Stanford CS224W: Deep Generative Models for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Announcements

- **Exam** opens this Thursday 11/21
  - 11/21 5**pm** to 11/23 5**am** (36 hour window)
  - 2 hours long (can't stop + start)
  - On gradescope – typeset your answers in Latex or upload images
  - Up to Lecture 13
  - Recitation: recording on Ed
  - More info on Website + Ed
- **Colab 4 + 5** due after the break (12/3 and 12/5)

Announcements

- **We need your Medium Account Usernames**
  - We will add all of you as writers to our CS224w publications.
  - Please fill out the google form on Ed with your medium account username by Wednesday EOD.
    - https://forms.gle/UtJ3x9dGpNTGCQz1A
    - This is a requirement!

Motivation for Graph Generation

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

Image credit: Medium

Social Networks

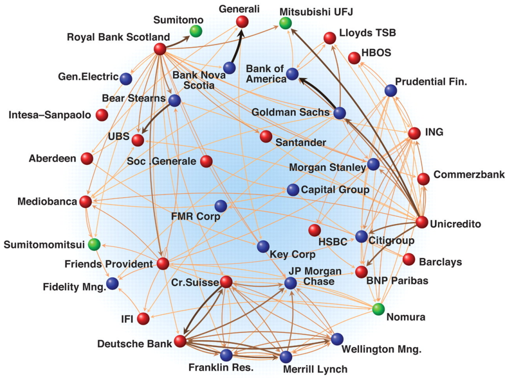


Image credit: Science

Economic Networks

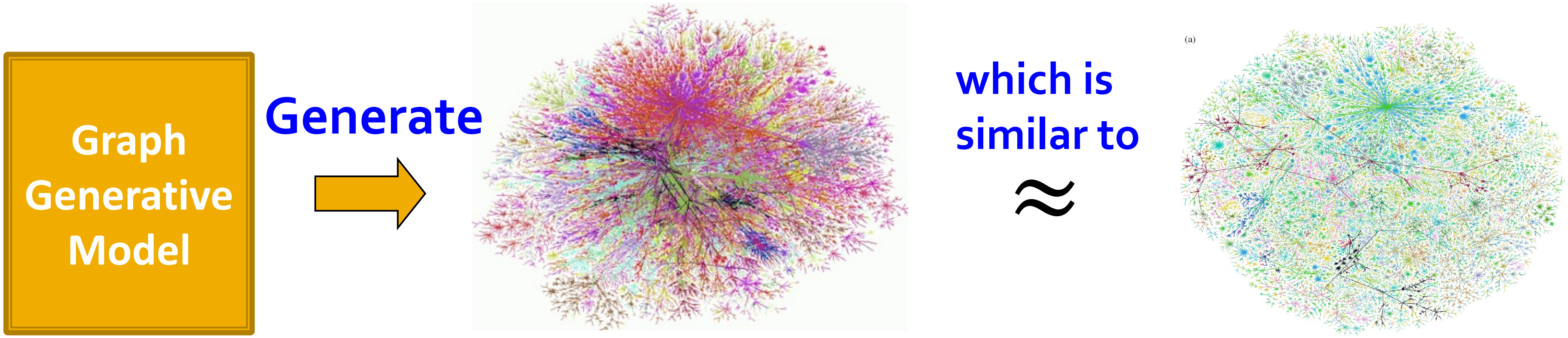
Image credit: Lumen Learning

Communication Networks

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

The Problem: Graph Generation

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



Graph Generative Model

Generate

which is similar to ≈

Synthetic graph

Real graph

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

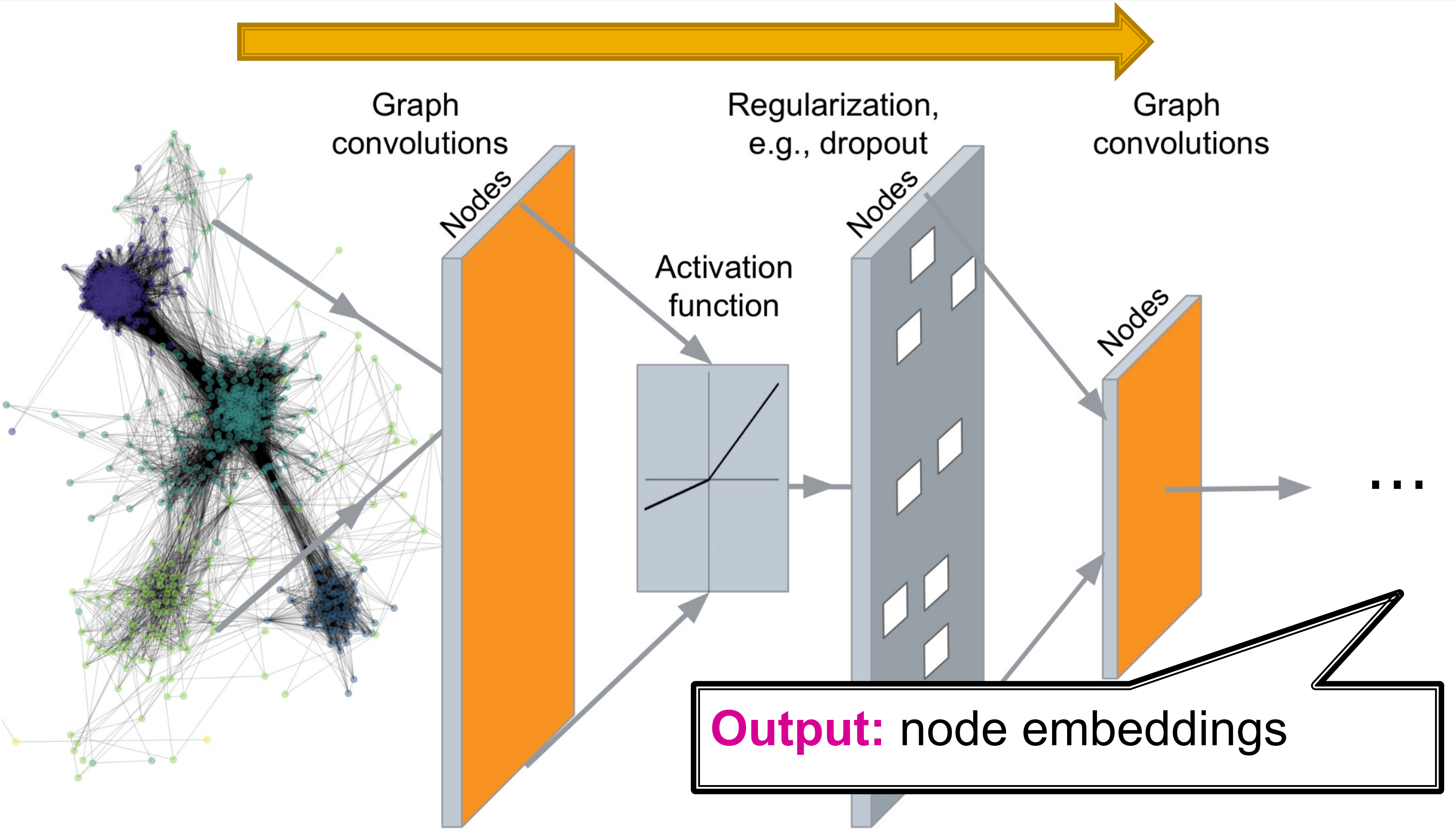
Why Do We Study Graph Generation

- **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 general novel graph instances
- **Anomaly detection** - We can decide if a graph is normal / abnormal

History of Graph Generation

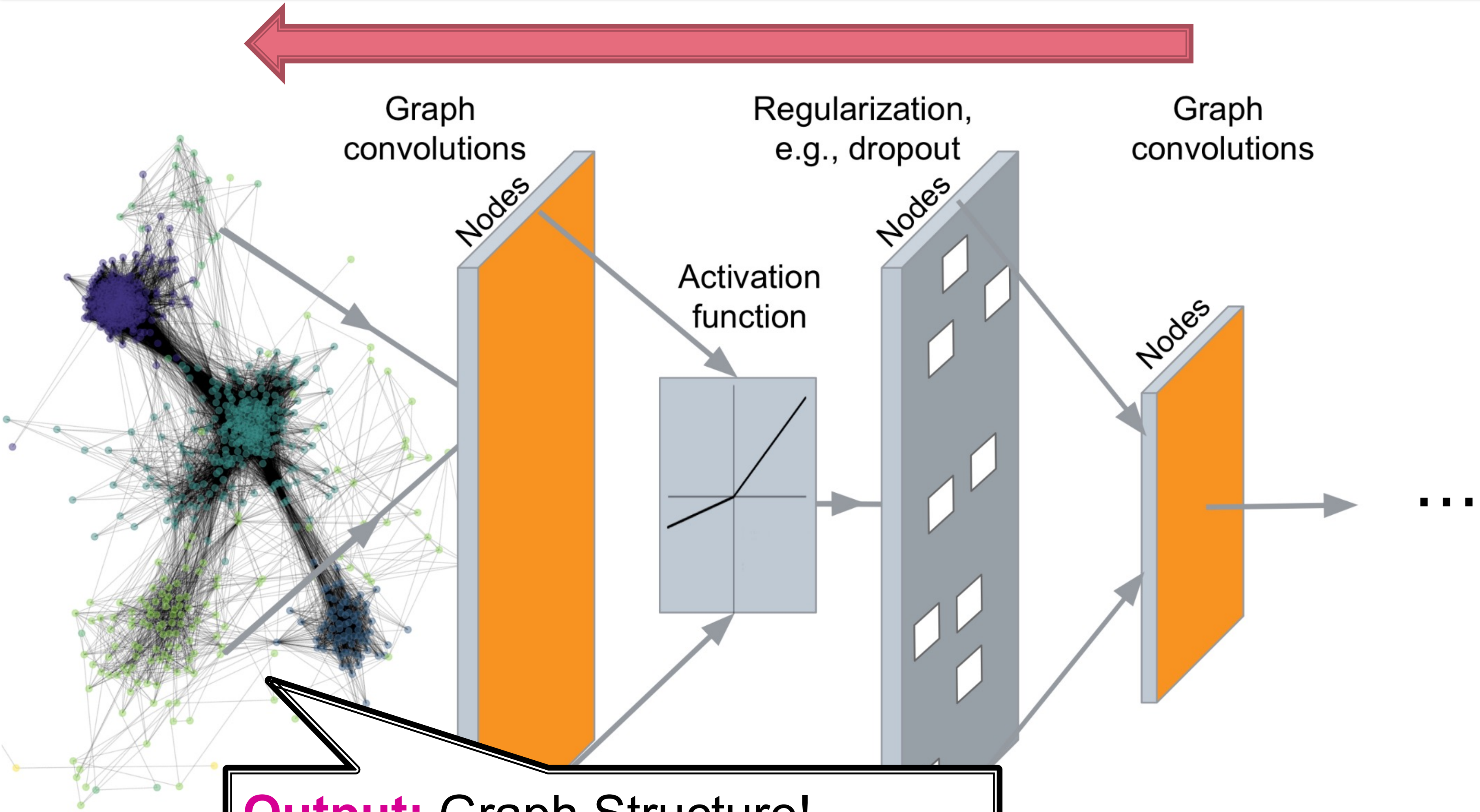
- **Step 1: Properties of real-world graphs**
  - A successful graph generative model should fit these properties
- **Step 2: Traditional graph generative models**
  - Each come with different assumptions on the graph formulation process
- **Step 3: Deep graph generative models**
  - Learn the graph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

So far: Deep Graph Encoders



Output: node embeddings

Today: Deep Graph Decoders



Output: Graph Structure!

Stanford CS224W: Machine Learning for Graph Generation

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Graph Generation Tasks

Task 1: Realistic graph generation

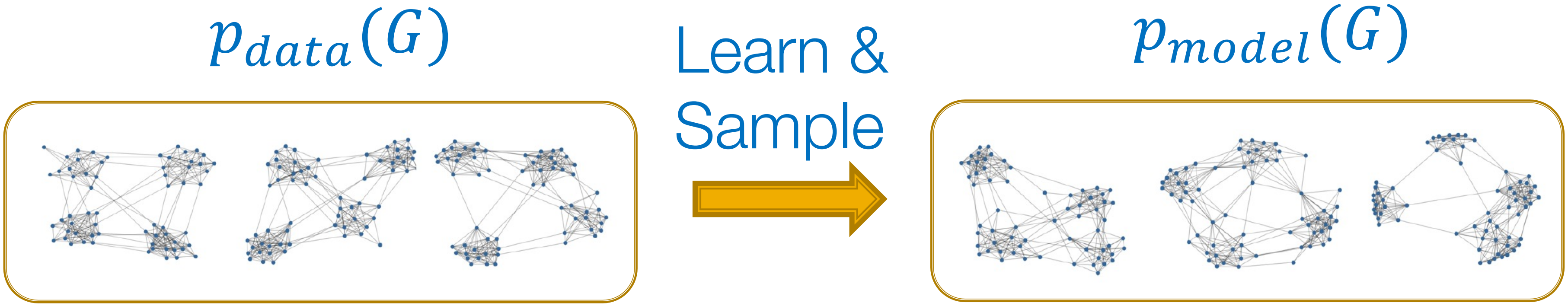
- Generate graphs that are **similar to a given set of graphs** **[Focus of this lecture]**

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

Setup:

- Assume we want to learn a generative model from a set of data points (i.e., graphs) $\{\boldsymbol{x}_i\}$
  - $p_{data}(\boldsymbol{x})$ is the data distribution, which is never known to us, but we have sampled $\boldsymbol{x}_i \sim p_{data}(\boldsymbol{x})$
  - $p_{model}(\boldsymbol{x}; \theta)$ is the model, parametrized by θ, that we use to approximate $p_{data}(\boldsymbol{x})$
- **Goal:**
  - **(1)** Make $p_{model}(\boldsymbol{x}; \theta)$ close to $p_{data}(\boldsymbol{x})$ **(Density estimation)**
  - **(2)** Make sure we can sample from $p_{model}(\boldsymbol{x}; \theta)$ **(Sampling)**
    - To generate new graphs, we sample from $p_{model}(\boldsymbol{x}; \theta)$

Generative Models Basics

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

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

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

- Find parameters θ^*, such that for observed data points $\boldsymbol{x}_i \sim p_{data}$ the $\sum_i \log p_{model}(\boldsymbol{x}_i; \theta^*)$ 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 $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$

- **Goal:** Sample from a complex distribution
- The most common approach:
  - **(1)** Sample from a simple noise distribution

$$\boldsymbol{z}_i \sim N(0,1)$$

  - **(2)** Transform the noise z_i via $f(\cdot)$

$$\boldsymbol{x}_i = f(\boldsymbol{z}_i; \theta)$$

  Then $\boldsymbol{x}_i$ 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:

- $\boldsymbol{p_{model}(x; \theta)}$ 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:

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

  - E.g., $\boldsymbol{x}$ is a vector, x_t is the t-th dimension; $\boldsymbol{x}$ 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)

Stanford CS224W: GraphRNN: Generating Realistic Graphs

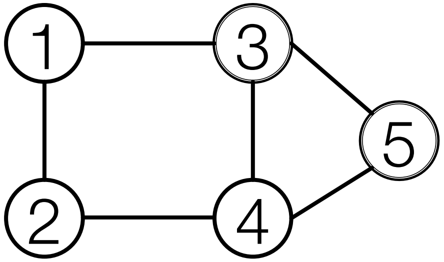
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

[You et al., ICML 2018]

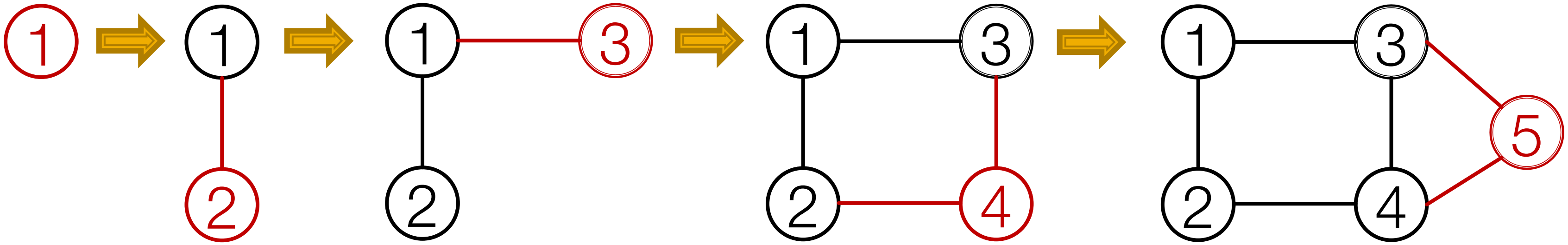
GraphRNN Idea

Generating graphs via sequentially adding nodes and edges

Graph G



Generation process S^{π}



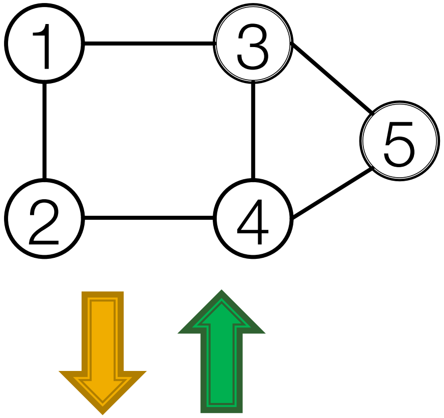
GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. J. You, R. Ying, X. Ren, W. L. Hamilton, J. Leskovec. *International Conference on Machine Learning (ICML)*, 2018.

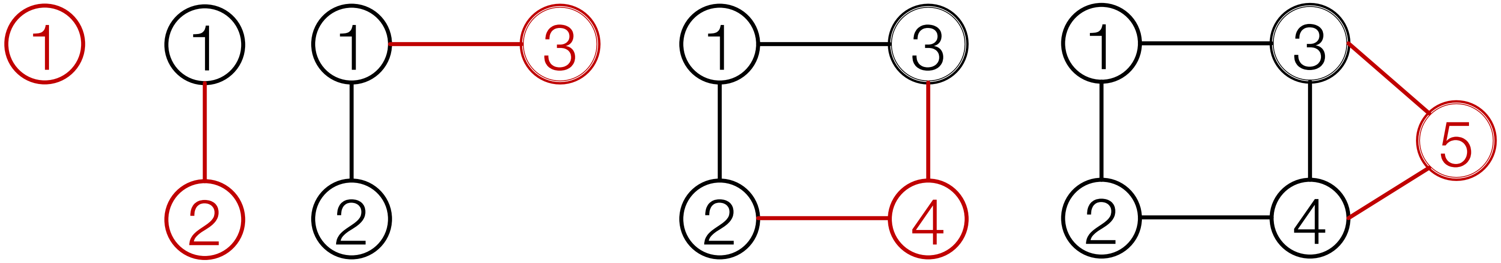
Model Graphs as Sequences

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

Graph G with node ordering π:

Sequence S^{π}:





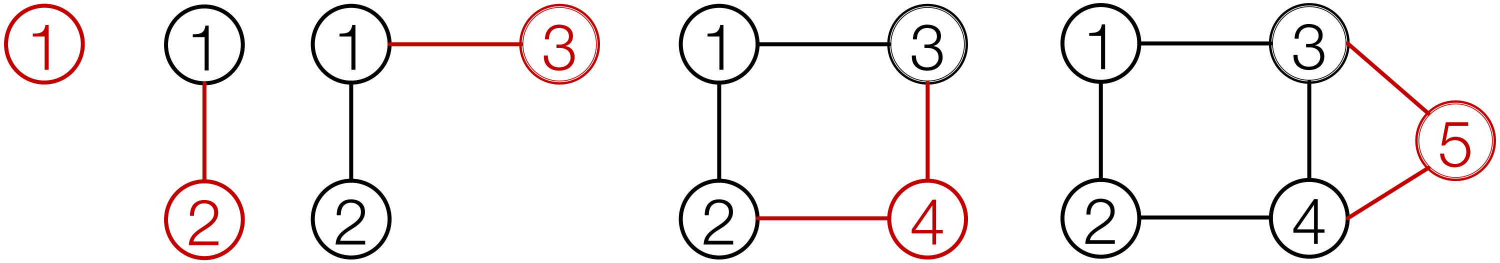
$$S^{\pi} = (\; S_1^{\pi} \;, S_2^{\pi} \;, \quad S_3^{\pi} \quad , \quad S_4^{\pi} \quad , \quad S_5^{\pi} \quad)$$

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



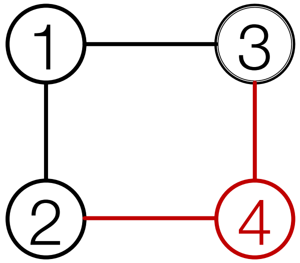
$$S^{\pi} = (\ S_1^{\pi}\ ,\ S_2^{\pi}\ ,\ S_3^{\pi}\ ,\ S_4^{\pi}\ ,\ S_5^{\pi}\ )$$

"Add node 1" . . . "Add node 5"

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^{π}

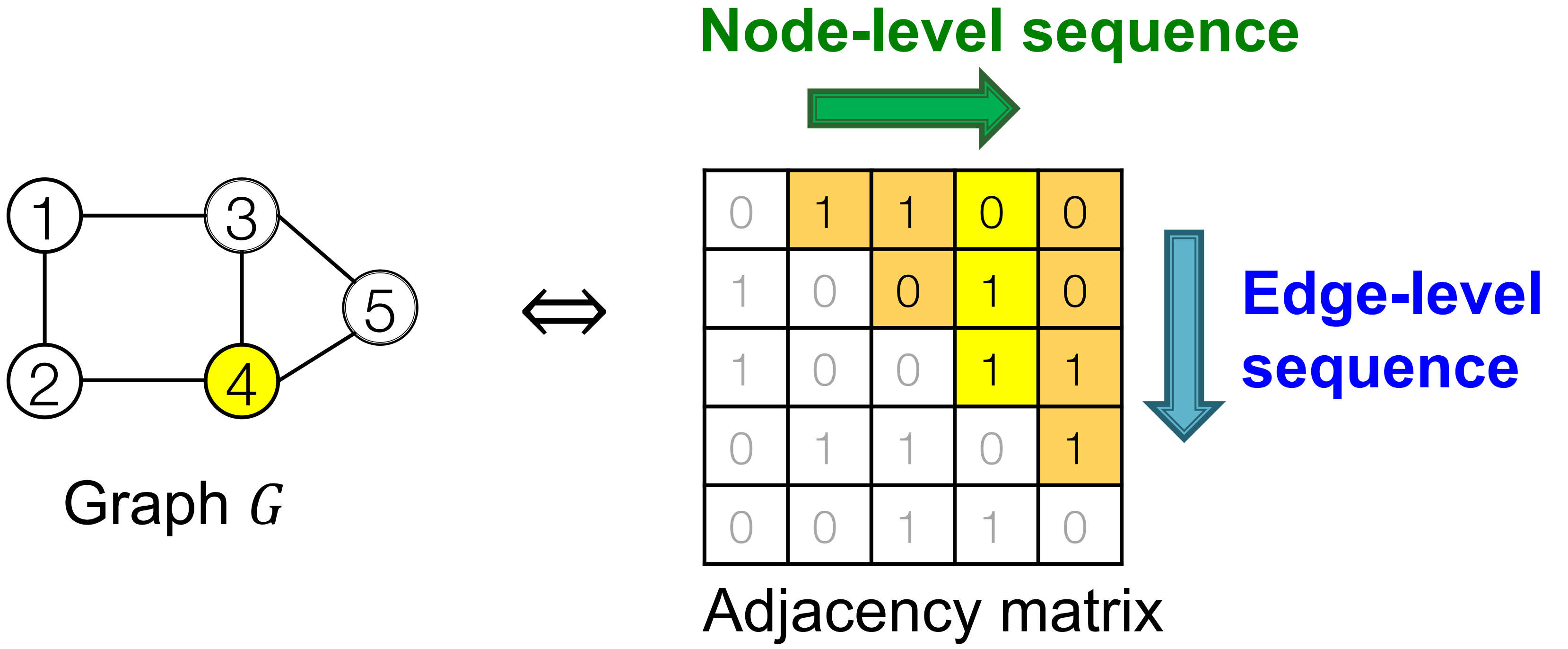
$$S_4^{\pi} = (\quad S_{4,1}^{\pi} \quad , \quad S_{4,2}^{\pi} \quad , \quad S_{4,3}^{\pi} \quad)$$

"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)



Graph G

Node-level sequence

Edge-level sequence

0	1	1	0	0
1	0	0	1	0
1	0	0	1	1
0	1	1	0	1
0	0	1	1	0

Adjacency matrix

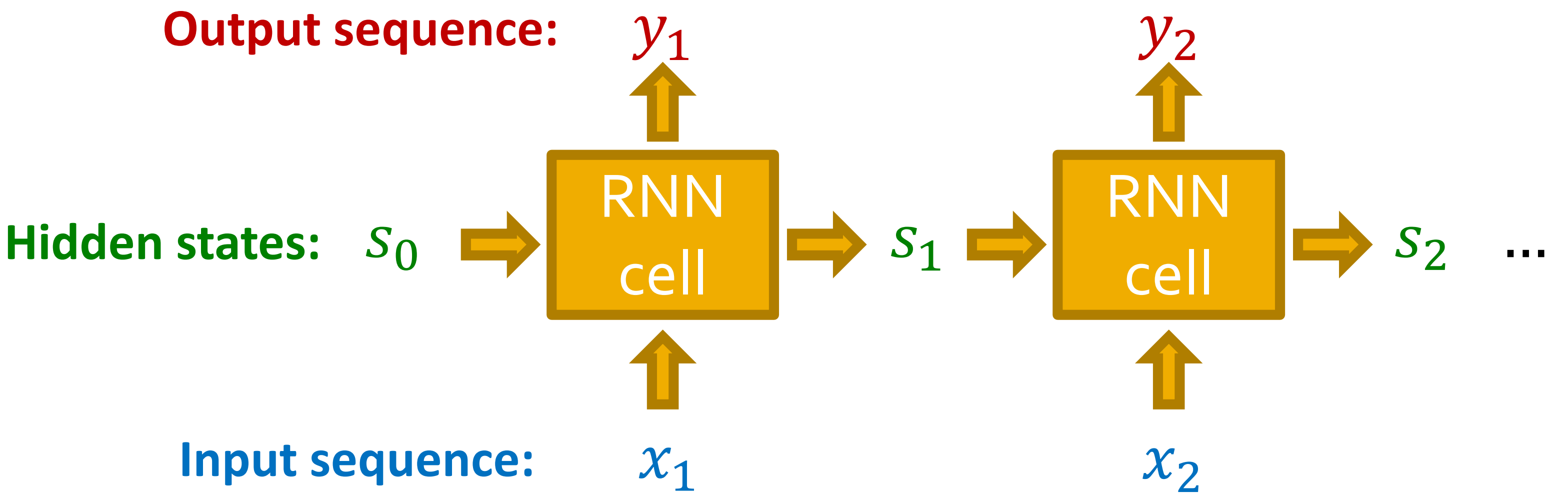
Model Graphs as Sequences

- **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!
- Same principles apply to Transformers

Background: Recurrent NNs

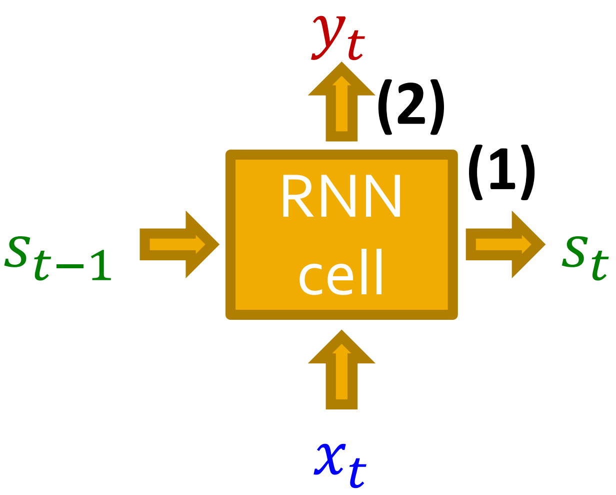
- RNNs are designed for **sequential data**
  - RNN sequentially takes **input sequence** to **update its hidden states**
  - The **hidden states** summarize all the information input to RNN
  - The update is conducted via **RNN cells**



Background: Recurrent NNs

- s_t: **State** of RNN after step t
- x_t: **Input** to RNN at step t
- y_t: **Output** of RNN at step t
- **RNN cell:** W, U, V: Trainable parameters

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



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.

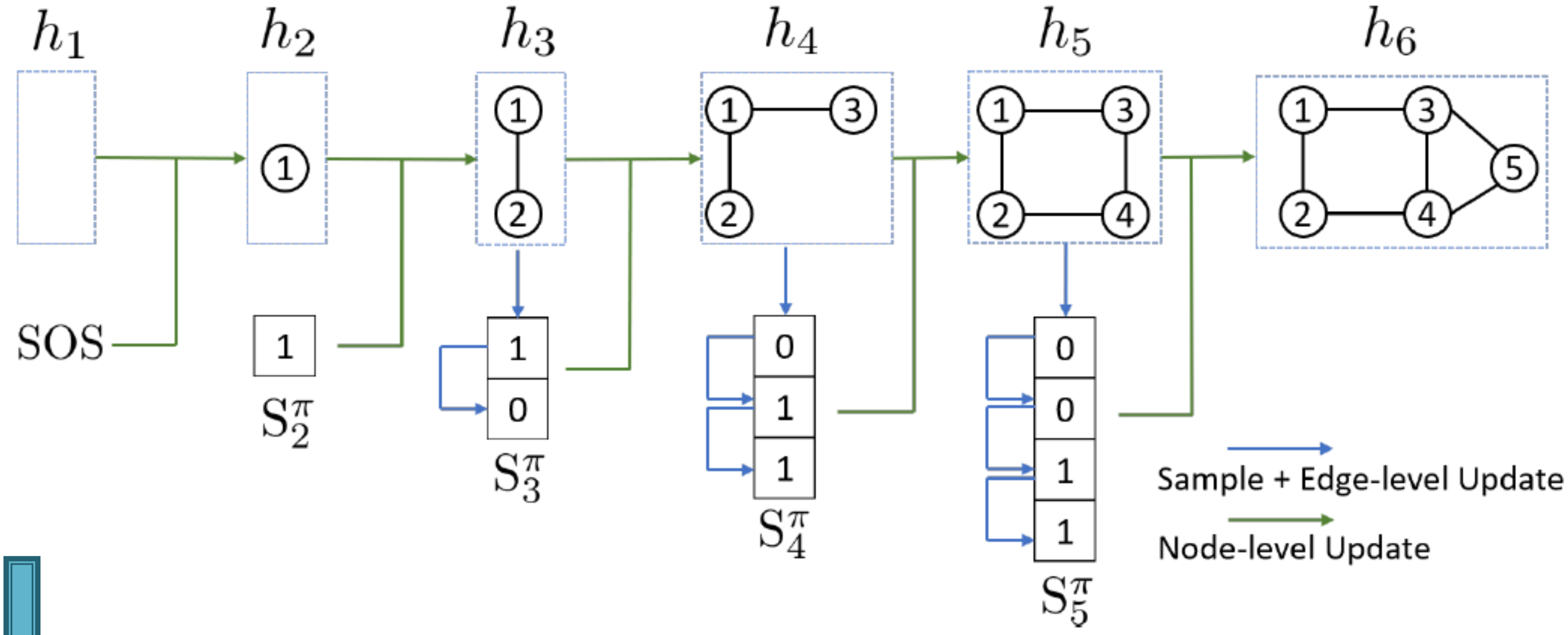
GraphRNN: Two levels of RNN

- **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

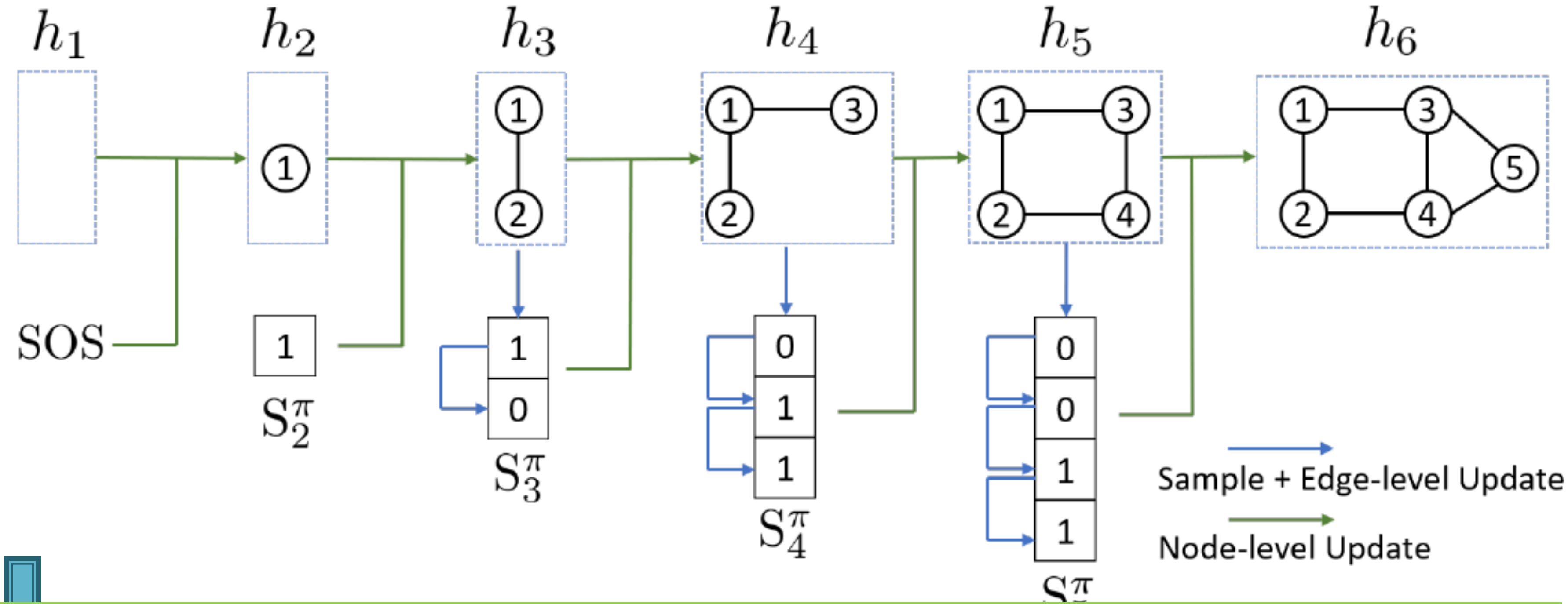
Node-level RNN generates the initial state for edge-level RNN



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

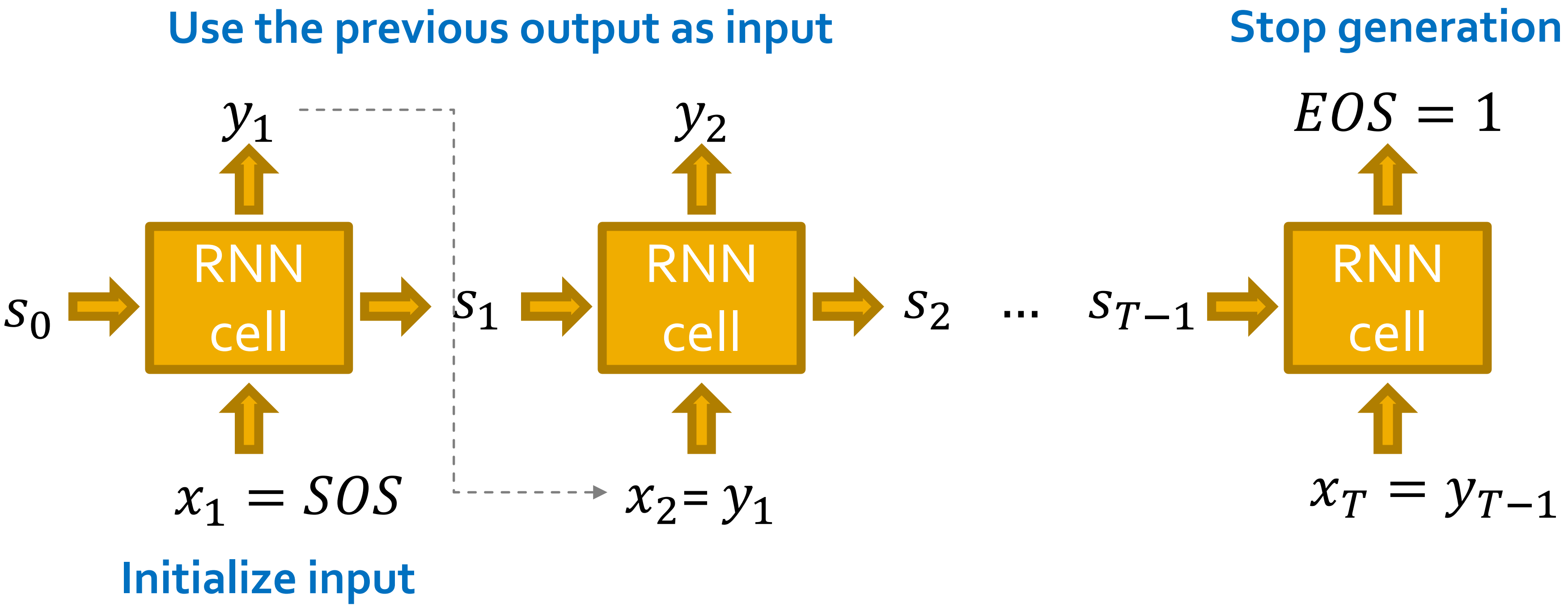
RNN for Sequence Generation

- **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

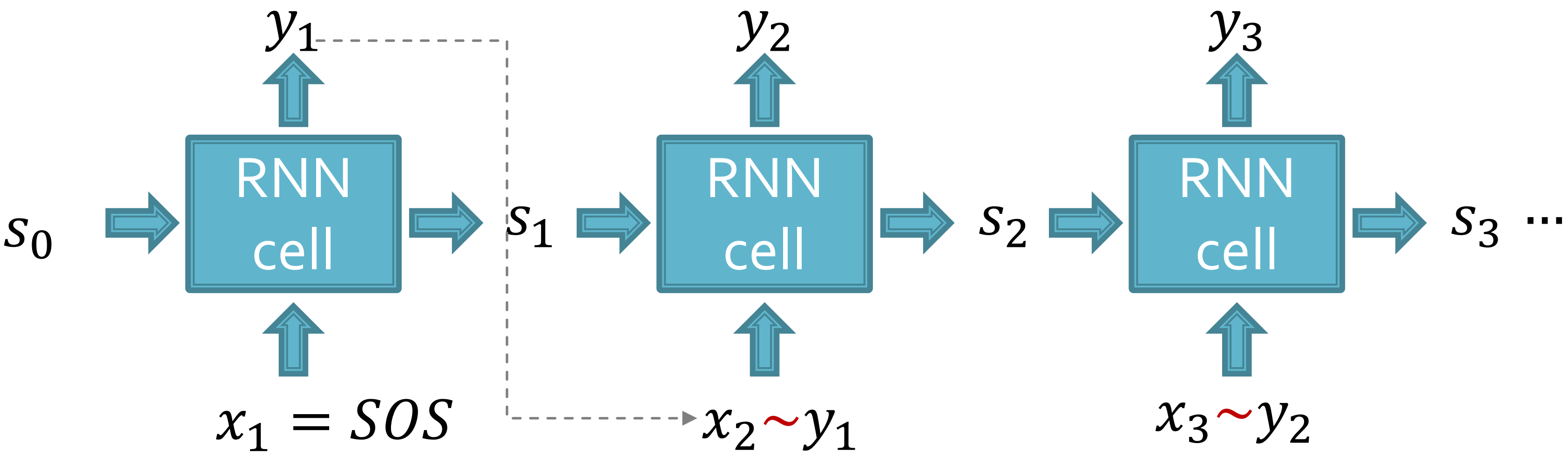


- This is good, but this model is **deterministic**

Towards Edge-Level RNN

Consider the Edge-level RNN for now.

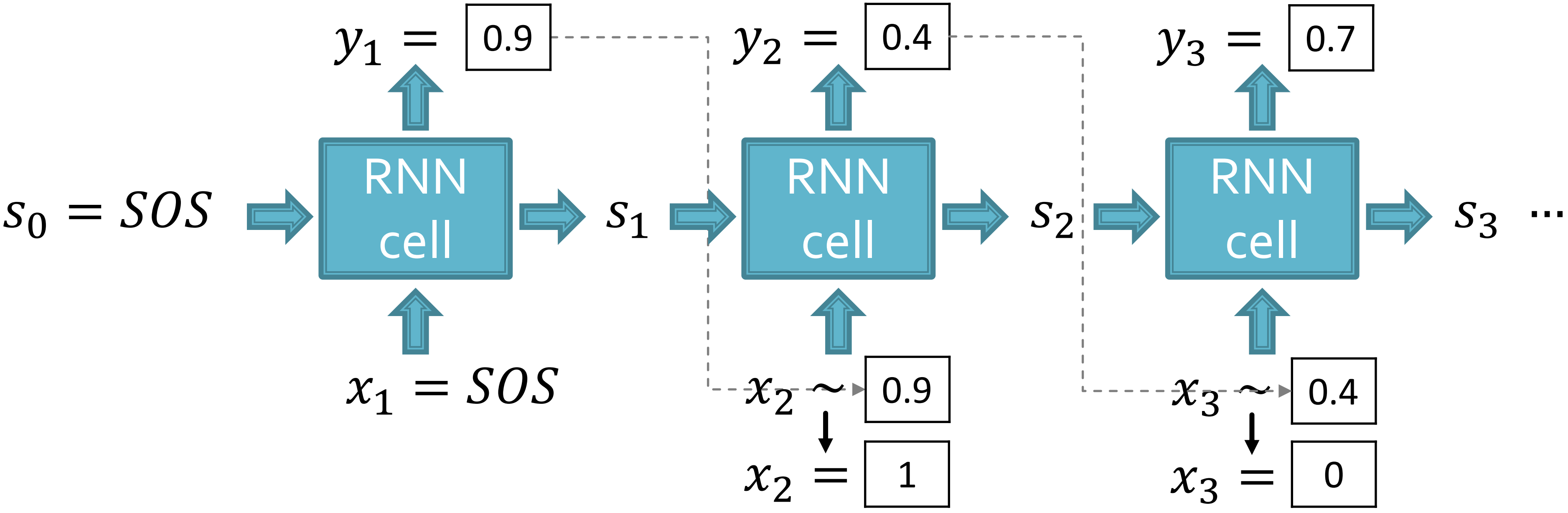
- **Our goal**: Model $\prod_{t=1}^{n} p_{model}(x_t | x_1, \dots, x_{t-1}; \theta)$
- Let $y_t = p_{model}(x_{t+1} | x_1, \dots, x_t; \theta)$
- Then we need to sample x_{t+1} from y_t: $x_{t+1} \sim y_t$
  - Each step of RNN outputs a **probability of a single edge**
  - We then sample from the distribution,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

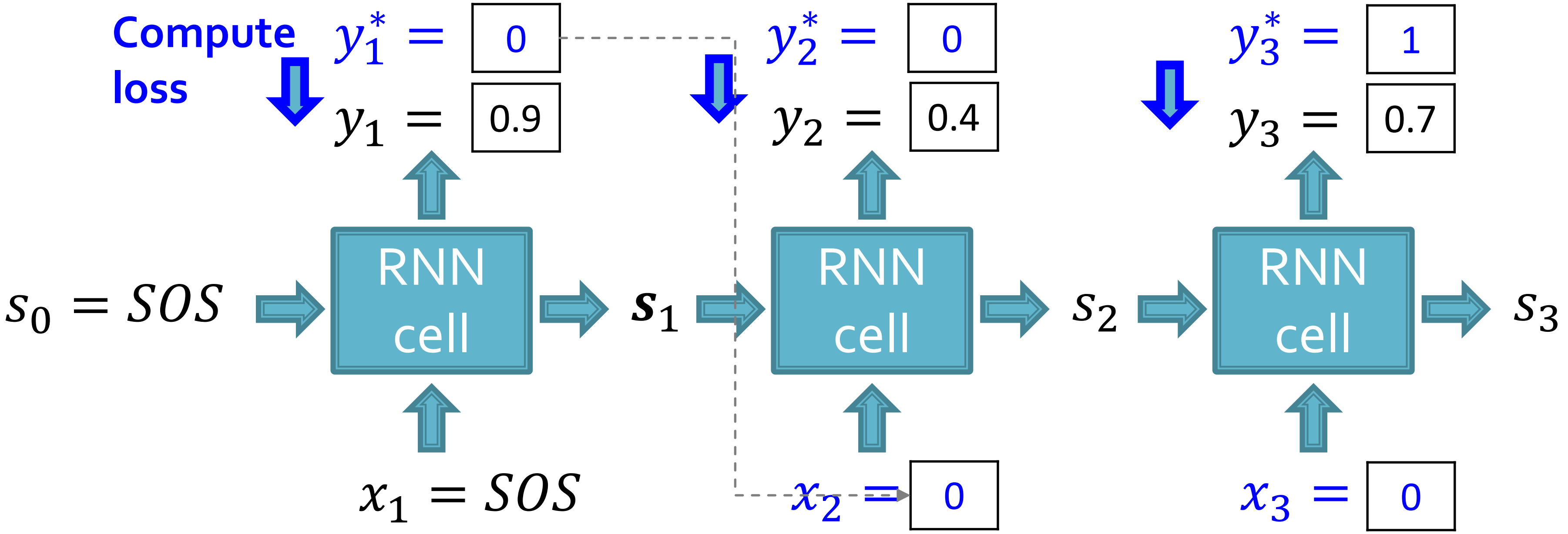


- **How do we use training data $x_1, x_2, \dots, x_n$?**

Edge-Level RNN at Training Time

Training the model:

- We observe a sequence y^* of edges [0,0,1,…]
- **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

$$y_1^* = \boxed{0}$$

$$y_1 = \boxed{0.9}$$

- 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 it output to initialize Edge RNN

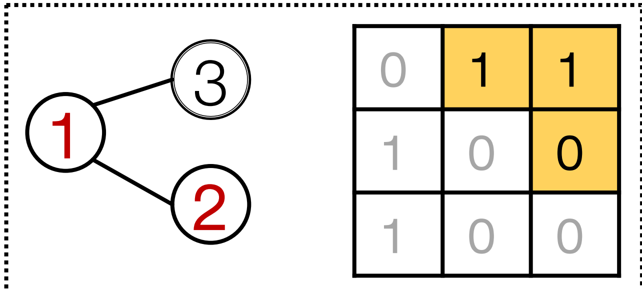
(2) **Add new edges for the new node:** We run Edge RNN to predict if the new node will connect to each of the previous node

(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.

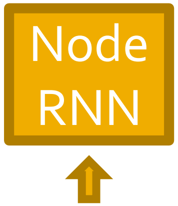
Put Things Together: Training

Assuming **Node 1** is in the graph
Now adding **Node 2**



0	1	1
1	0	0
1	0	0

Observed graph

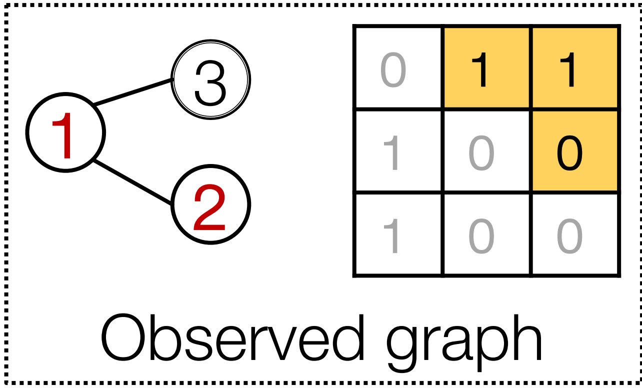

Node RNN

SOS

Start the node RNN

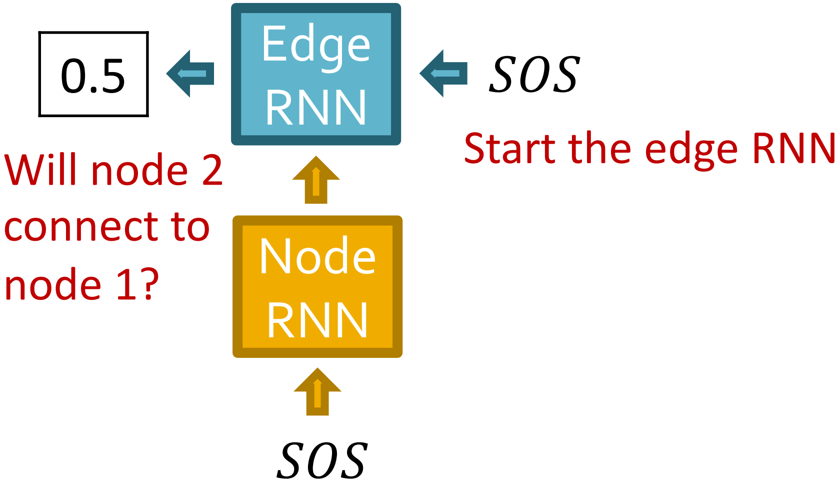
Put Things Together: Training

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



0	1	1
1	0	0
1	0	0

Observed graph

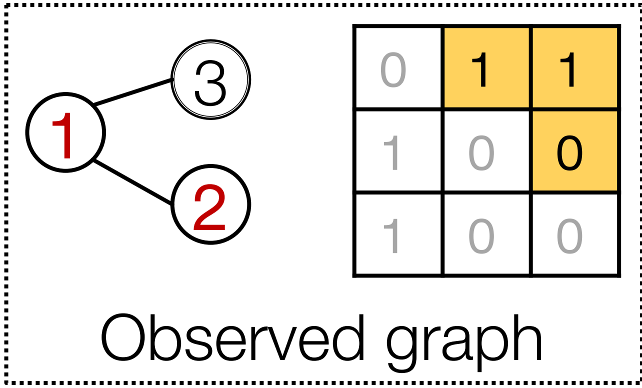


Start the edge RNN

Will node 2 connect to node 1?

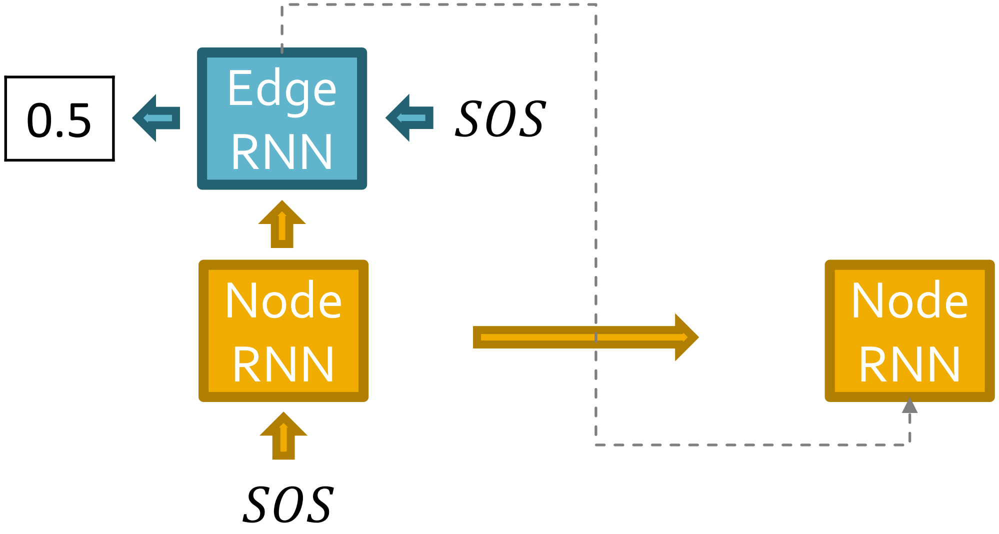
Put Things Together: Training

Update Node RNN using
Edge RNN's hidden state



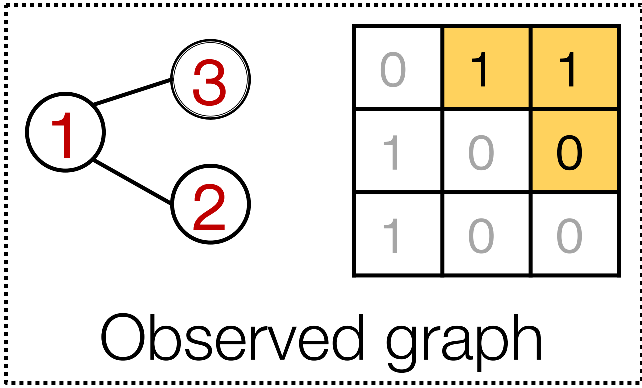
0	1	1
1	0	0
1	0	0

Observed graph

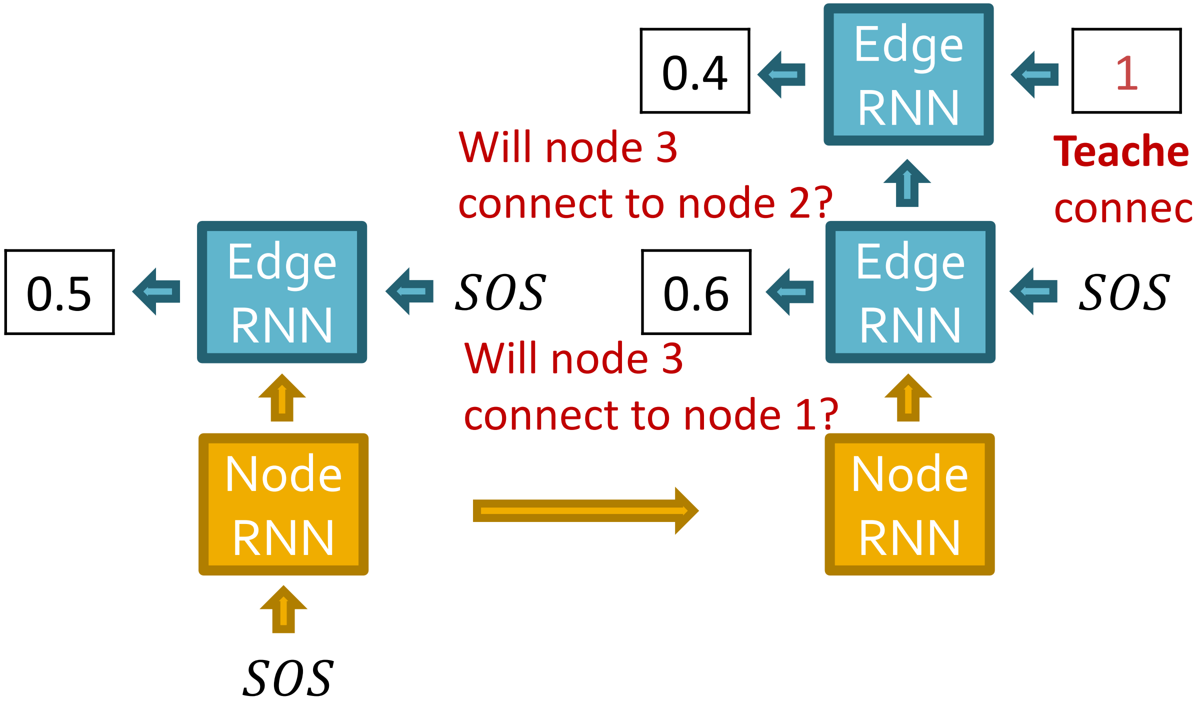


Put Things Together: Training

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



0	1	1
1	0	0
1	0	0



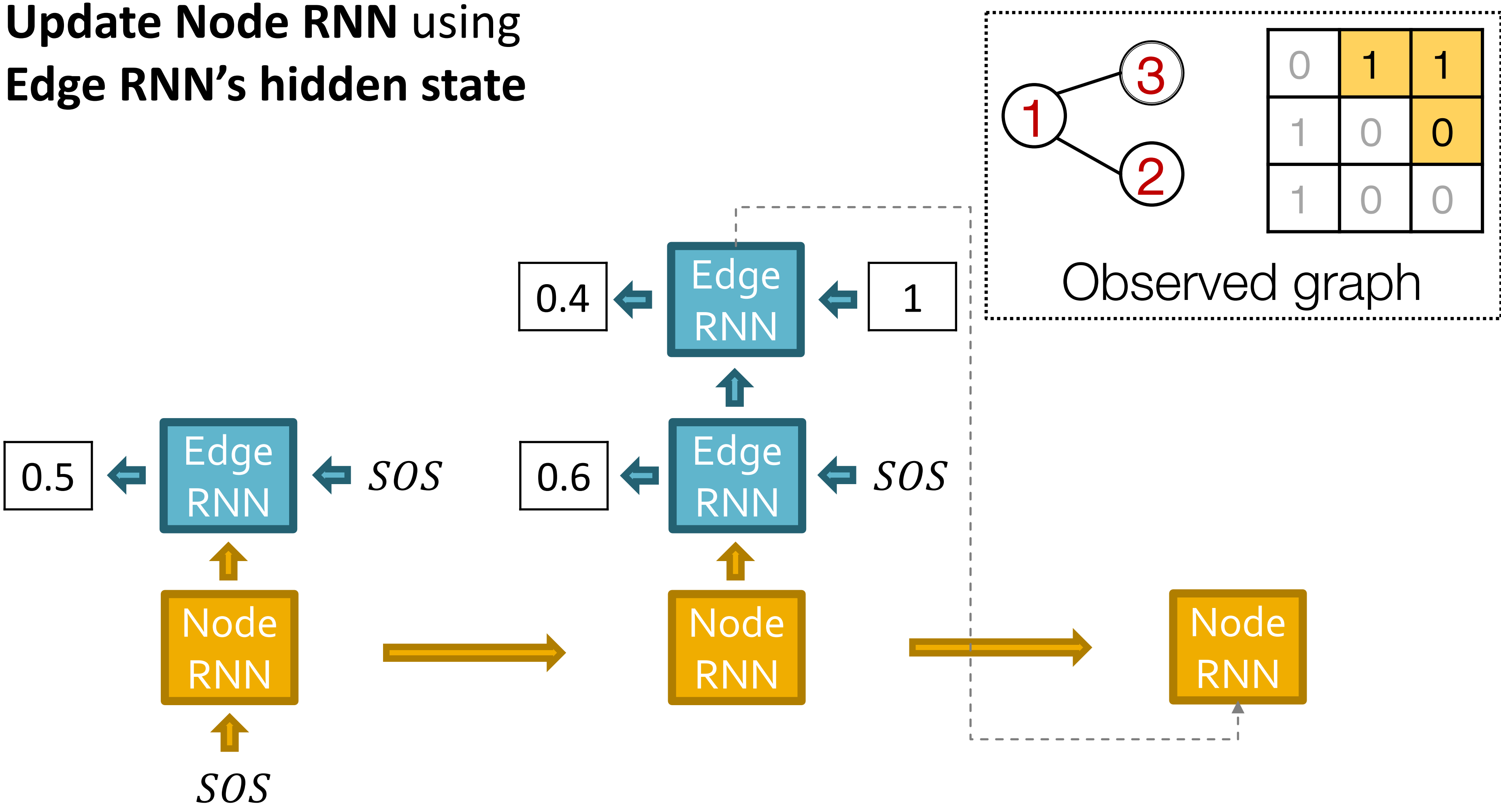
Will node 3 connect to node 2?

Teacher forcing: node 3 will connect to node 1

Will node 3 connect to node 1?

Put Things Together: Training

Update Node RNN using **Edge RNN's hidden state**

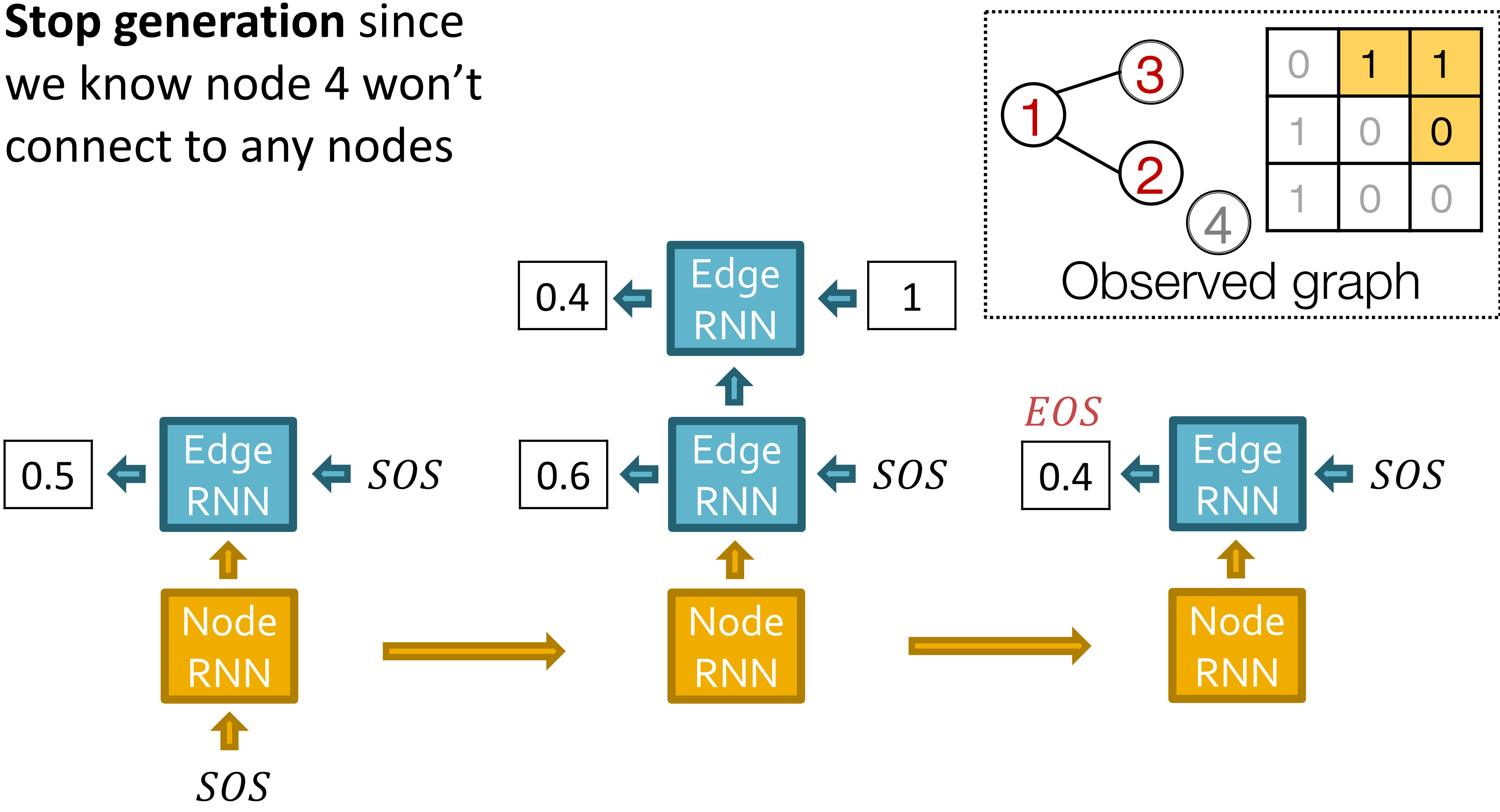


0	1	1
1	0	0
1	0	0

Observed graph

Put Things Together: Training

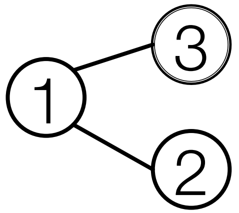
Stop generation since we know node 4 won’t connect to any nodes



0	1	1
1	0	0
1	0	0

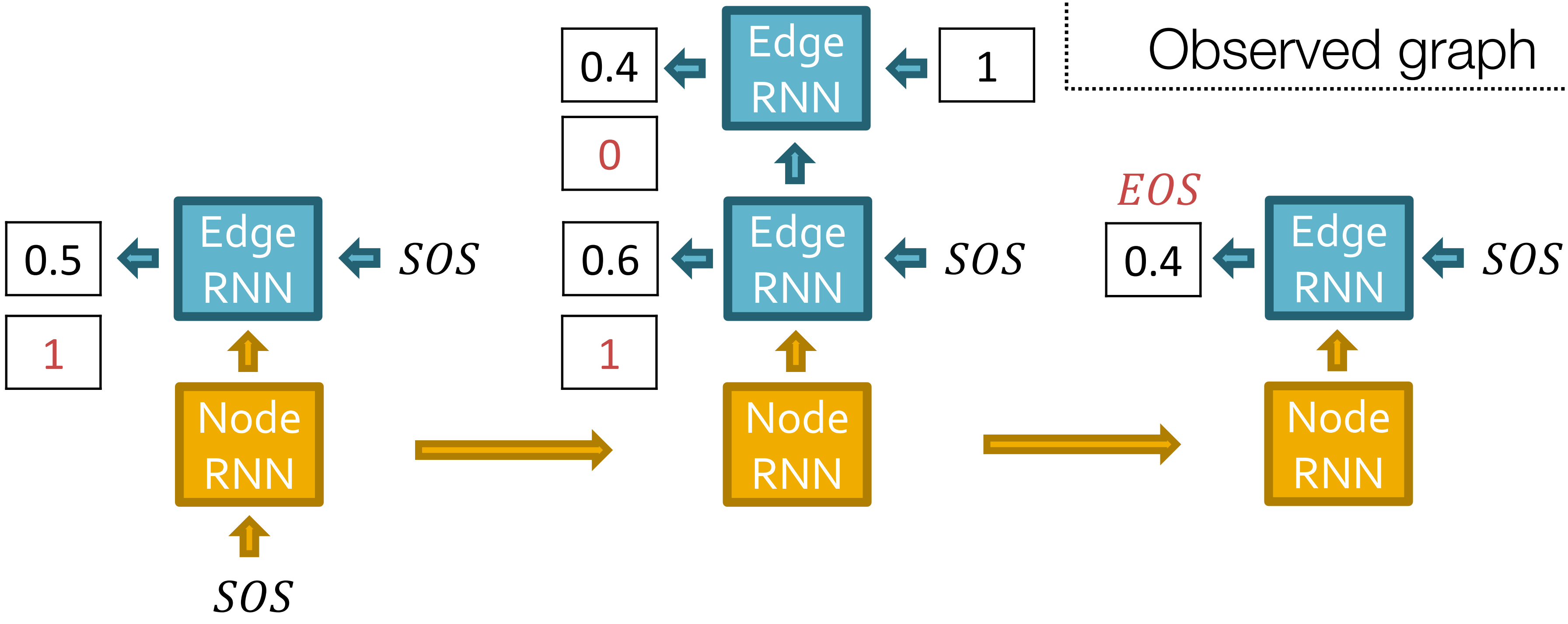
Put Things Together: Training

For each prediction, we **get supervision from the ground truth**



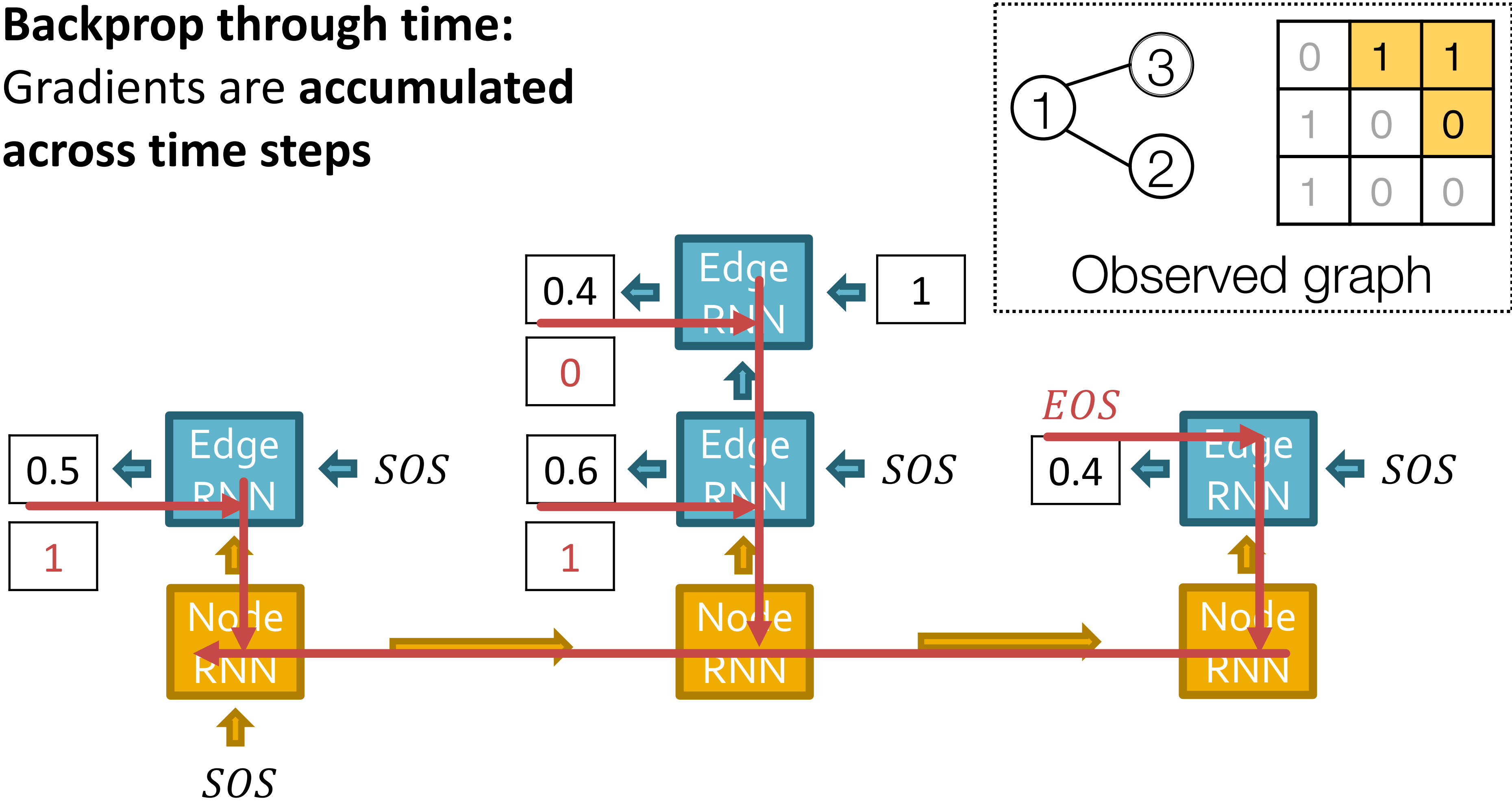
0	1	1
1	0	0
1	0	0

Observed graph



Put Things Together: Training

Backprop through time:
Gradients are **accumulated across time steps**

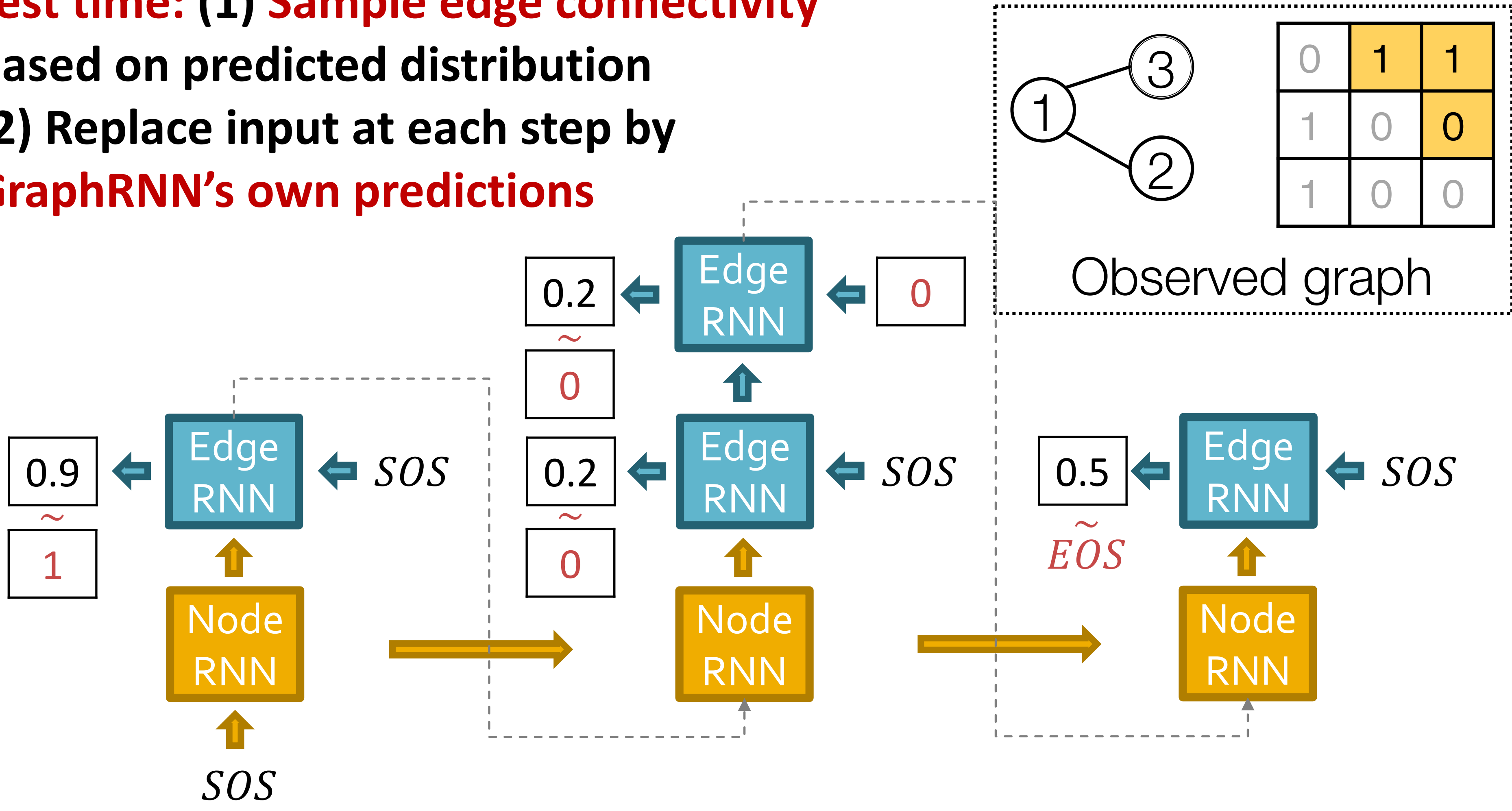


0	1	1
1	0	0
1	0	0

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

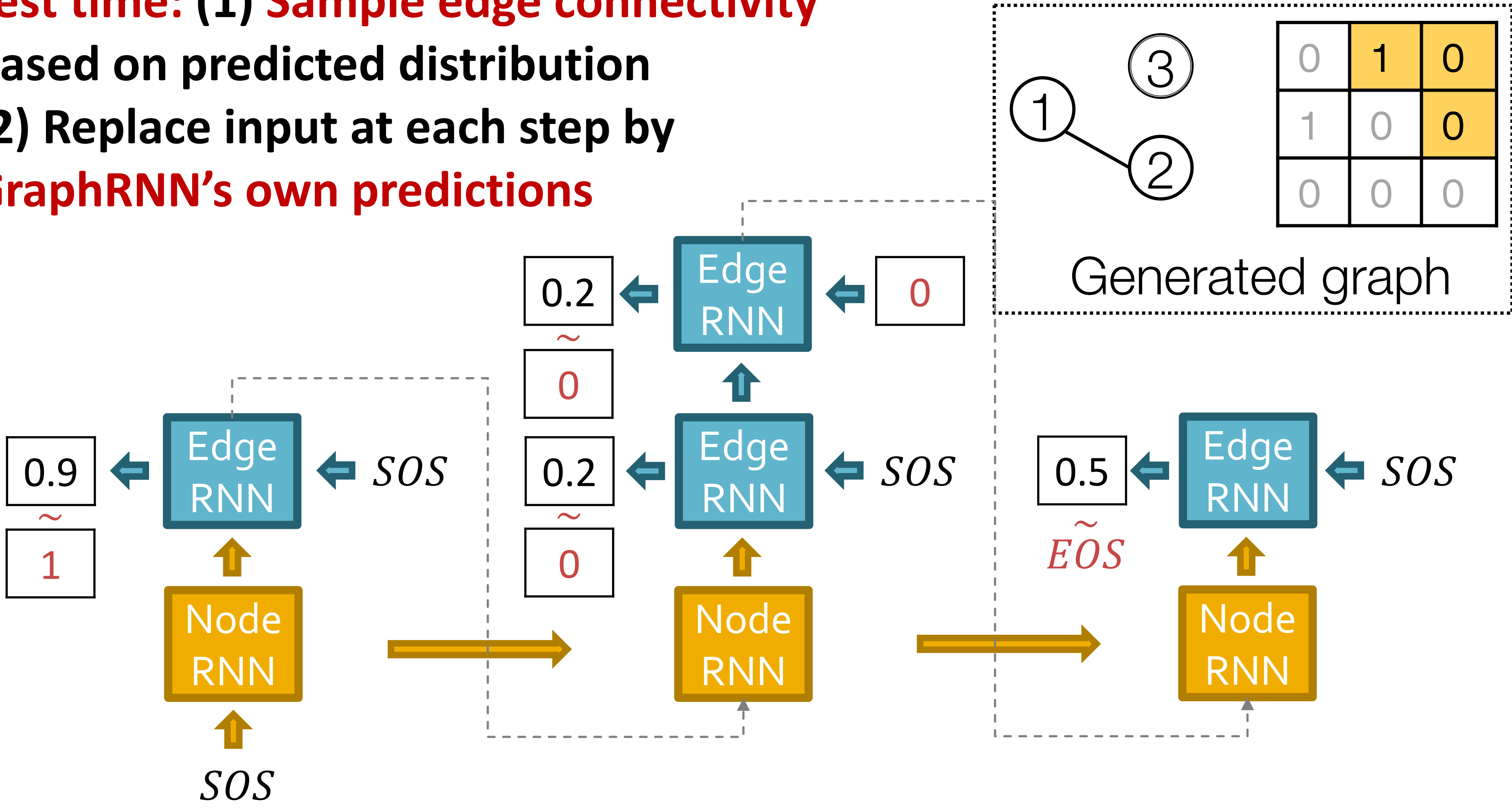


0	1	1
1	0	0
1	0	0

Observed graph

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



0	1	0
1	0	0
0	0	0

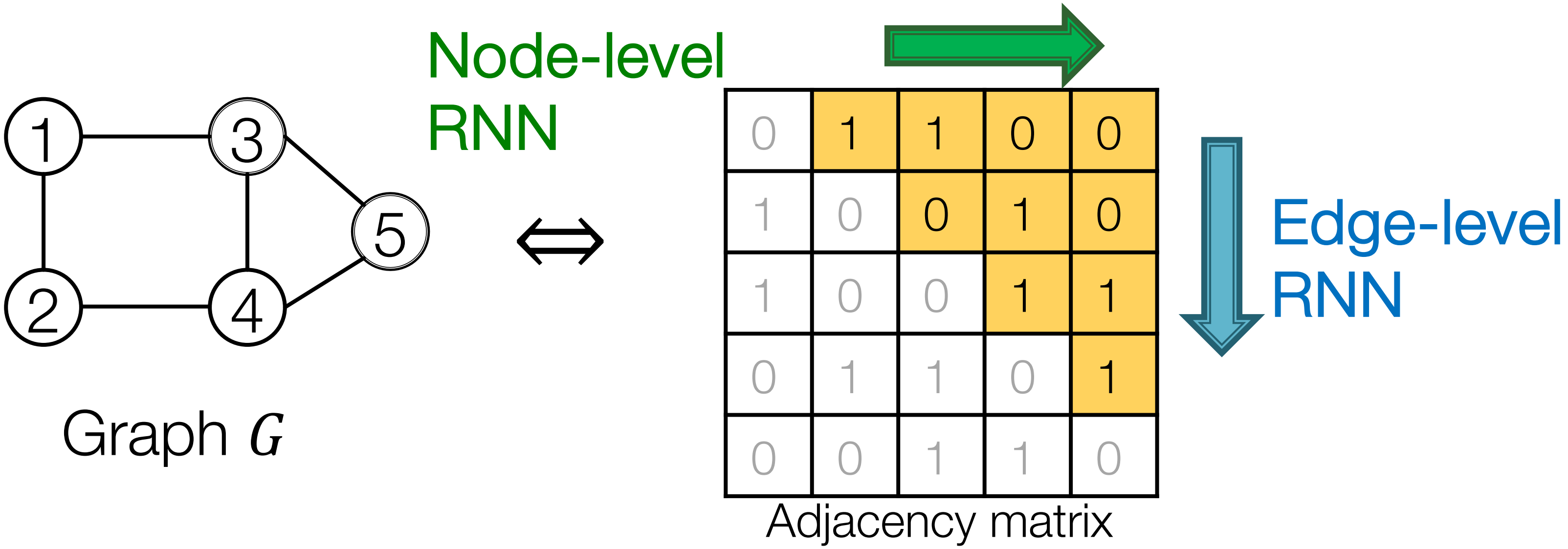
Generated graph

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



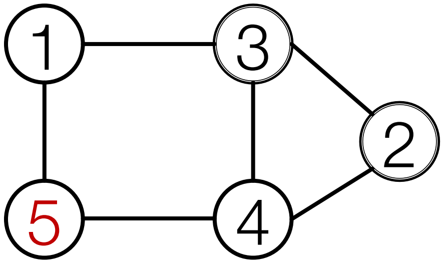
0	1	1	0	0
1	0	0	1	0
1	0	0	1	1
0	1	1	0	1
0	0	1	1	0

Stanford CS224W: Scaling Up and Evaluating Graph Generation

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

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

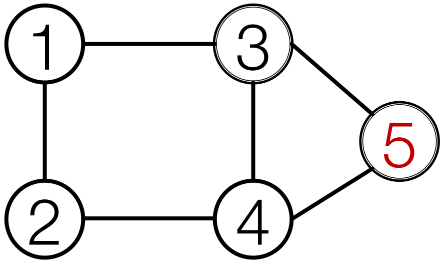
"Recipe" to generate the left graph:

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

How do we limit this complexity?

Solution: Tractability via BFS

- **Breadth-First Search node ordering**



BFS ordering

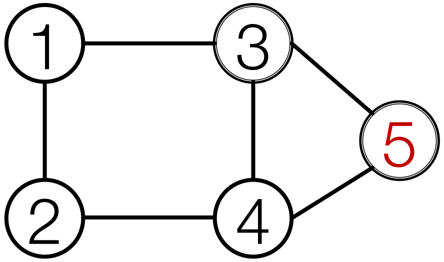
"Recipe" to generate the left graph:

- 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 ordering

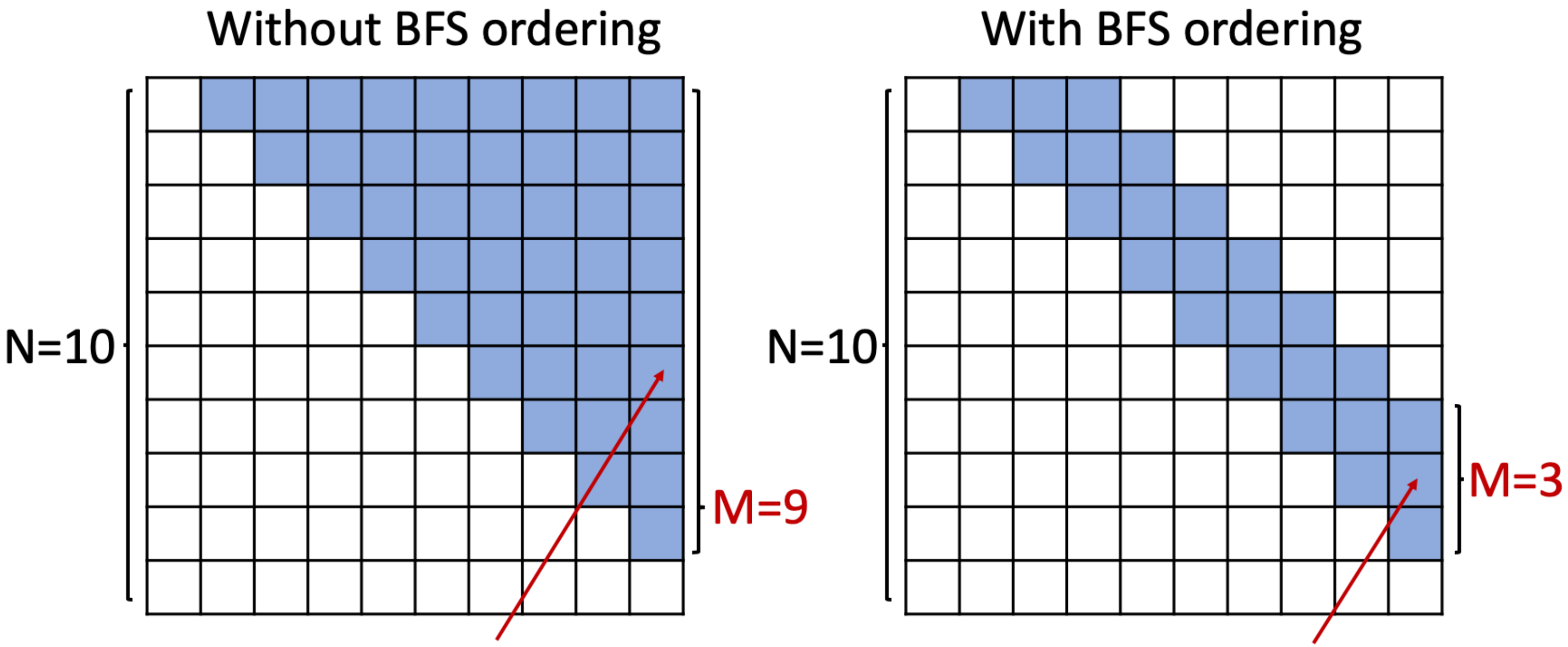
BFS node ordering: Node 5 will never connect to node 1 (only need memory of 2 “steps” rather than $n-1$ steps)

- **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

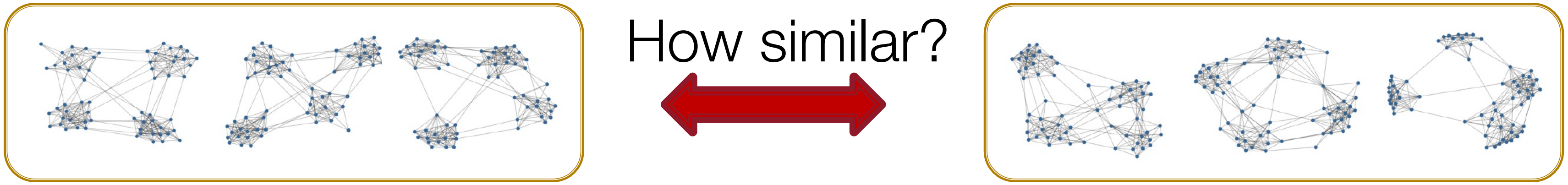


Connectivity with
All Previous nodes

Connectivity only with
nodes in the BFS frontier

Evaluating Generated Graphs

- **Task: Compare two sets of graphs**

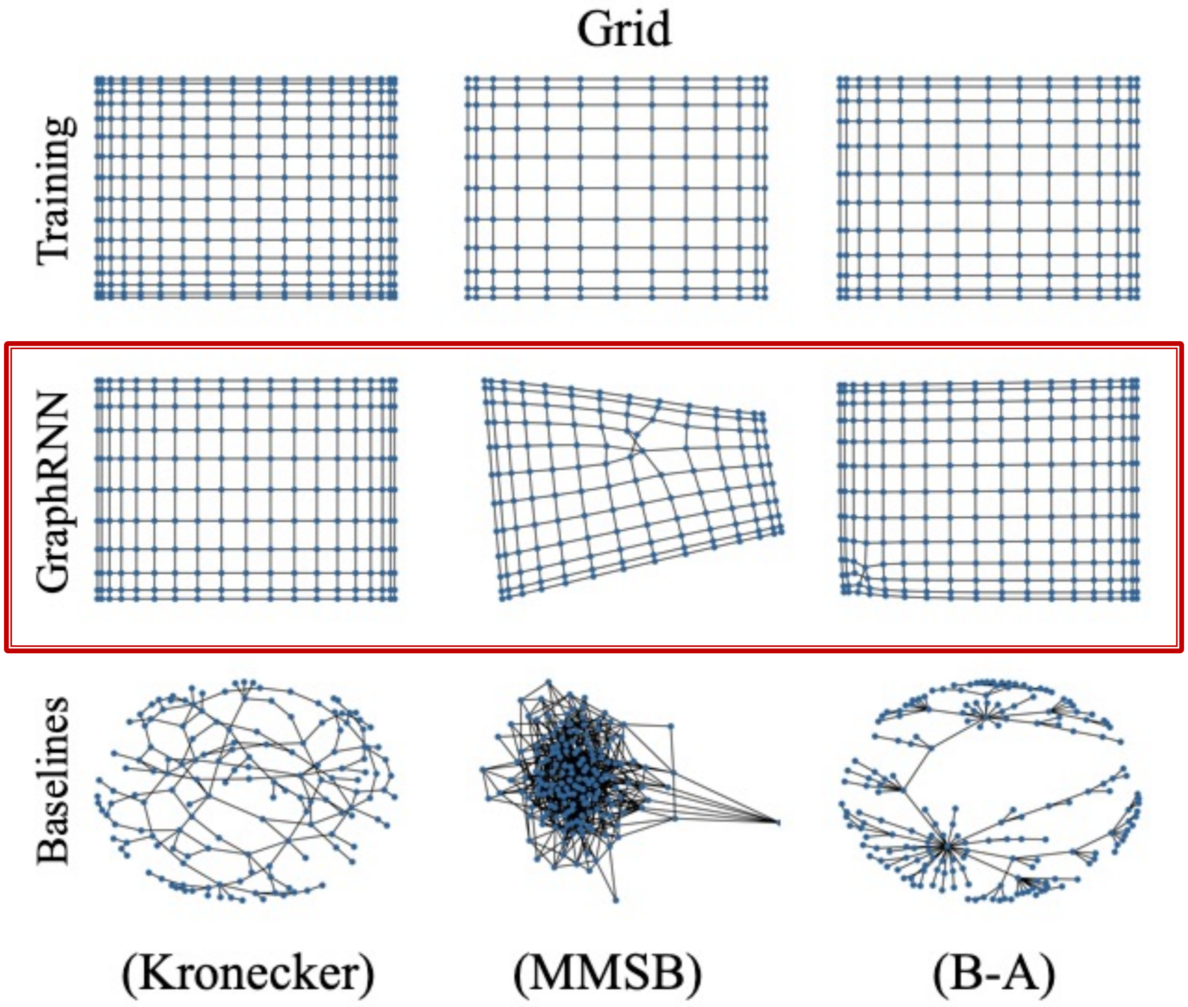


How similar?

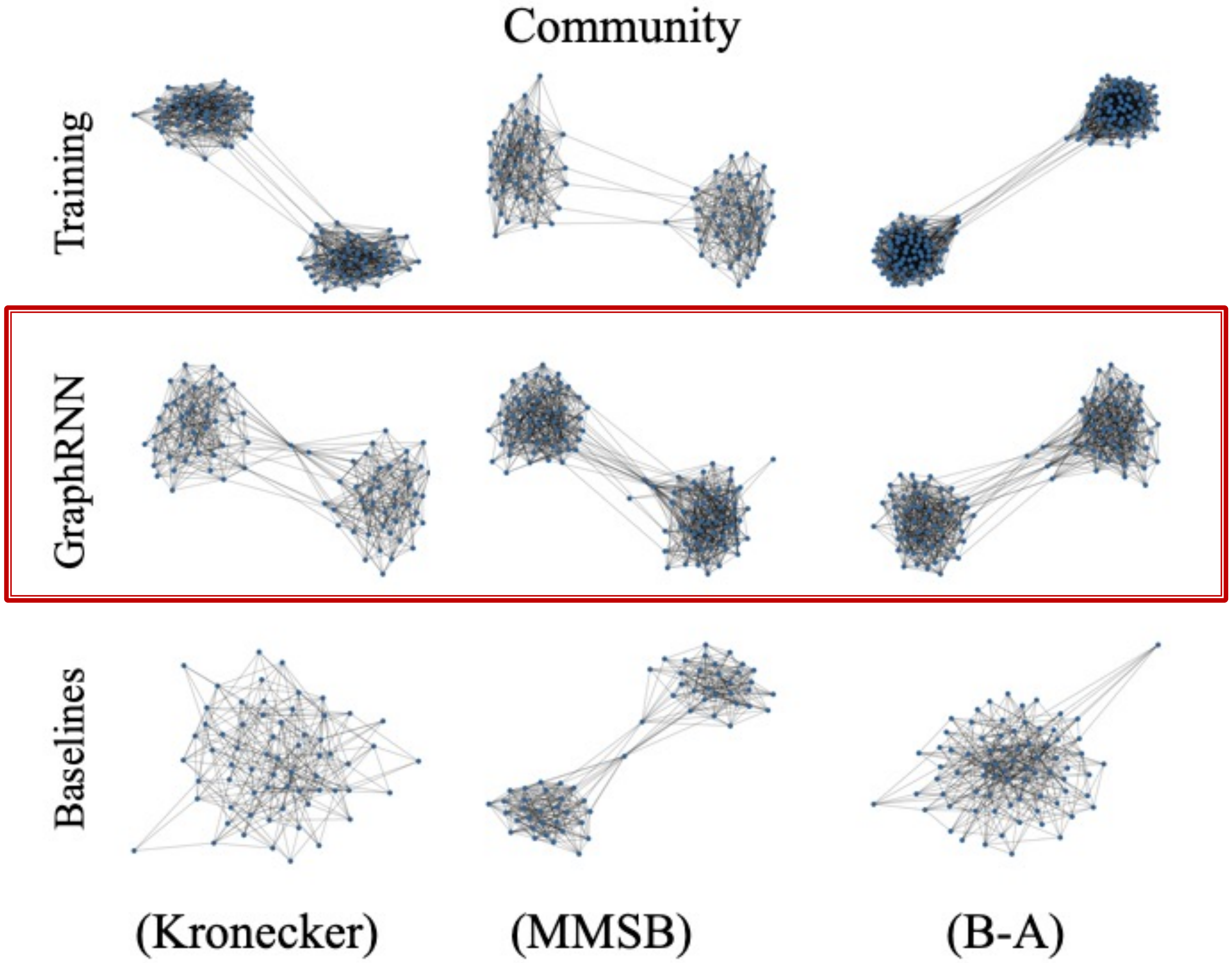
- **Goal:** Define similarity metrics for graphs

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

(1) Visual Similarity



(1) Visual Similarity



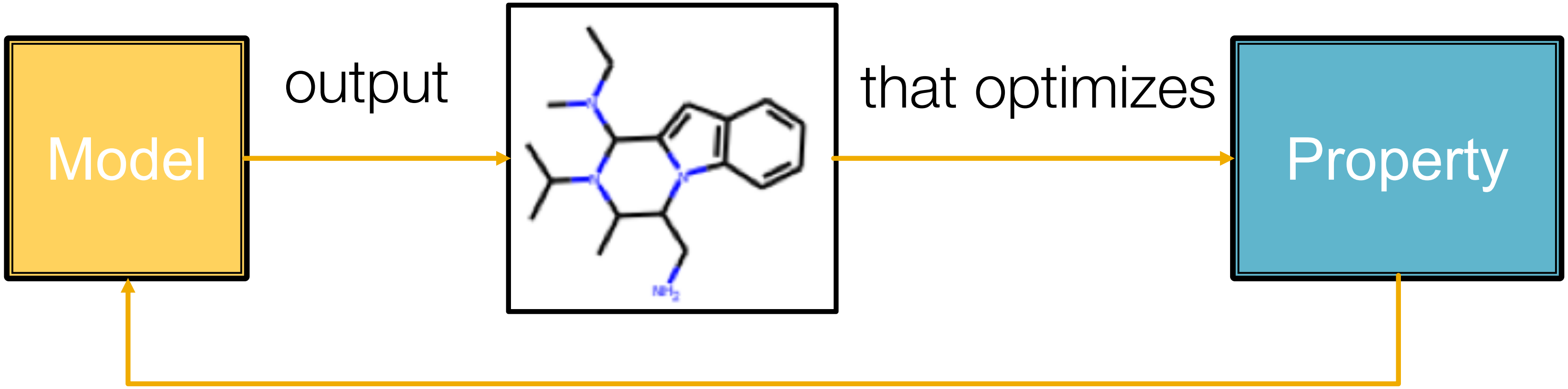
Stanford CS224W: Application of Deep Graph Generative Models to Molecule Generation

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

[You et al., NeurIPS 2018]

Application: Drug Discovery

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



e.g., `drug_likeness=0.95`

Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation. J. You, B. Liu, R. Ying, V. Pande, J. Leskovec. *Neural Information Processing Systems (NeurIPS)*, 2018.

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)

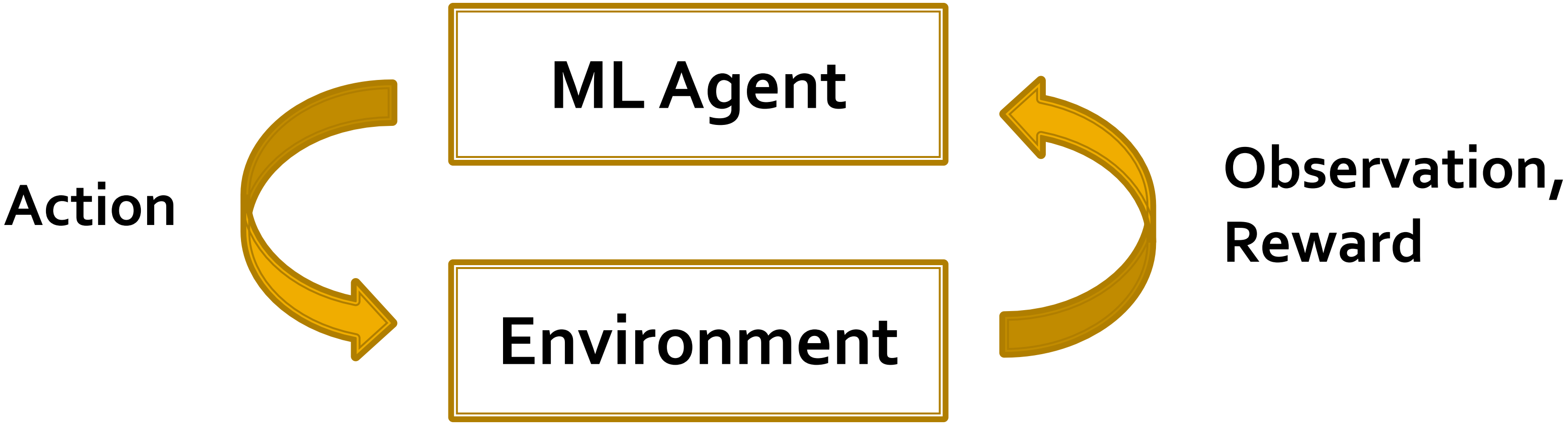
Including a "Black-box" to Graph Generation:

Objectives like drug-likeness are governed by physical law which is assumed to be unknown to us.

  - 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:

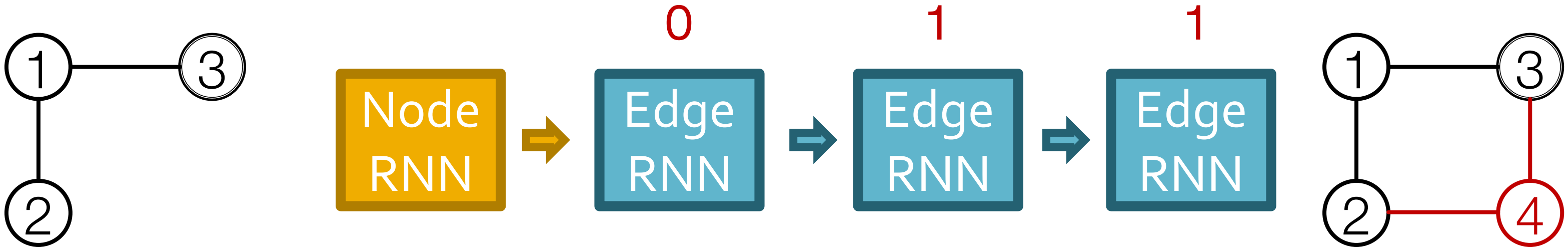
- Graph Neural Network captures graph structural information
- Reinforcement learning guides the generation towards the desired objectives
- Supervised training imitates examples in given datasets

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

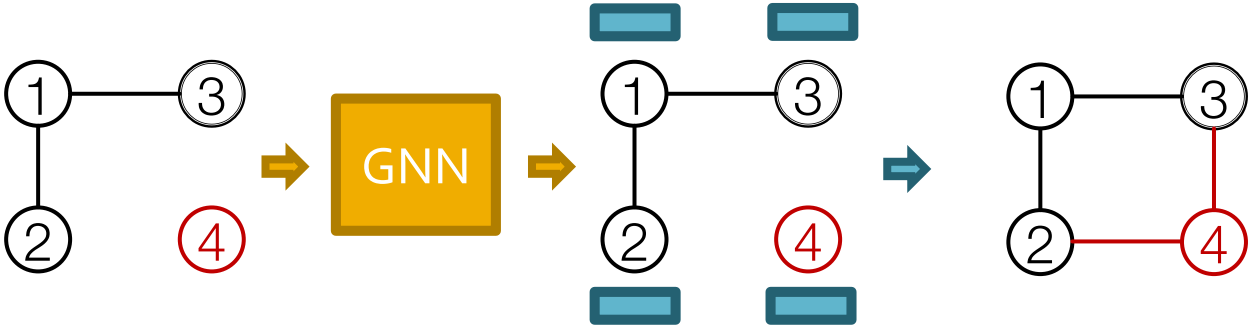
GCPN vs. GraphRNN

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



RNN hidden state captures the generated graph so far

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



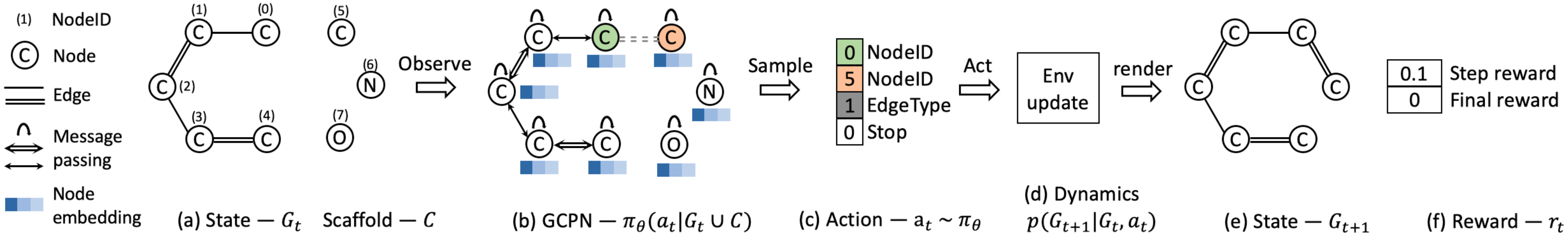
Node embeddings

Predict potential links using node embeddings

Recall the link prediction head:

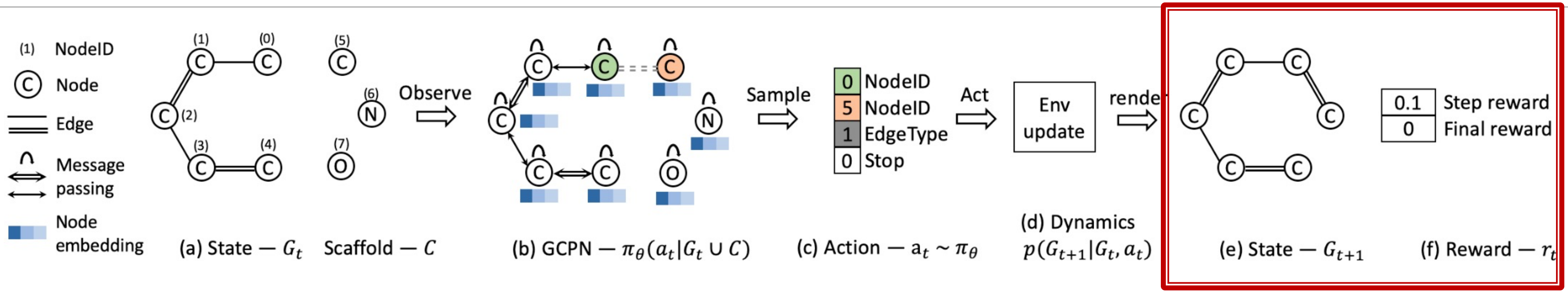
$$\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}) := \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$$

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

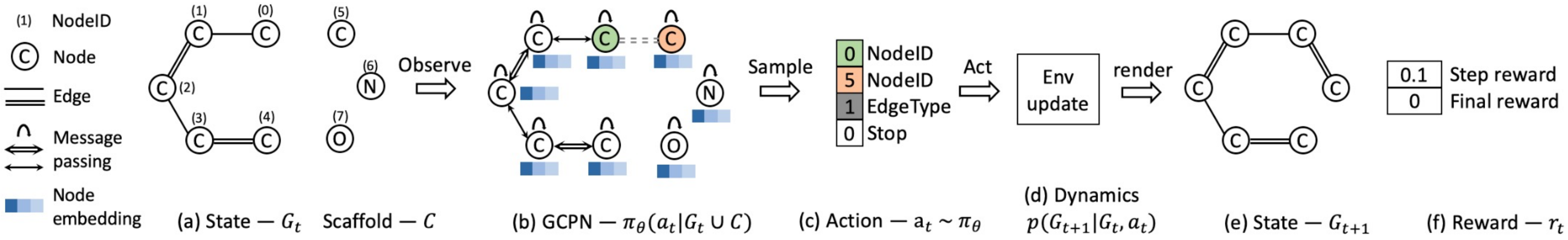
How Do We Set the Reward?



- **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?



(a) State — G_t Scaffold — C

(b) GCPN — $\pi_\theta(a_t|G_t \cup C)$

(c) Action — $a_t \sim \pi_\theta$

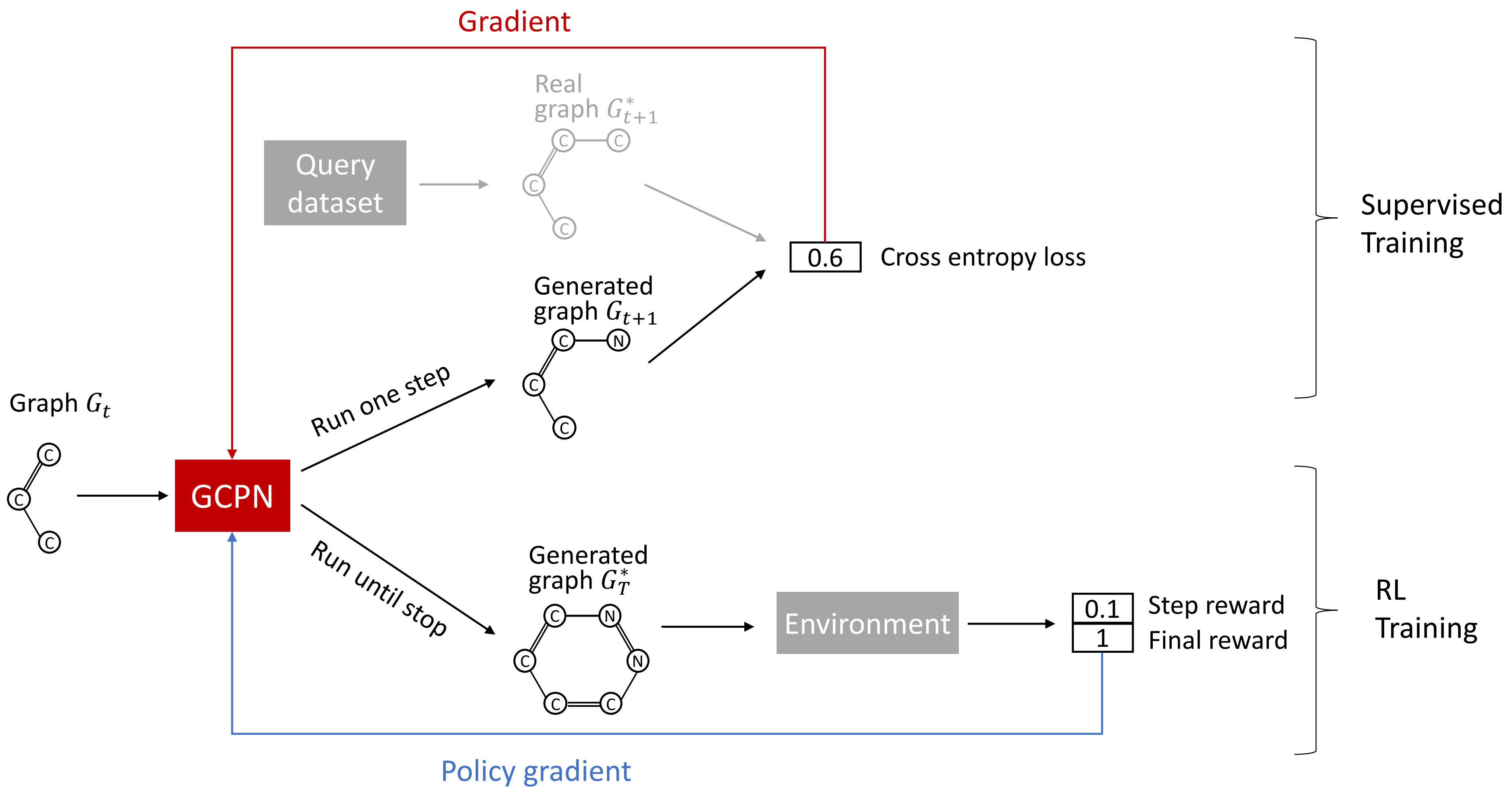
(d) Dynamics $p(G_{t+1}|G_t, a_t)$

(e) State — G_{t+1}

(f) Reward — r_t

- **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.
  - Refer to any RL course, e.g., CS234

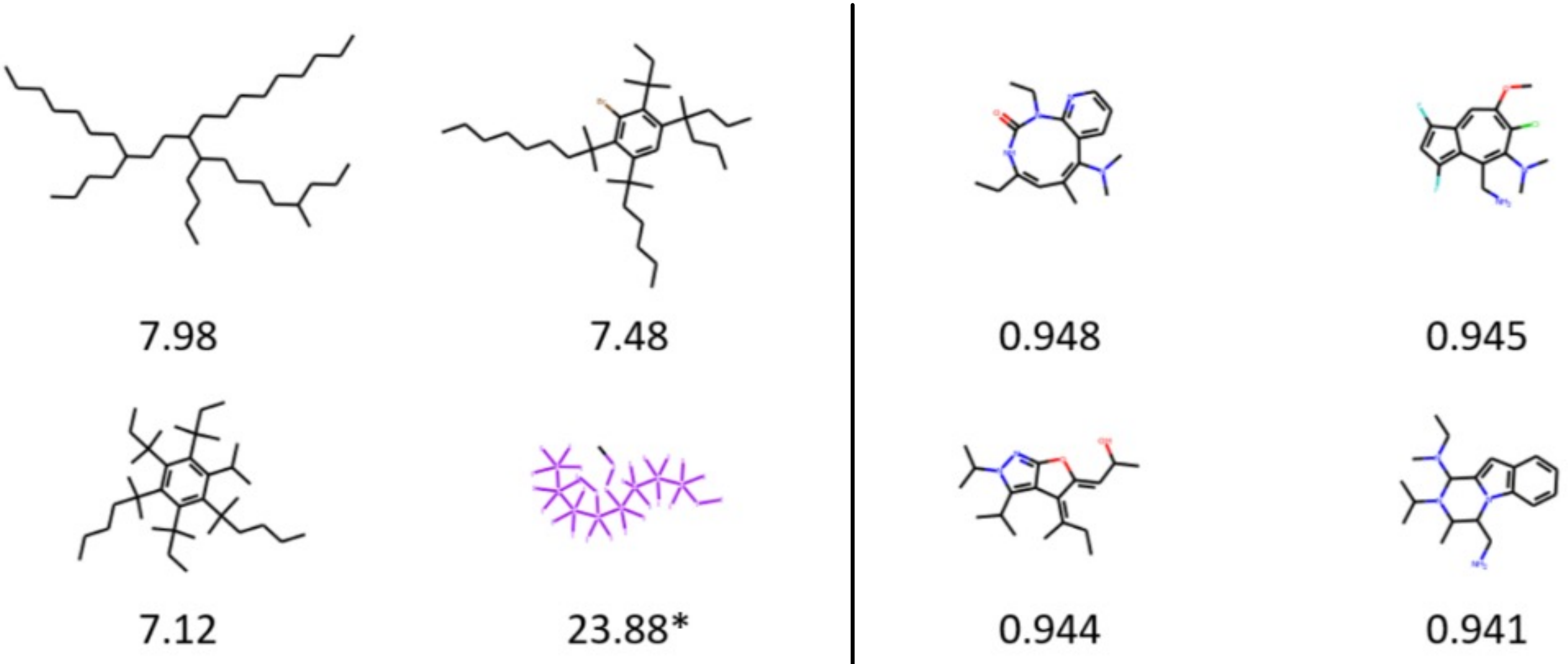
Training GCPN



Qualitative Results

Visualization of GCPN graphs:

- Property optimization Generate molecules with high specified property score



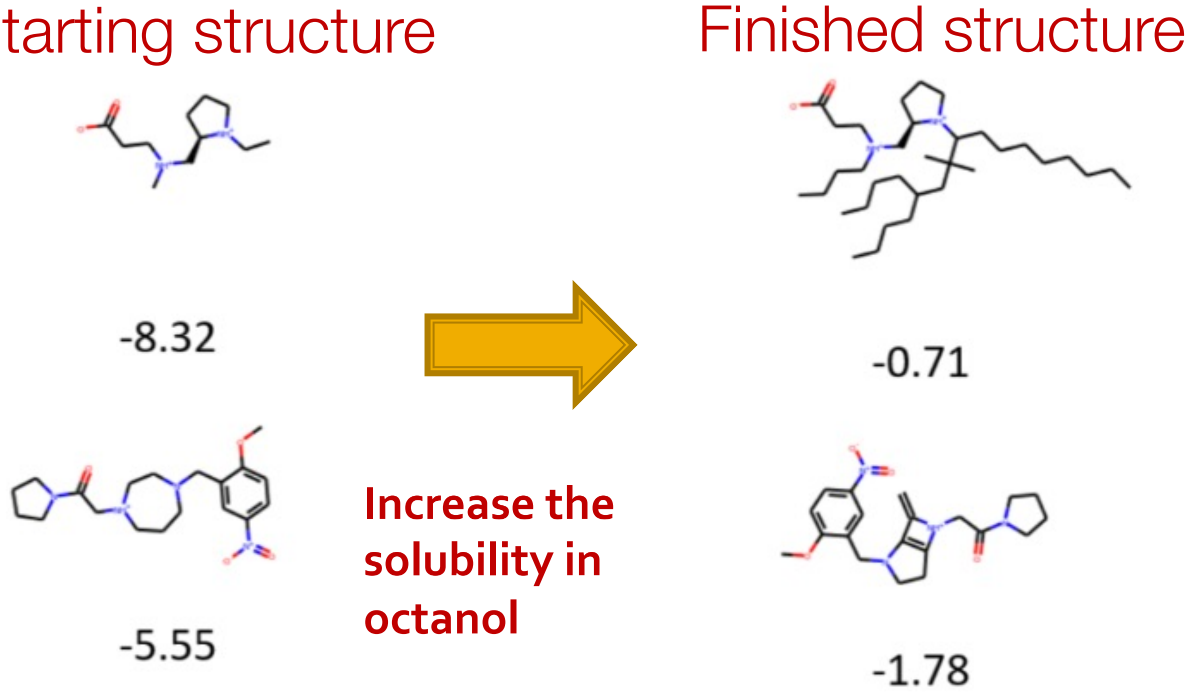
(a) Penalized logP optimization

(b) QED optimization

Qualitative Results

Visualization of GCPN graphs:

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



Increase the solubility in octanol

(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