Yale

Deep Generative Models for Graphs

CPSC483: Deep Learning on Graph-Structured Data

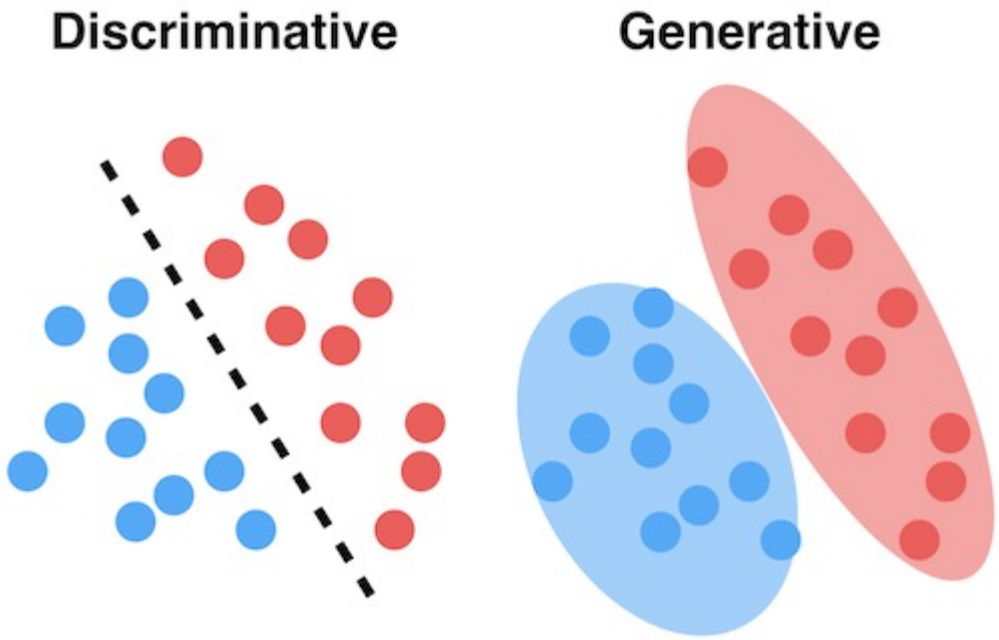
Rex Ying

Readings

- Readings are updated on the website (syllabus page)
- **Lecture 10 readings:**
  - Position-aware Graph Neural Networks
  - Identity-aware Graph Neural Networks
- **Lecture 11 readings:**
  - Traditional graph generative models:
    - Erdős–Rényi model, Barabási–Albert model, Watts–Strogatz model, Kronecker Graphs
    - More generators from NetworkX (under Random Graphs)

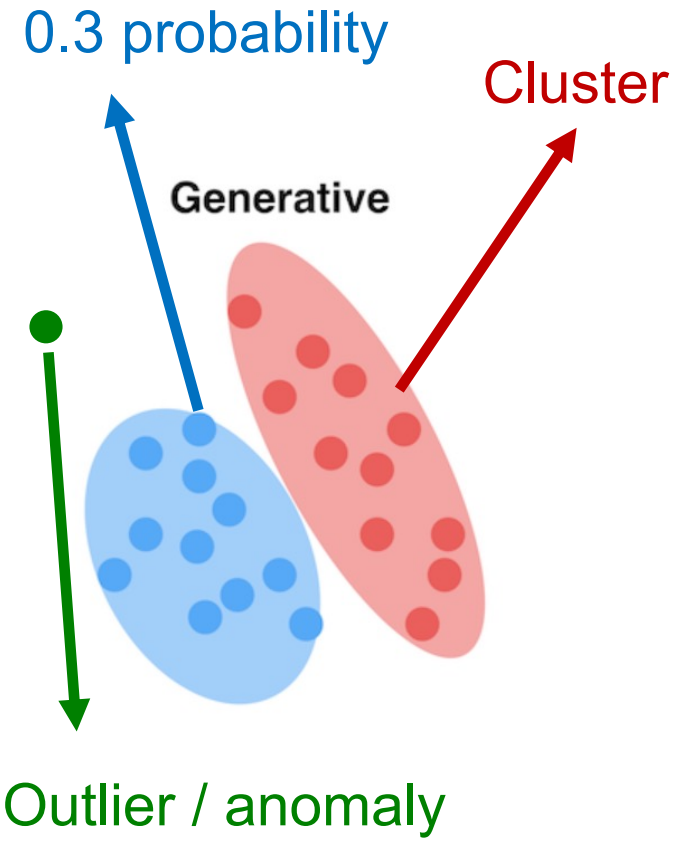
Generative Model

- 2 fundamental philosophies to approach machine learning
- **Discriminative model:** find decision boundaries between classes
  - **Classical models:** decision tree, random forest, logistic regression, SVM …
  - **Deep models:** any feedforward neural nets (CNN, GNN, RNN, transformer …)
- **Generative model:** model the underlying (conditional) distribution of data
  - **Classical models:** naïve Bayes, graphical models…
  - **Deep models:** VAE, deep autoregressive model, GAN, normalizing flow, diffusion model …



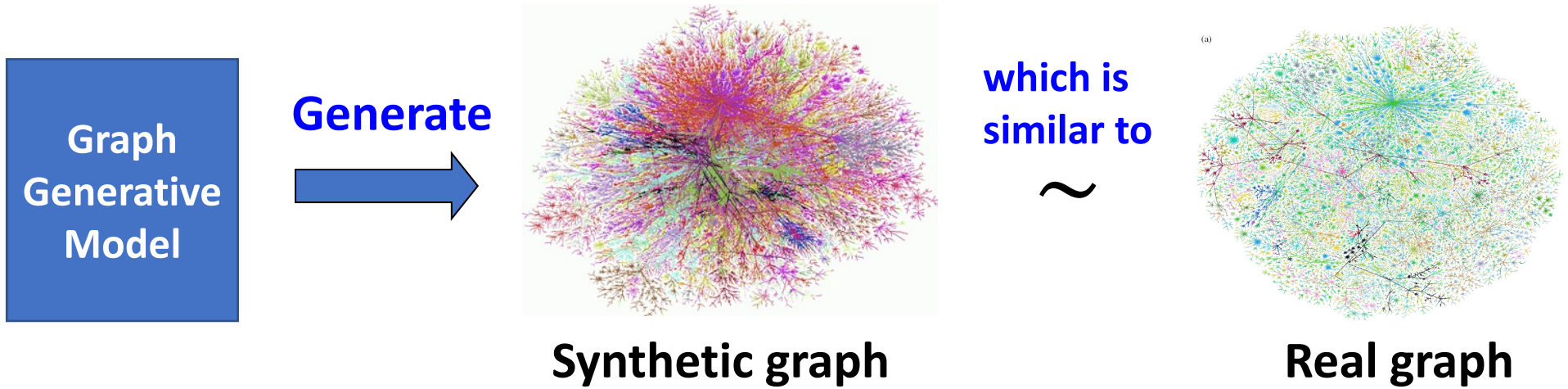
Why Generative Model?

- Generative models are typically useful for **unsupervised** learning
  - Clustering (community detection)
- Provides a **probabilistic** interpretation for the prediction (posterior distribution)
  - anomaly detection, uncertainty approximation …
- Most importantly: we can **sample** from the model!
  - For example, image generation from text:



Recap: Graph Generation Problem

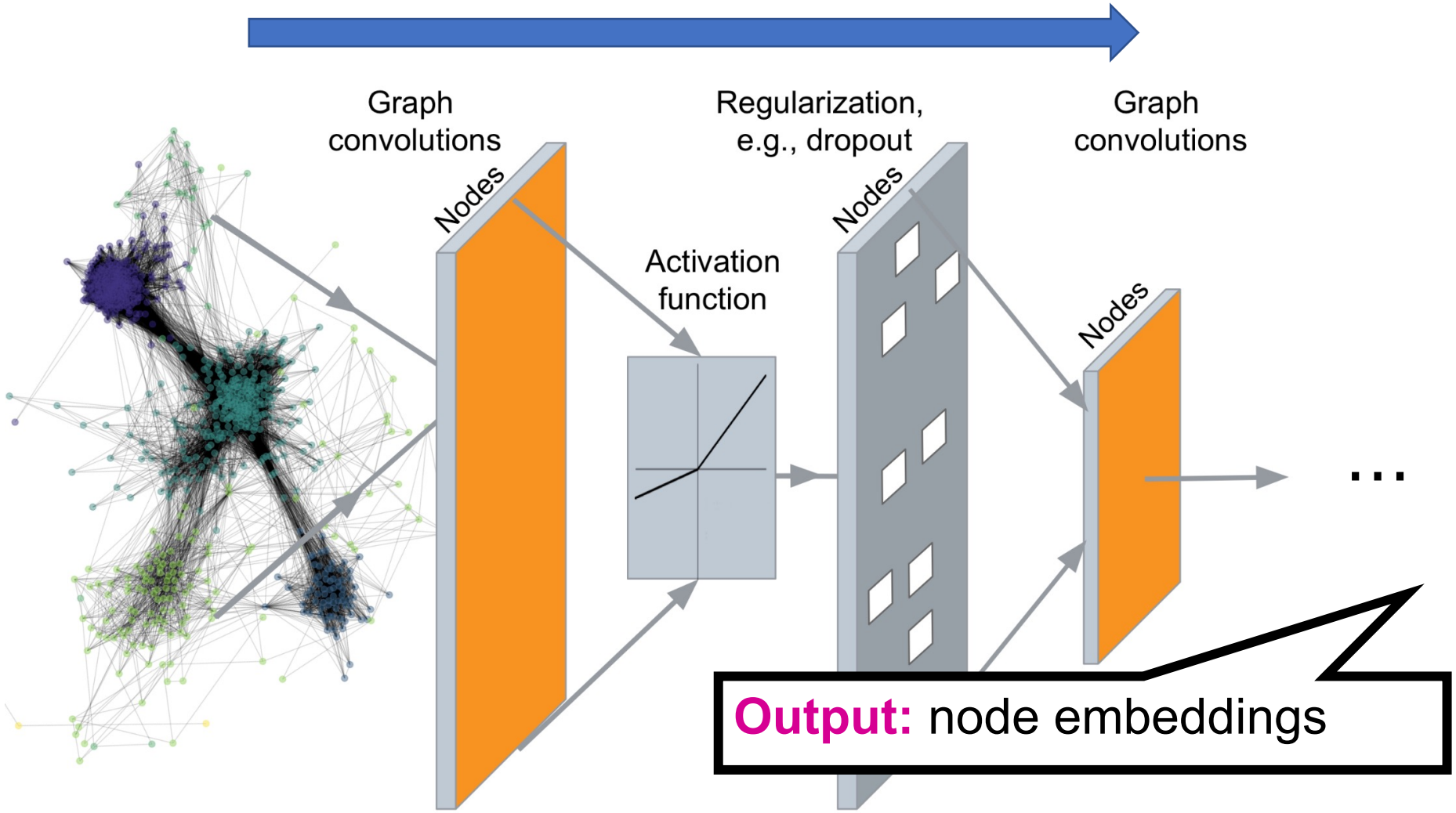
- We want to generate realistic graphs, using **graph generative models**



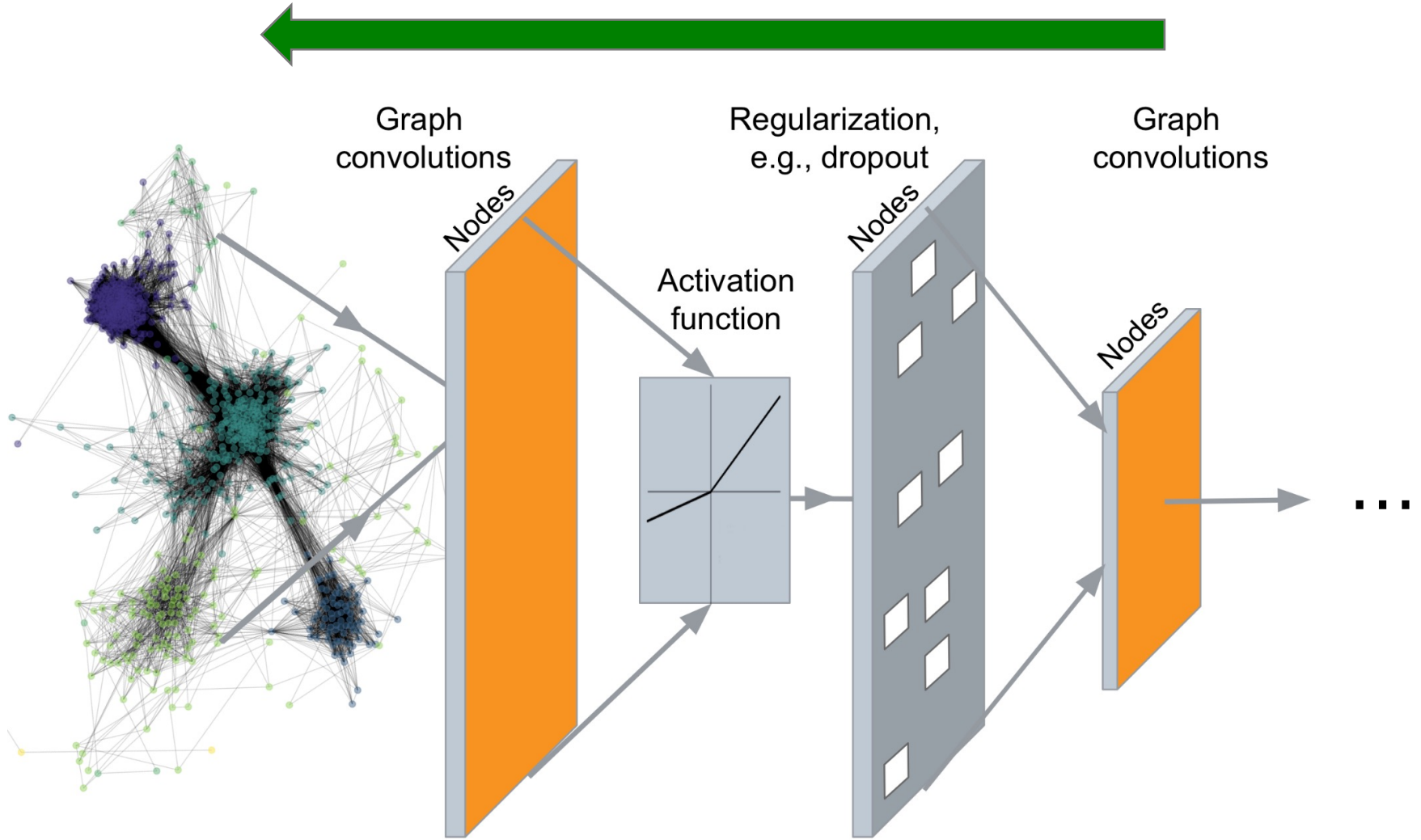
Roadmap of Graph Generation

- **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 (see readings)
- **Step 3: Deep graph generative models**
  - Learn the graph formation process from the data
  - **This lecture!**

Deep Graph Encoders



Today: Deep Graph Decoders



Deep Generative Models for Graphs

- **Machine Learning for Graph Generation**
- **GraphRNN: Generating Realistic Graphs**
- **Application of Deep Graph Generative Models**

Deep Generative Models for Graphs

- **Machine Learning for Graph Generation**
- **GraphRNN: Generating Realistic Graphs**
- **Application of Deep Graph Generative Models**

Graph Generation Tasks

Task 1: Realistic graph generation

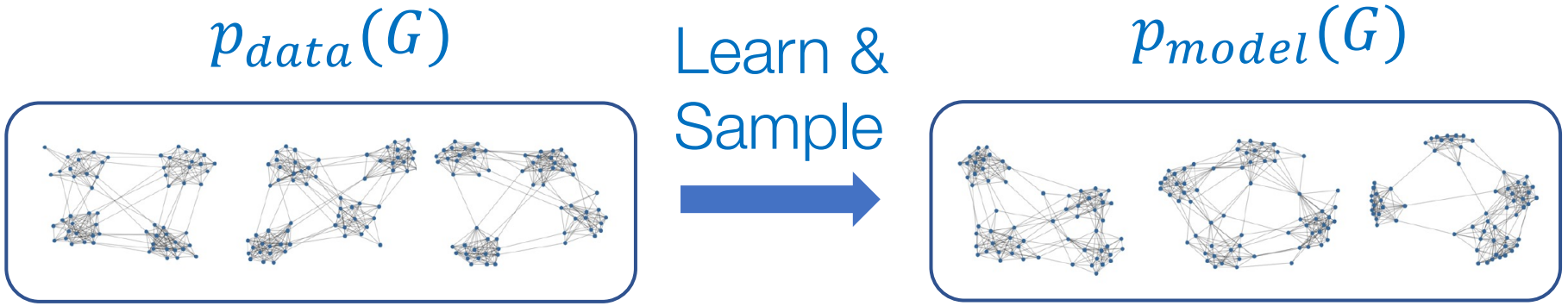
- Generate graphs that are **similar to a given set of graphs**

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 $p_{model}(G)$



Graph Generative Model Problem Setup

Setup:

- Assume we want to learn a generative model from a set of data points (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 $p_{model}(x; \theta)$ close to $p_{data}(x)$ (**Density estimation**)
  - **(2)** Make sure we can sample from $p_{model}(x; \theta)$ (**Sampling**)
    - We need to generate examples (graphs) from $p_{model}(\boldsymbol{x}; \theta)$

Generative Models Basics: Density Estimation

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

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

  - Find parameters θ^*, such that for observed data points $x_i \sim p_{data}$, $\sum_i \log p_{model}(x_i;\theta^*)$ 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: **(maximum likelihood)**

Generative Models Basics: Sampling

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

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

$$z_i \sim N(0,1)$$

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

$$x_i = f(z_i; \theta)$$

  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:

- $\boldsymbol{p_{model}(x;\theta)}$ is used for **both density estimation and sampling**
  - 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:

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

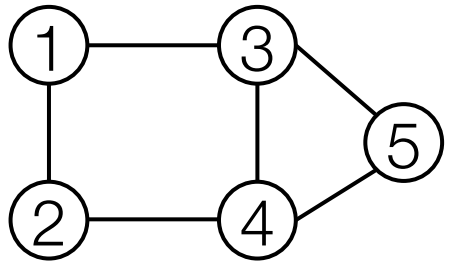
    - E.g., $\boldsymbol{x}$ is a vector, x_t is the t-th dimension;
      $\boldsymbol{x}$ is a sentence, x_t is the t-th word.
    - **In our case:** x_t will be the t-th action (add node, add edge)

Deep Generative Models for Graphs

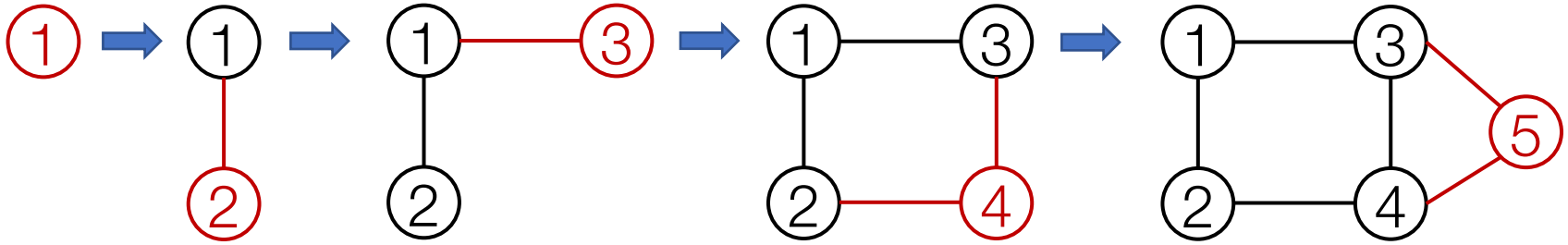
- Machine Learning for Graph Generation
- **GraphRNN: Generating Realistic Graphs**
- Application of Deep Graph Generative Models

GraphRNN Idea

Generating graphs via sequentially adding nodes and edges



Generation process S^{π}

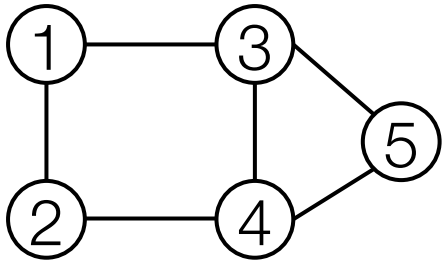


GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. J. You, R. Ying, X. Ren, W. L. Hamilton, J. Leskovec. *International Conference on Machine Learning (ICML)*, 2018.

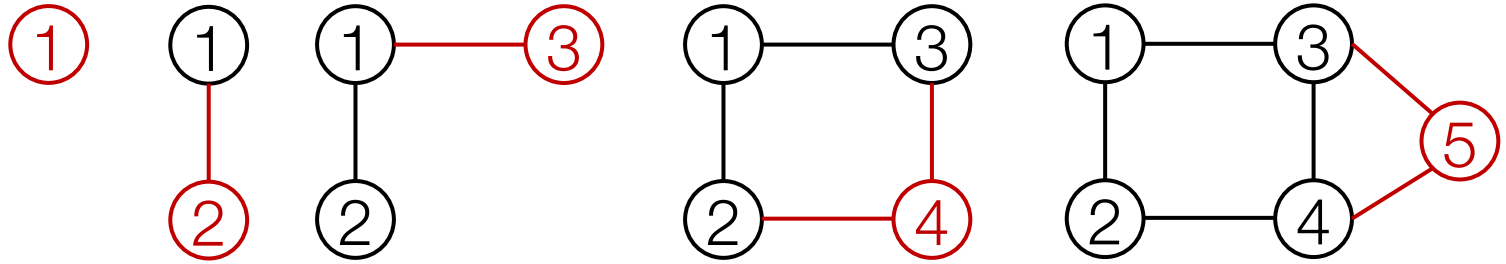
Model Graphs as Sequences

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}:



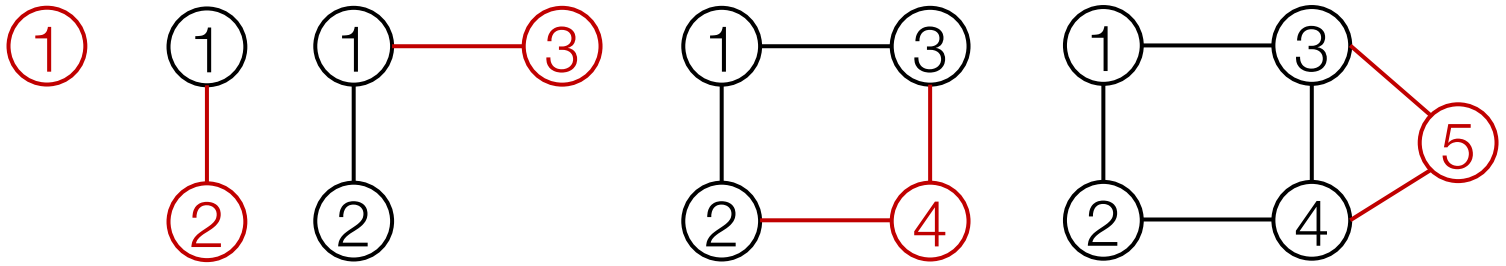
$$S^{\pi} = (S_1^{\pi}, S_2^{\pi}, S_3^{\pi}, S_4^{\pi}, S_5^{\pi})$$

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



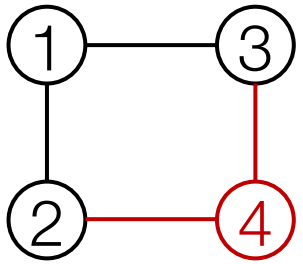
$$S^\pi = (\ S_1^\pi\ ,\ S_2^\pi\ ,\ S_3^\pi\ ,\ S_4^\pi\ ,\ S_5^\pi\ )$$

"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}

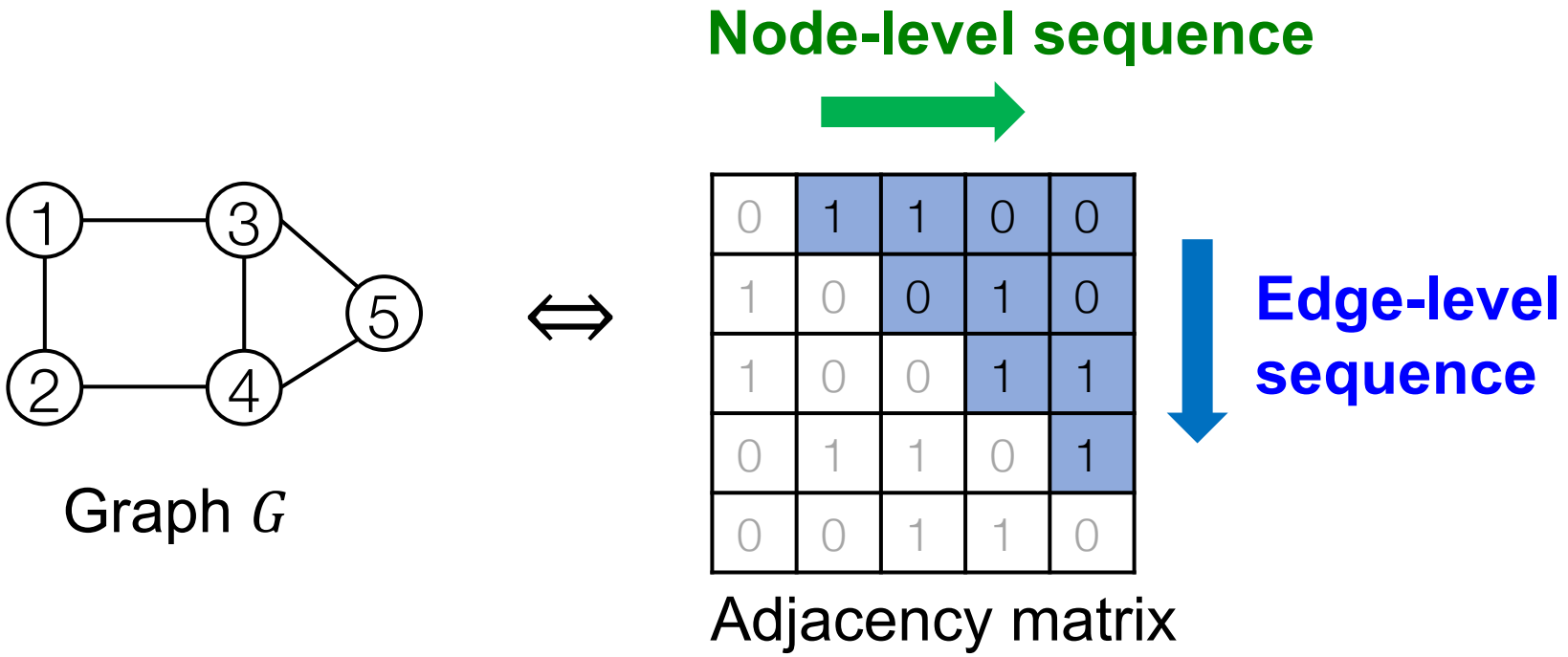
$$S_4^{\pi} = \left(S_{4,1}^{\pi}, S_{4,2}^{\pi}, S_{4,3}^{\pi} \right)$$

"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

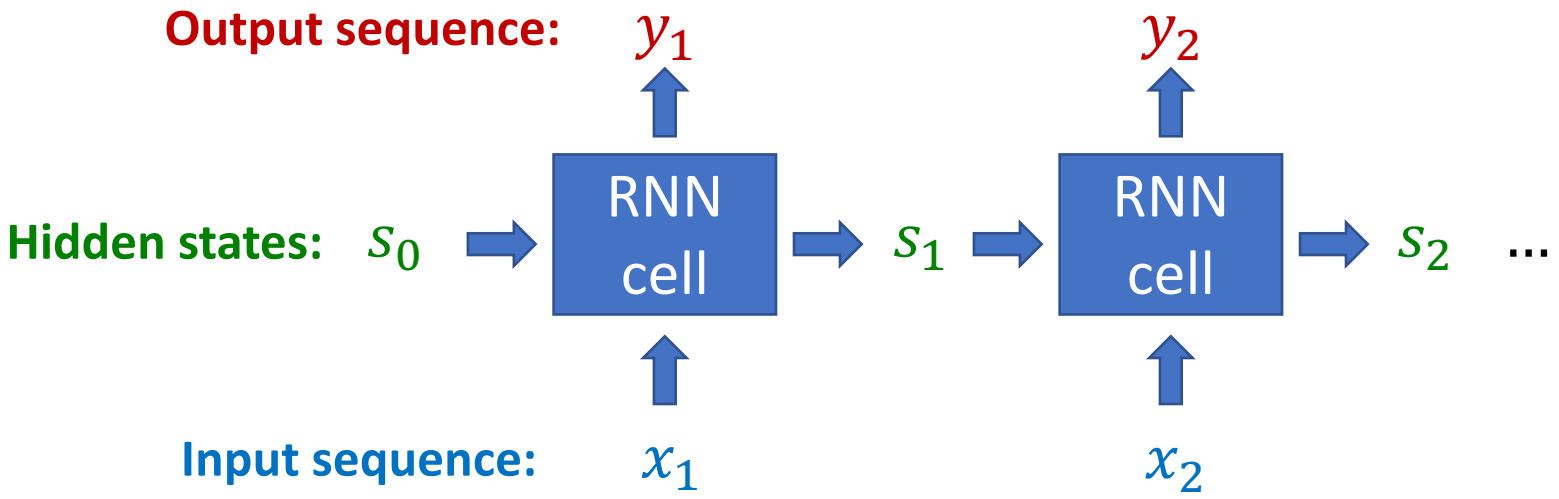
- **We have transformed graph generation problem into a sequence generation problem**

- **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!

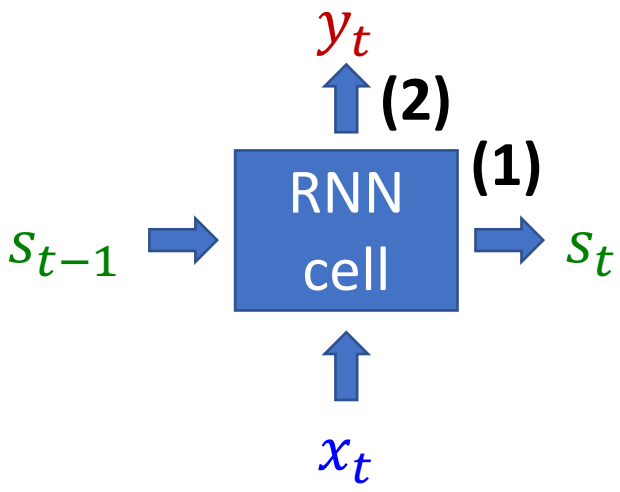
Background: Recurrent NNs

- 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
- x_t: **Input** to RNN at step t
- 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.

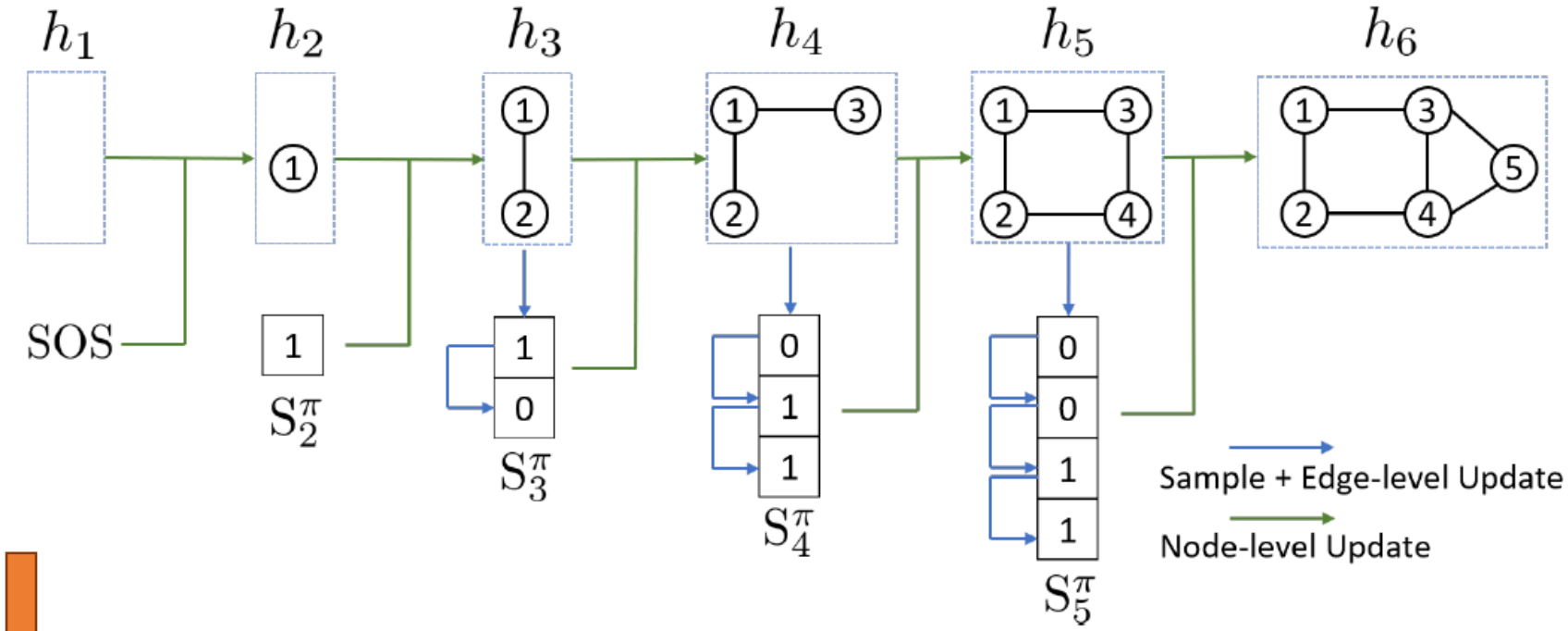
GraphRNN: Two levels of RNN

- **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

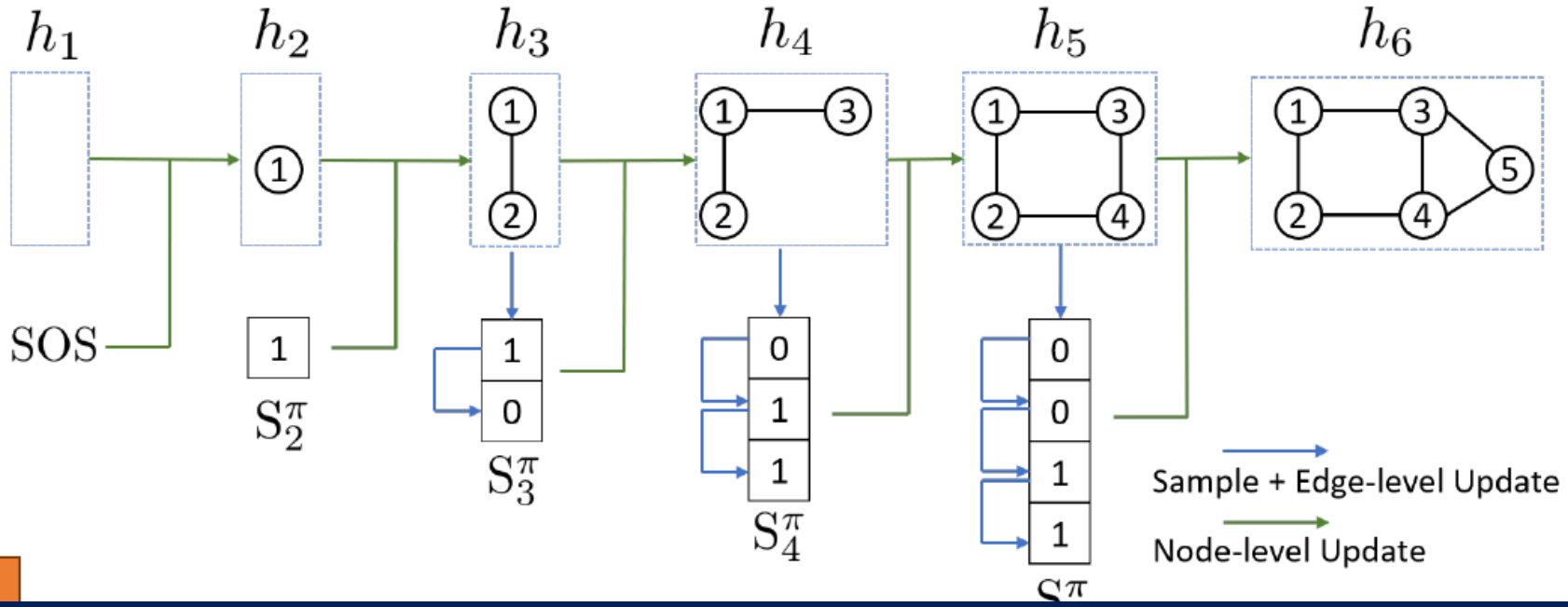
Node-level RNN generates the initial state for edge-level RNN



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?

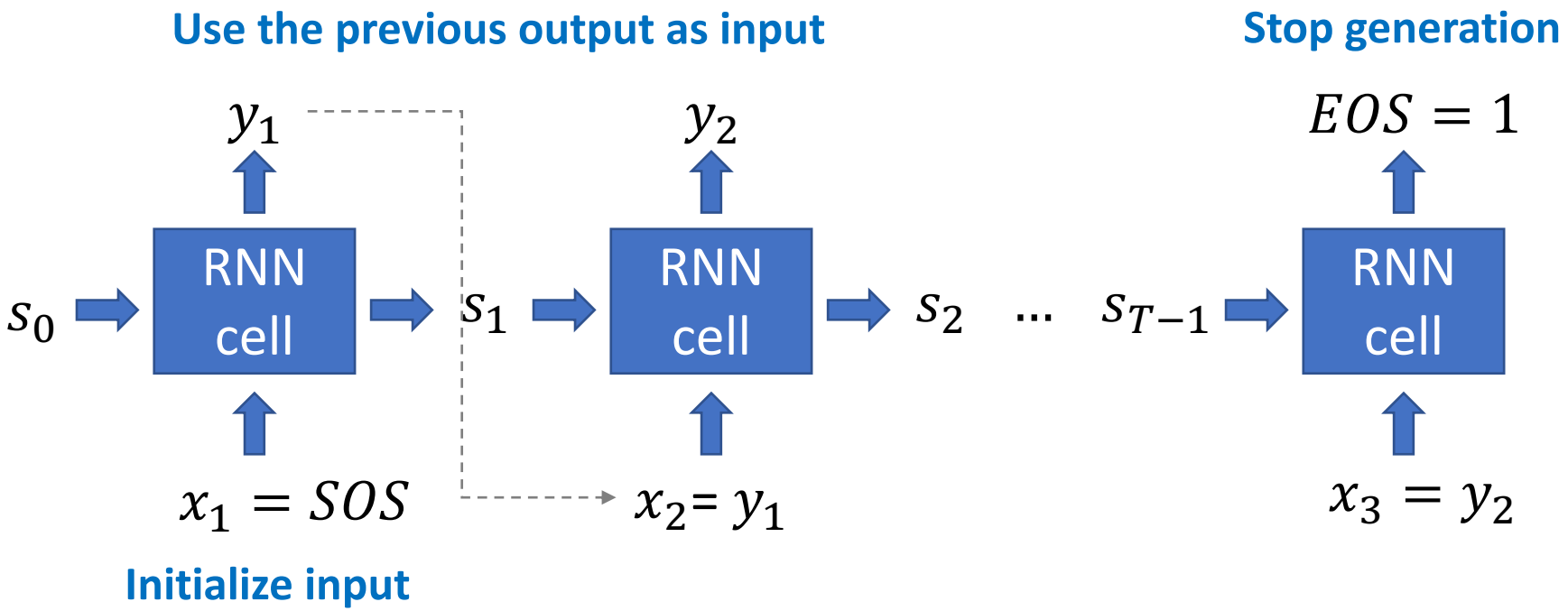
RNN for Sequence Generation

- **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

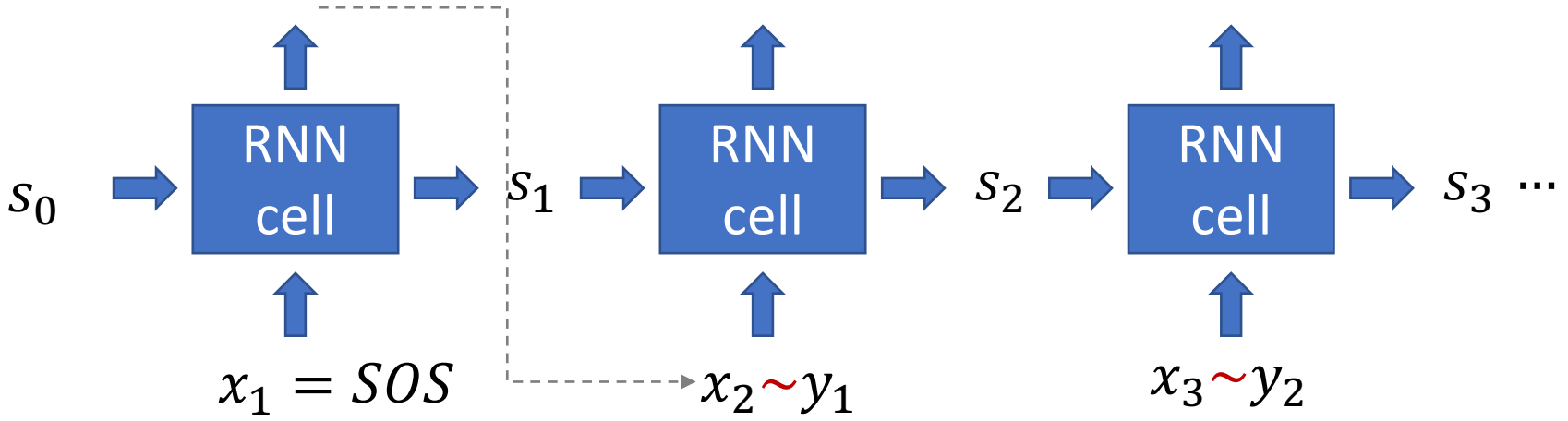
RNN for Sequence Generation



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

RNN for Sequence Generation

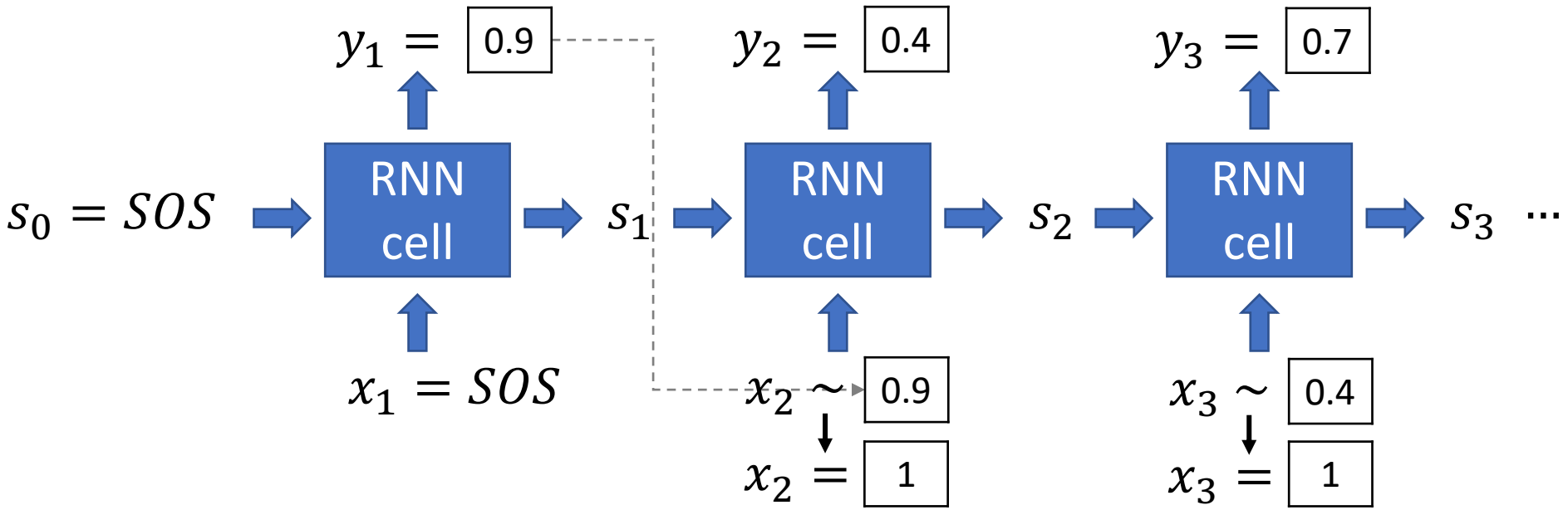
- **Remember our goal**: Use RNN to model $\prod_{k=1}^{n} p_{model}(x_t|x_1, \dots, x_{t-1}; \theta)$
- Let $y_t = p_{model}(x_t|x_1, \dots, x_{t-1}; \theta)$
- Then we need to sample x_{t+1} from y_t: $x_{t+1} \sim y_t$
  - Each step of RNN outputs a **probability of a single edge**
  - We then sample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



RNN at Test Time

Suppose we already have trained the model

- At each step, the model predicts y_t (a **scalar**), following a **Bernoulli distribution**
- p means value 1 has probability p, value 0 has probability $1 - p$

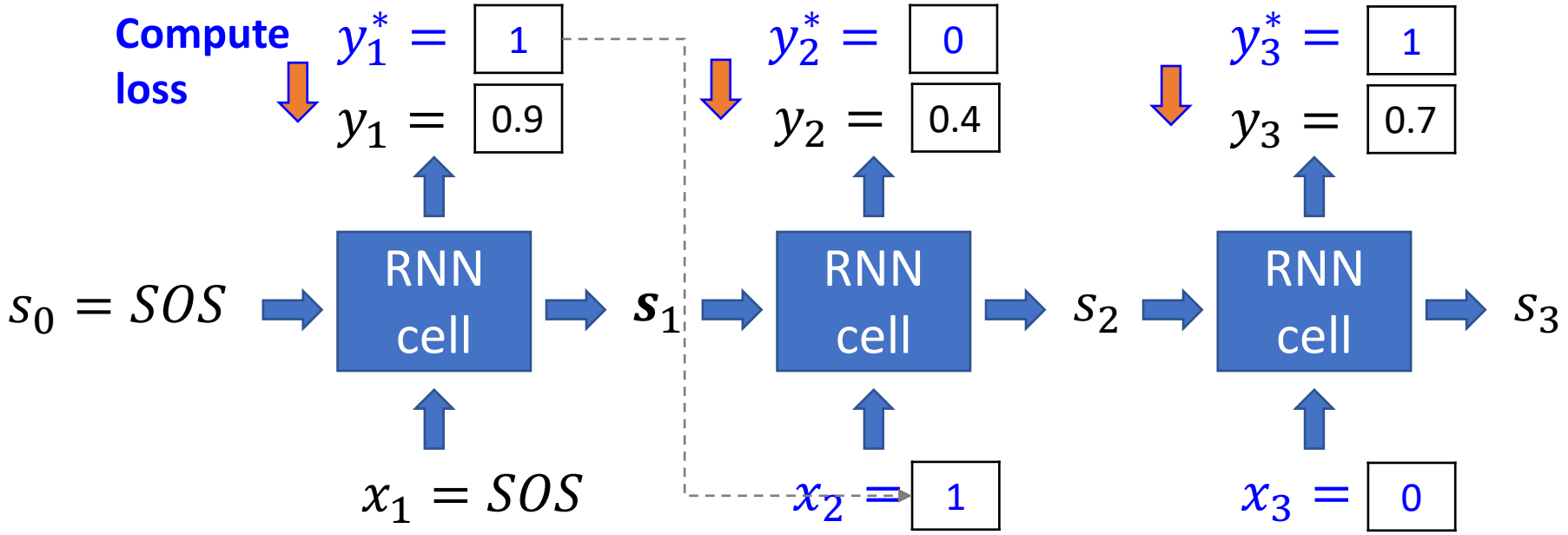


- **How do we use training data $x_1, x_2, \ldots, x_n$?**

RNN at Training Time

Training the model:

- We observe a sequence y^* of edges [1,0,…]
- **Principle**: **Teacher Forcing** -- Replace input and output by the real sequence



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

$y_1^* =$ 1

$y_1 =$ 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, and this loss will **backprop through RNN parameters!**

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

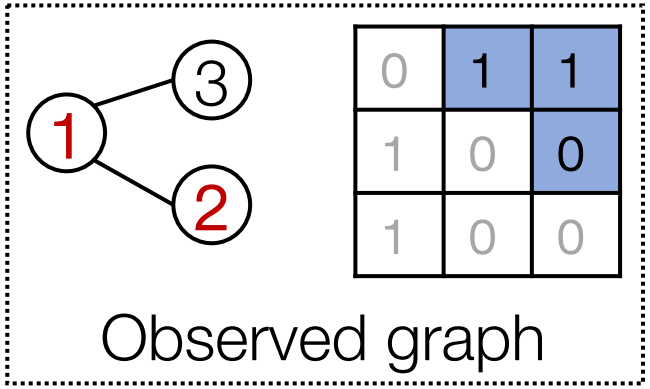
(3) **Add another new node:** We use the last hidden state of Edge RNN to run Node RNN for another step

(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

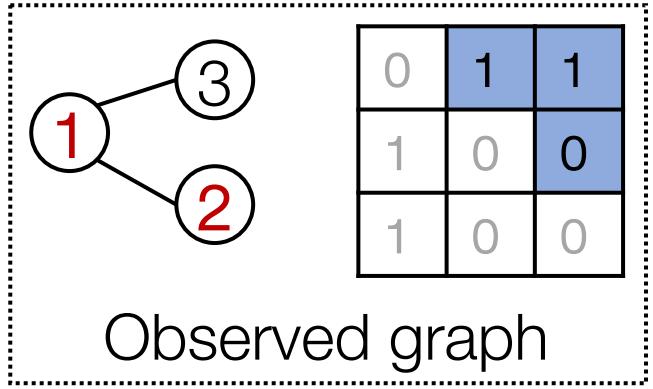
Now adding **Node 2**

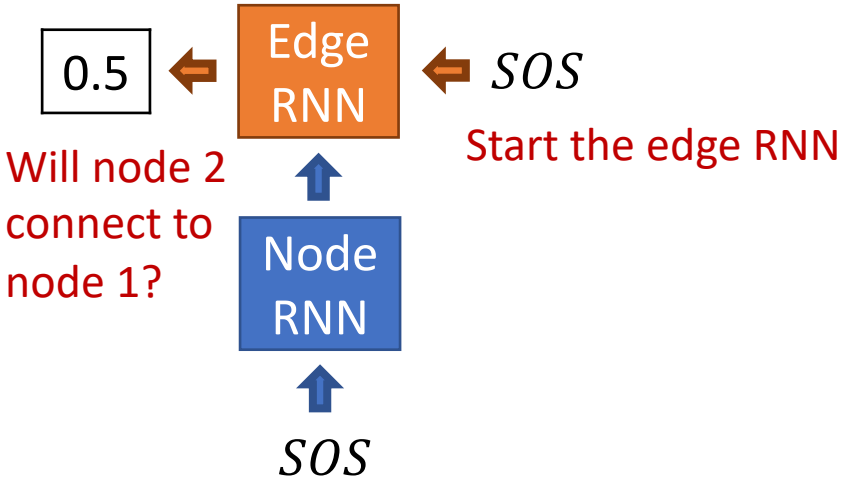


Start the node RNN

Put Things Together: Training

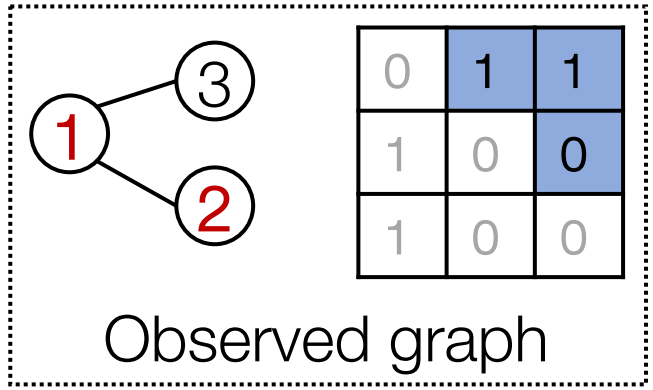
Edge RNN predicts how
Node 2 connects to **Node 1**

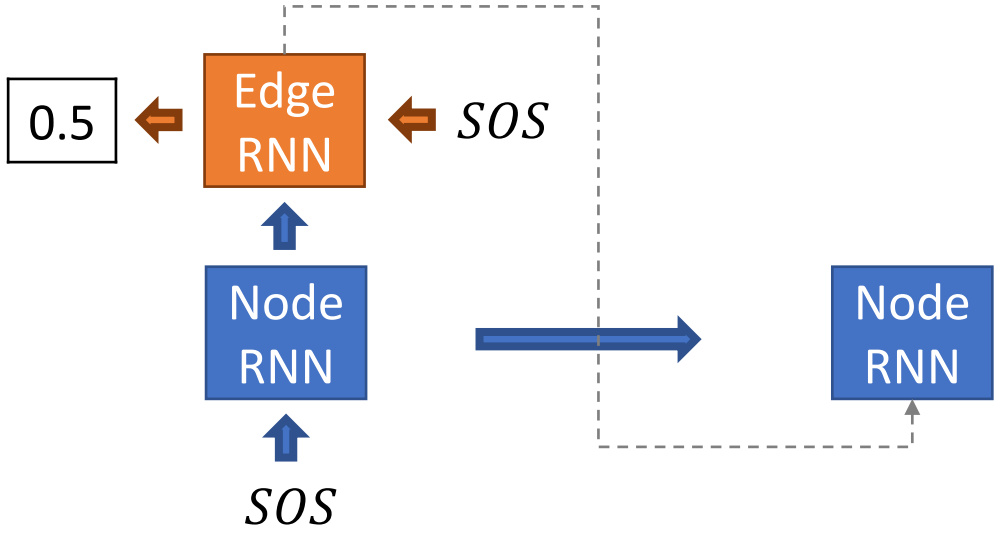




Put Things Together: Training

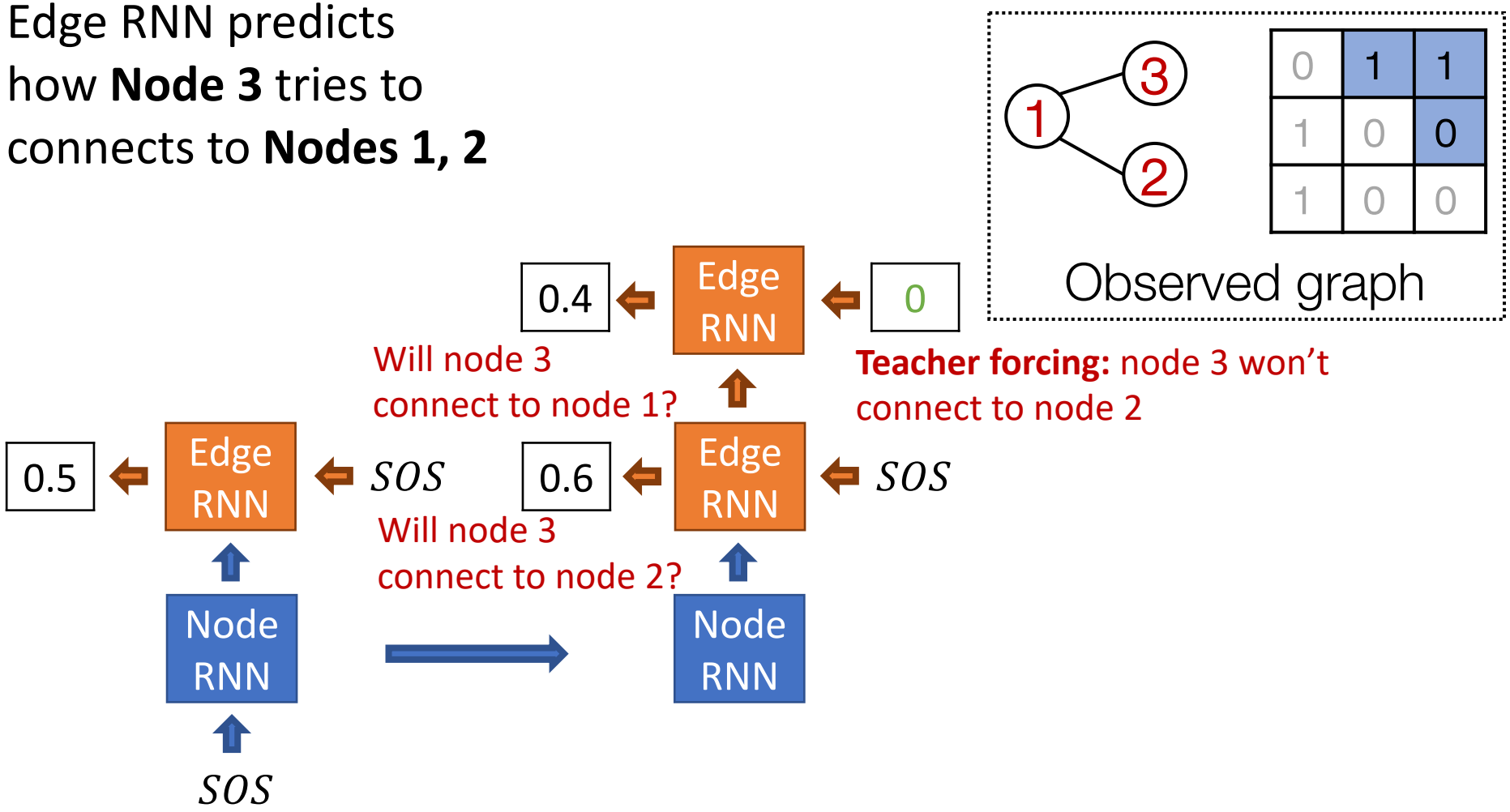
Update Node RNN using
Edge RNN's hidden state





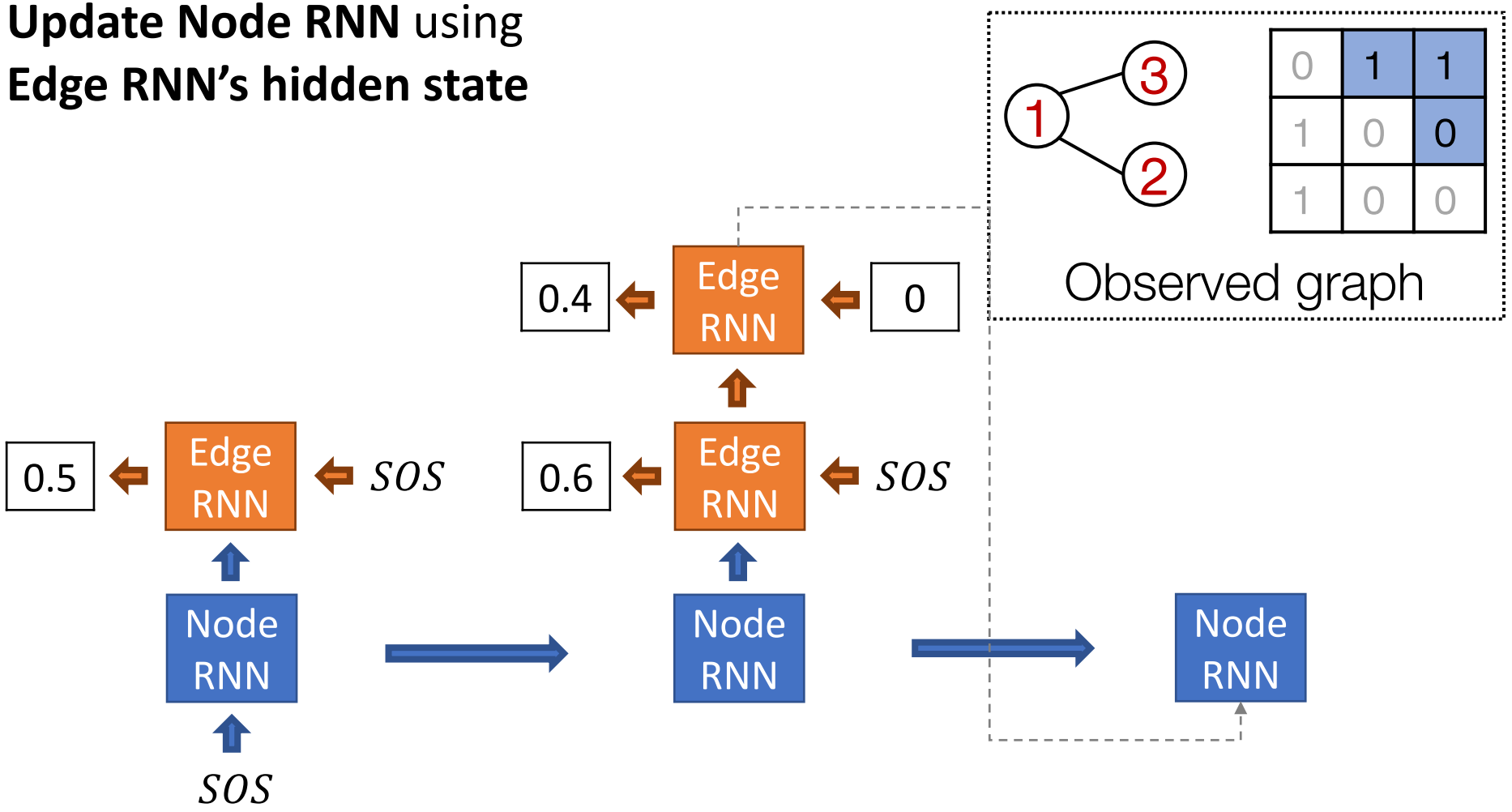
Put Things Together: Training

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



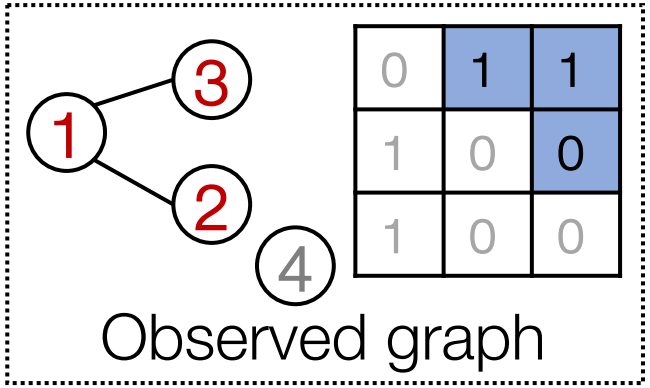
Put Things Together: Training

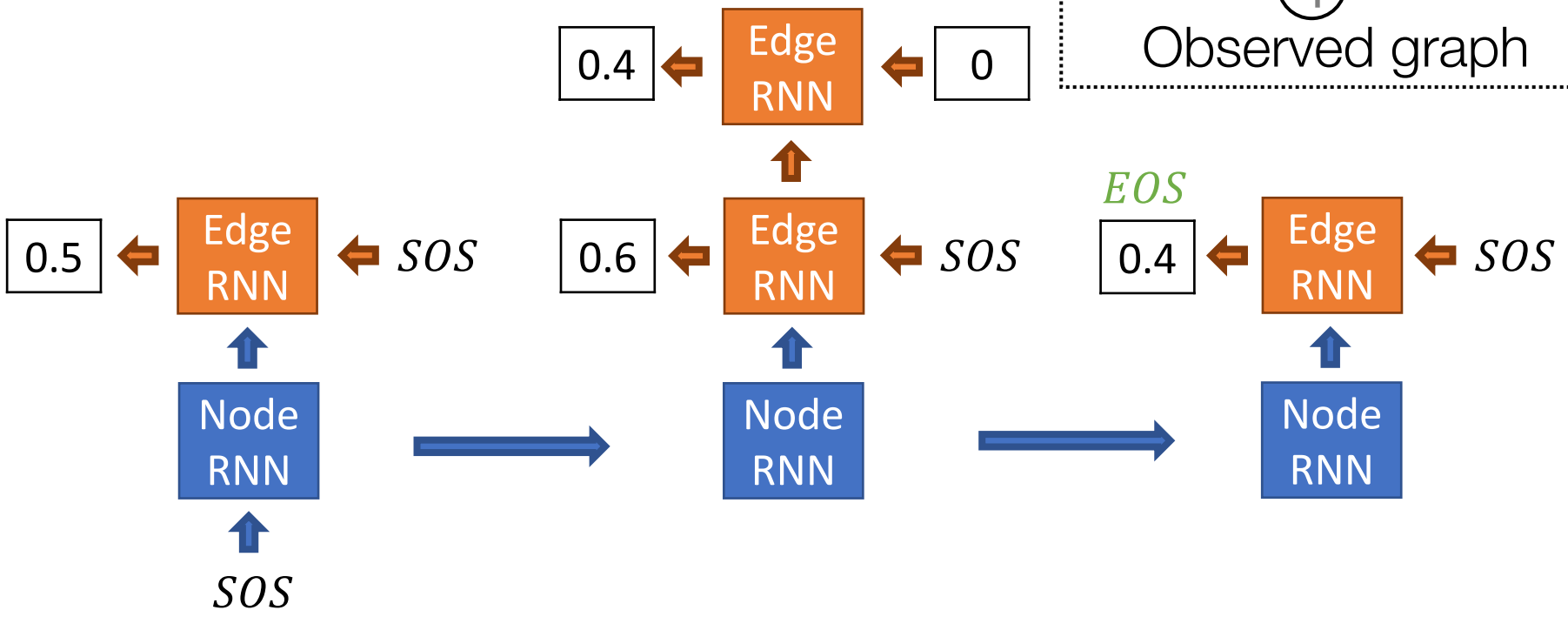
Update Node RNN using
Edge RNN's hidden state



Put Things Together: Training

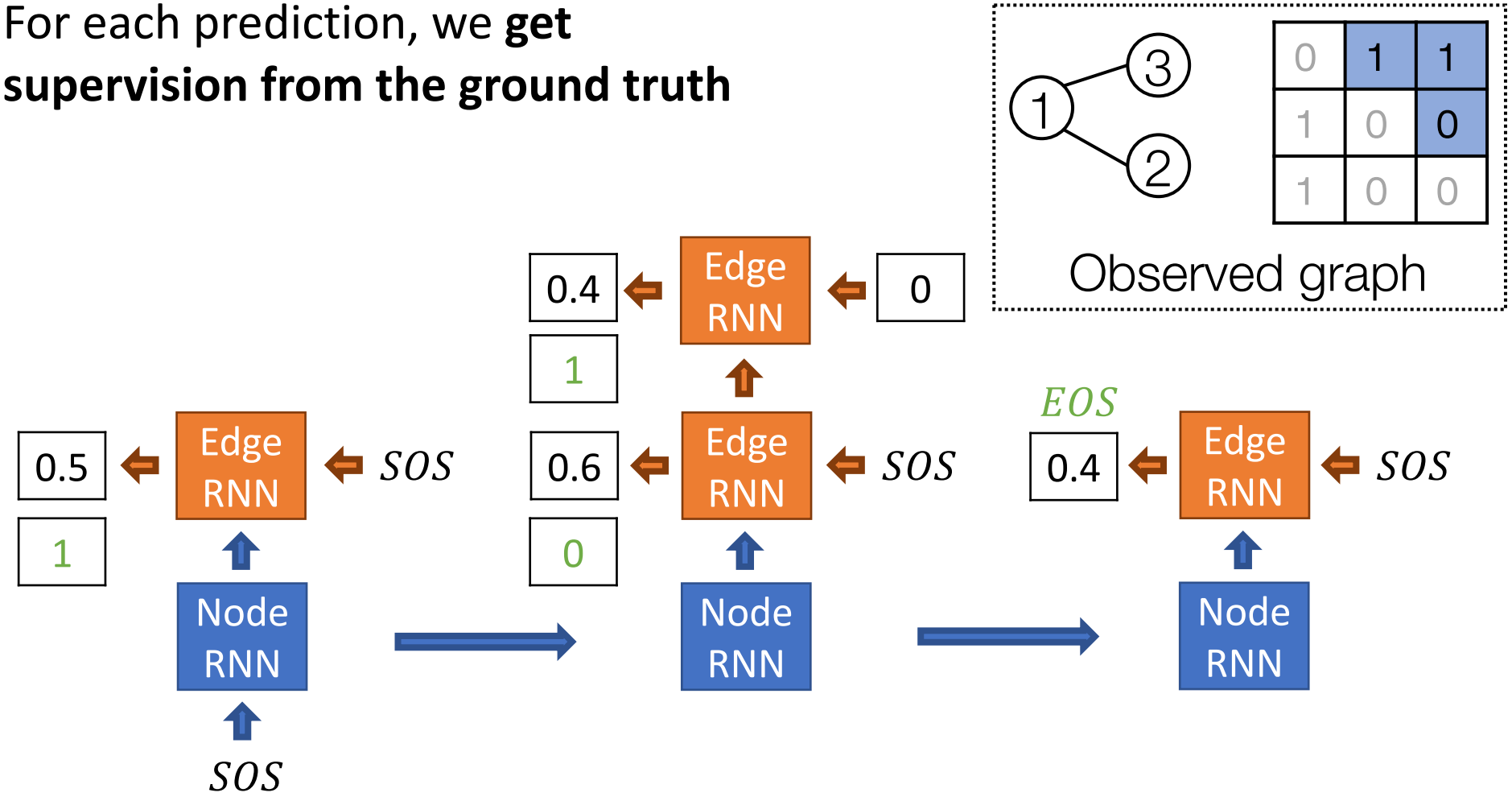
Stop generation since we know node 4 won't connect to any nodes





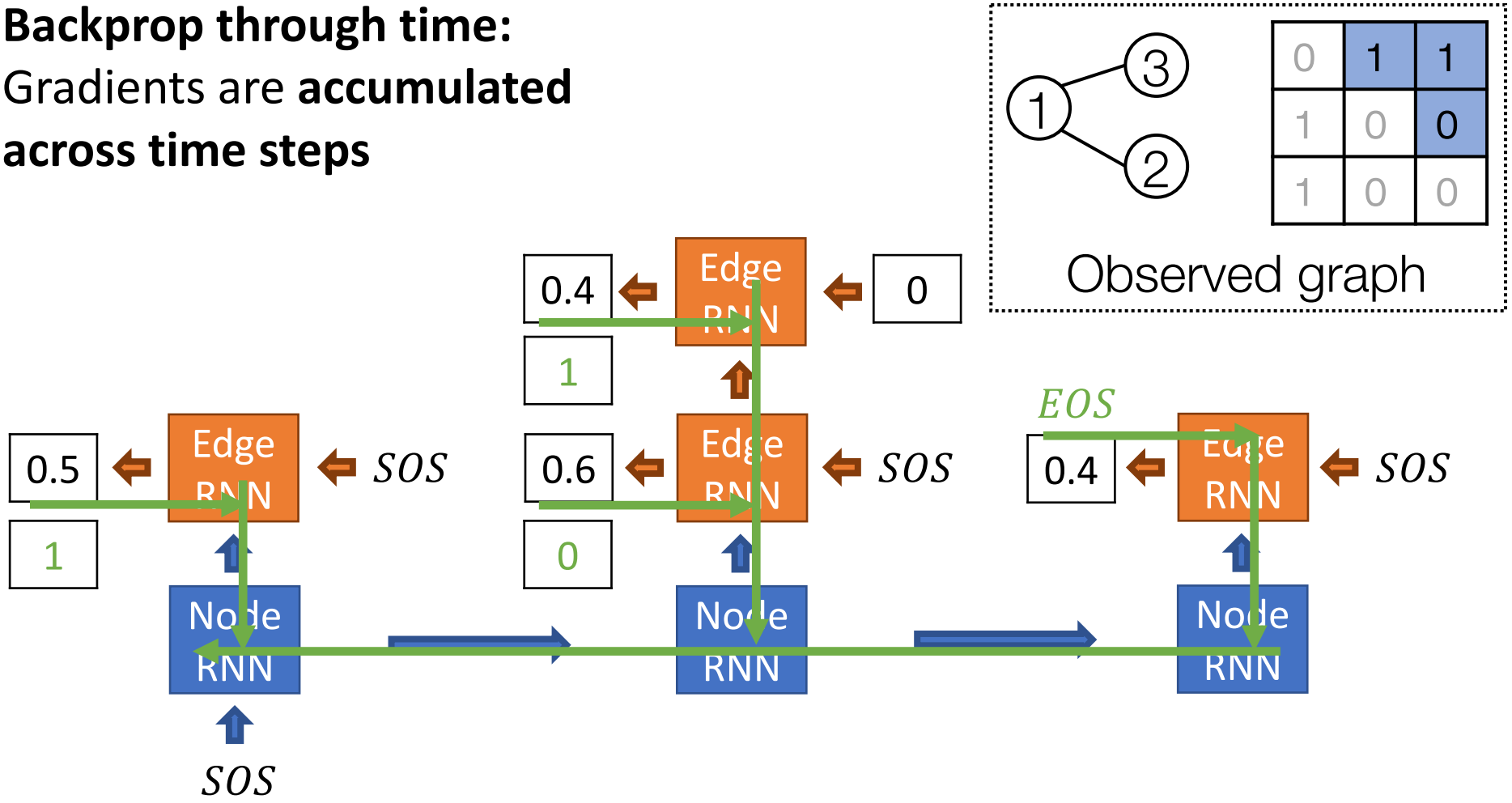
Put Things Together: Training

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



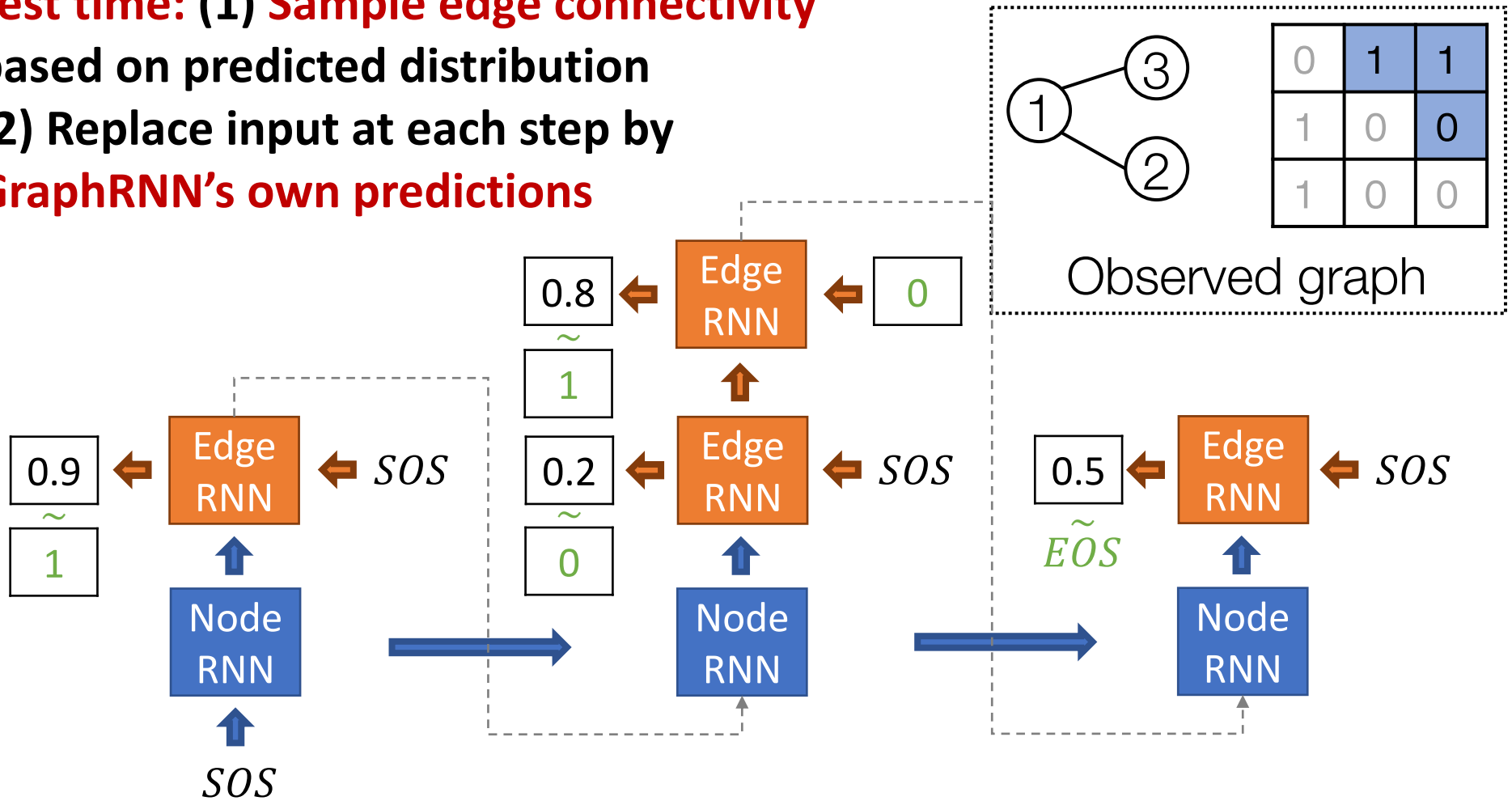
Put Things Together: Training

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

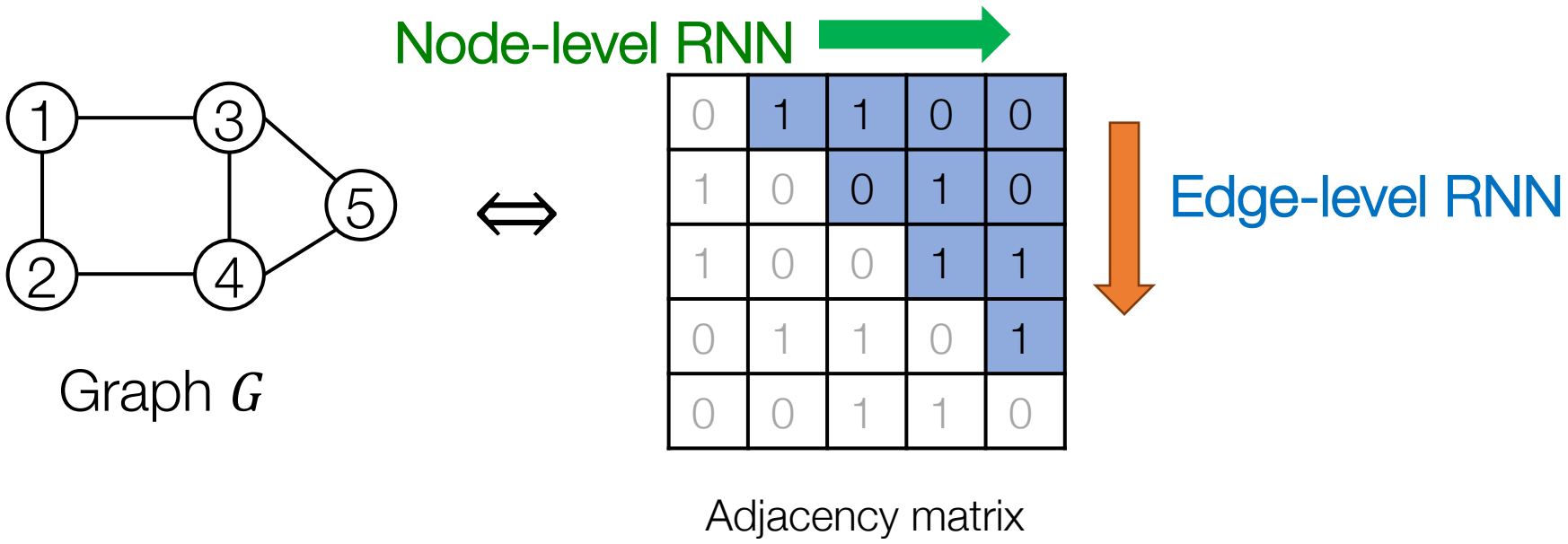


GraphRNN: Two levels of RNN

Quick Summary of GraphRNN:

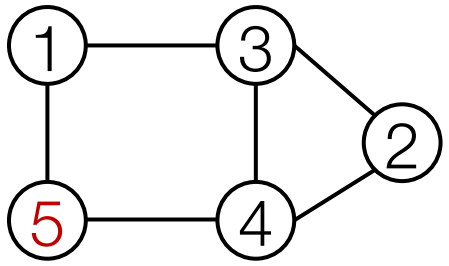
- Generate a graph by generating a **two-level sequence**
- Use **RNN** to generate the sequences

- **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



"Recipe" to generate the left graph:

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

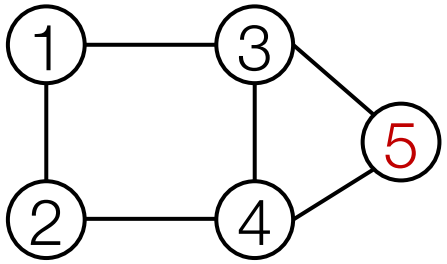
Random node ordering:

Node 5 may connect to any/all previous nodes

How do we limit this complexity?

Solution: Tractability via BFS

- **Breadth-First Search node ordering**



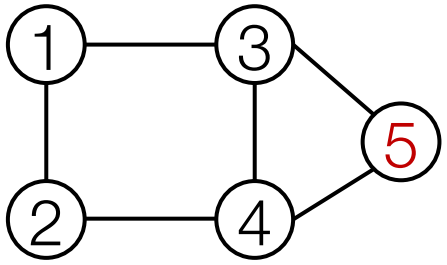
BFS ordering

"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

- **Breadth-First Search node ordering**



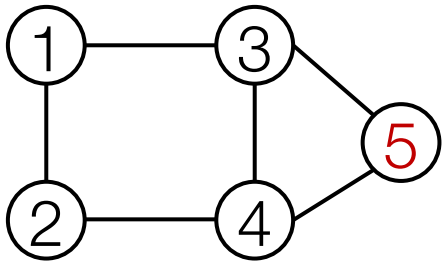
BFS ordering

"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

- **Breadth-First Search node ordering**



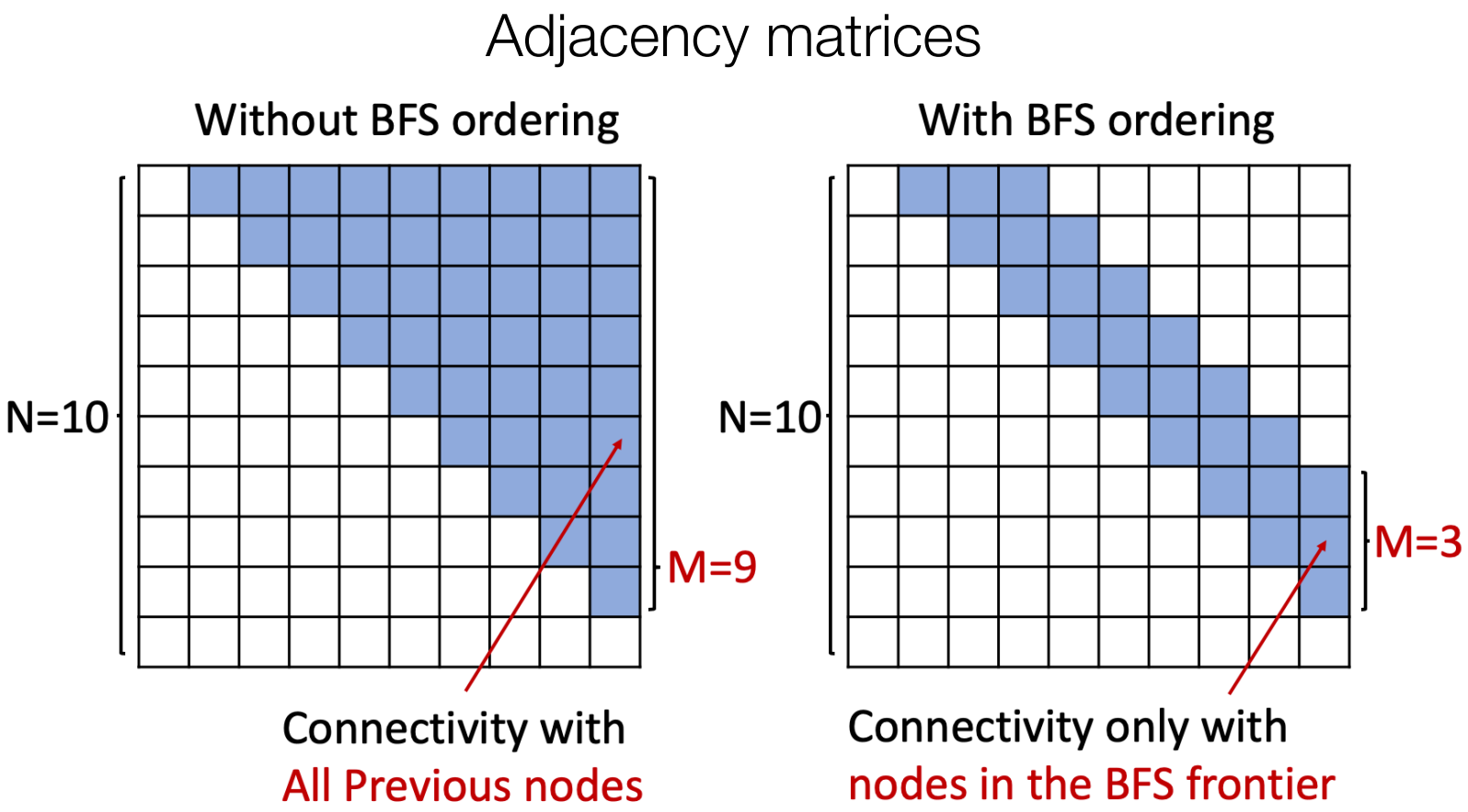
BFS ordering

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

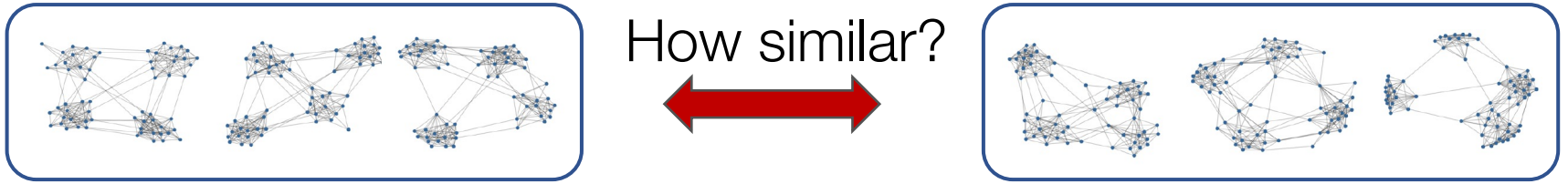
Solution: Tractability via BFS

- **BFS reduces the number of steps for edge generation**



Evaluating Generated Graphs

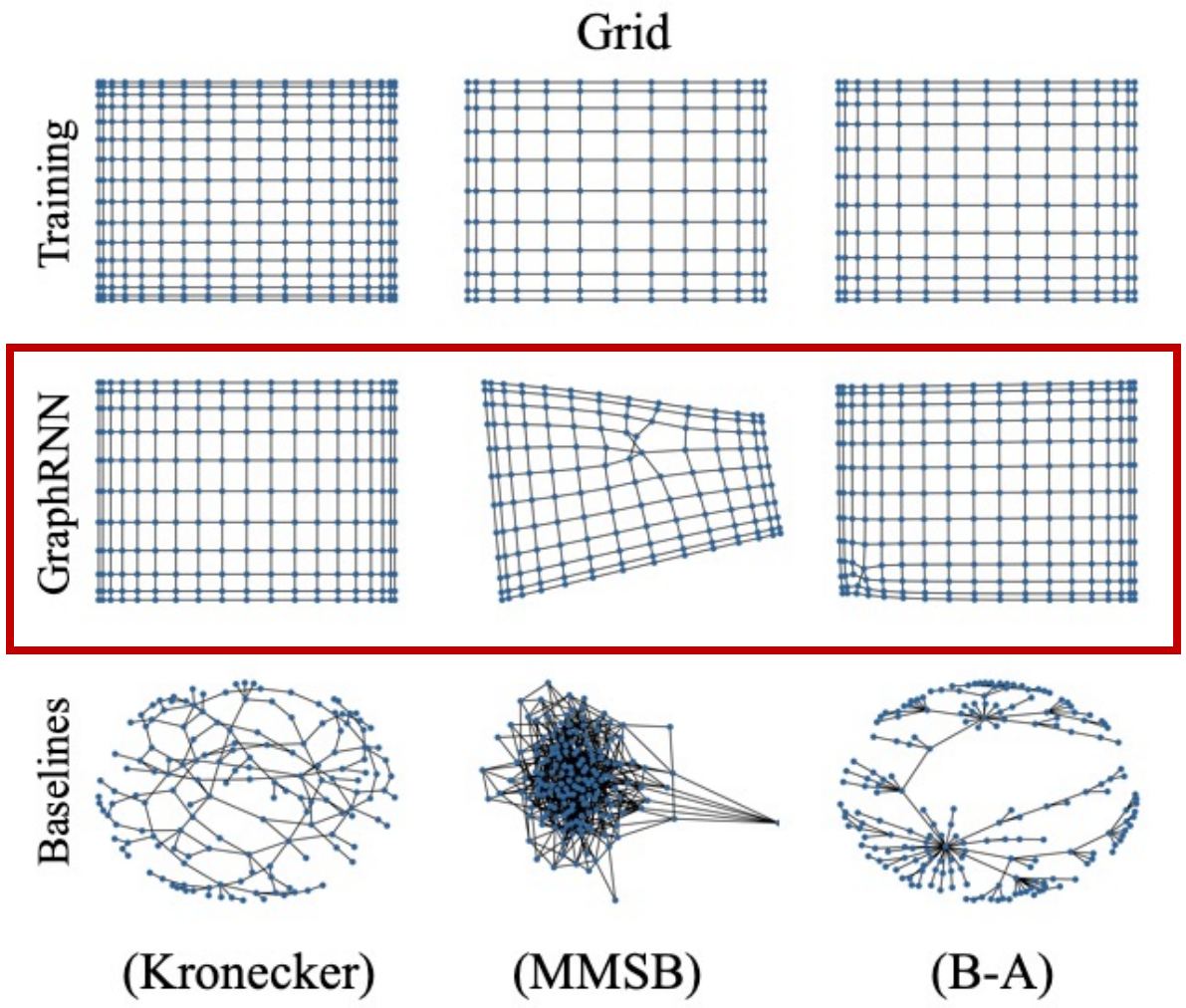
- **Task: Compare two sets of graphs**



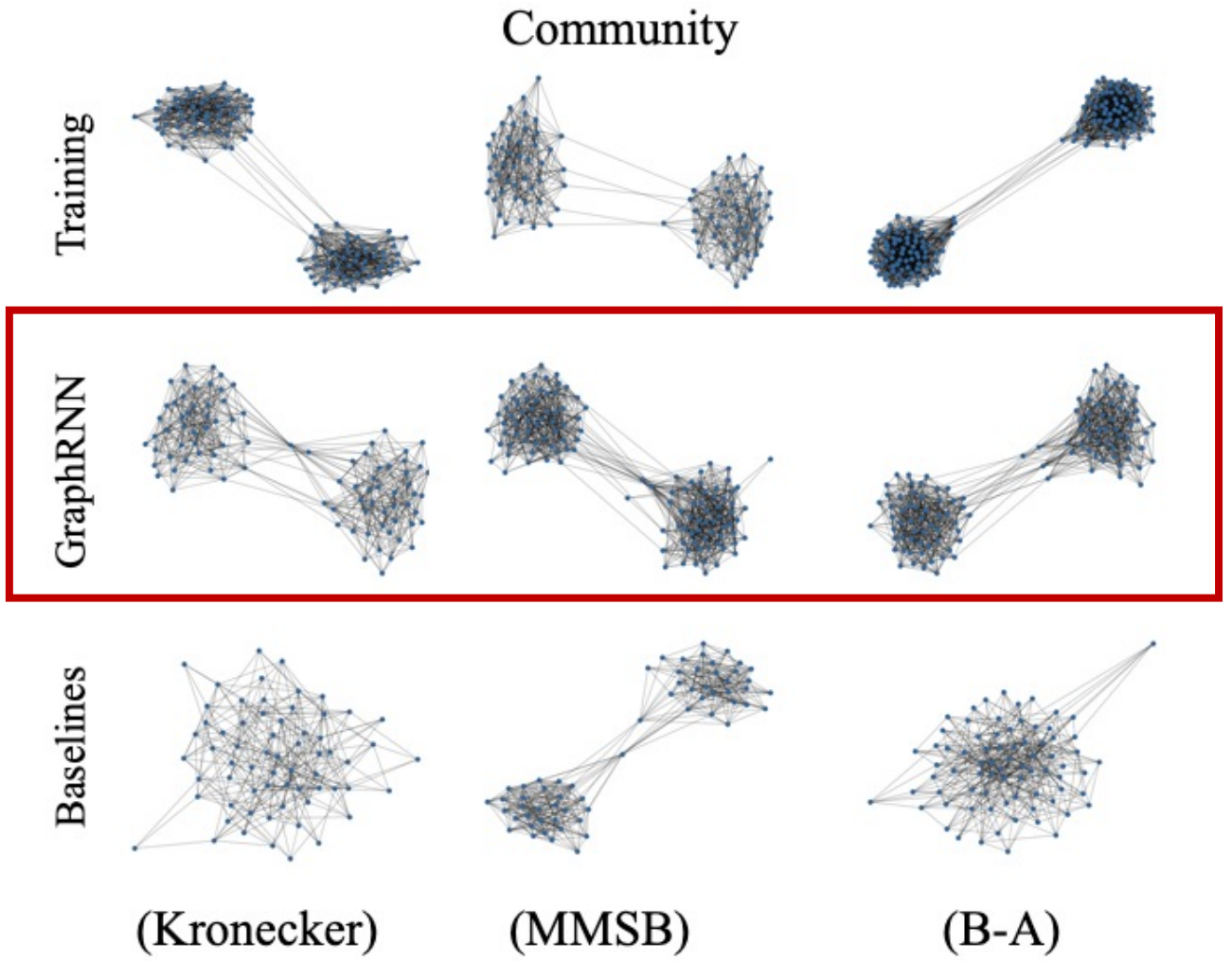
- **Goal:** Define similarity metrics for graphs

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

(1) Visual Similarity



(1) Visual Similarity

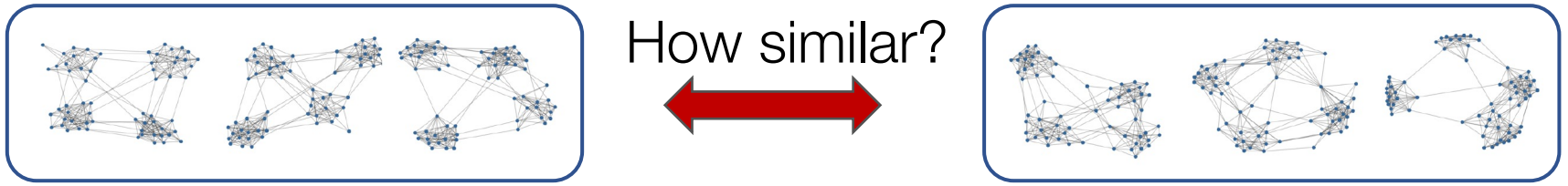


(2) Graph statistics similarity

- **Can we do more rigorous comparison?**
- **Issue:** Direct comparison between two graphs is hard (Graph Edit Distance is NP)!
- **Solution:** Compare **graph statistics**!
- Typical Graph Statistics:
  - Degree distribution (Deg.)
  - Clustering coefficient distribution (Clus.)
  - Orbit count statistics (Orbit)
- **Note:** Each statistic is a probability distribution

(2) Graph statistics similarity

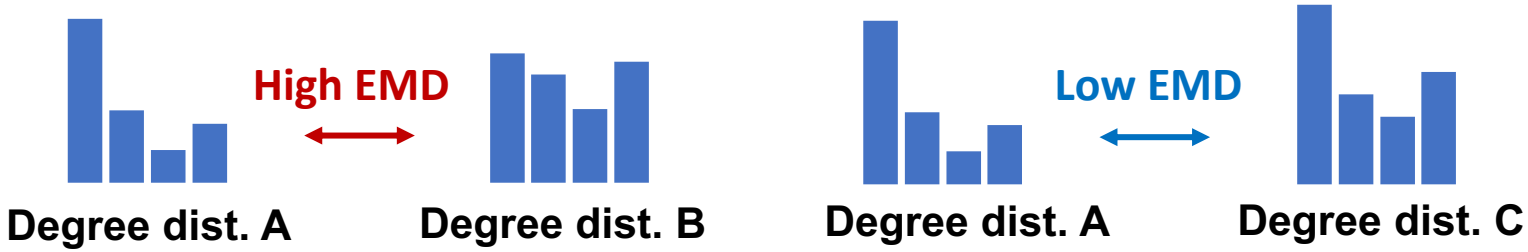
- **Issue:** want to compare sets of training graph statistics and generated graph statistics



- **Solution:**
- **Step 1:** How to compare two graph statistics
  - Earth Mover Distance (EMD)
- **Step 2:** How to compare sets of graph statistics
  - Maximum Mean Discrepancy (MMD) based on EMD

(2) Graph statistics similarity

- **Step 1: Earth Mover Distance (EMD)**
  - Compare **similarity between 2 distributions**
  - **Intuition:** Measure the minimum effort that **move earth from one pile to the other**



The EMD can be solved as the optimal flow and is found by solving this linear optimization problem.

$$\mathrm{WORK}(F, \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{m} \sum_{j=1}^{n} f_{ij} d_{ij}$$

We want to find a flow F, with f_{ij} the flow between p_i and q_j, that minimizes the overall cost. d_{ij} is the ground distance between p_i and q_j.

(2) Graph statistics similarity

- **Step 2: Maximum Mean Discrepancy (MMD)**
  - Compare **similarity between 2 sets**, based on the similarity between set elements

$\{2,3,5\}$ **High MMD** $\{6,1,2\}$ $\{2,3,5\}$ **Low MMD** $\{5,4,2\}$

Similarity between set elements: L2 distance

(Each element is a scalar)

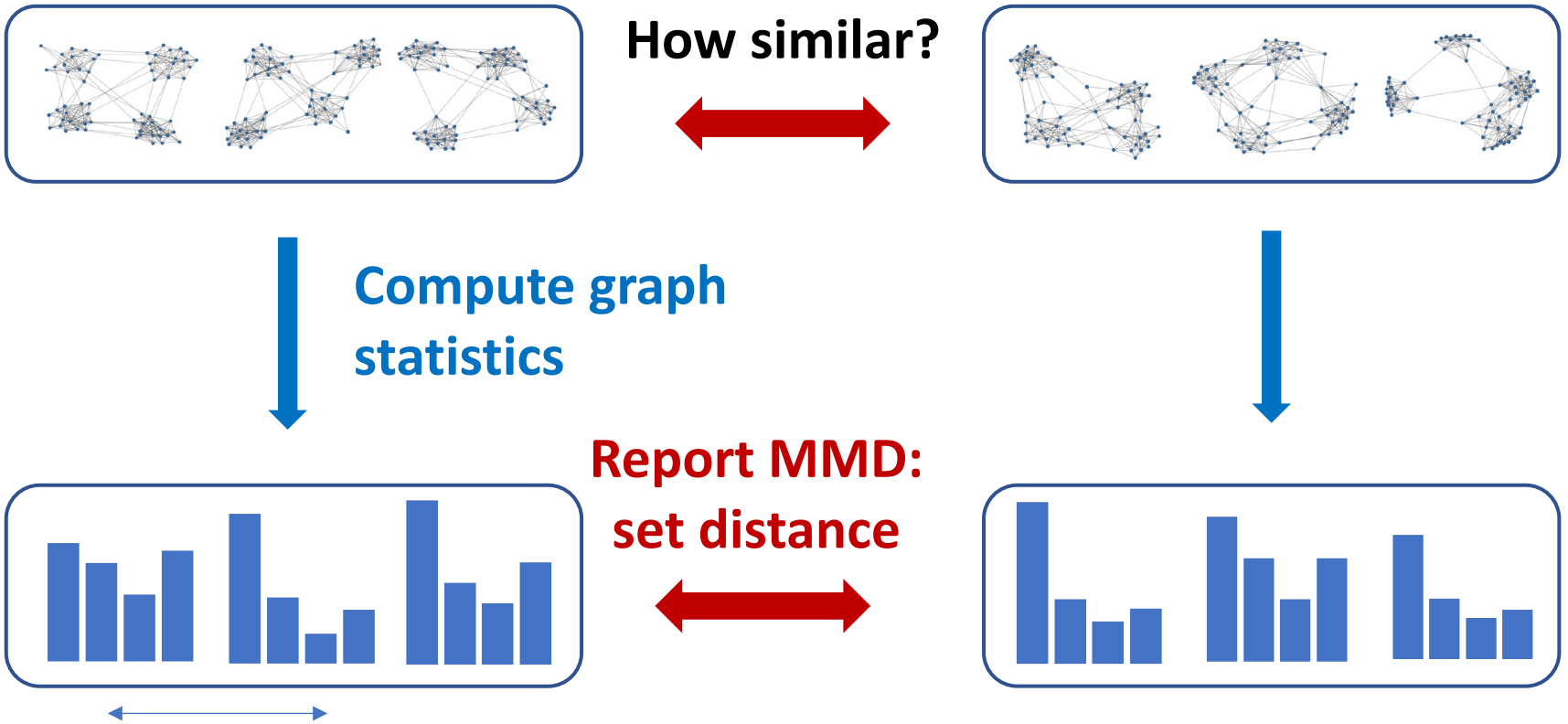
  - **Recall: We compare 2 sets of graph statistics (distributions)**

Similarity between set elements: EMD

(Each element is a distribution)

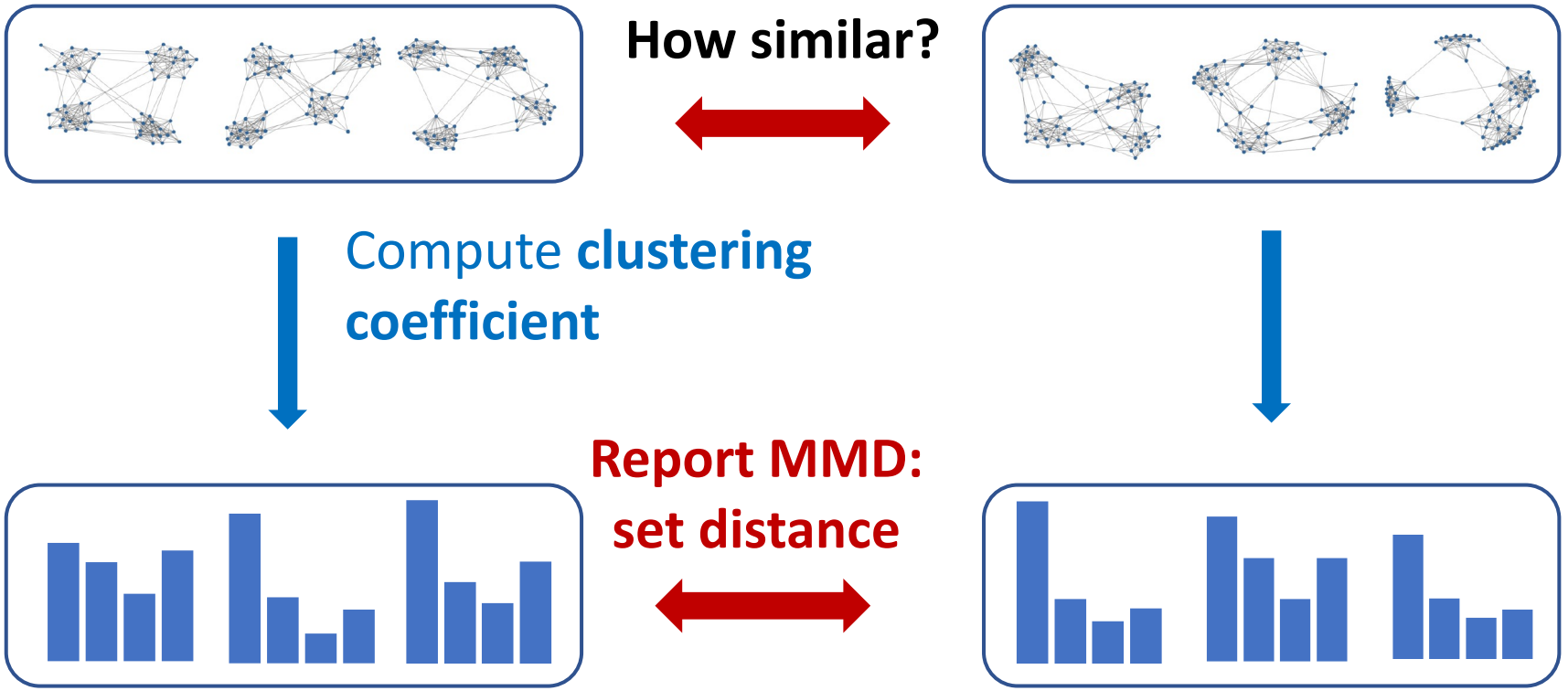
(2) Graph statistics similarity

- Putting things together



(2) Graph statistics similarity

- **Example**



	Erdos-Renyi	Kronecker	GraphRNN
MMD score	1.24	1.67	**0.002**

(2) Graph statistics similarity

- Compared to traditional graph generative models, GraphRNN can generated graphs with statistics **very close to ground-truth graphs**

Table 1. Comparison of GraphRNN to traditional graph generative models using MMD. $(\max(|V|), \max(|E|))$ of each dataset is shown.

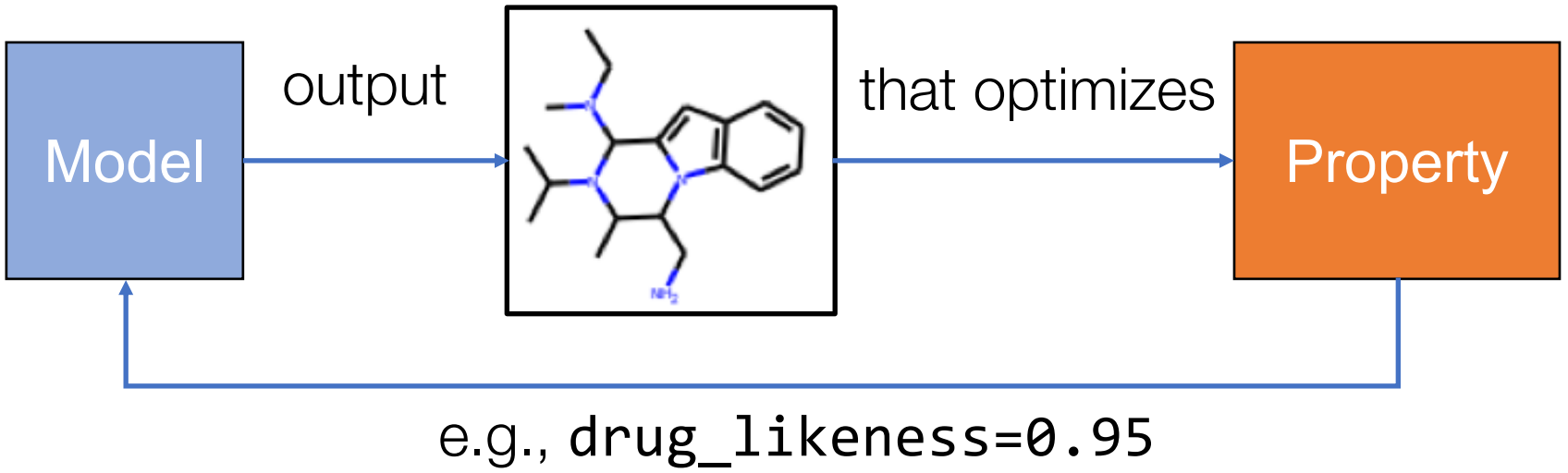
	Community (160,1945)			Ego (399,1071)			Grid (361,684)			Protein (500,1575)		
	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit	Deg.	Clus.	Orbit
E-R	0.021	1.243	0.049	0.508	1.288	0.232	1.011	0.018	0.900	0.145	1.779	1.135
B-A	0.268	0.322	0.047	0.275	0.973	0.095	1.860	0	0.720	1.401	1.706	0.920
Kronecker	0.259	1.685	0.069	0.108	0.975	0.052	1.074	0.008	0.080	0.084	0.441	0.288
MMSB	0.166	1.59	0.054	0.304	0.245	0.048	1.881	0.131	1.239	0.236	0.495	0.775
GraphRNN-S	0.055	0.016	0.041	0.090	**0.006**	0.043	0.029	10^{-5}	0.011	0.057	**0.102**	**0.037**
GraphRNN	**0.014**	**0.002**	**0.039**	**0.077**	0.316	**0.030**	$\mathbf{10^{-5}}$	**0**	$\mathbf{10^{-4}}$	**0.034**	0.935	0.217

Deep Generative Models for Graphs

- Machine Learning for Graph Generation
- GraphRNN: Generating Realistic Graphs
- **Application of Deep Graph Generative Models**

Application: Drug Discovery [You et al., NeurIPS 2018]

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



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:

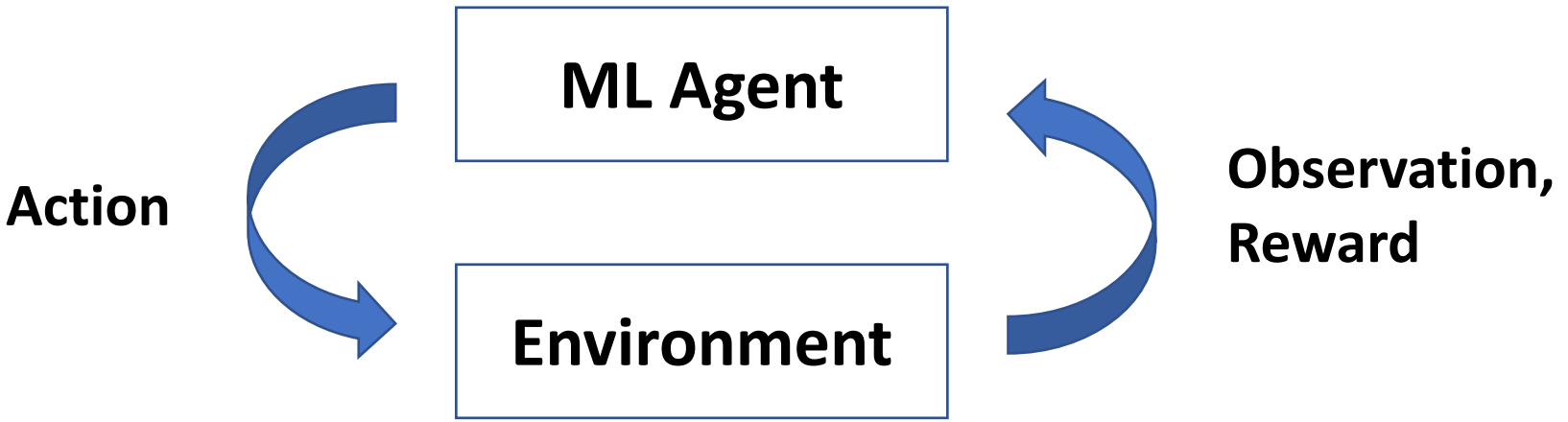
- **Optimize a given objective** (High scores)
  - e.g., drug-likeness
- **Obey underlying rules** (Valid)
  - e.g., chemical validity rules

Including "Black-box" in ML:

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

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

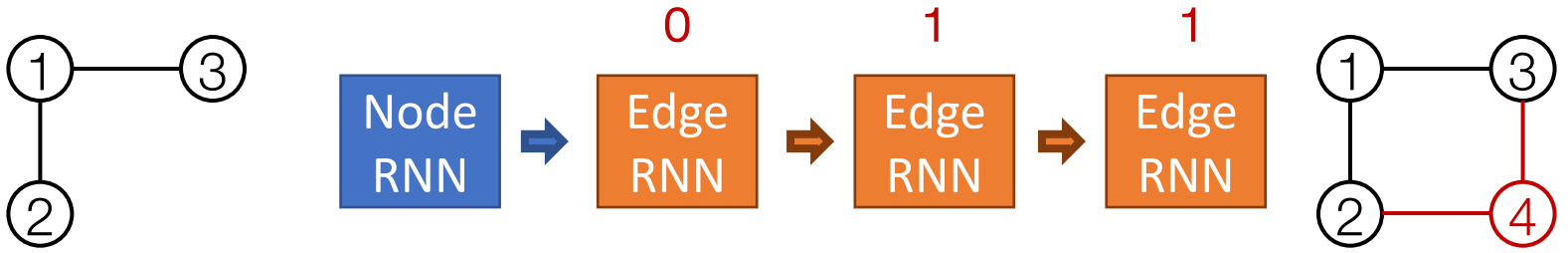
- **Commonality of GCPN & GraphRNN:**
  - Generate graphs sequentially
  - Imitate a given graph dataset

- **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**

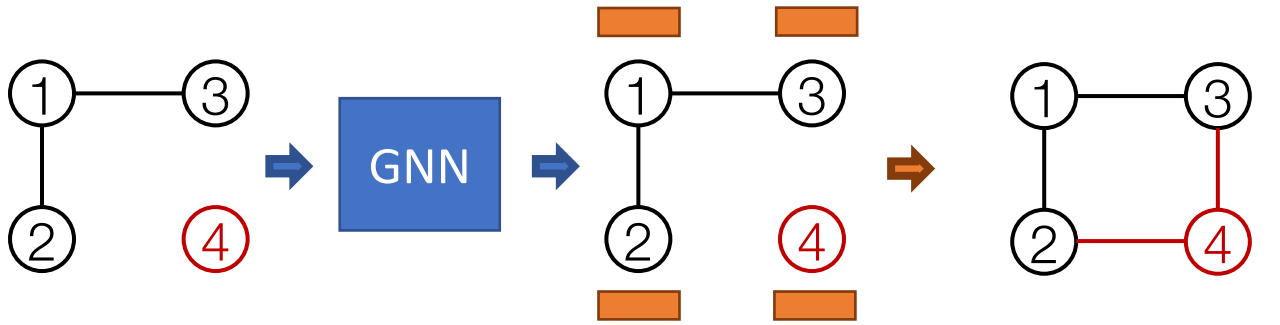


RNN hidden state captures the generated graph so far

GCPN vs GraphRNN

- **Sequential graph generation**

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



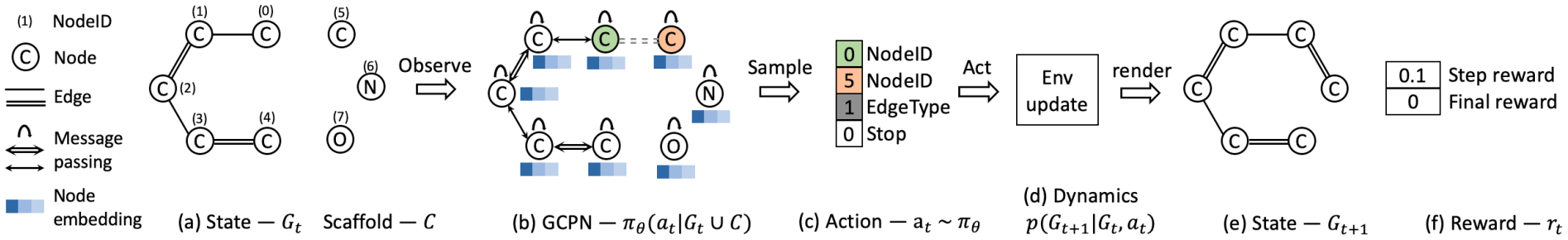
Node embeddings

Predict potential links using node embeddings

Recall the link prediction head:

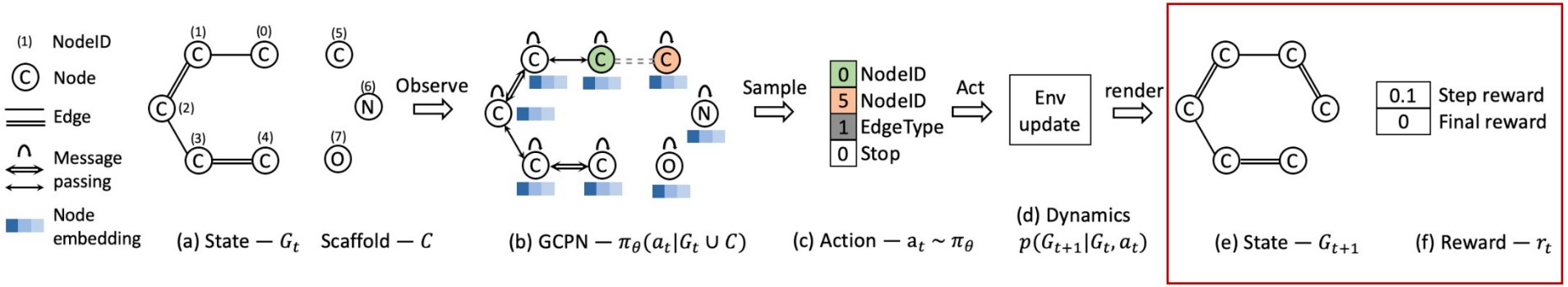
$\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}) :=$
$\text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$

Overview of GCPN



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

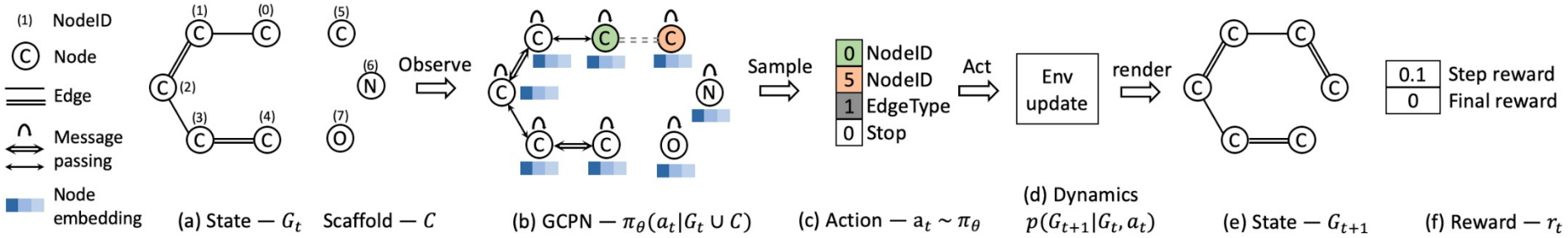
How Do We Set the Reward?



- **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

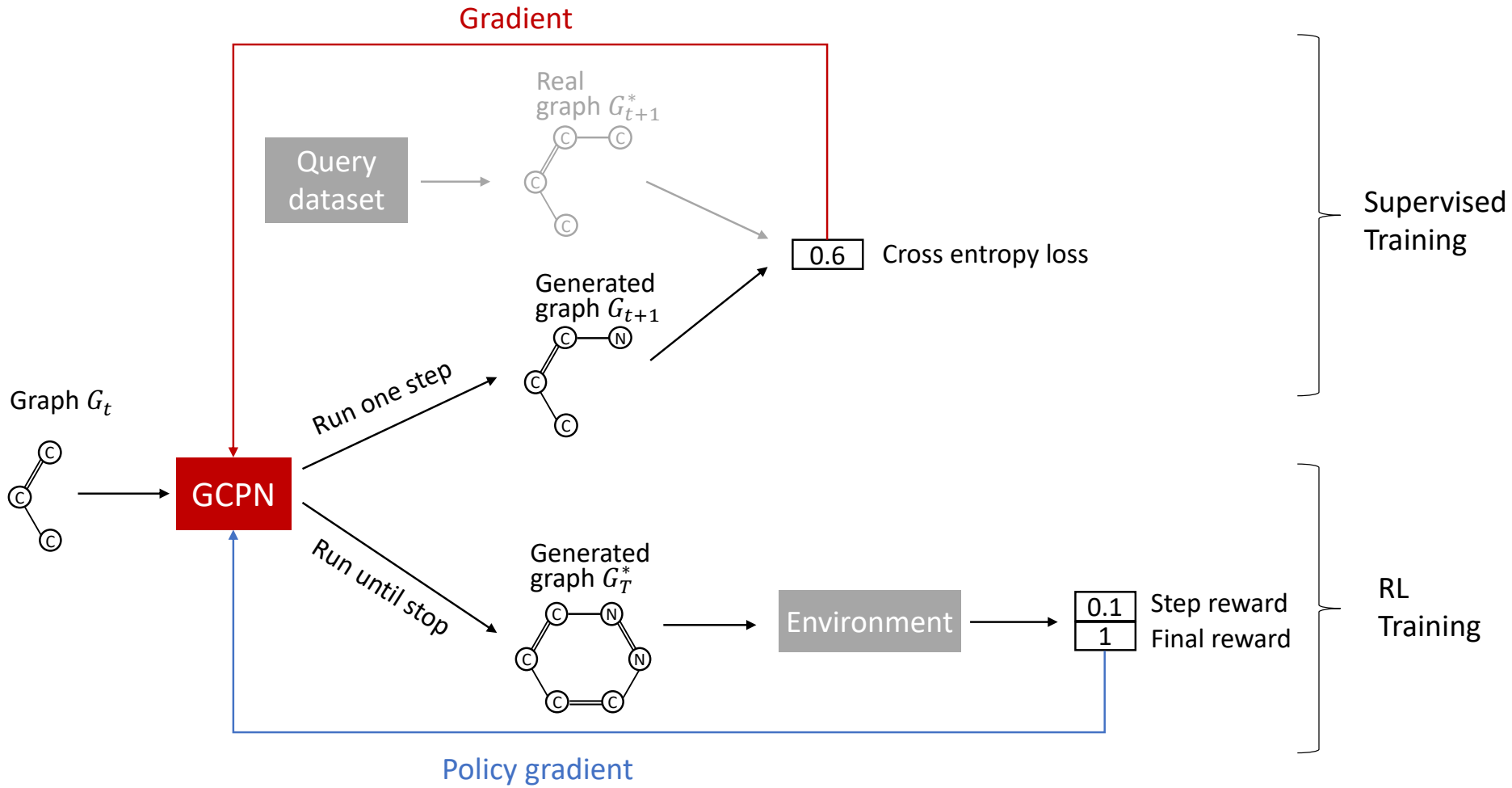
Reward = Final reward + Step reward

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

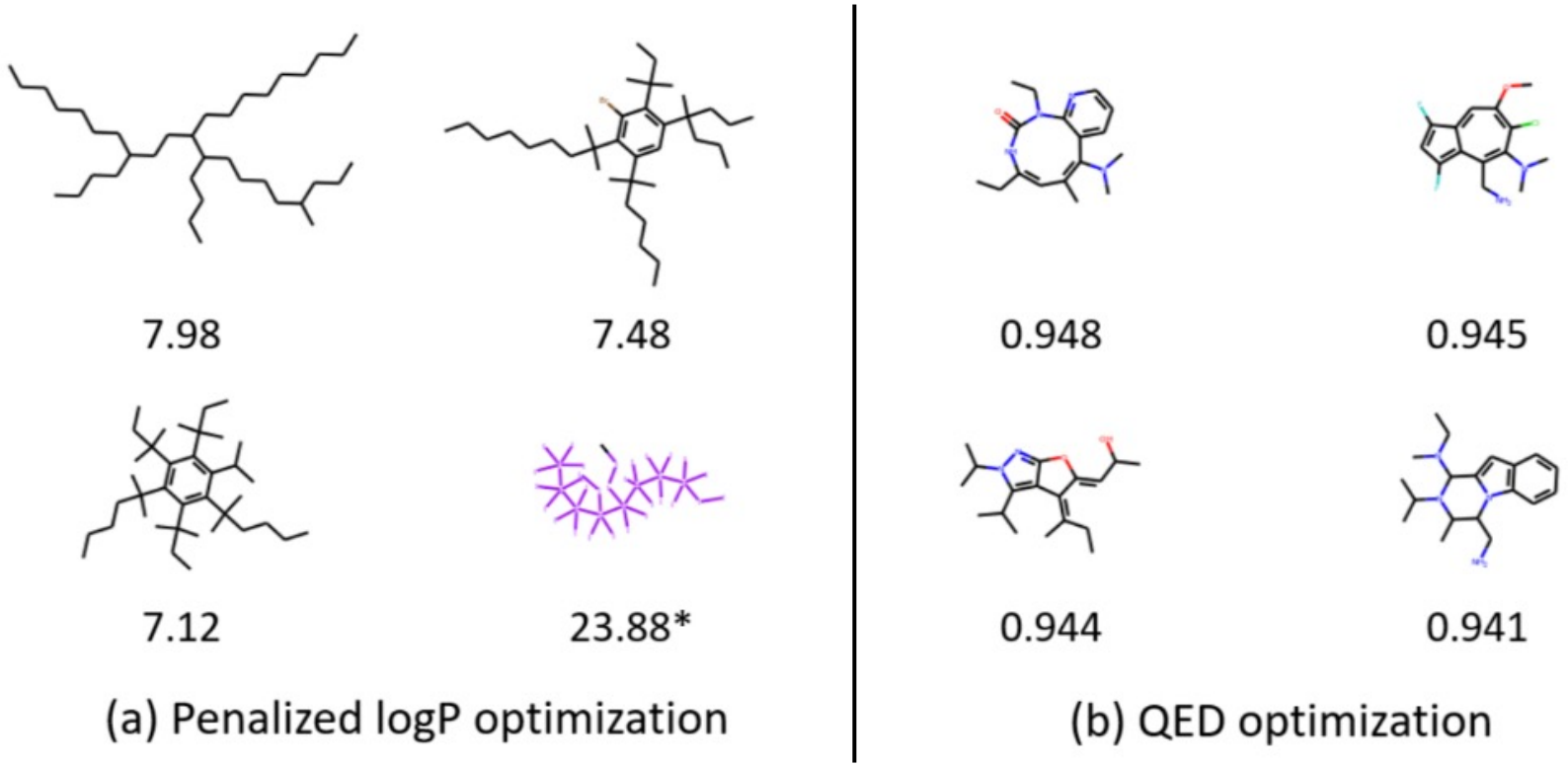
Training GCPN



Qualitative Results

Visualization of GCPN graphs:

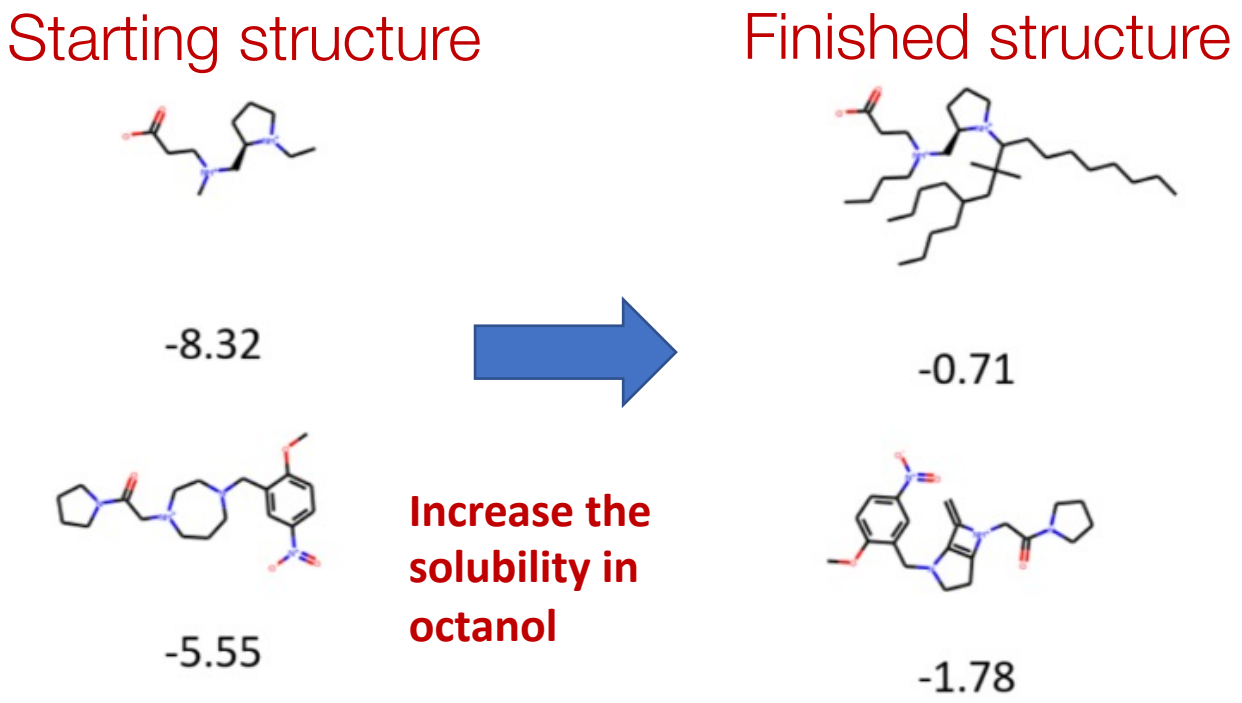
- Property optimization Generate molecules with high specified property score



Qualitative Results

Visualization of GCPN graphs:

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



(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