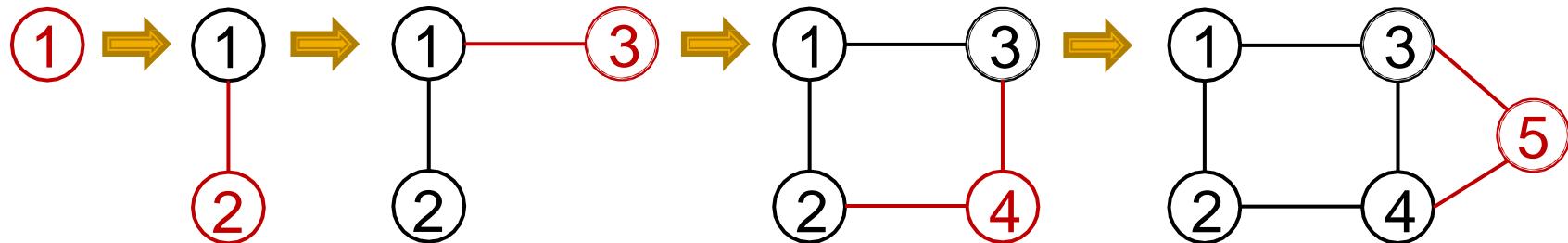
GraphRNN Idea

Generating graphs via sequentially adding nodes and edges

Graph G



Generation process S^π



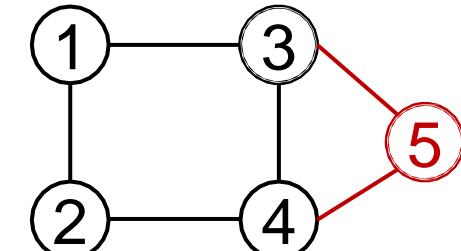
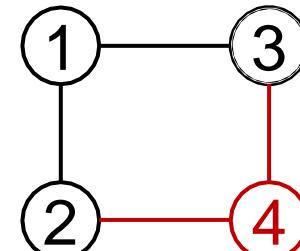
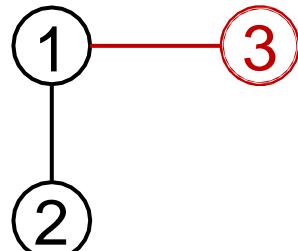
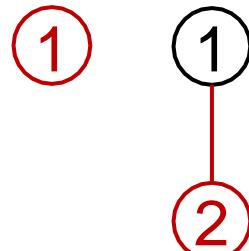
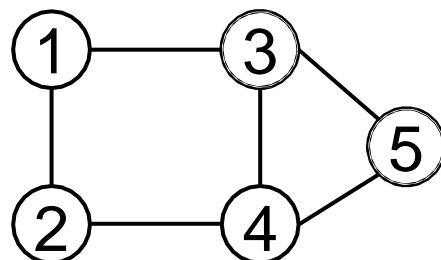
Model Graphs as Sequences

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

Graph G with
node ordering π :



Sequence S^π :



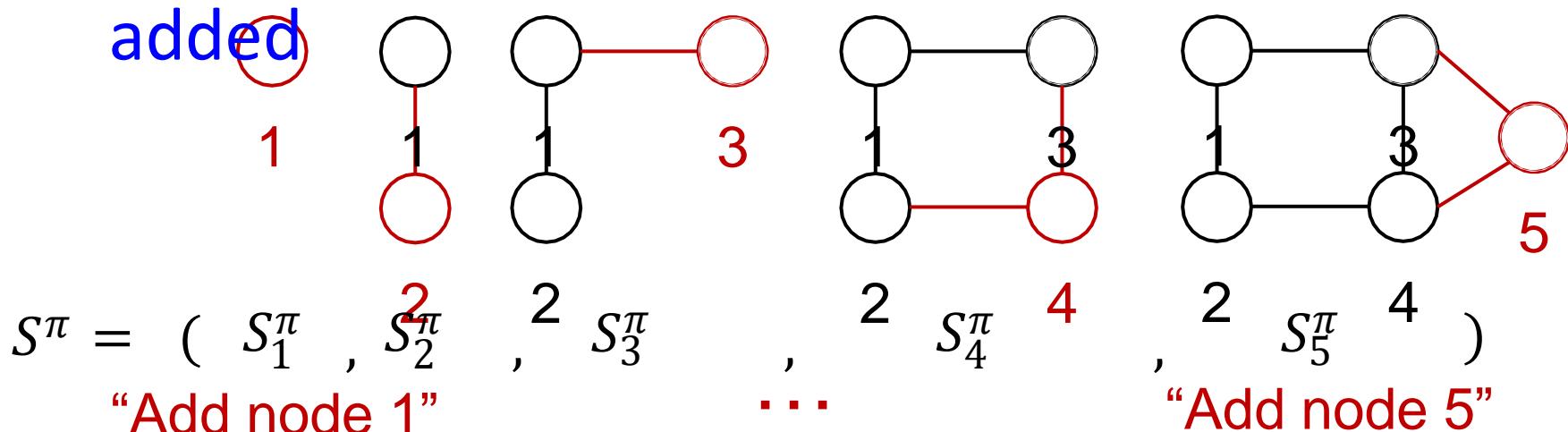
$$S^\pi = (S_1^\pi, S_2^\pi, S_3^\pi, S_4^\pi, S_5^\pi)$$

Model Graphs as Sequences

The sequence S^π has **two levels**

(S is a sequence of sequences):

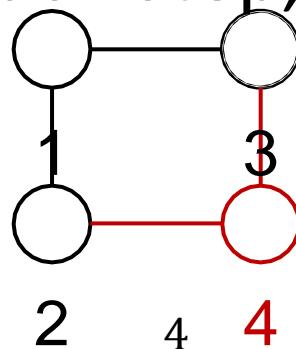
- **Node-level:** add nodes, one at a time
 - **Edge-level:** add edges between existing nodes
- **Node-level:** At each step, a **new node is added**



Model Graphs as Sequences

The sequence S^π has **two levels**:

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



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

“Not connect 4, 1” “Connect 4, 2” “Connect 4, 3”

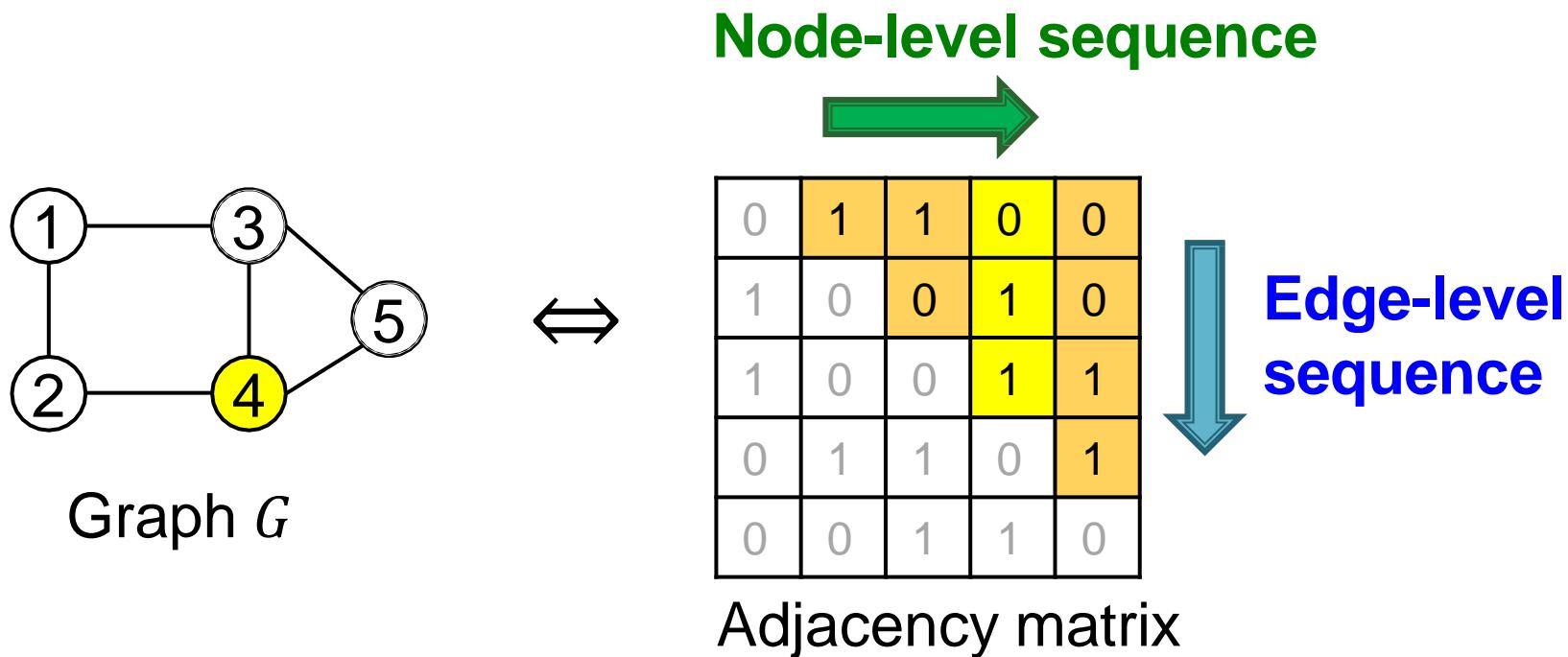
0

1

1

Model Graphs as Sequences

- Summary: A graph + a node ordering
= A sequence of sequences
- Node ordering is randomly selected (we will come back to this)

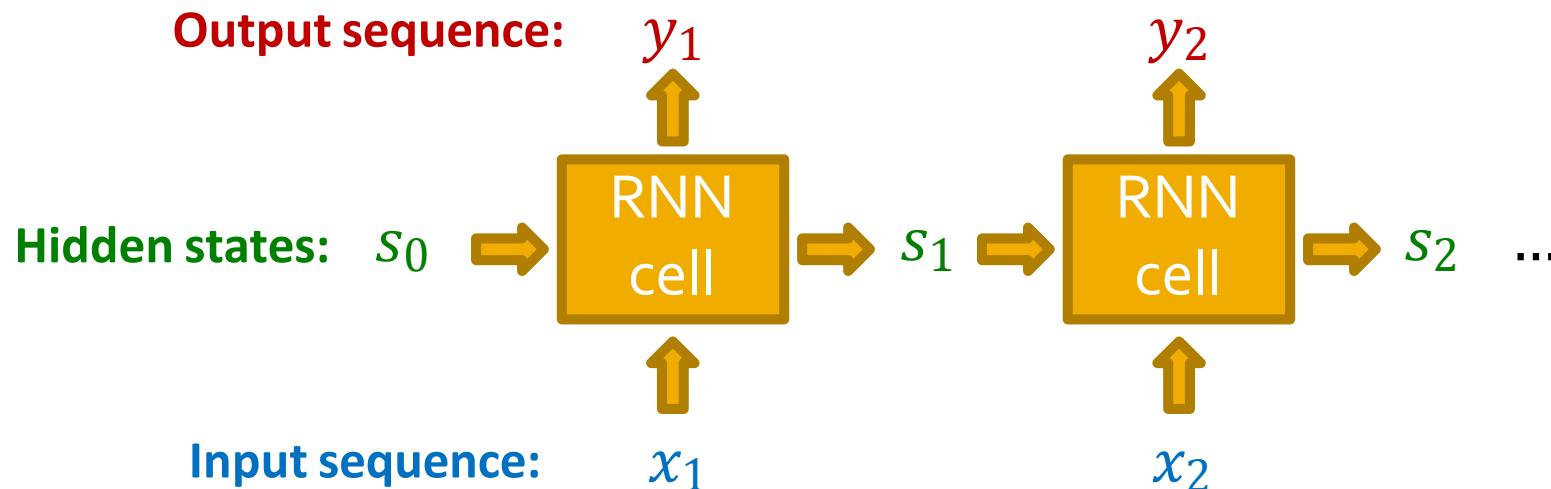


Model Graphs as Sequences

- 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: Use Recurrent Neural Networks (RNNs) to model these processes!

Background: Recurrent NNs

- 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 update is conducted via **RNN cells**



Background: Recurrent NNs

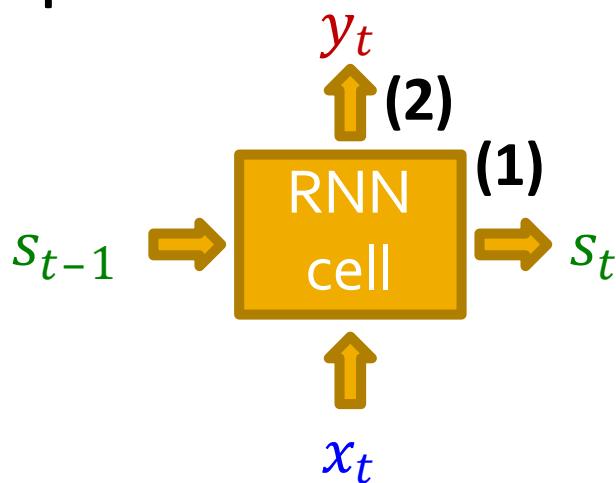
- s_t : State of RNN after step

t

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

- y_t : Output of RNN at step t

- RNN cell: W, U, V : Trainable parameters



The RNN cell:

- (1) Update hidden state:

$$s_t = \sigma(W \cdot x_t + U \cdot s_{t-1})$$

- (2) Output prediction:

$$y_t = V \cdot s_t$$

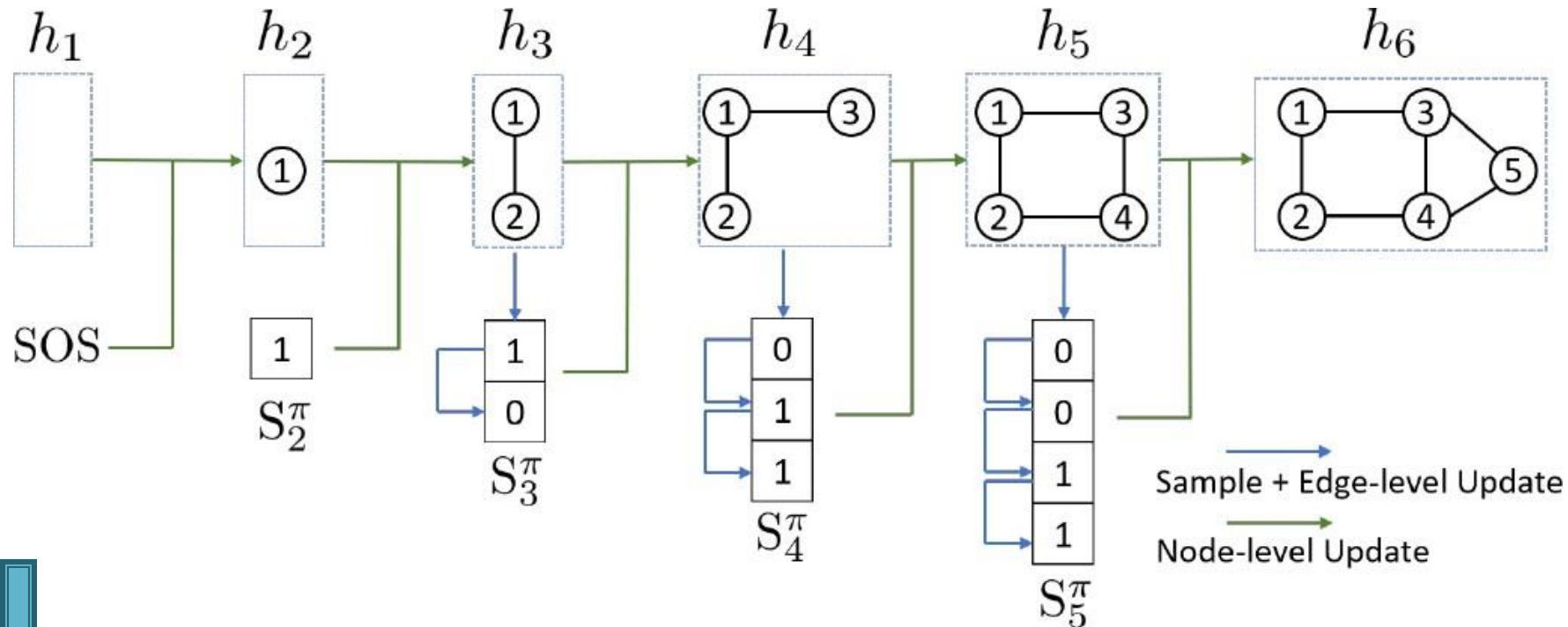
- More expressive cells: GRU, LSTM, etc.

GraphRNN: Two levels of RNN

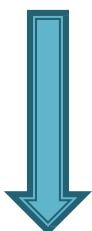
- GraphRNN has a **node-level RNN** and an **edge-level RNN**
- Relationship between the two RNNs:
 - Node-level RNN generates the initial state for edge-level RNN
 - Edge-level RNN sequentially predict if the new node will connect to each of the previous node

GraphRNN: Two levels of RNN

Node-level RNN generates the initial state for edge-level RNN

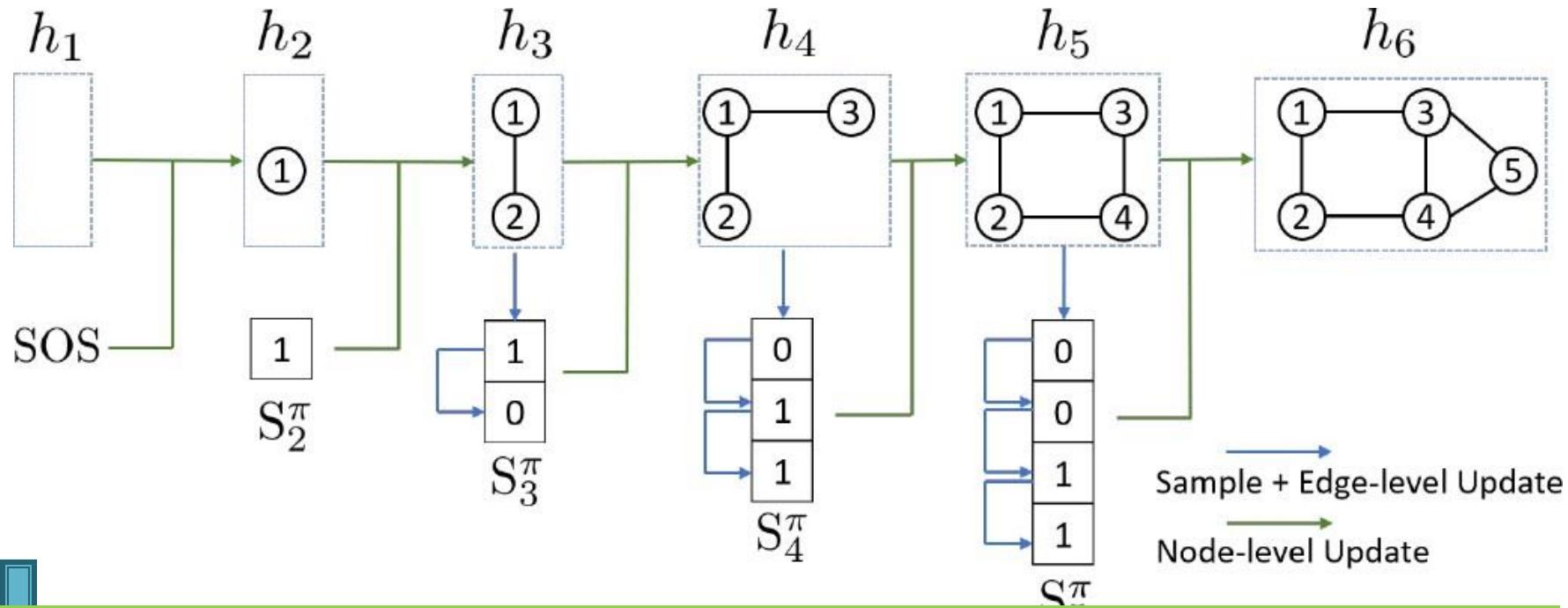


Edge-level RNN sequentially predict if the new node will connect to each of the previous node



GraphRNN: Two levels of RNN

Node-level RNN generates the initial state for edge-level RNN



Next: How to generate a sequence with RNN?

results

RNN for Sequence Generation

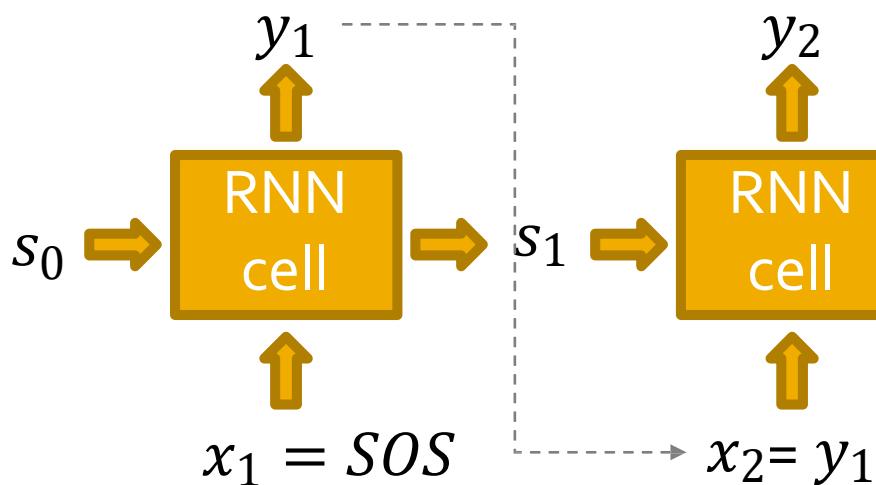
- **Q:** How to use RNN to generate sequences?
- **A:** Let $x_{t+1} = y_t$ (Use the previous output as input)

- **Q:** How to initialize the input sequence?
- **A:** Use **start of sequence token (SOS)** as the initial input
 - SOS is usually a vector with all zero/ones

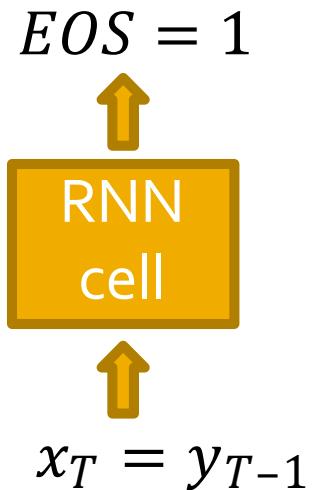
- **Q:** When to stop generation?
- **A:** Use **end of sequence token (EOS)** as an extra RNN output
 - If output EOS=0, RNN will continue generation
 - If output EOS=1, RNN will stop generation

RNN for Sequence Generation

Use the previous output as input



Stop generation



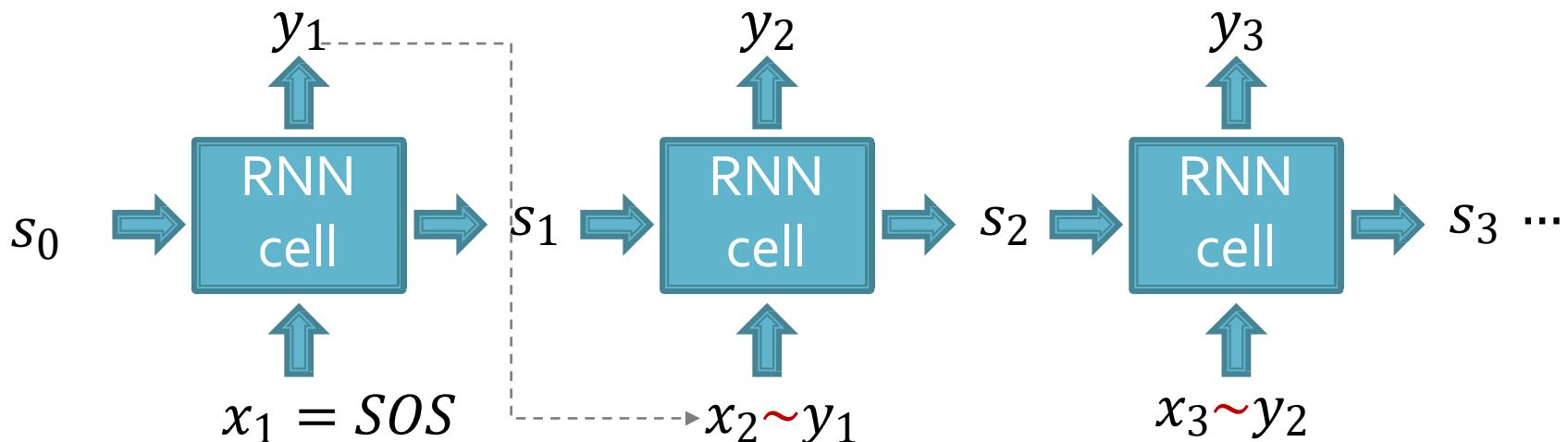
Initialize input

- This is good, but this model is deterministic

Towards Edge-Level RNN

Consider the Edge-level RNN for now.

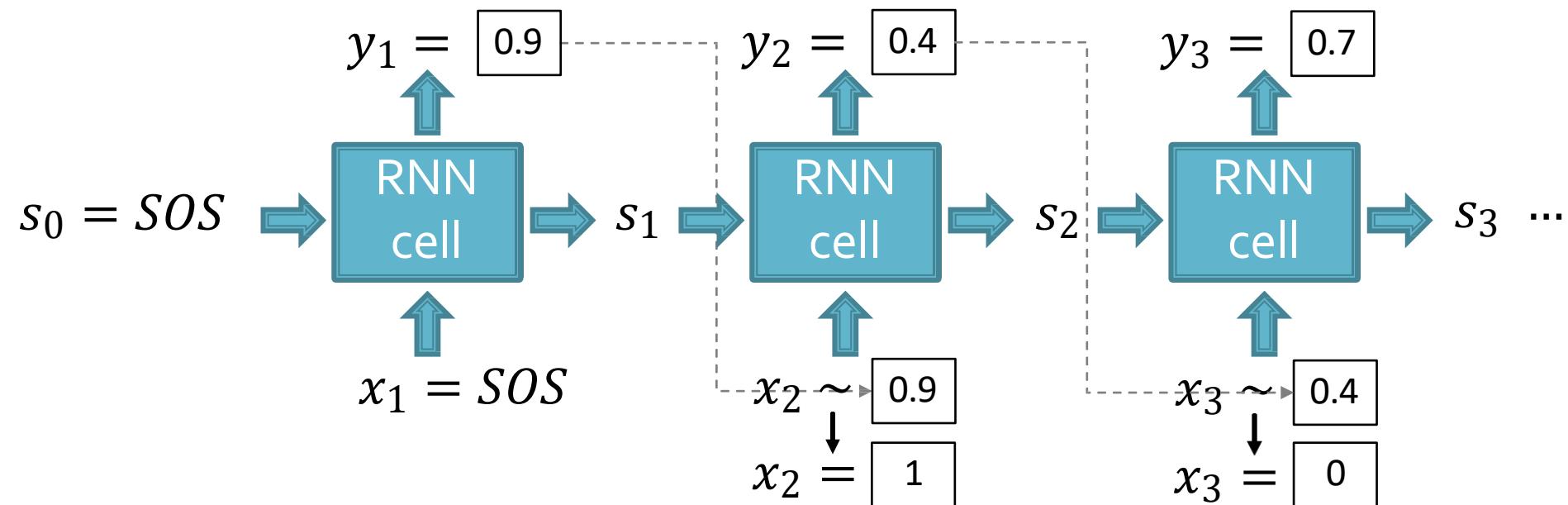
- Our goal: Model $\prod_{k=1}^n p_{model}(x_t|x_1, \dots, x_{t-1}; \theta)$
- Let $y_t = p_{model}(x_t|x_1, \dots, x_{t-1}; \theta)$
- Then we need to sample x_{t+1} from y_t : $x_{t+1} \sim y_t$
 - Each step of RNN outputs a **probability of a single edge**
 - We then sample from the distribution,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



Towards Edge-Level RNN

Suppose we already have trained the edge-level RNN

- y_t is a scalar, following a Bernoulli distribution
- \boxed{p} means value 1 has prob. p , value 0 has prob. $1 - p$



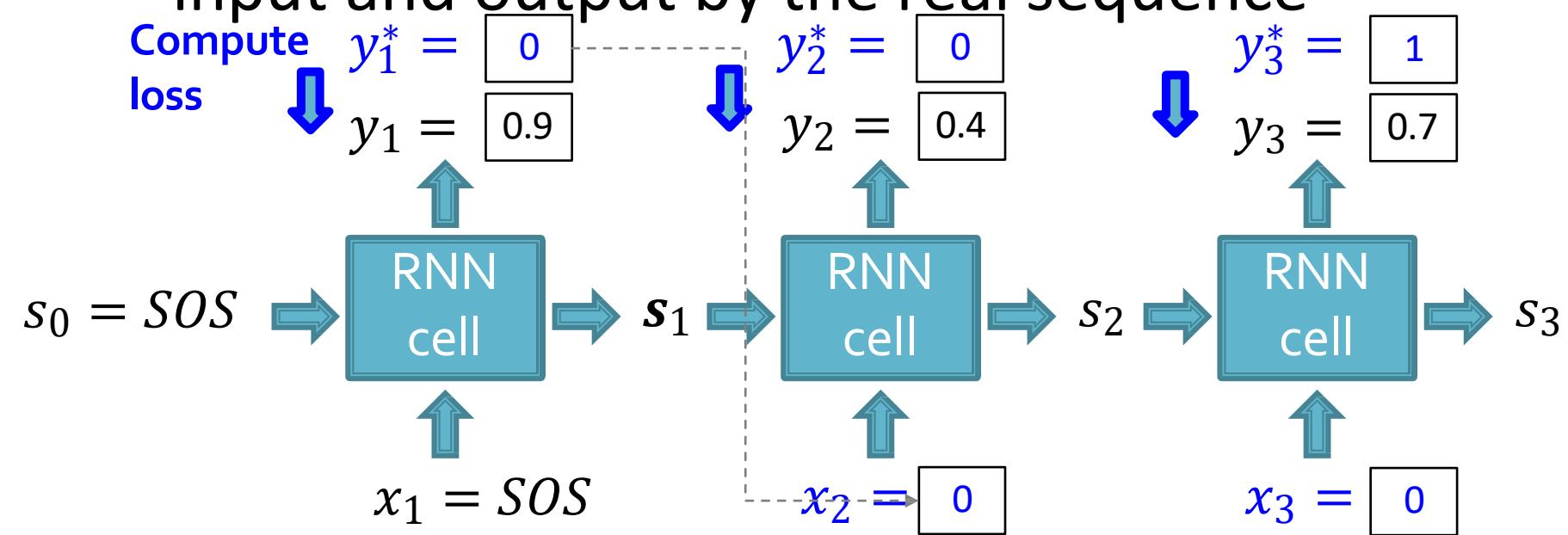
- How do we use training data x_1, x_2, \dots, x_n ?

Edge-Level RNN at Training Time

Training the model:

- We observe a sequence y^* of edges $[0,0,1,\dots]$

- Principle: Teacher Forcing** -- Replace input and output by the real sequence



Edge-Level RNN at Training Time

- Loss L : **Binary cross entropy**

- **Minimize:**

$$L = -[y_1^* \log(y_1) + (1 - y_1^*) \log(1 - y_1)]$$

Compute
loss

↓

$$\begin{aligned} y_1^* &= \boxed{0} \\ y_1 &= \boxed{0.9} \end{aligned}$$

- If $y_1^* = 1$, we minimize $-\log(y_1)$, making y_1 higher
- If $y_1^* = 0$, we minimize $-\log(1 - y_1)$, making y_1 lower
- This way, y_1 is **fitting** the data samples y_1^*
- **Reminder:** y_1 is computed by RNN, this loss will **adjust RNN parameters accordingly**, using back propagation!

Putting Things Together

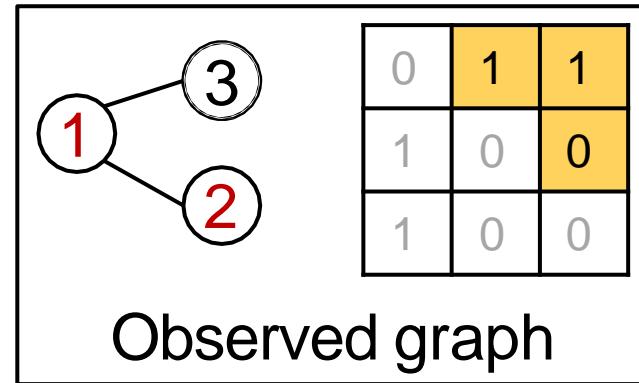
Our Plan:

- (1) Add a new node:** We run Node RNN for a step, and use its output to initialize Edge RNN
- (2) Add new edges for the new node:** We run Edge RNN to predict if the new node will connect to each of the previous nodes
- (3) Add another new node:** We use the last hidden state of Edge RNN to run Node RNN for another step
- (4) Stop graph generation:** If Edge RNN outputs EOS at step 1, we know no edges are connected to the new node. We stop the graph generation.

Put Things Together: Training

Assuming **Node 1** is in the graph

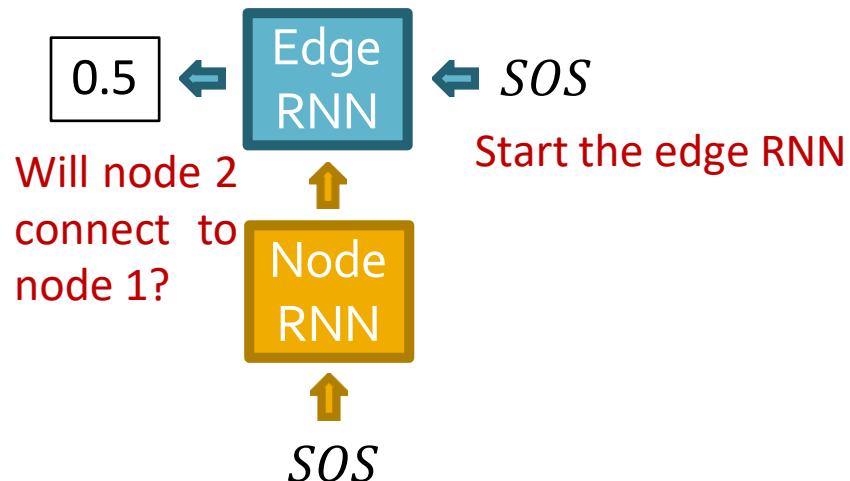
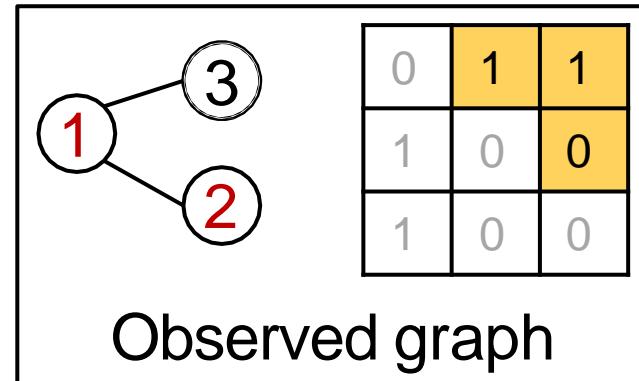
Now adding **Node 2**



Start the node RNN

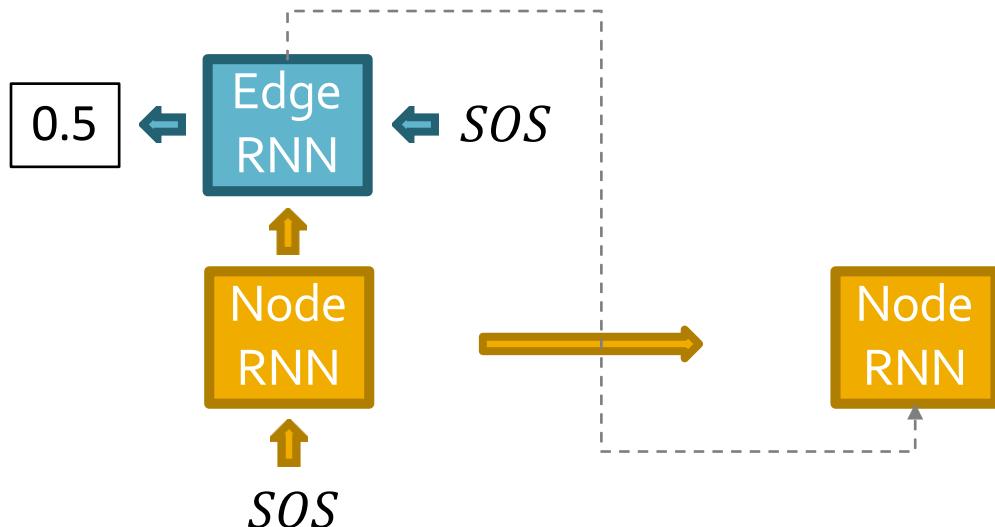
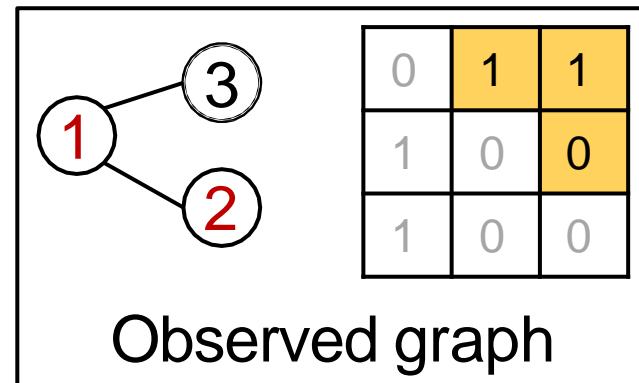
Put Things Together: Training

Edge RNN predicts how
Node 2 connects to **Node 1**



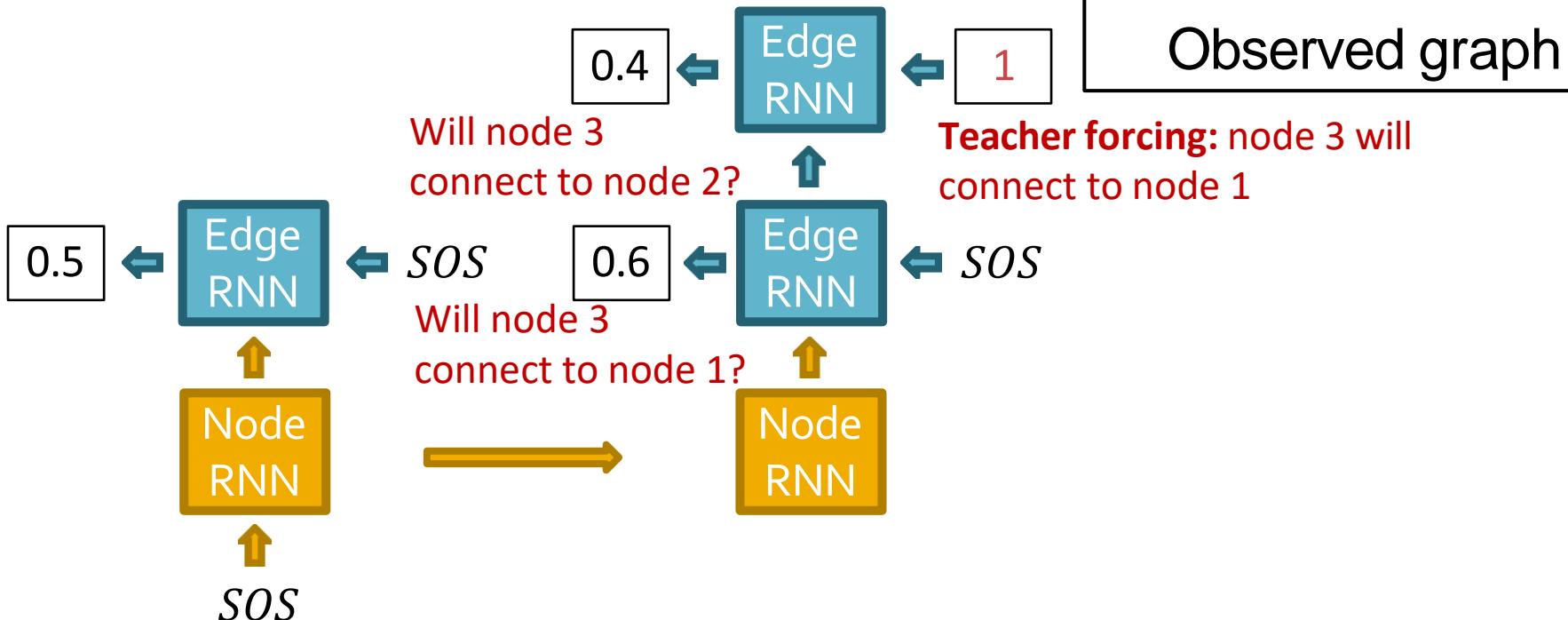
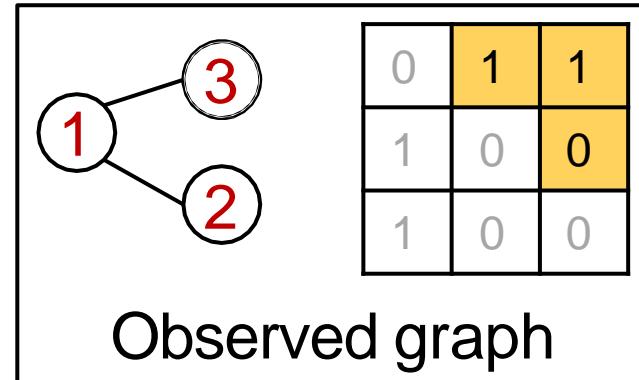
Put Things Together: Training

Update Node RNN using
Edge RNN's hidden state



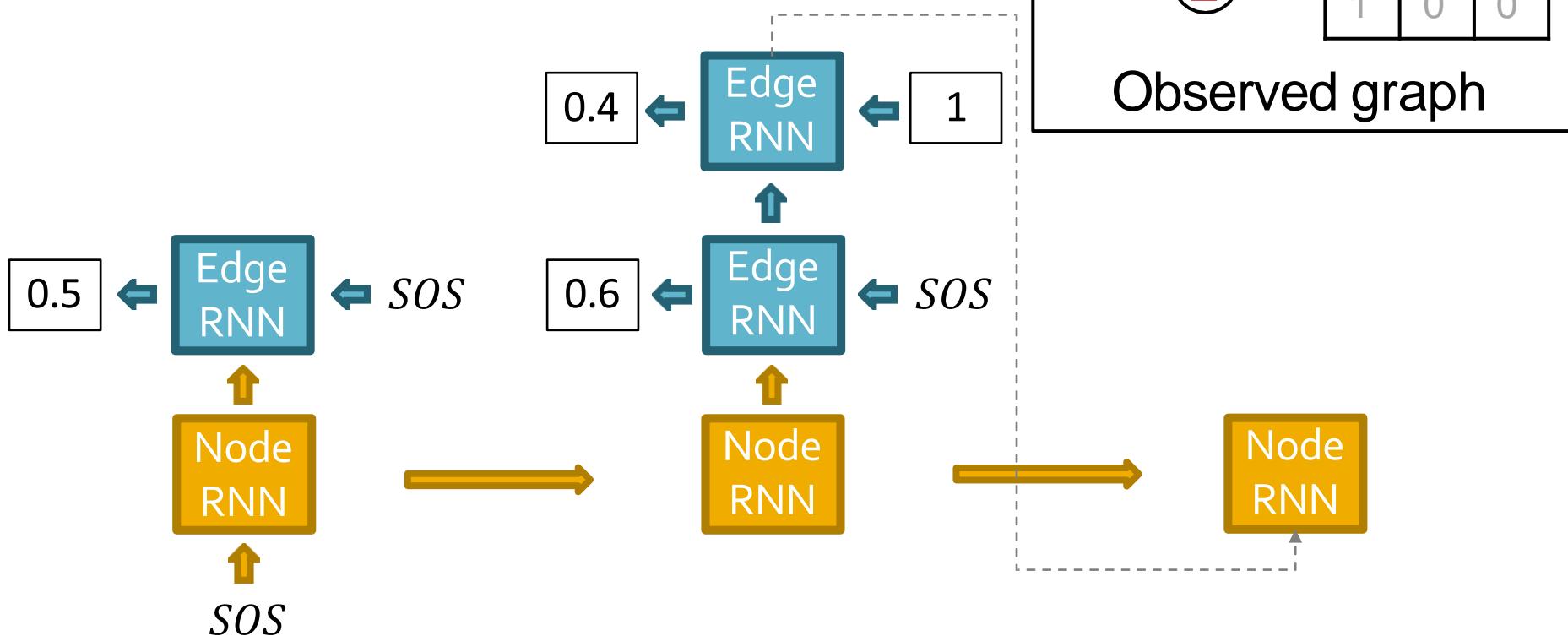
Put Things Together: Training

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



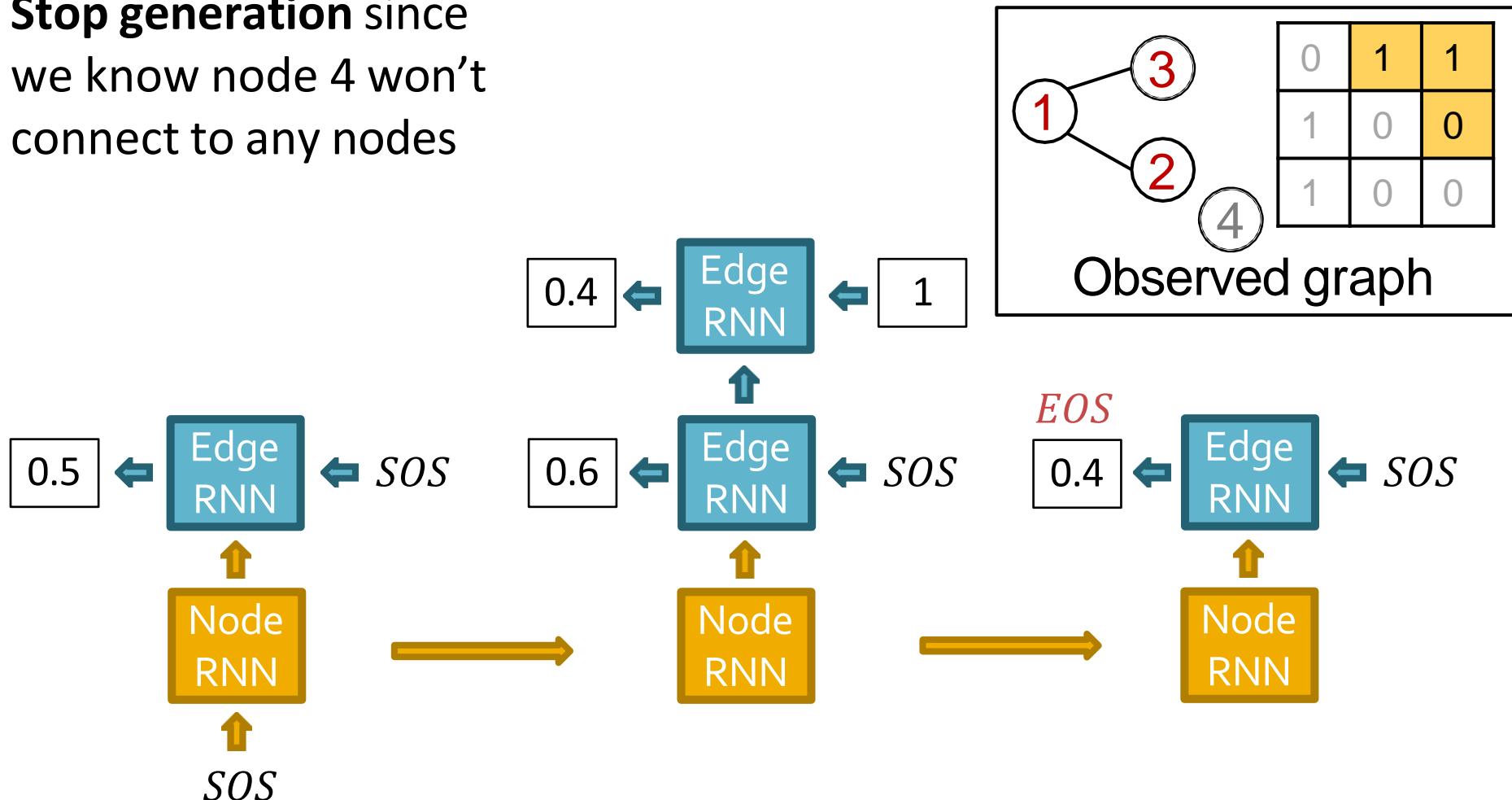
Put Things Together: Training

Update Node RNN using
Edge RNN's hidden state



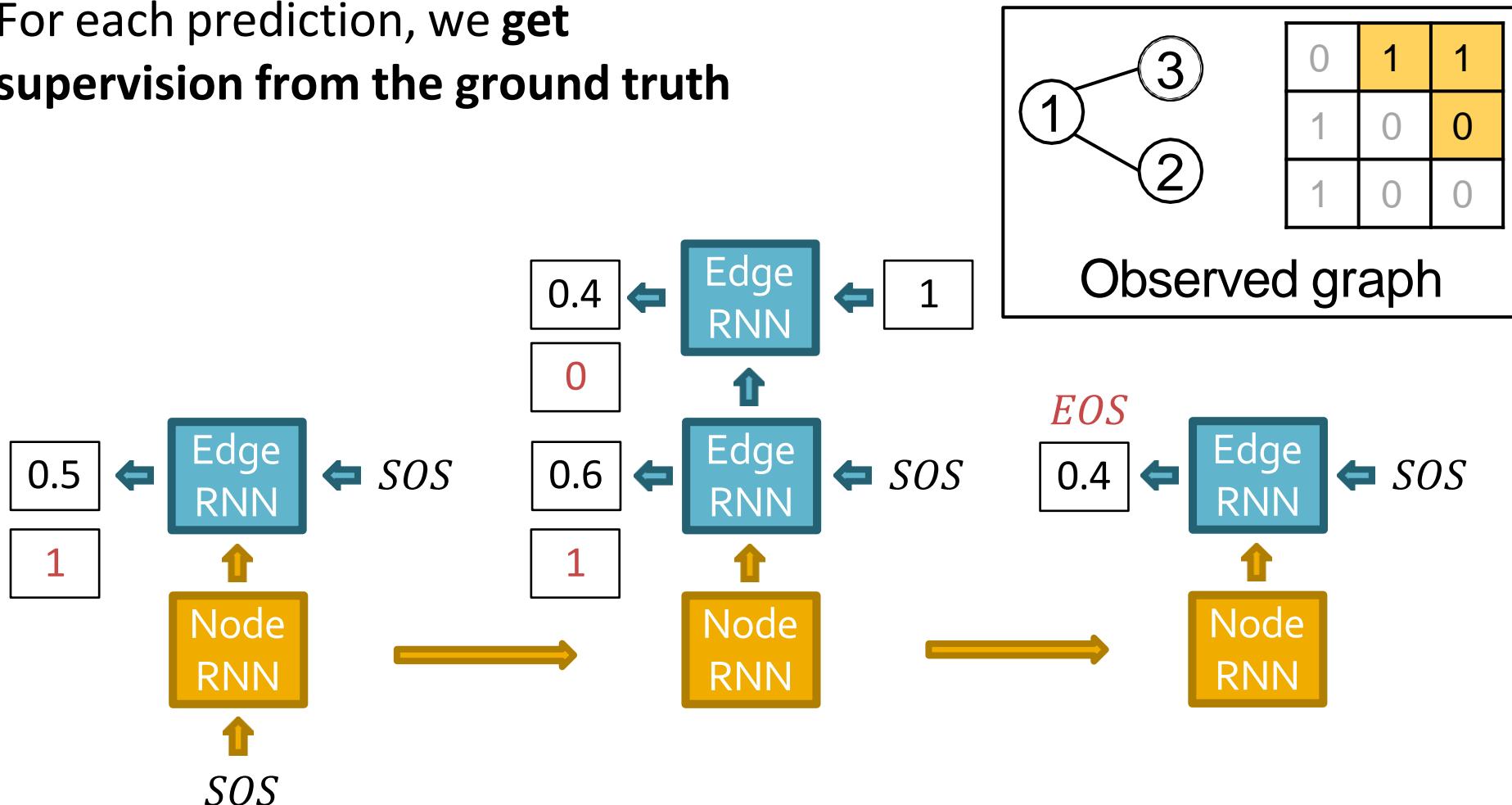
Put Things Together: Training

Stop generation since we know node 4 won't connect to any nodes



Put Things Together: Training

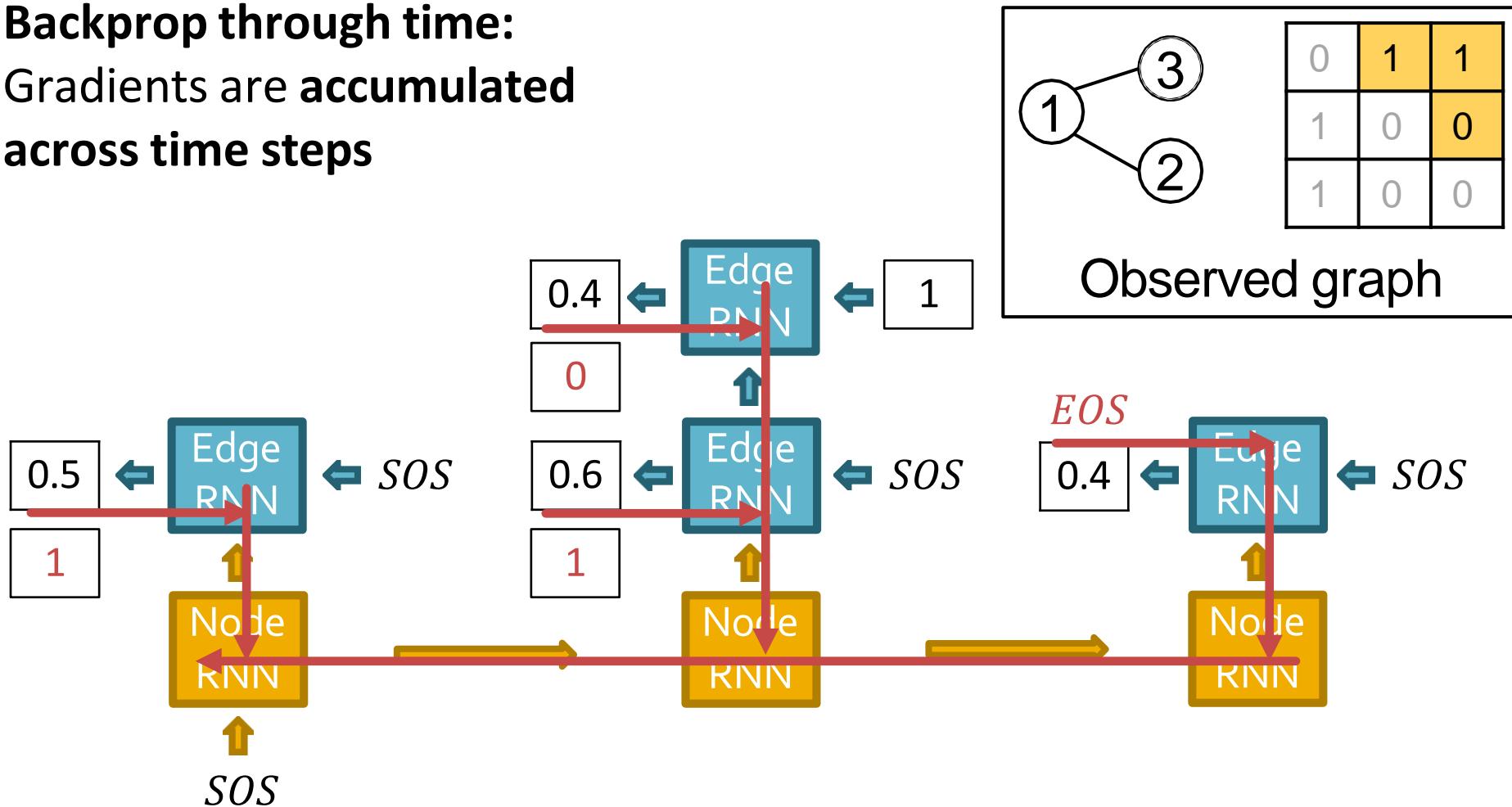
For each prediction, we get
supervision from the ground truth



Put Things Together: Training

Backprop through time:

Gradients are accumulated
across time steps

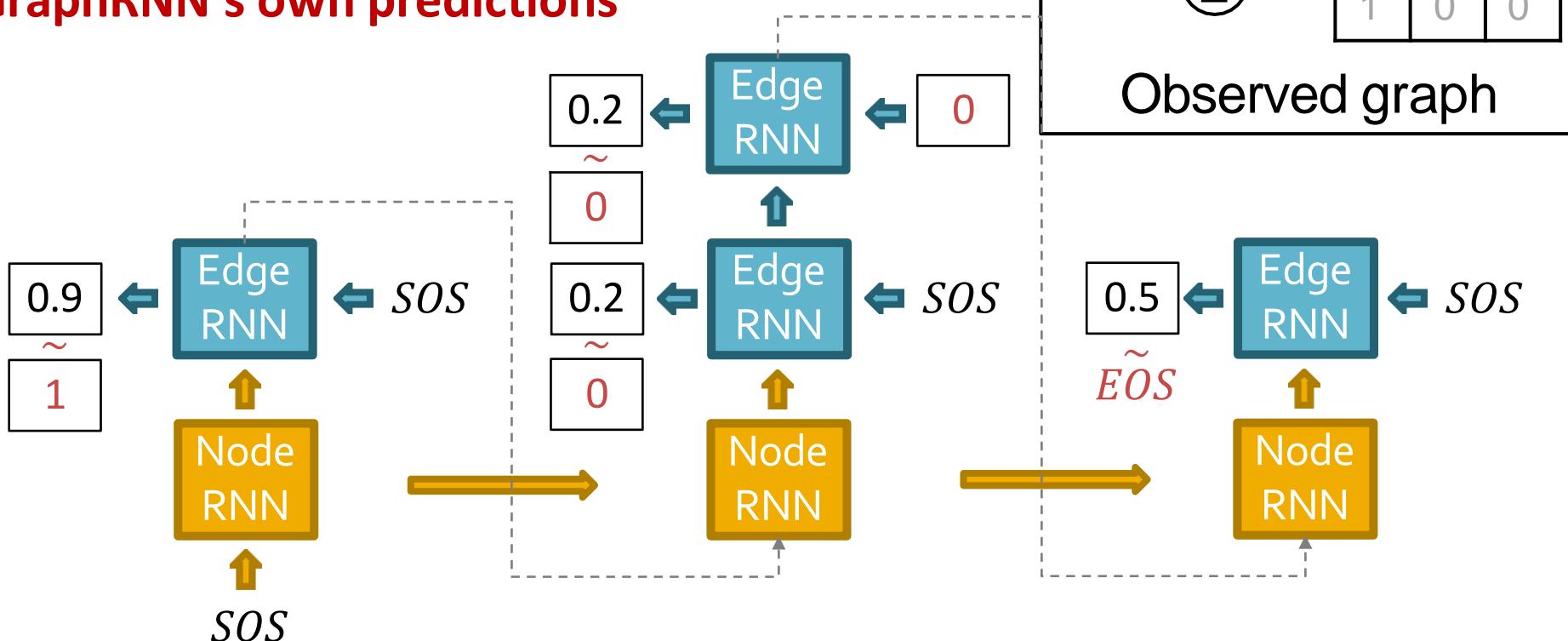


Put Things Together: Test

Test time: (1) Sample edge connectivity

based on predicted distribution

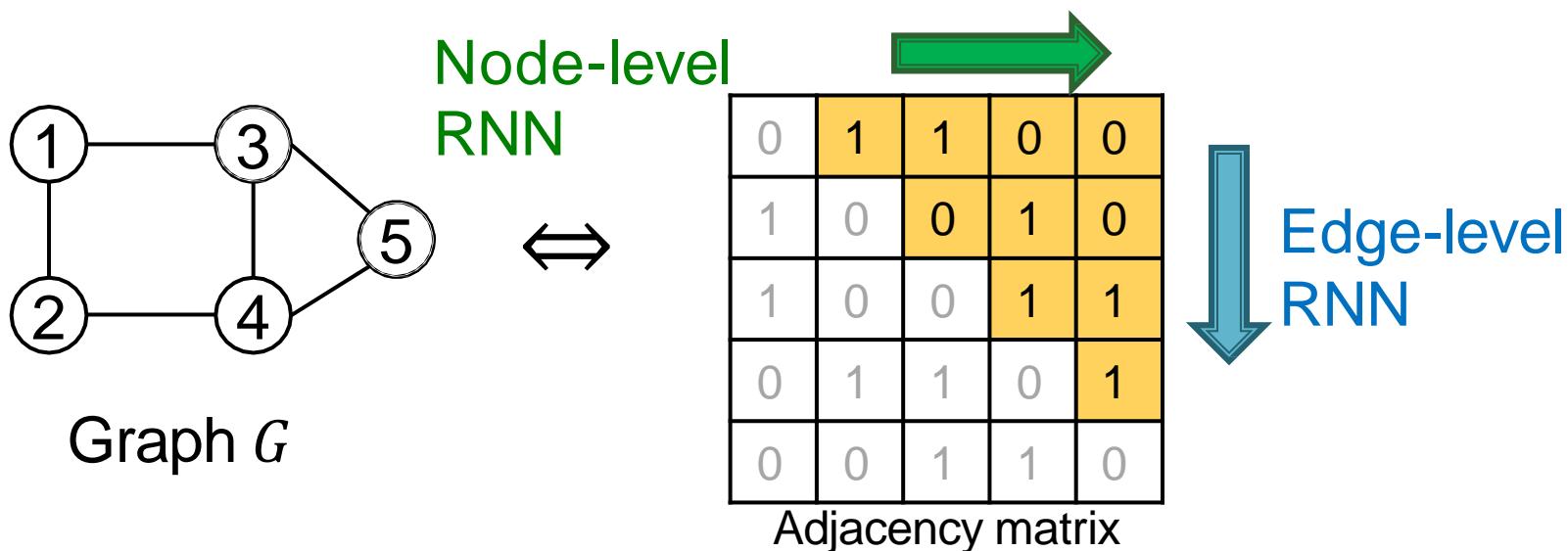
(2) Replace input at each step by
GraphRNN's own predictions



GraphRNN: Two levels of RNN

Quick Summary of GraphRNN:

- Generate a graph by generating a two-level sequence
 - Use RNN to generate the sequences
- **Next:** Making GraphRNN tractable, proper evaluation



Issue: Tractability

- Any node can connect to any prior node
- Too many steps for edge generation
 - Need to generate full adjacency matrix
 - Complex too-long edge dependencies

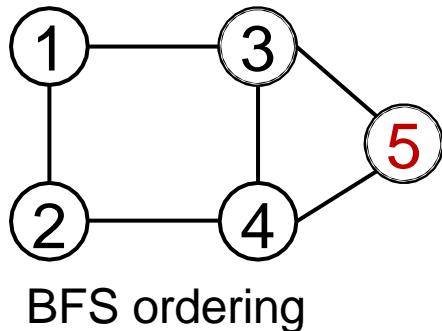
Random node ordering:

Node 5 may connect to any/all previous nodes

How do we limit this complexity?

Solution: Tractability via BFS

■ Breadth-First Search node ordering



“Recipe” to generate the left graph:

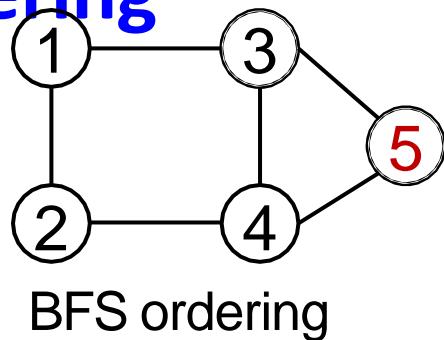
- Add node 1
- Add node 2
- Connect 2 with 1
- Add node 3
- Connect 3 with 1
- Add node 4
- Connect 4 with 3 and 2

■ BFS node ordering:

- Since Node 4 doesn't connect to Node 1
- We know all Node 1's neighbors have already been traversed
- Therefore, Node 5 and the following nodes will never connect to node 1
- We only need memory of 2 “steps” rather than $n - 1$ steps

Solution: Tractability via BFS

■ Breadth-First Search node ordering



BFS node ordering: Node 5 will never connect to node 1
(only need memory of 2 “steps” rather than $n - 1$ steps)

■ Benefits:

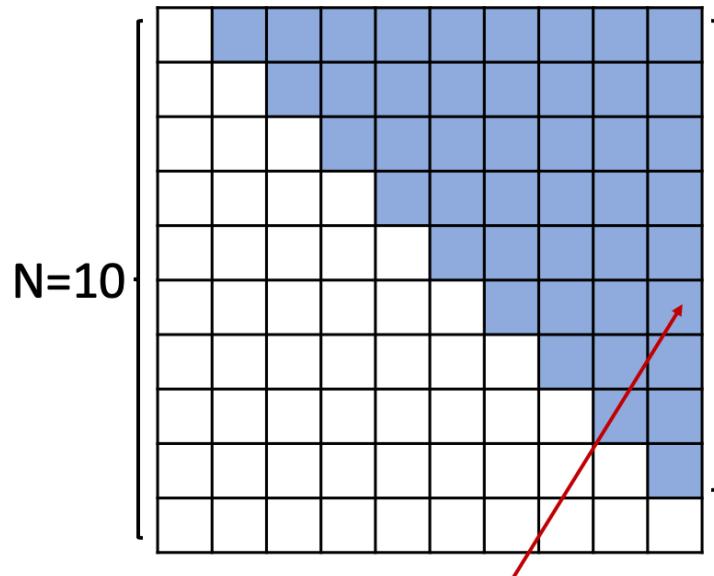
- Reduce possible node orderings
 - From $O(n!)$ to number of distinct BFS orderings
- Reduce steps for edge generation
 - Reducing number of previous nodes to look at

Solution: Tractability via BFS

- BFS reduces the number of steps for edge generation

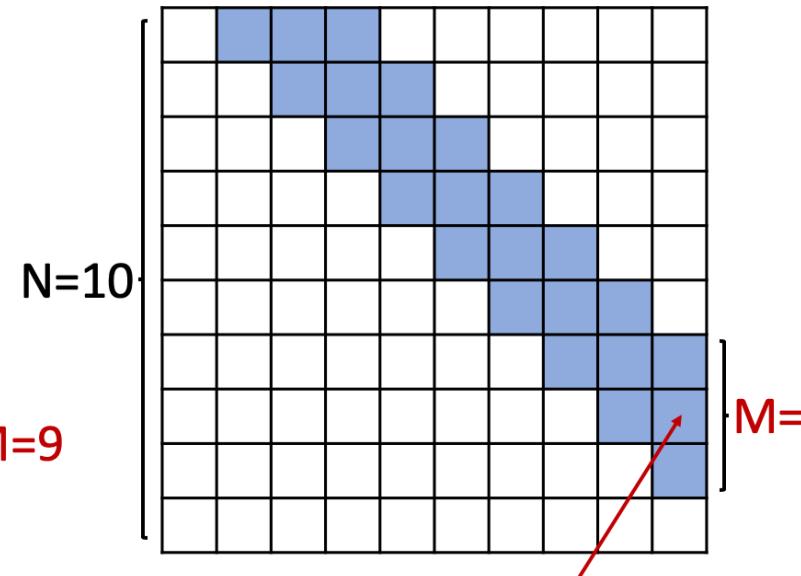
Adjacency matrices

Without BFS ordering



Connectivity with
All Previous nodes

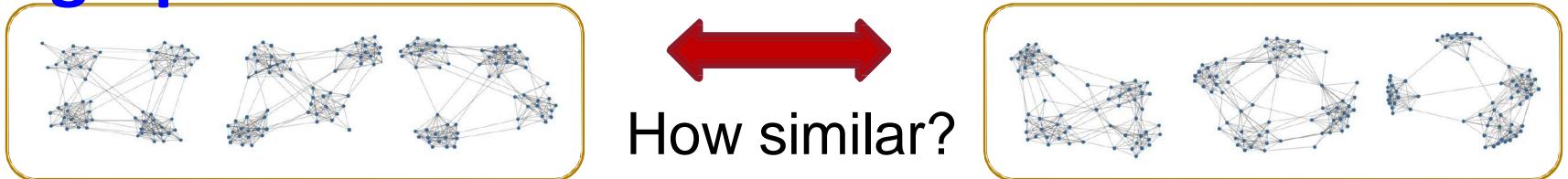
With BFS ordering



Connectivity only with
nodes in the BFS frontier

Evaluating Generated Graphs

- Task: Compare two sets of graphs

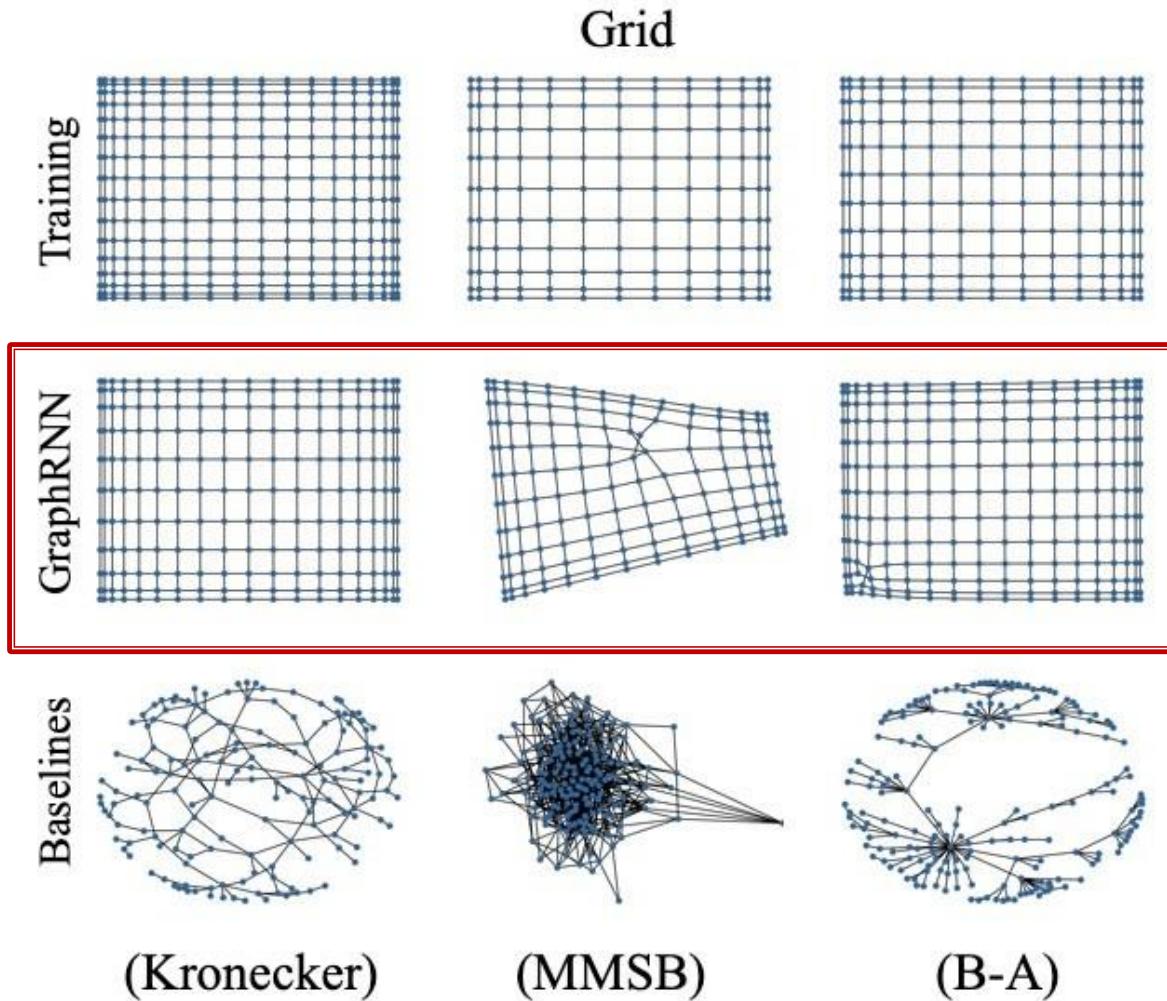


- Goal: Define similarity metrics for graphs

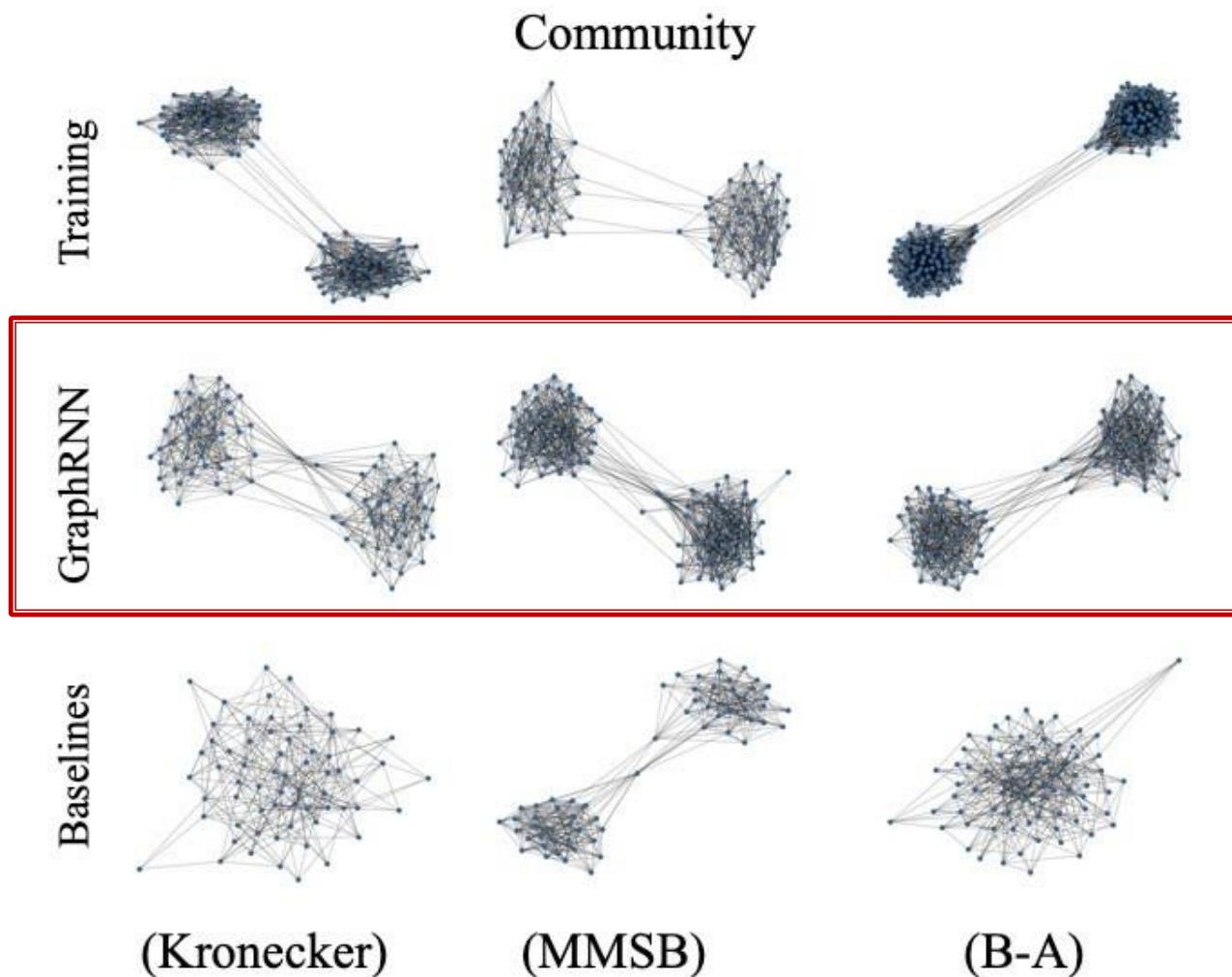
- Solution

- (1) Visual similarity
- (2) Graph statistics similarity

(1) Visual Similarity

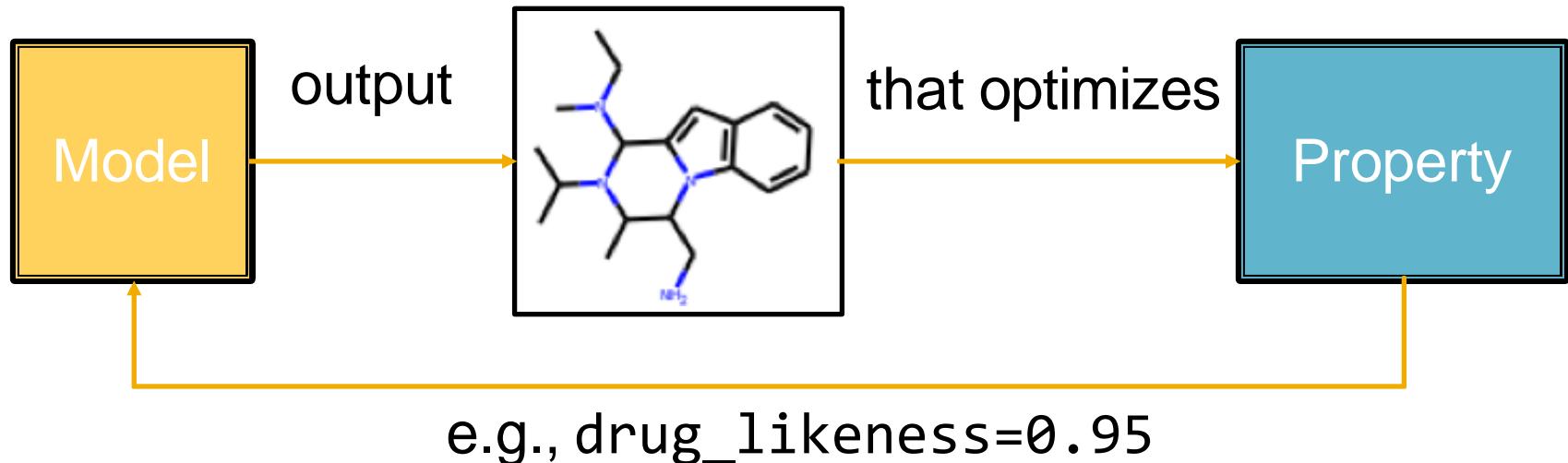


(1) Visual Similarity



Application: Drug Discovery

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



Goal-Directed Graph Generation

Generating graphs that:

- Optimize a given objective (High scores)
 - e.g., drug-likeness
- Obey underlying rules (Valid)
 - e.g., chemical validity rules
- Are learned from examples (Realistic)
 - Imitating a molecule graph dataset
 - We have just covered this part

The Hard Part:

Generating graphs that:

- Optimize a given objective (High scores)
- Including a “Black-box” to Graph Generation:
Objectives like drug-likeness are governed by
- physical law which is assumed to be
 - Covered this part when introducing GraphRNN

Idea: Reinforcement Learning

- A ML agent **observes** the environment, takes an **action** to interact with the environment, and receives positive or negative **reward**
- The agent then **learns from this loop**
- **Key idea:** Agent can directly learn from environment, which is a **blackbox** to the agent



Solution: GCPN

Graph Convolutional Policy Network (GCPN)

combines **graph representation + RL**

Key component of GCPN:

- **Graph Neural Network** captures graph structural information
- **Reinforcement learning** guides the generation towards the desired objectives
- **Supervised training** imitates examples in given datasets

GCPN vs. GraphRNN

■ Commonality of GCPN & GraphRNN:

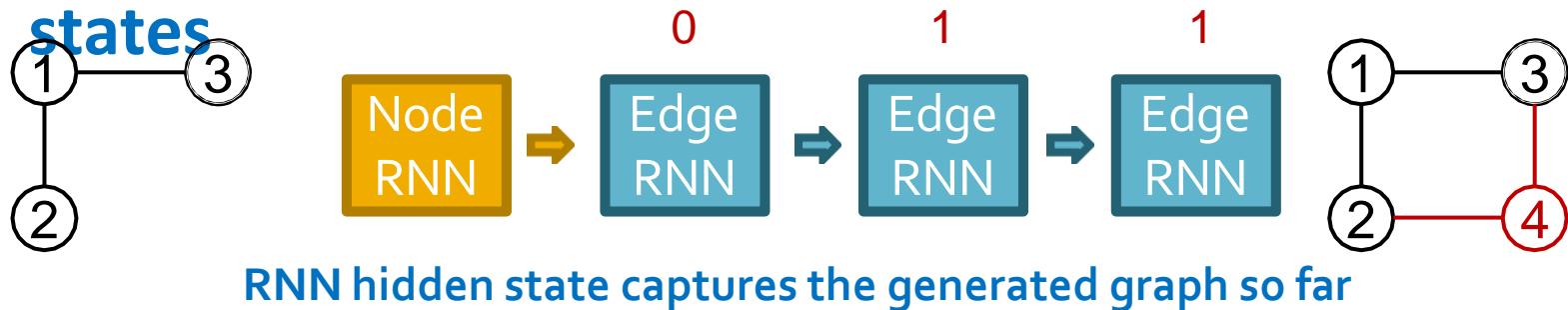
- Generate graphs sequentially
- Imitate a given graph dataset

■ Main Differences:

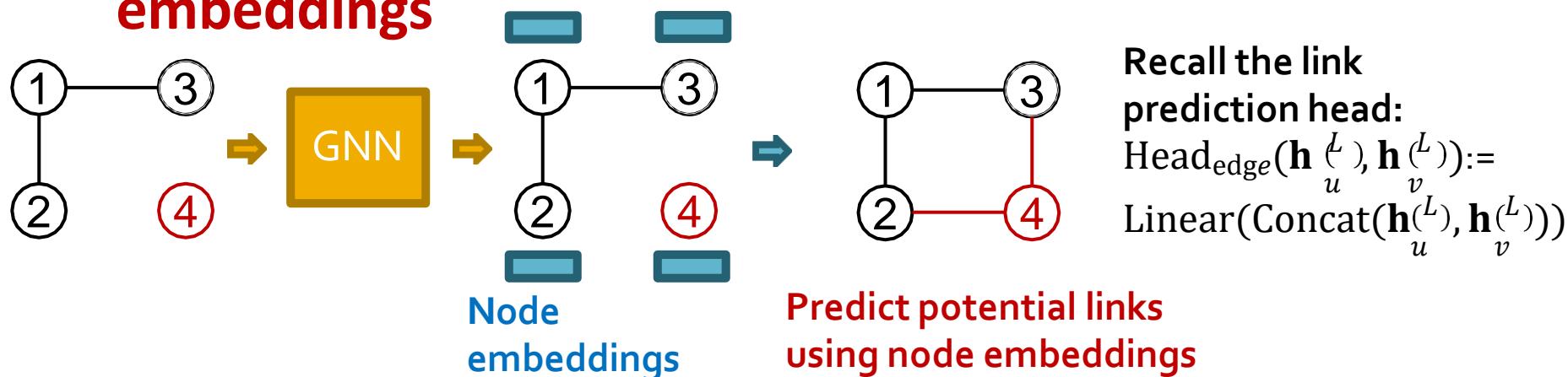
- GCPN uses **GNN** to predict the generation action
 - **Pros:** GNN is more expressive than RNN
 - **Cons:** GNN takes longer time to compute than RNN
- GCPN further uses **RL** to direct graph generation to our goals
 - RL enables goal-directed graph generation

GCPN vs. GraphRNN

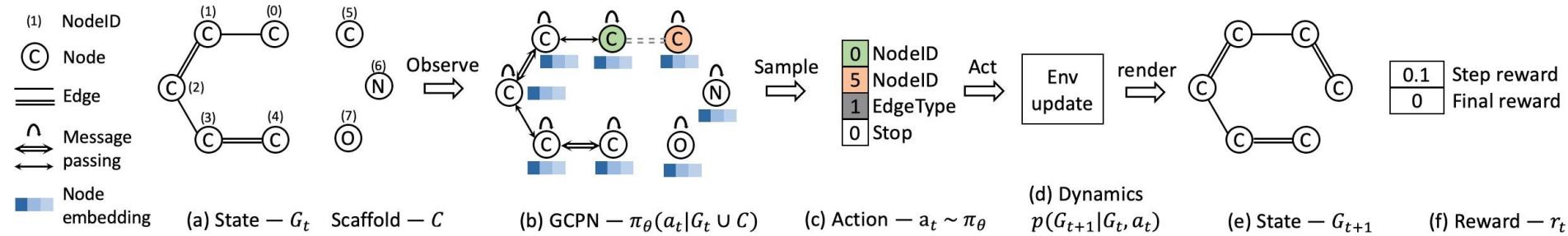
- Sequential graph generation
- GraphRNN: predict action based on **RNN hidden**



- GCPN: predict action based on **GNN node embeddings**

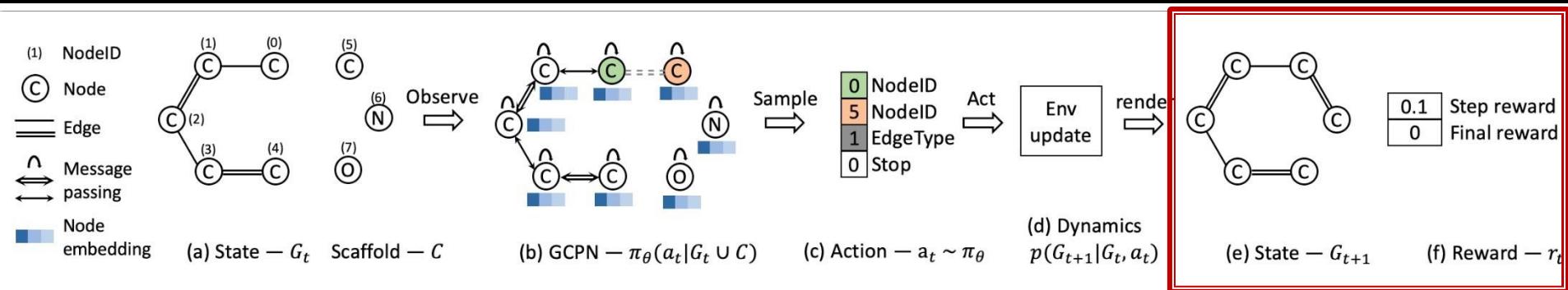


Overview of GCPN



- **(a) Insert nodes**
- **(b,c) Use GNN to predict which nodes to connect**
- **(d) Take an action (check chemical validity)**
- **(e, f) Compute reward**

How Do We Set the Reward?



■ Step reward: Learn to take valid action

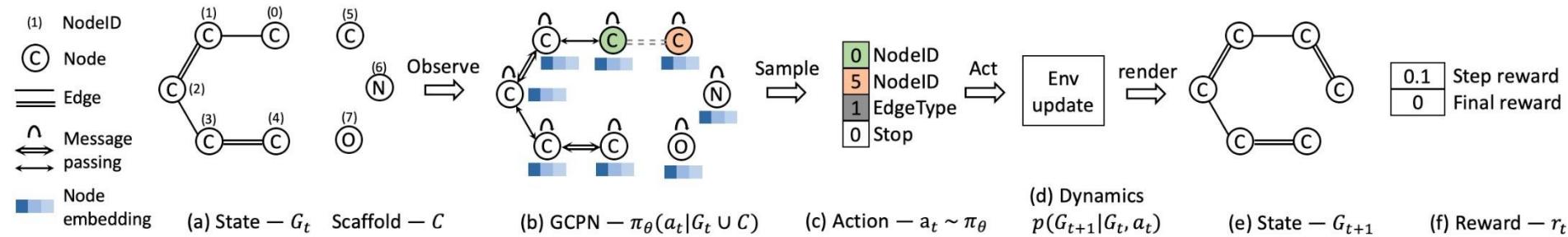
- At each step, assign small positive reward for valid action

■ Final reward: Optimize desired properties

- At the end, assign positive reward for high desired property

Reward = Final reward + Step reward

How Do We Train?



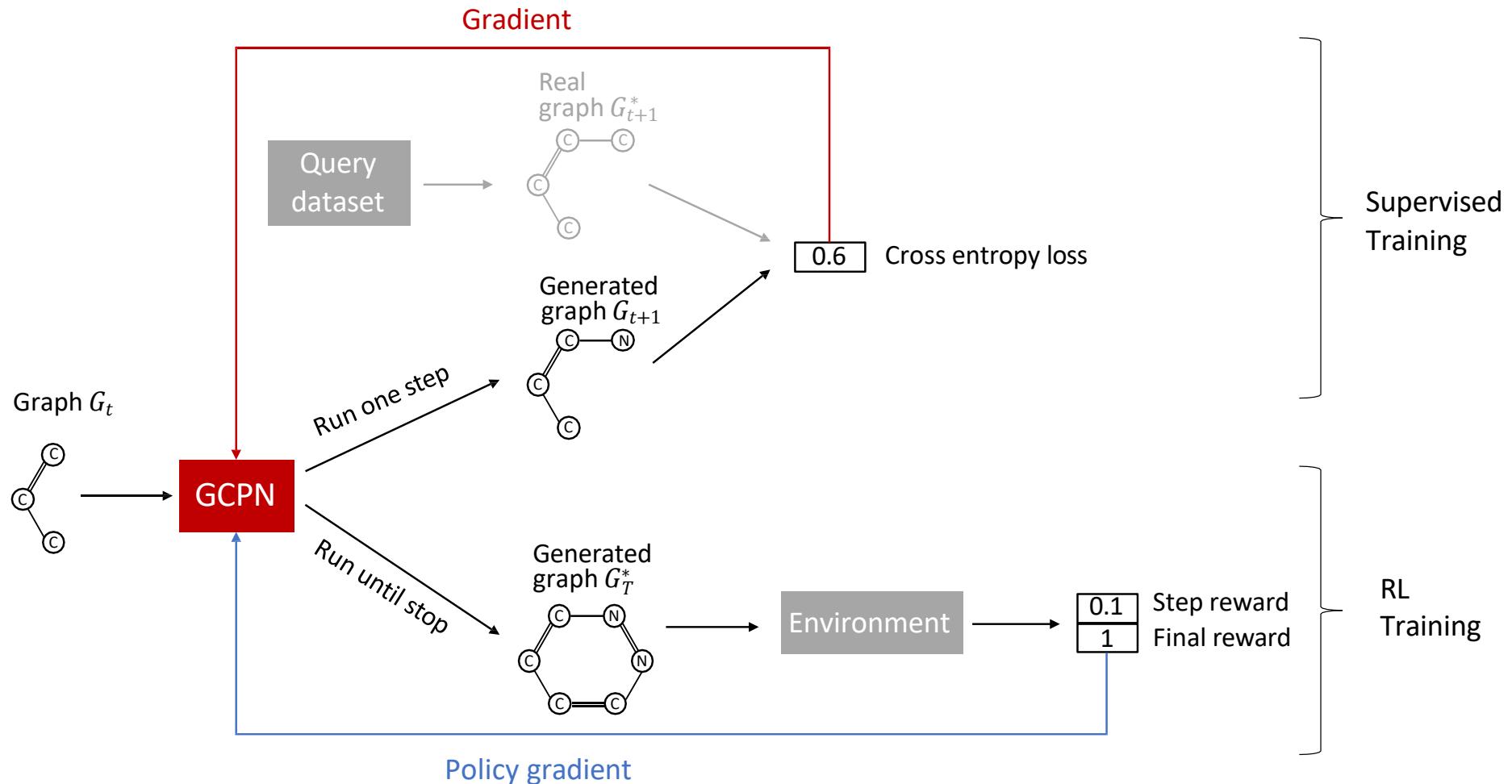
■ TWO PARTS:

■ **(1) Supervised training:** Train policy by imitating the action given by real observed graphs. Use gradient.

- We have covered this idea in GraphRNN

■ **(2) RL training:** Train policy to optimize rewards. Use standard policy gradient algorithm.

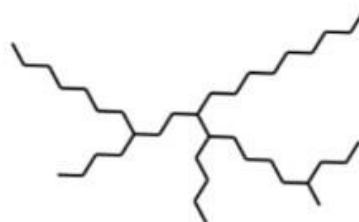
Training GCPN



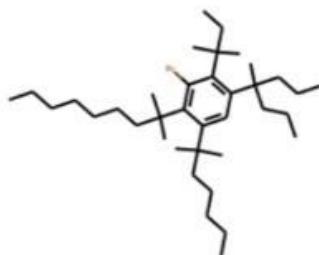
Qualitative Results

Visualization of GCPN graphs:

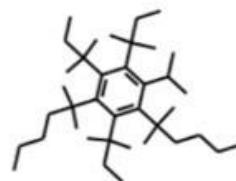
- **Property optimization** Generate molecules with high specified property score



7.98

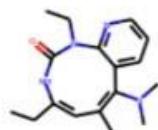


7.48

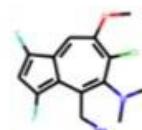


7.12

(a) Penalized logP optimization



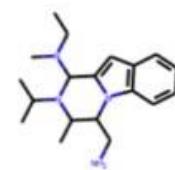
0.948



0.945



0.944



0.941

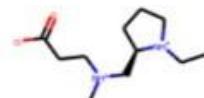
(b) QED optimization

Qualitative Results

Visualization of GCPN graphs:

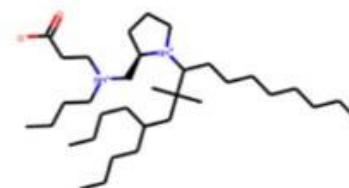
- Constrained optimization: Edit a given molecule for a few steps to achieve higher property score

Starting structure



-8.32

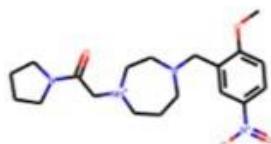
Finished structure



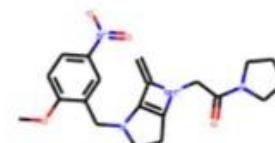
-0.71



Increase the
solubility in
octanol



-5.55



-1.78

(c) Constrained optimization of penalized logP

Summary of Graph Generation

- Complex graphs can be successfully generated via **sequential generation using deep learning**
- Each step a decision is made based on **hidden state**, which can be
 - **Implicit:** vector representation, decode with RNN
 - **Explicit:** intermediate generated graphs, decode with GCN
- Possible tasks:
 - **Imitating** a set of given graphs
 - **Optimizing** graphs towards given goals

Refs

[12-deep-generation.pdf \(stanford.edu\)](#)