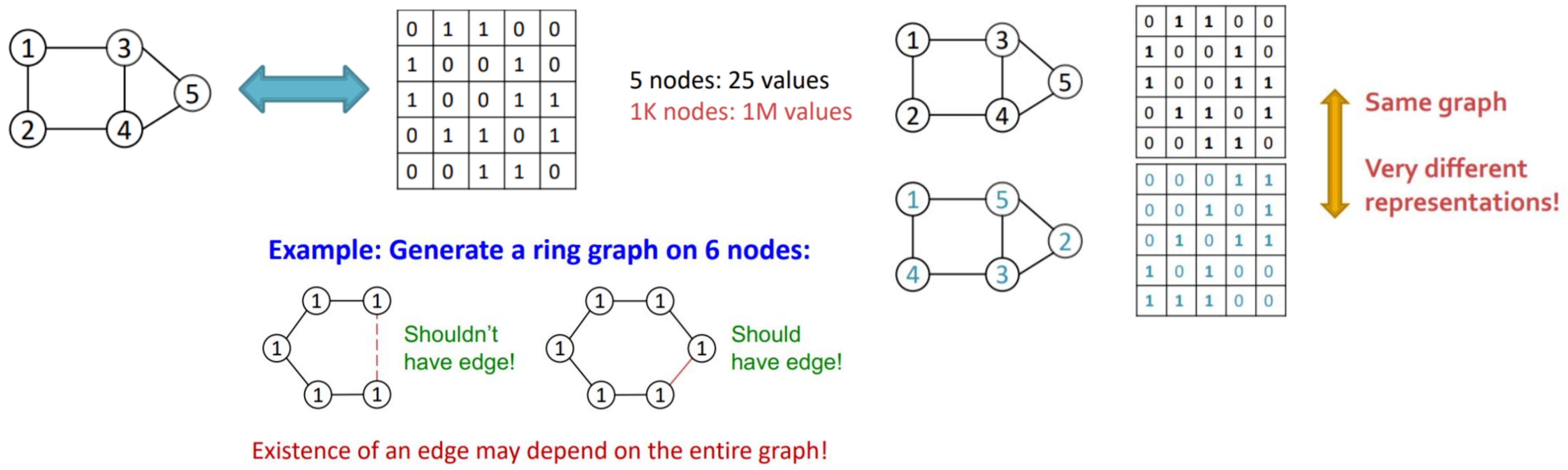
## Summary Questions of the lecture

- Why is the Graph Generation Tasks hard?
- GGT is difficult because 1) the output graph space is large and variable, 2) a single graph has multiple representations, and 3) determining edge connections depend on long-range and extensive information.



5 nodes: 25 values
1K nodes: 1M values
Same graph
Very different representations!
Example: Generate a ring graph on 6 nodes:
Shouldn't have edge!
Should have edge!
Existence of an edge may depend on the entire graph!

| | | | | |
|---|---|---|---|---|
| 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 |

| | | | | |
|---|---|---|---|---|
| 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 |

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |

## Summary Questions of the lecture

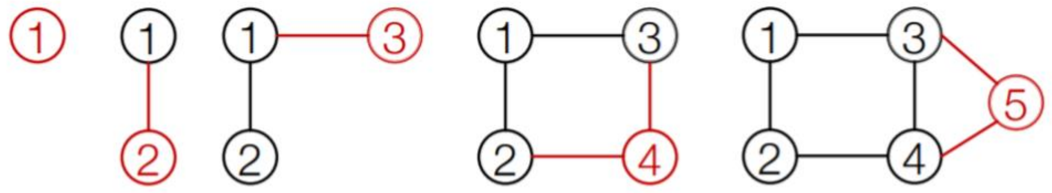
- Discuss the machine learning background of auto-regressive model to obtain the graph generative models.
- Auto-regressive models sequentially predict graph-generation actions (adding a node and one or more edges) on the current time step based on all past actions. These actions are represented by random variables, whose distribution is estimated from the given data distribution with a parametric distribution model in a maximum likelihood manner, and then actions are sampled from the distribution by the estimated model.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathbb{E}_{\boldsymbol{x}\sim P_{data}(\boldsymbol{x})} \log P_{model}(\boldsymbol{x}|\boldsymbol{\theta})$$

$$P_{model}(\boldsymbol{x}|\boldsymbol{\theta}) = \prod_{t=1}^{n} P_{model}(x_t|x_1, \dots, x_{t-1}, \boldsymbol{\theta}) \qquad x_t \sim P_{data}(\boldsymbol{x})$$

## Summary Questions of the lecture

- Explain GraphRNN which is a sequence process to generate a graph with a node ordering.
- GraphRNN generates a graph through a sequenc of sequences. That is, on time-step $t$, node $t$ is first added to the graph, which forms node level sequence. Then, at the node level action $t$, edge level sequence is conducted by determining whether the node is connected to each of the existing nodes $1, 2, \cdots, t-1$.



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

“Add node 1” ... “Add node 5”

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

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

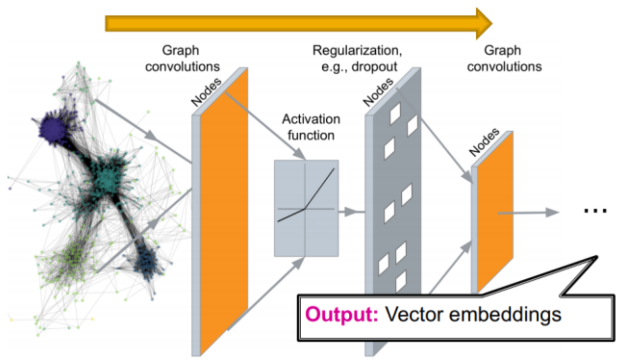
0 1 1

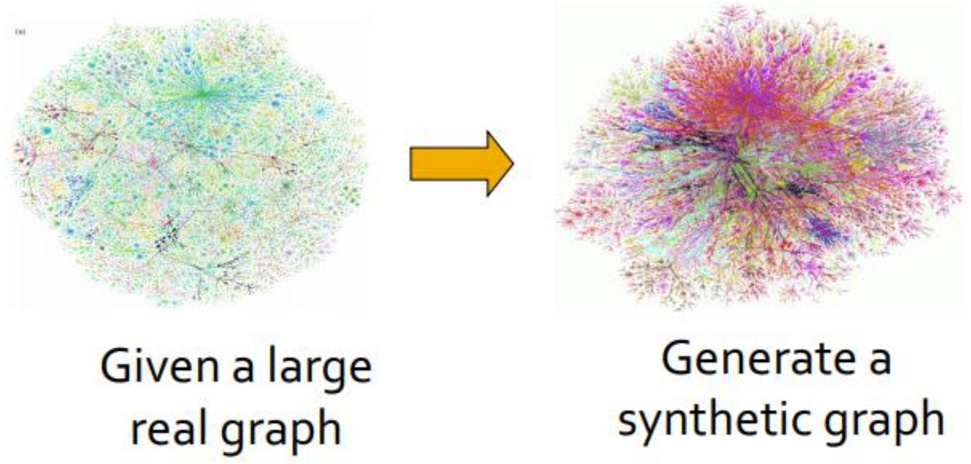
## Outline of Lecture (5)

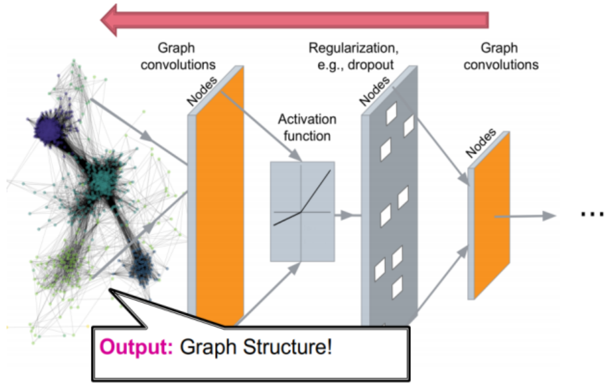
- Random Walks and Diffusion
- Diffusion in GCN
  - Propagation using graph diffusion
    - APPNP: Predict Then Propagate [IC LR'19]
    - Graph Diffusion-Embedding Network s [CVPR'19]
  - Making a new graph
    - Diffusion Improves Graph Learning [NIPS'19]
    - SSL with Graph Learning-Convolutio nal Networks [CVPR'19]
- Deep Generative Models For Graph
  - Problem of Graph Generation
  - ML Basics for Graph Generation
  - GraphRNN : Generating Realistic Graphs
  - Applications and Open Questions

# DGMG: (Continue) Deep Generative Models For Graph

## *Deep Graph Encoder, Graph Generation*







## Deep Graph Decoders

### Outline of Contents

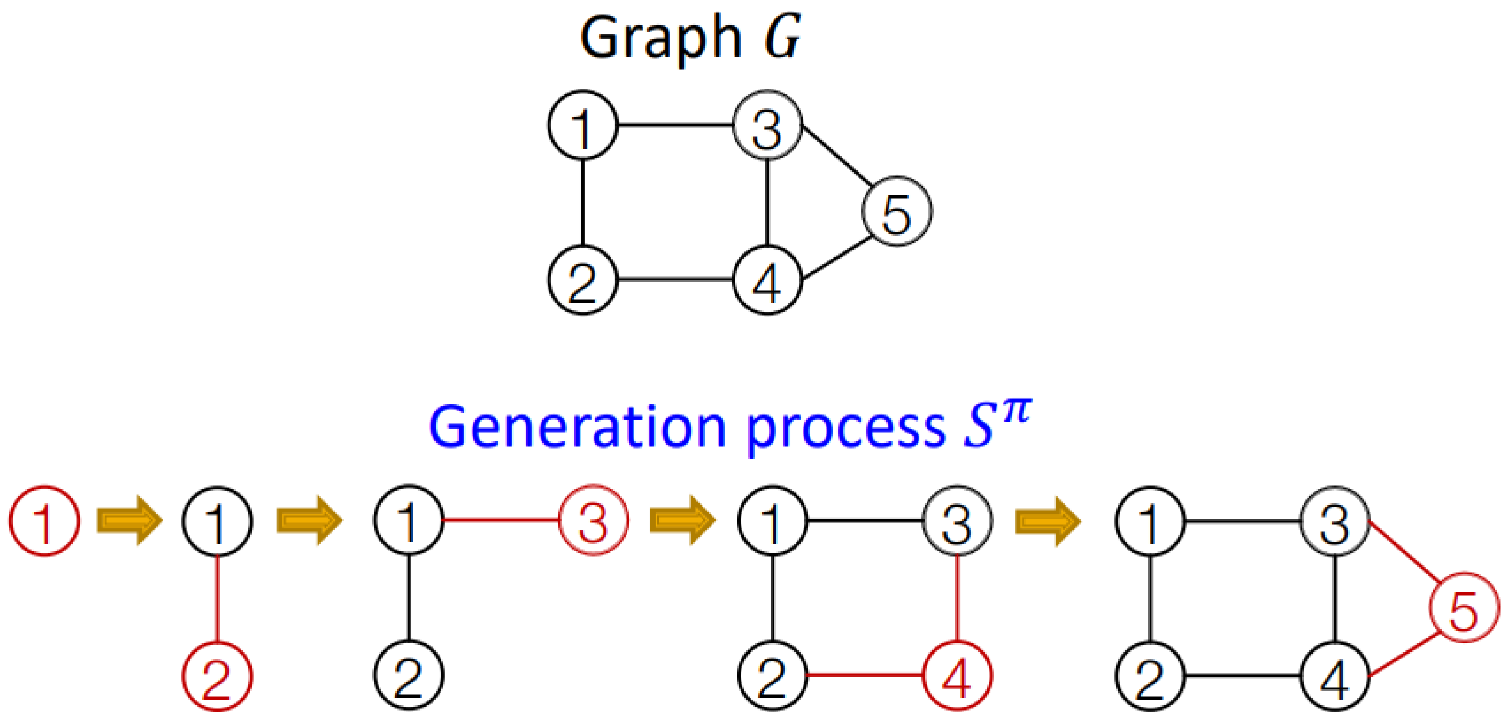
GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models

(J. You et al., ICML 2018)

## GraphRNN

### IDEA

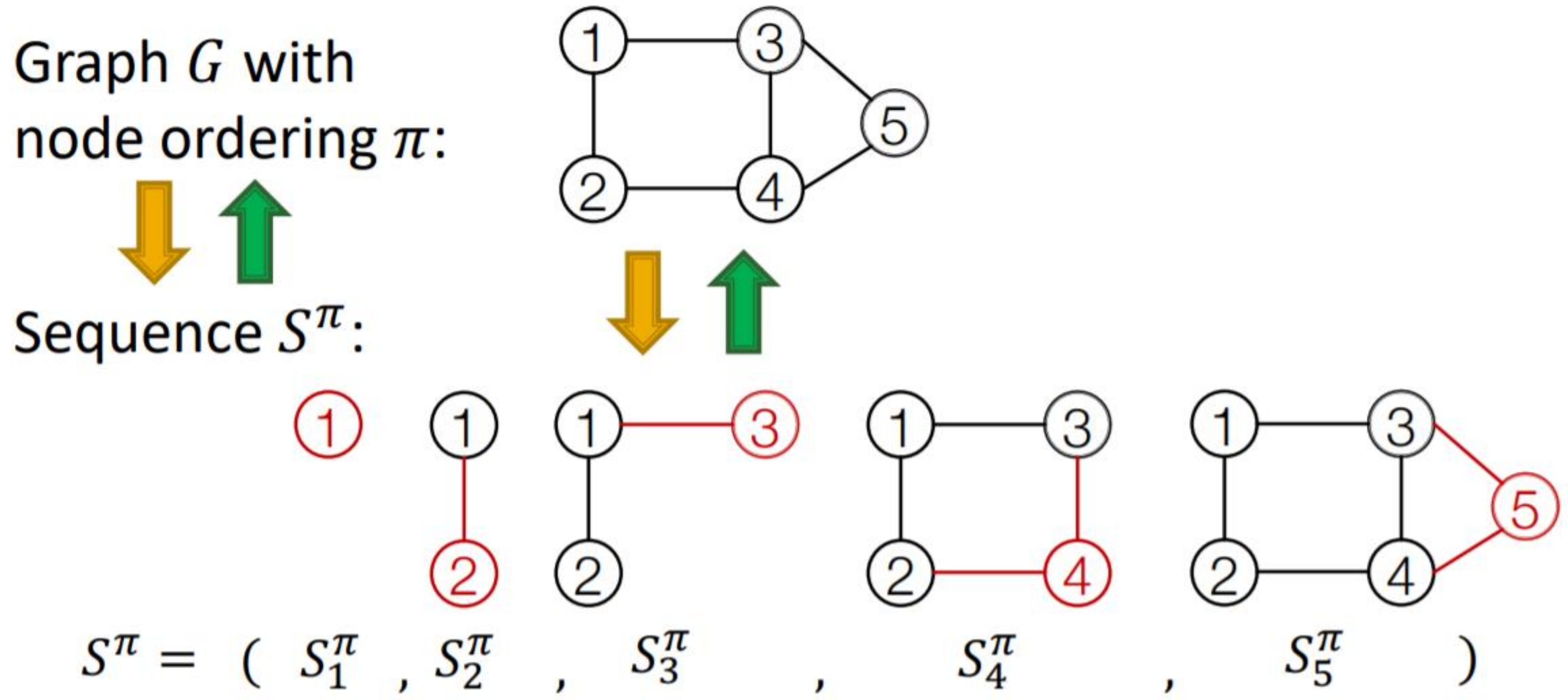
- Generating graphs via sequentially adding nodes and edges



## GraphRNN

**Model** Graphs as Sequences

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

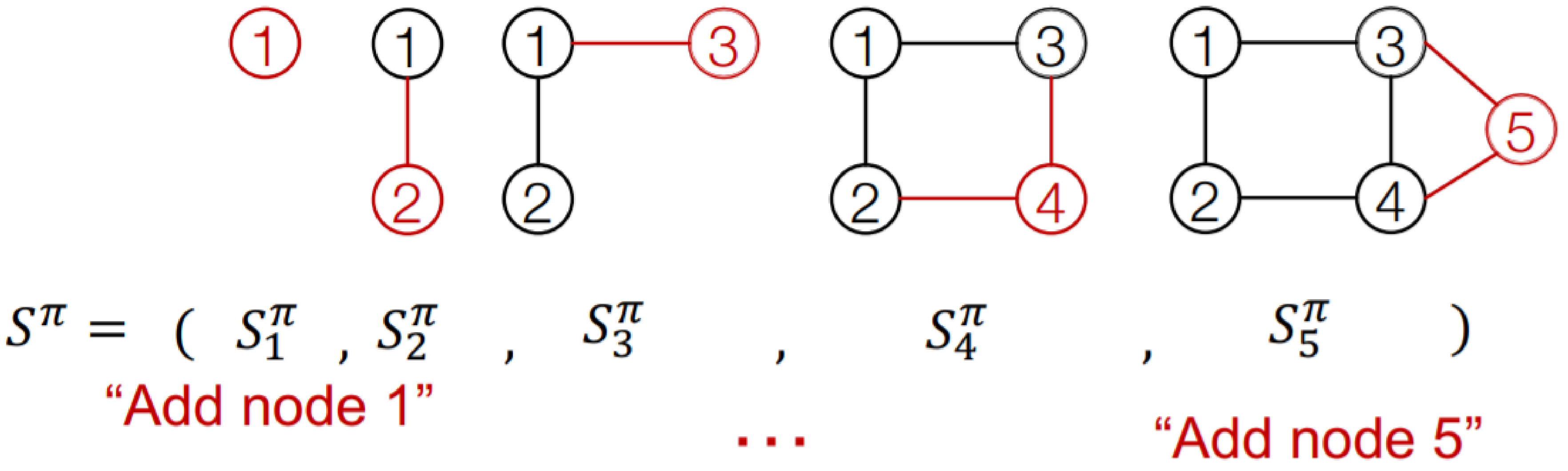

Graph $G$ with
node ordering $\pi$:
Sequence $S^{\pi}$:
1
2
3
4
5
$S^{\pi} = ( S^{\pi}_1 , S^{\pi}_2 , S^{\pi}_3 , S^{\pi}_4 , S^{\pi}_5 )$

## GraphRNN

**The sequence** $S^{\pi}$ 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

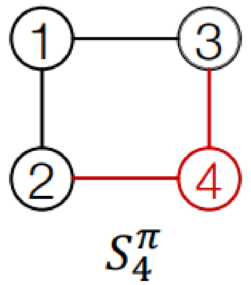


$$S^{\pi} = ( S_1^{\pi}, S_2^{\pi}, S_3^{\pi}, S_4^{\pi}, S_5^{\pi} )$$

## GraphRNN

**The sequence** $S^{\pi}$ has two levels: ($S$ is a sequence of sequences)

- Each Node-level step is an edge-level sequence (multiple edges per each node)
- Edge-level: At each edge-level step, add a new edge

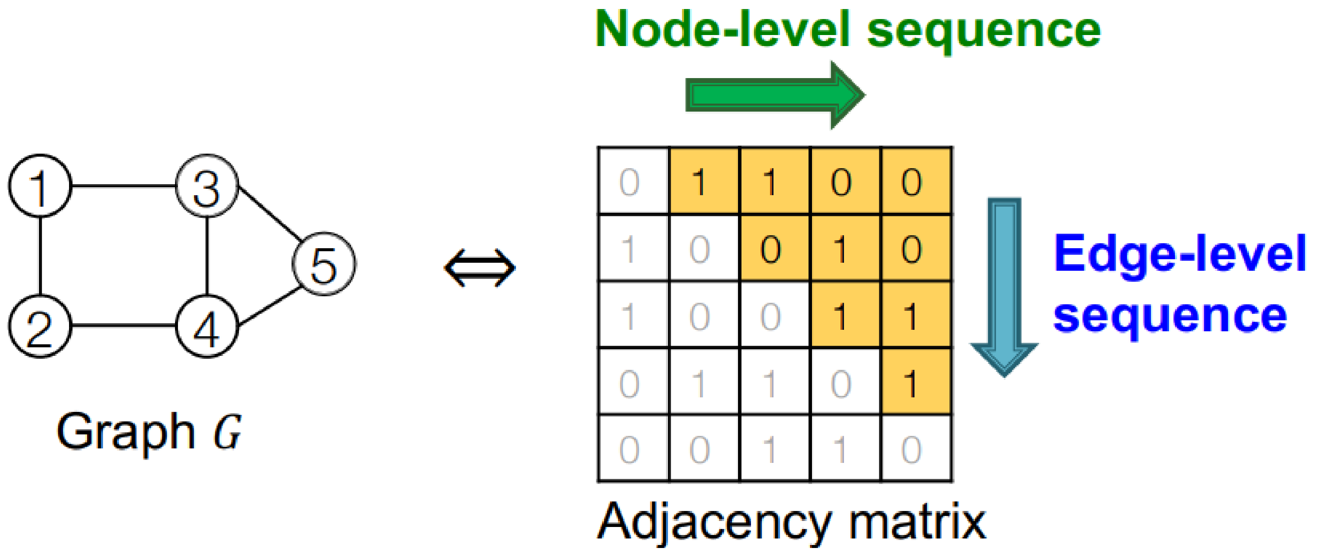


$$S_4^{\pi} = \left( \underset{\substack{\text{“Not connect 4, 1”} \\ 0}}{S_{4,1}^{\pi}}, \quad \underset{\substack{\text{“Connect 4, 2”} \\ 1}}{S_{4,2}^{\pi}}, \quad \underset{\substack{\text{“Connect 4, 3”} \\ 1}}{S_{4,3}^{\pi}} \right)$$

## GraphRNN

**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)



## GraphRNN

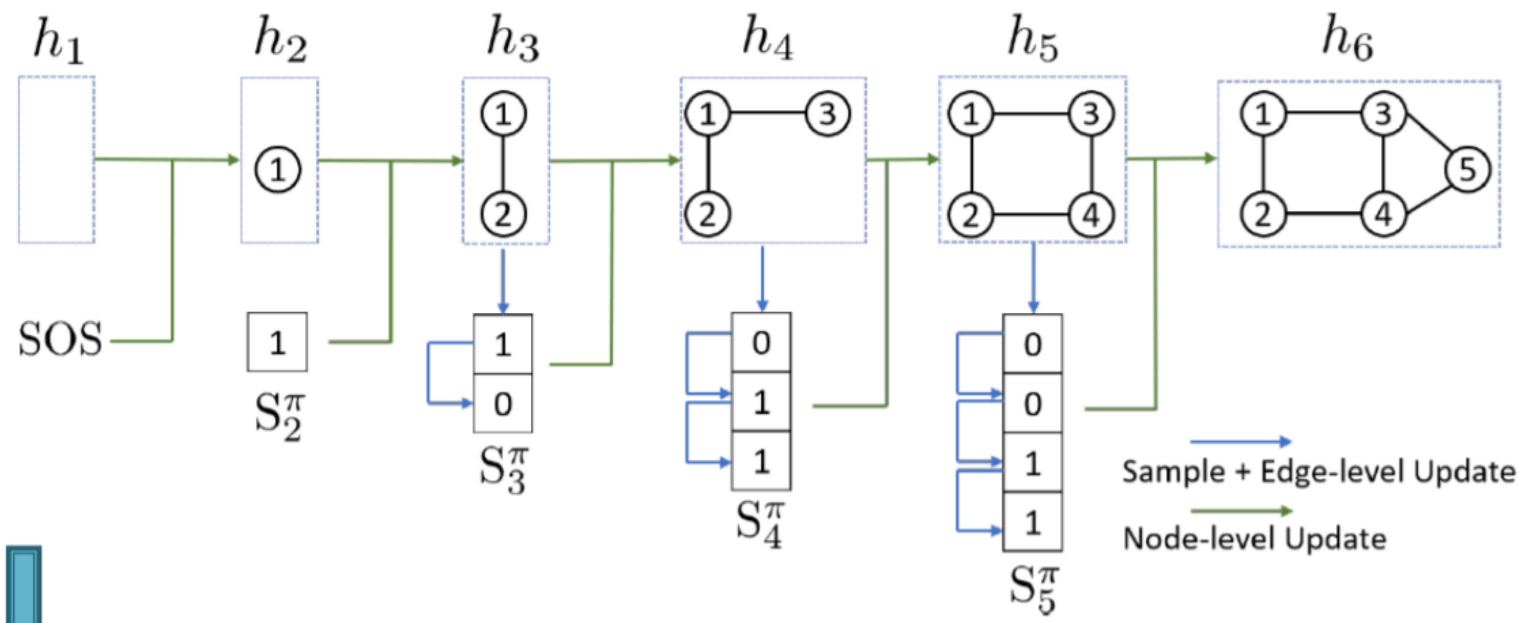
### Two levels of RNN

- 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 generates edges for the new node, then updates node-level RNN state using generated results

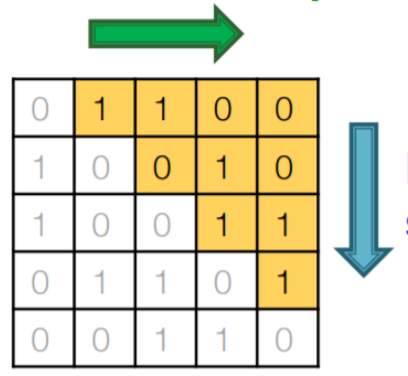
# GraphRNN

## Two levels of RNN

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



Node-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 |

Edge-level sequence

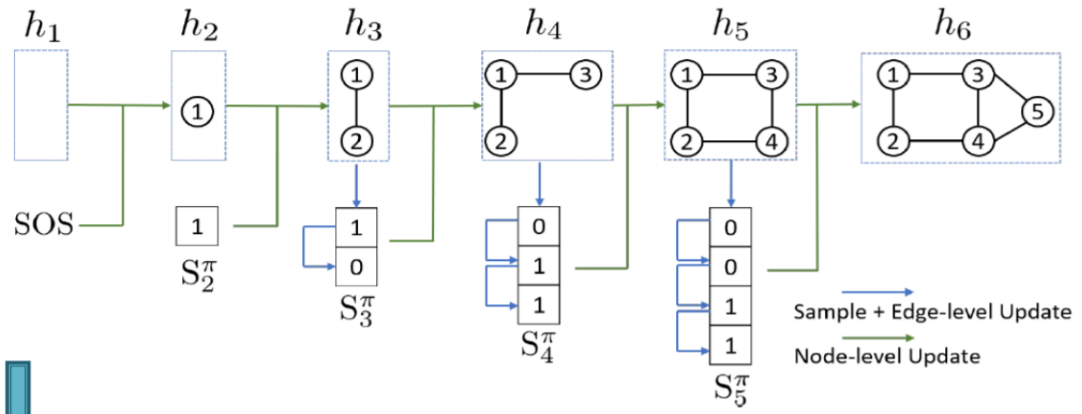
Adjacency matrix

Edge-level RNN generates edges for the new node, then update node-level RNN state using generated results

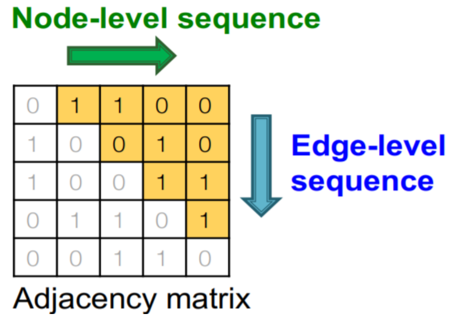
# GraphRNN

## Two levels of RNN

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



Edge-level RNN generates edges for the new node, then update node-level RNN state using generated results



| | | | | |
|---|---|---|---|---|
| 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 |

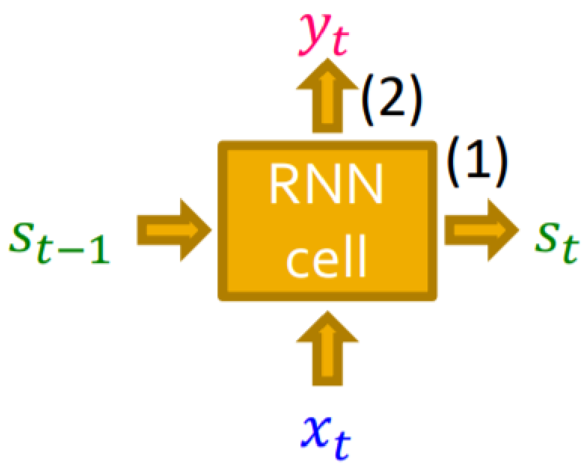
Adjacency matrix

**Next:** How to generate a sequence with RNN?

## GraphRNN

**RNN:** Recurrent Neural Networks

- $s_t$: State of RNN after time $t$
- $x_t$: Input to RNN at time $t$
- $y_t$: Output of RNN at time $t$
- $W, U, V$: parameter matrices, $\sigma(\cdot)$ : non-linear activation function



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

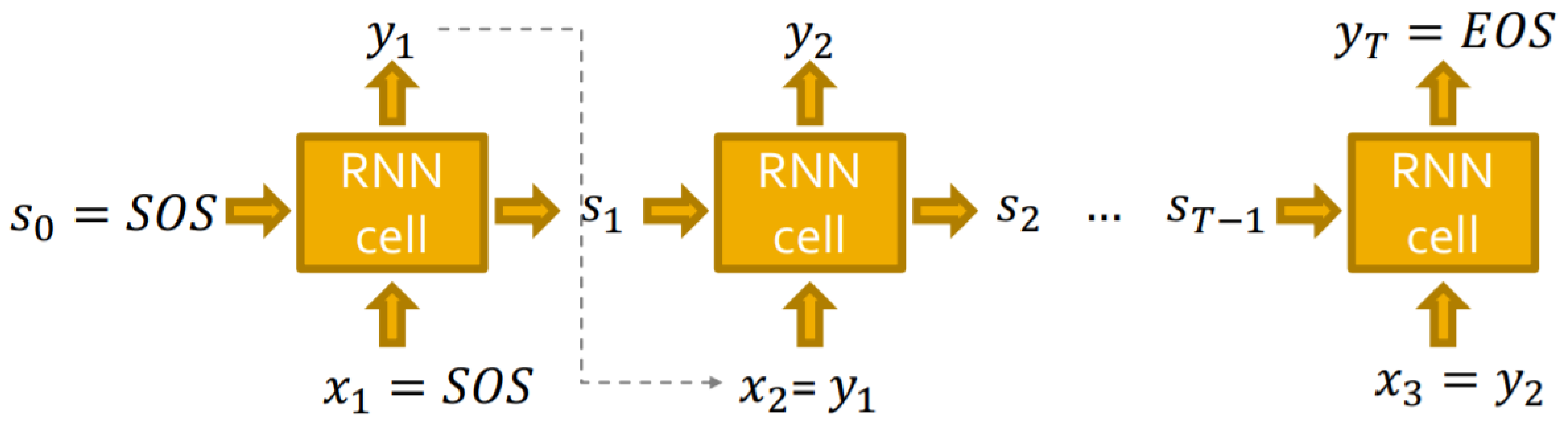
(2) $y_t = V \cdot s_t$

- More expressive cells: GRU, LSTM, etc.

## GraphRNN

### RNN for Sequence Generation

- **Q:** How to use RNN to generate sequences?
- **A:** Let $x_{t+1} = y_t$
- **Q:** How to initialize $s_0$, $x_1$? When to stop generation?
- **A:** Use start/end of sequence token (SOS, EOS)- e.g., zero vector



- This is good, but this model is deterministic

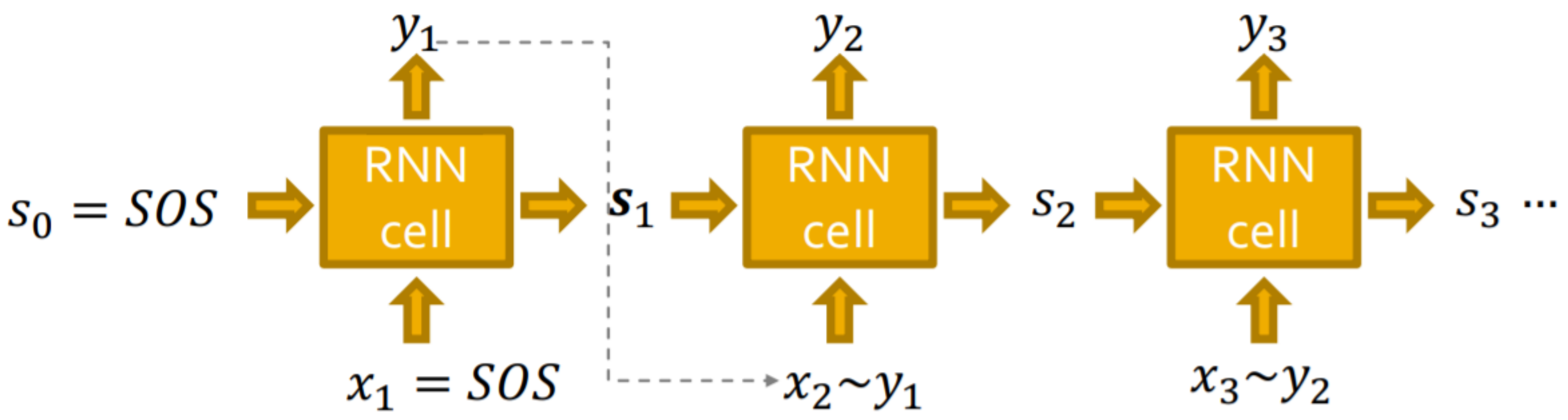
## GraphRNN

### RNN for Sequence Generation

- Remember our goal: Use RNN to model

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

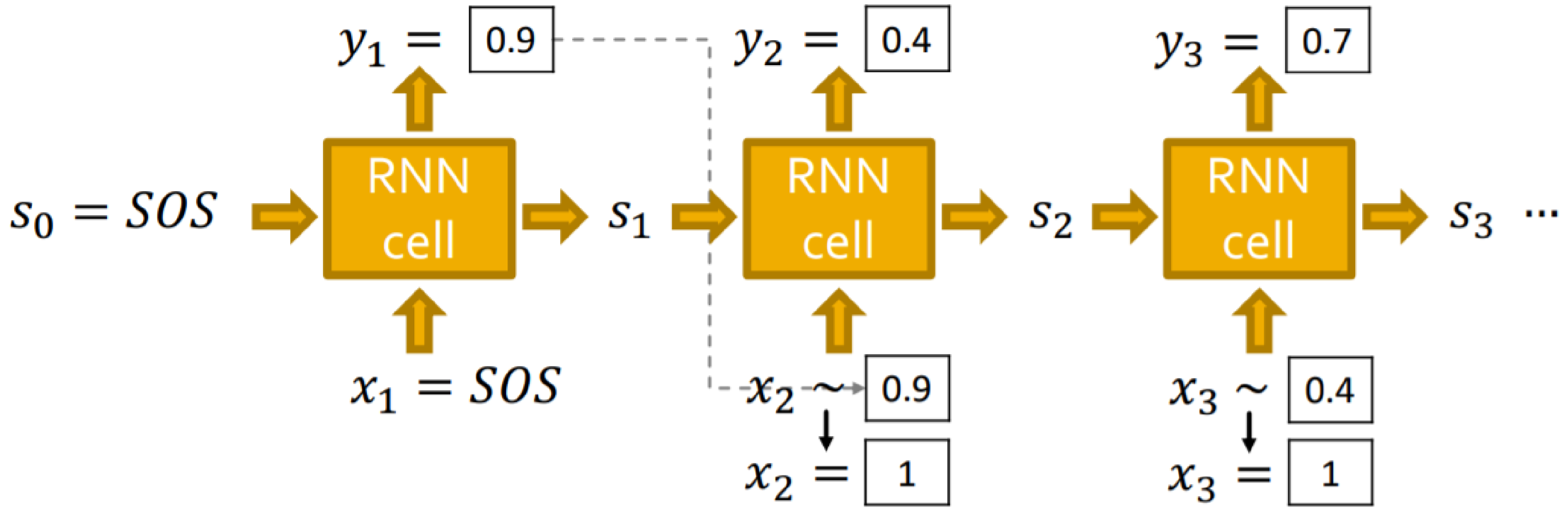
- Let $y_t = P_{model}(x_t|x_1, \dots, x_{t-1}, \boldsymbol{\theta})$
- Then $x_{t+1}$ is a sample from $y_t$: $x_{t+1} \sim y_t$
  - Each step of RNN outputs a probability vector
  - We then sample from the vector, and feed sample to next step:



# GraphRNN

## RNN at Test Time

- Suppose we already have trained the model
  - $y_t$ follows Bernoulli distribution (choice of $P_{model}(x_t|x_1, \ldots, x_{t-1}, \boldsymbol{\theta})$)
  - $y_t = p$ means $x_{t+1}$ has value 1 with prob. $p$, and 0 with prob. 1 − $p$

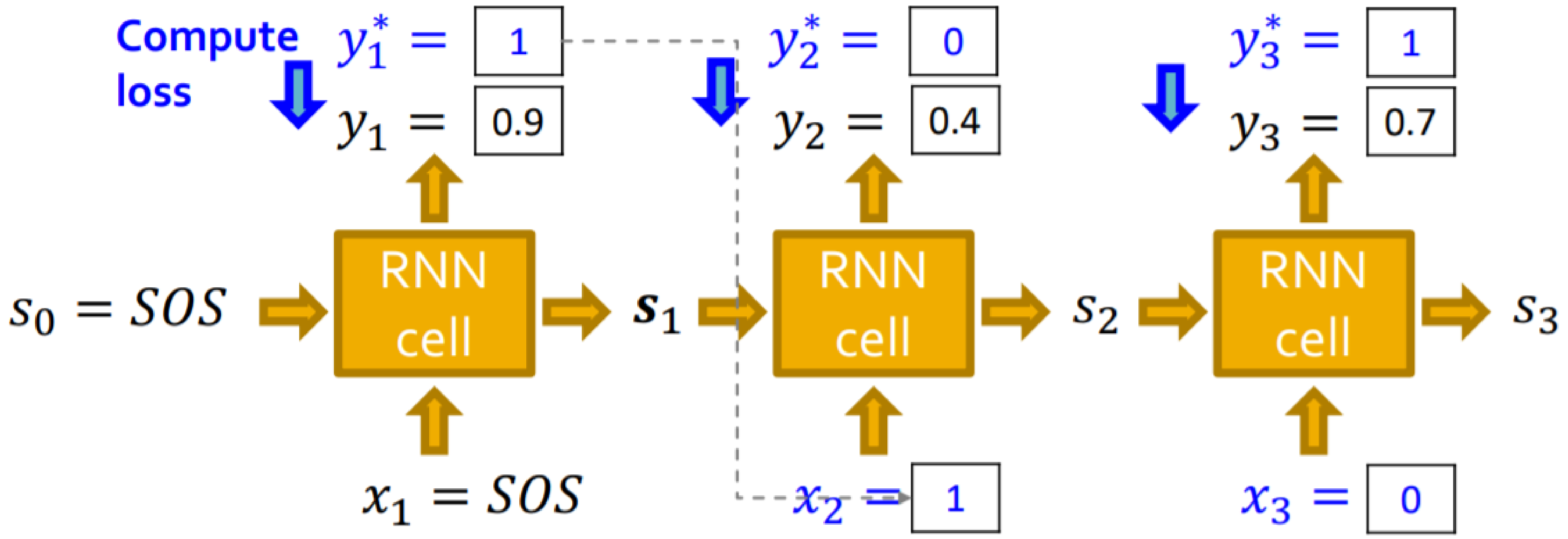


- Right now everything is generated by the model
- How do we use training data $x_1, \ldots, x_n$?

## GraphRNN

### Training **RNN**

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



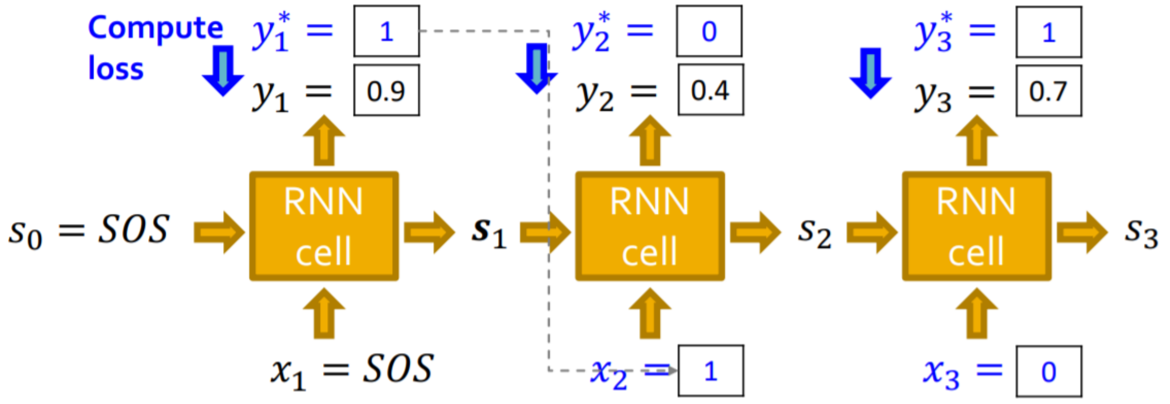
# GraphRNN

## Training **RNN**

- Loss $L$: Binary cross entropy
- Minimize:

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

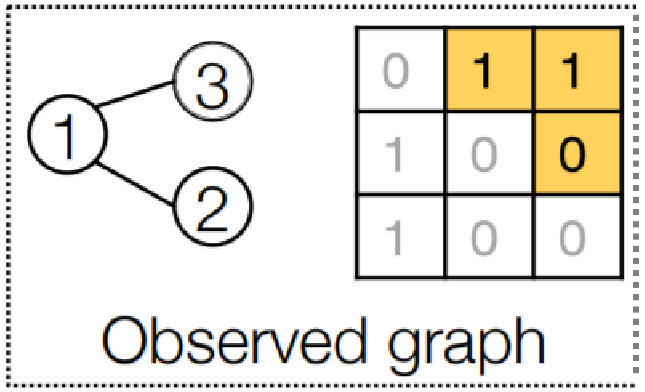
- If $y_i^* = 1,$ we minimize $-\log y_i$, making $y_i$ higher to approach 1
- If $y_i^* = 0,$ we minimize $-\log(1 - y_i)$, making $y_i$ lower to approach 0
- $y_i$ is computed by RNN, this loss will adjust RNN parameters accordingly, using back propagation!

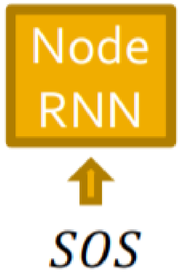


## GraphRNN

### Put Things Together: Training

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

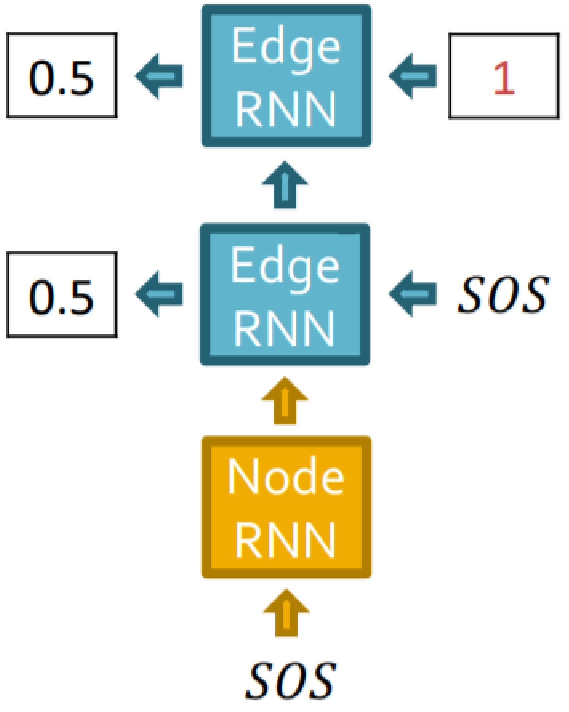


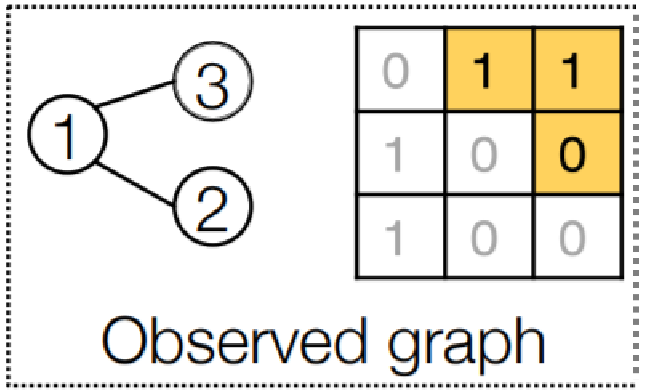


# GraphRNN

## Put Things Together: Training

- Edge RNN predicts how Node 2 connects to Node 1

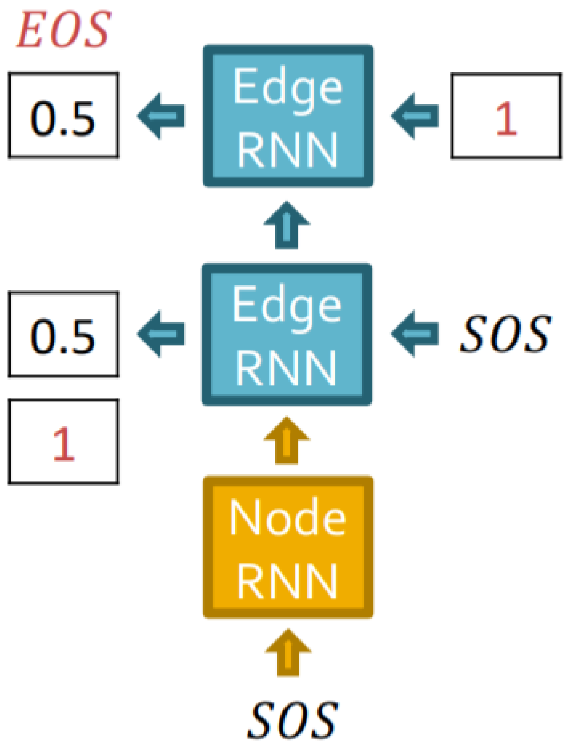


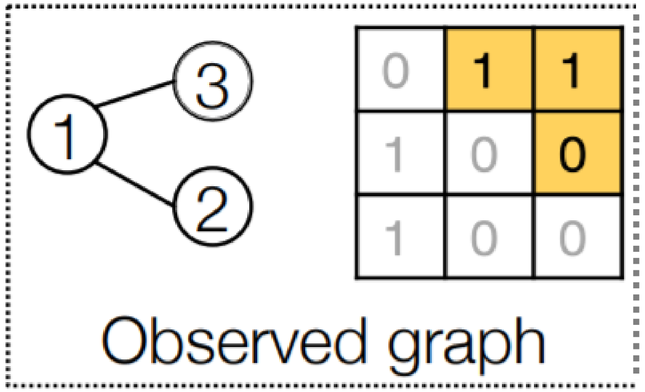


## GraphRNN

### Put Things Together: Training

- Edge RNN gets supervisions from ground truth

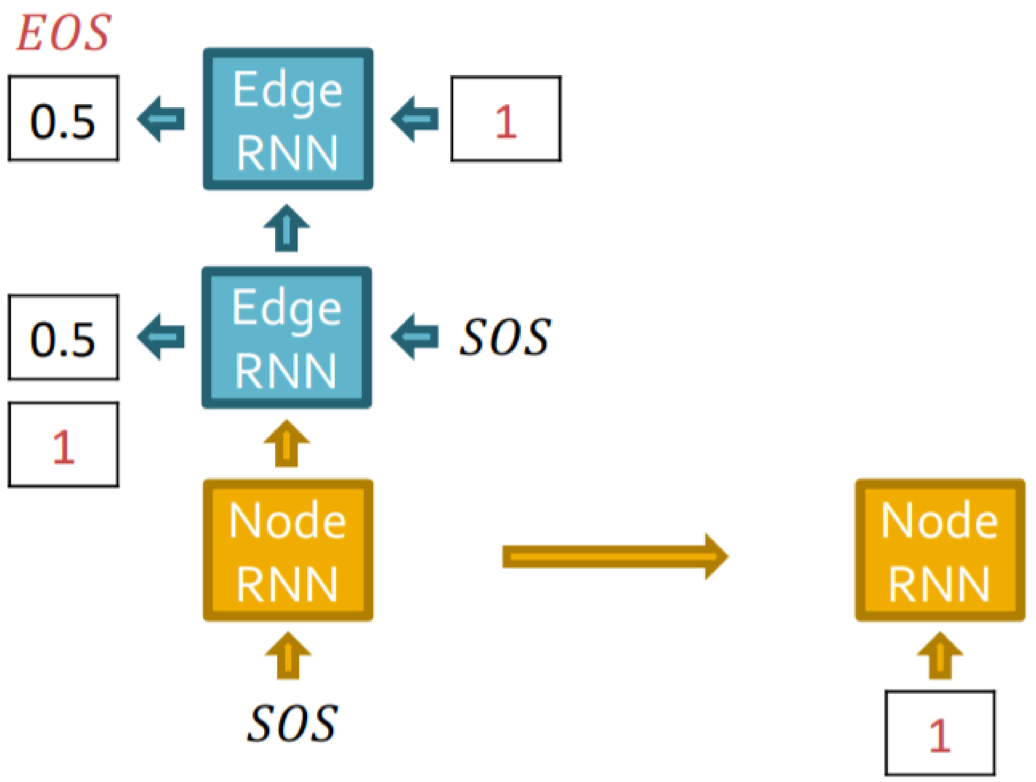


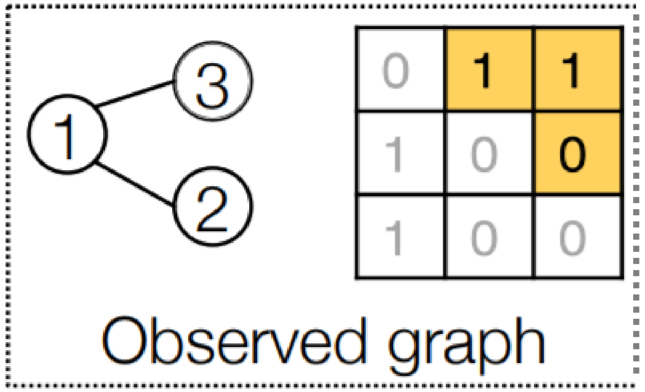


## GraphRNN

### Put Things Together: Training

- New edges are used to update Node RNN

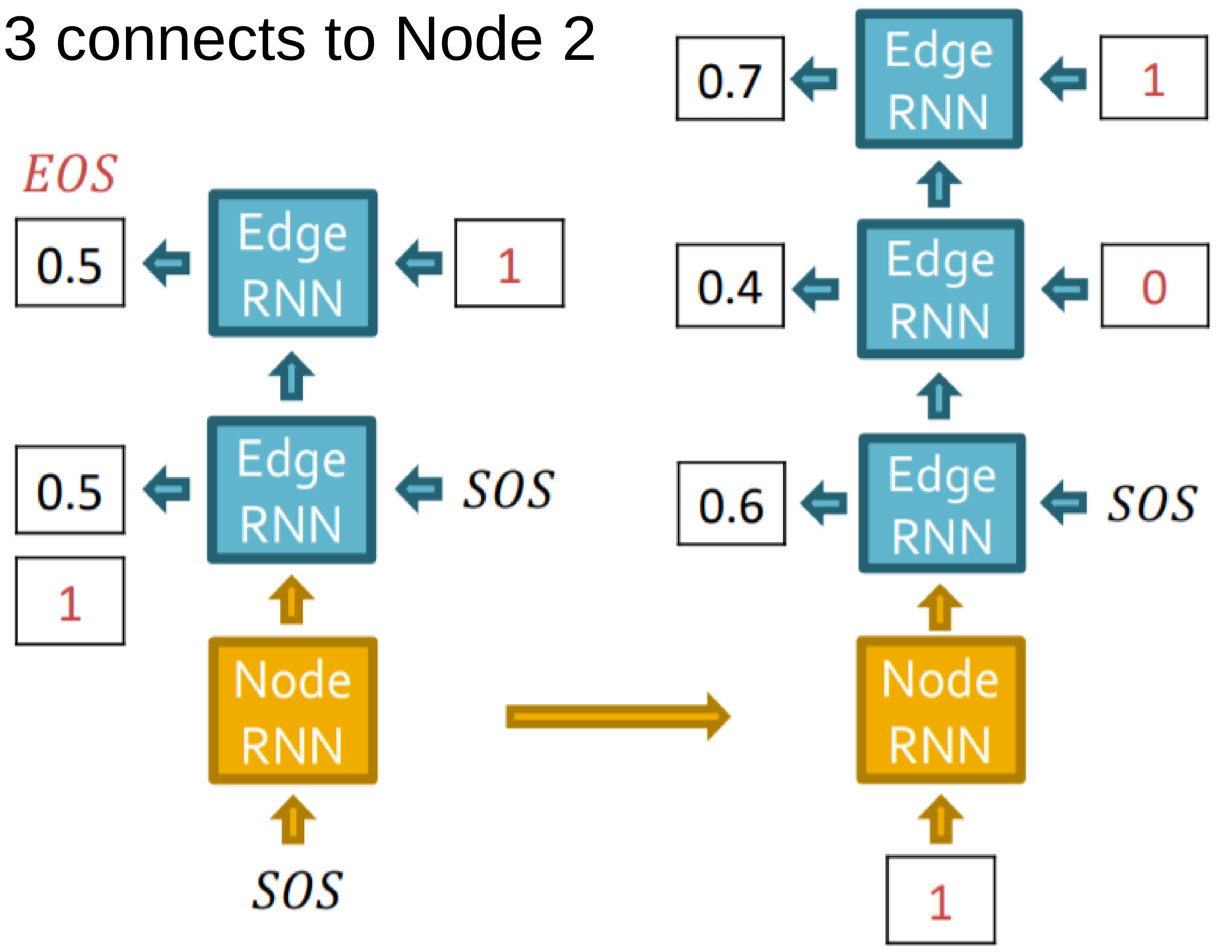


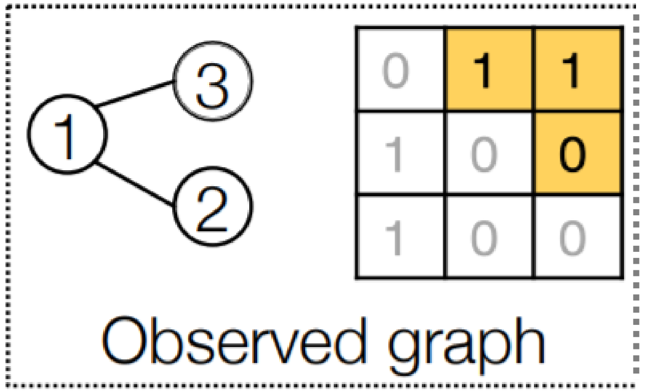


## GraphRNN

### Put Things Together: Training

- Edge RNN predicts how Node 3 connects to Node 2

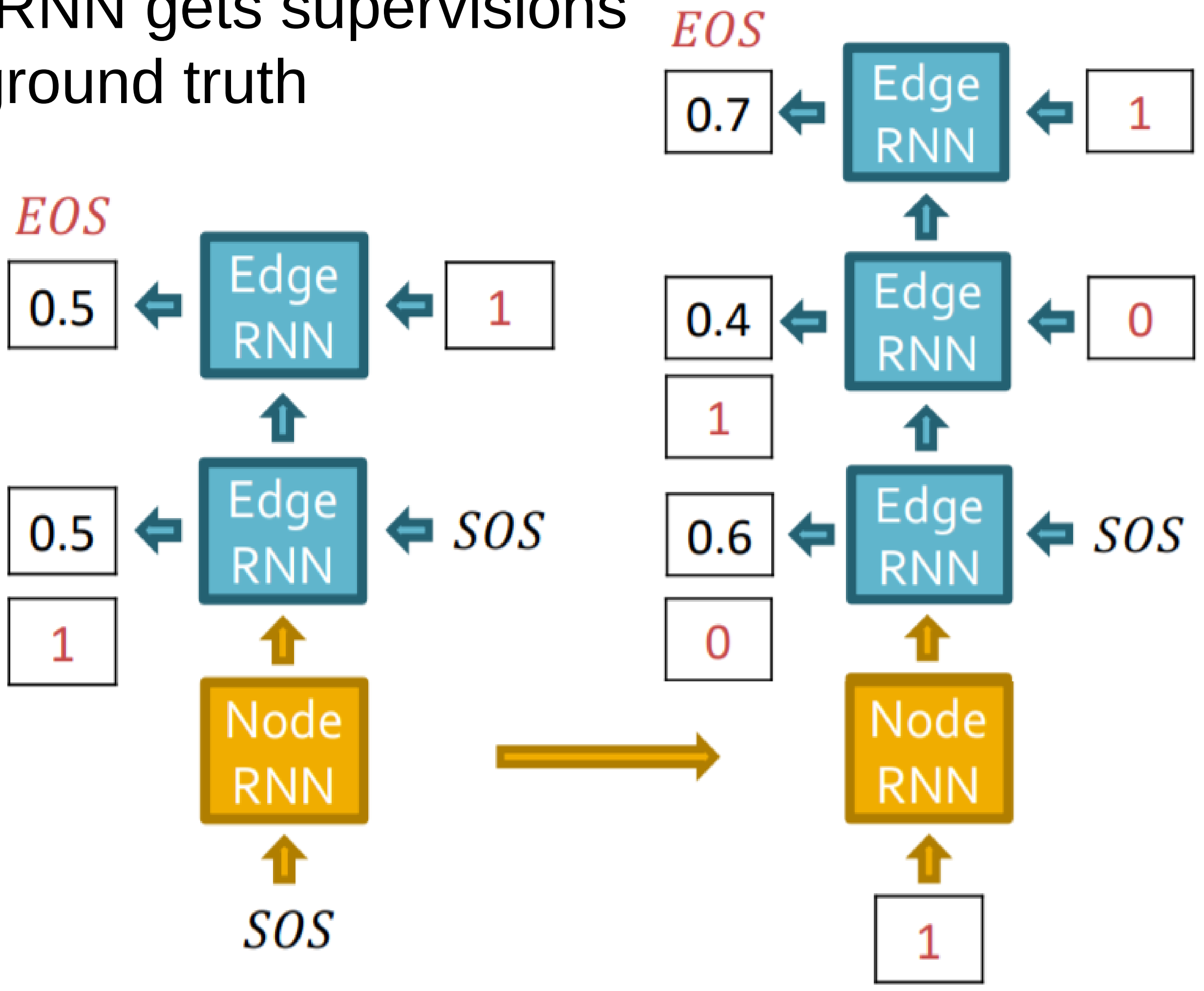


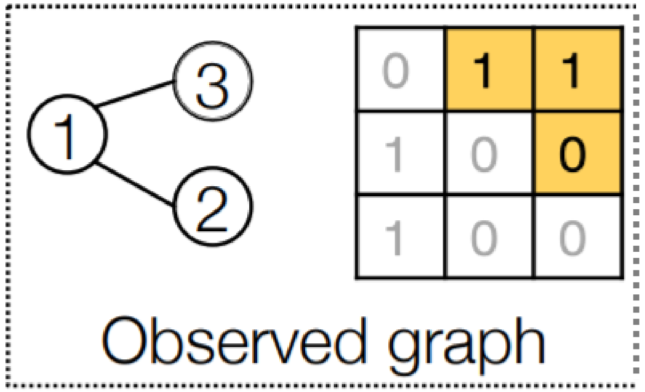


## GraphRNN

### Put Things Together: Training

- Edge RNN gets supervisions from ground truth



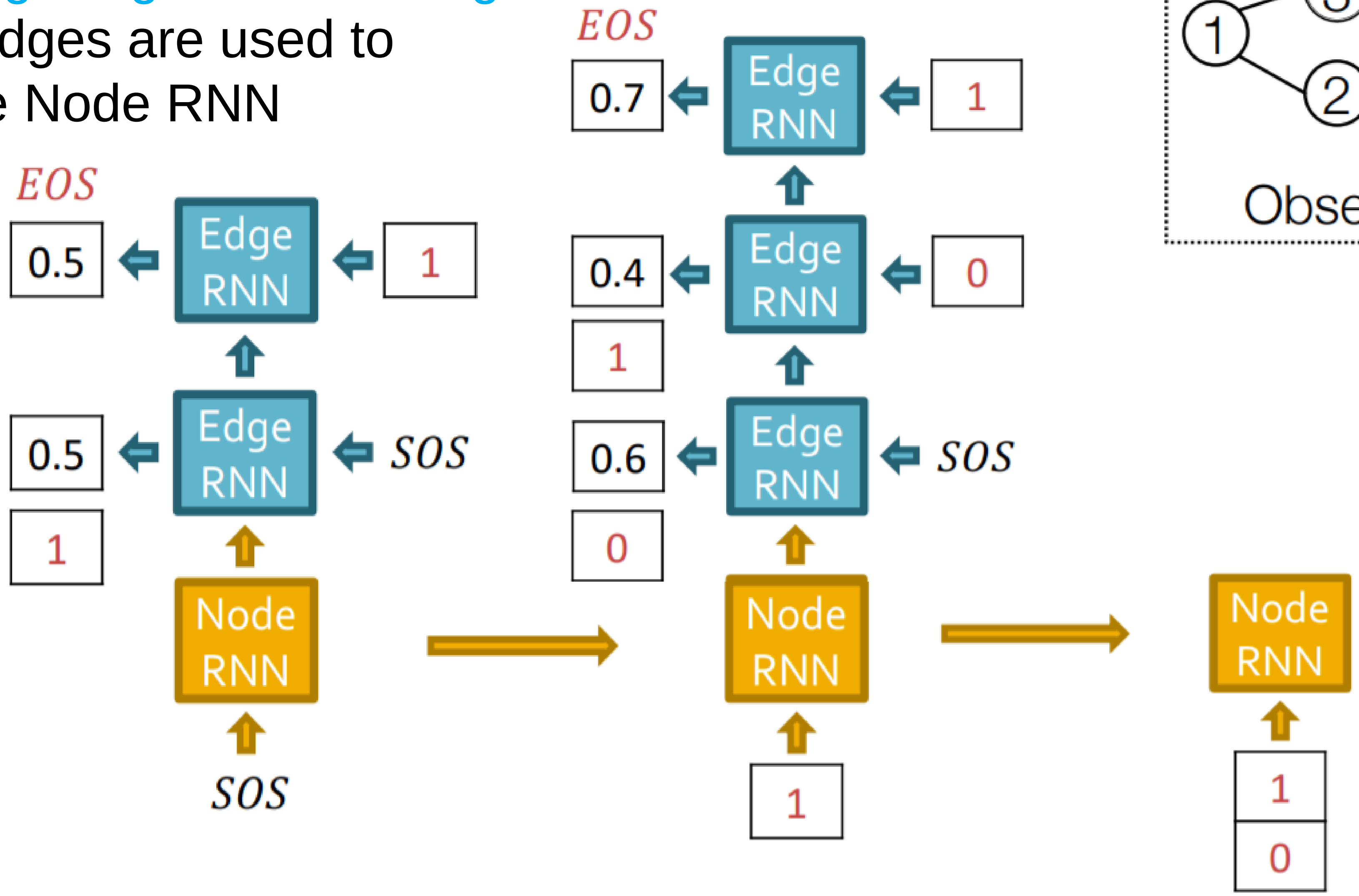


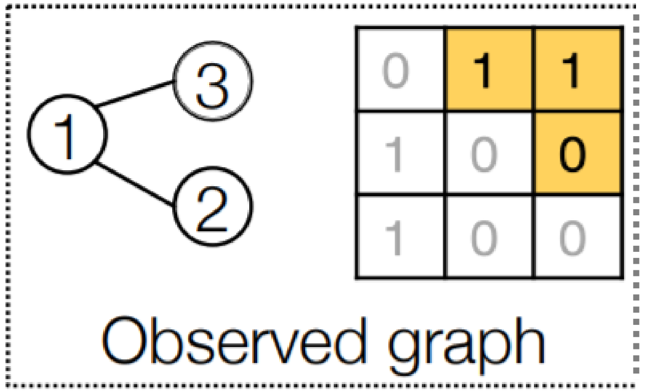
|   |   |   |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

# GraphRNN

## Put Things Together: Training

- New edges are used to update Node RNN



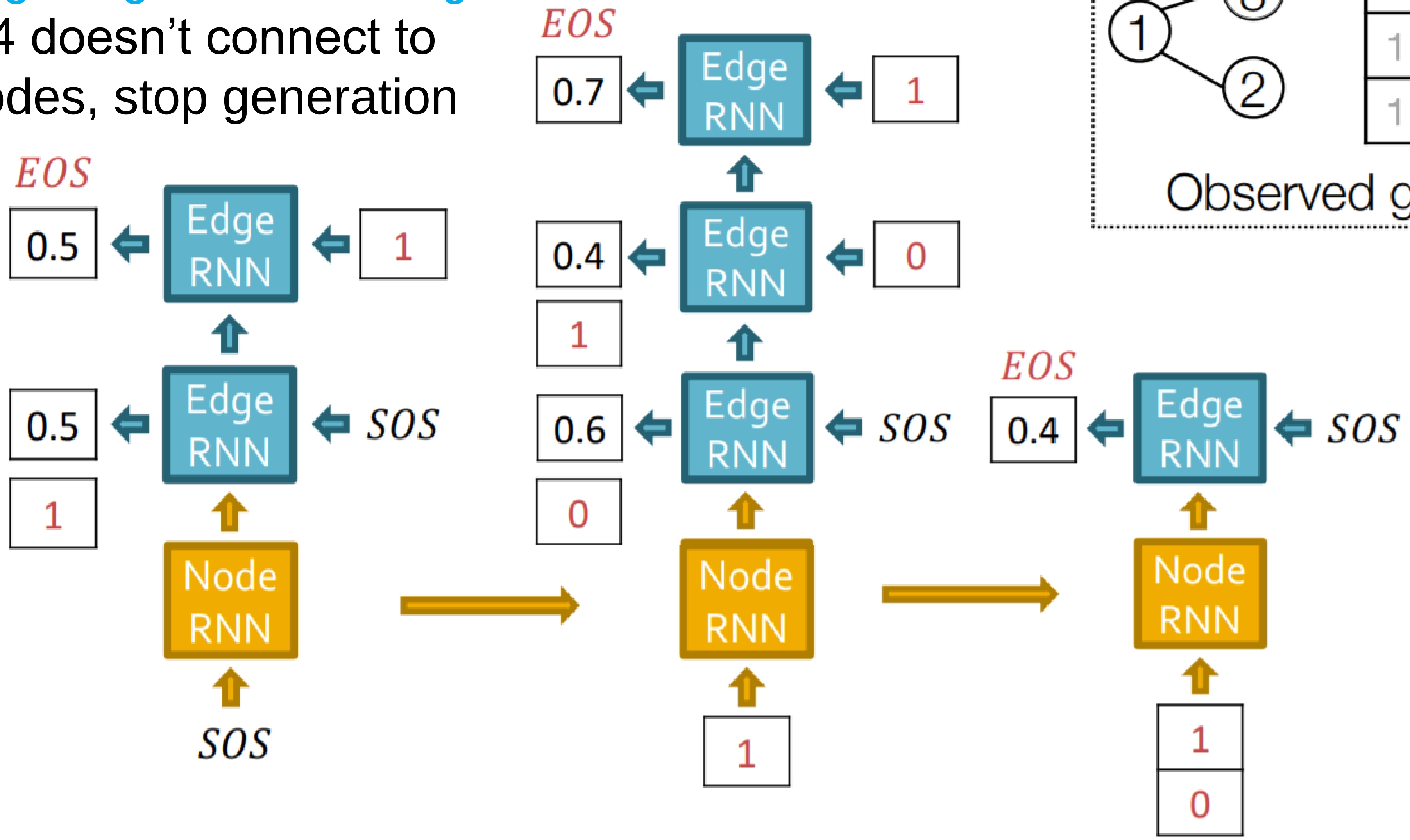


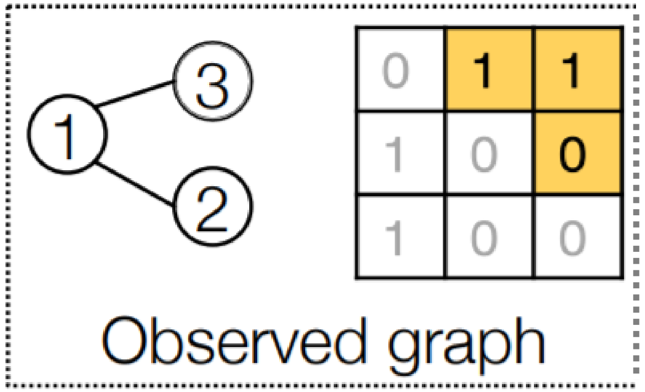
|  |  |  |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

## GraphRNN

### Put Things Together: Training

- Node 4 doesn't connect to any nodes, stop generation

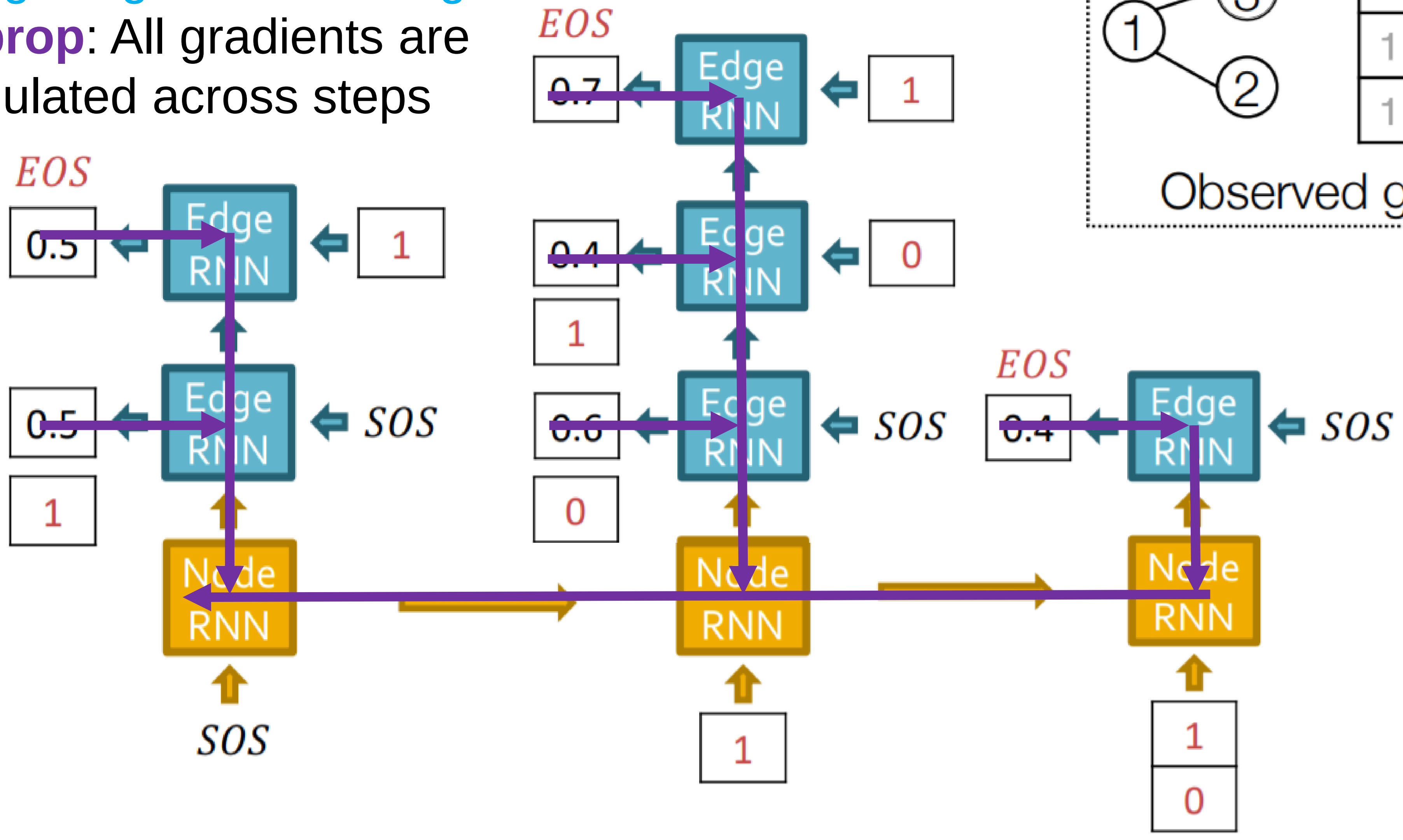


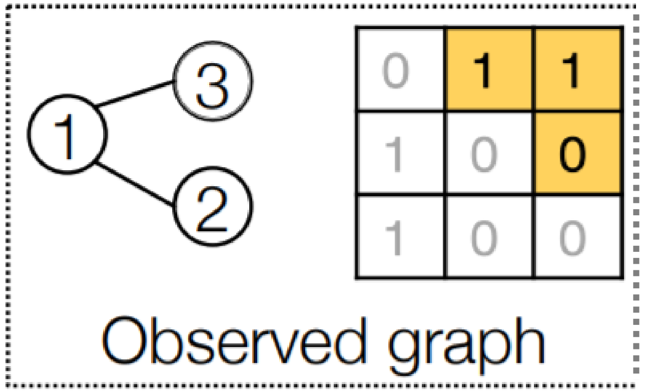


## GraphRNN

### Put Things Together: Training

- **Backprop**: All gradients are accumulated across steps



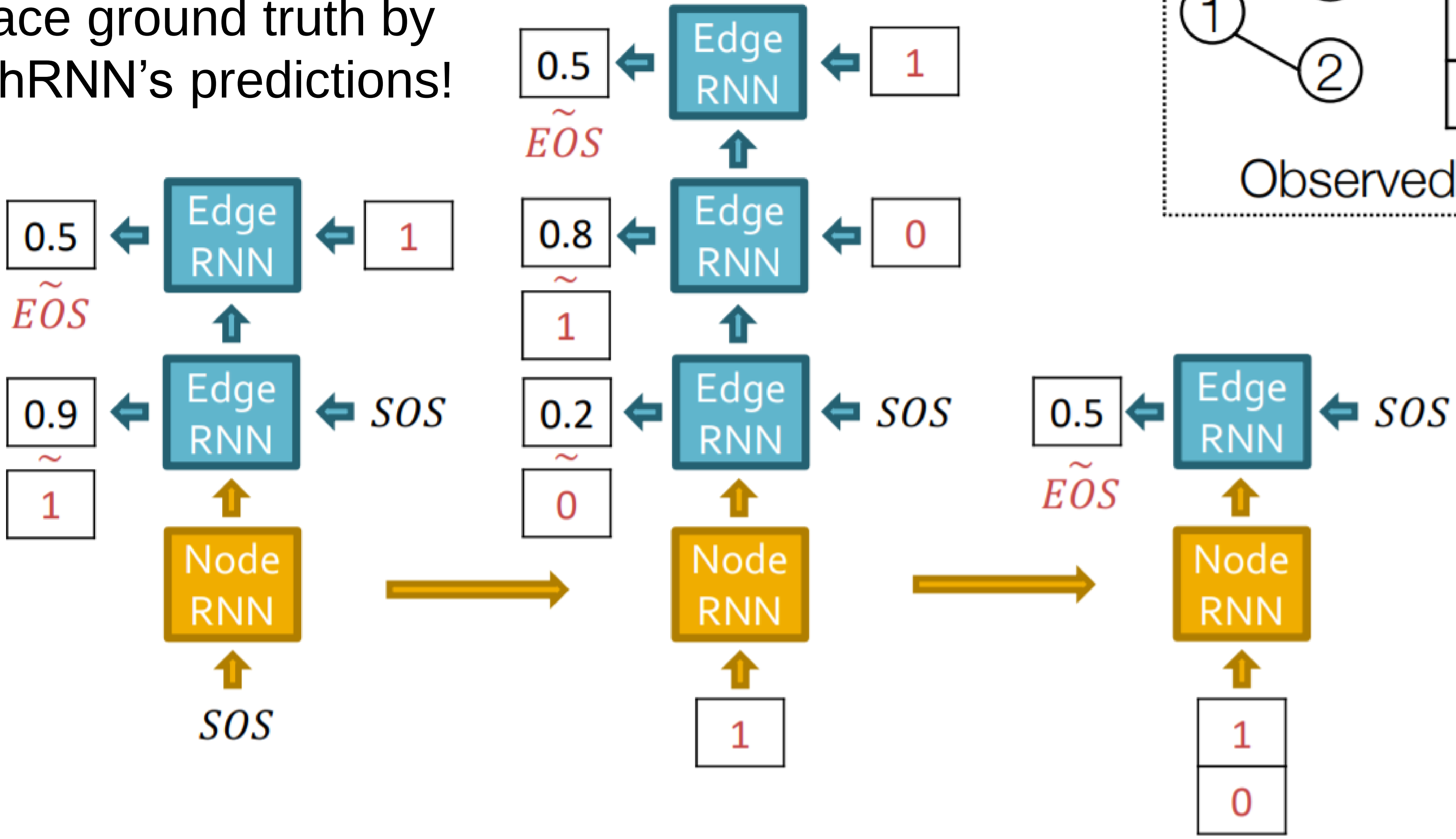


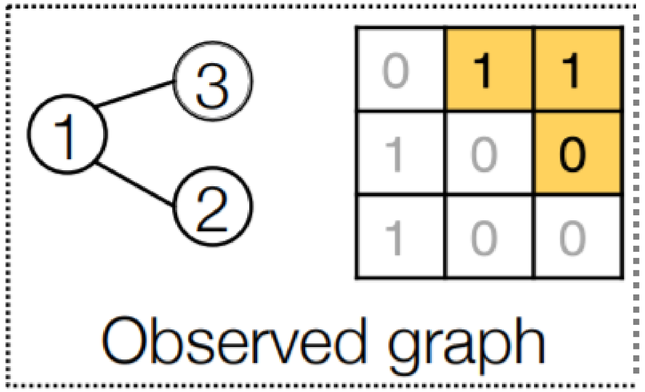
|  |  |  |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

## GraphRNN

### Put Things Together: **Test**

- Replace ground truth by GraphRNN's predictions!





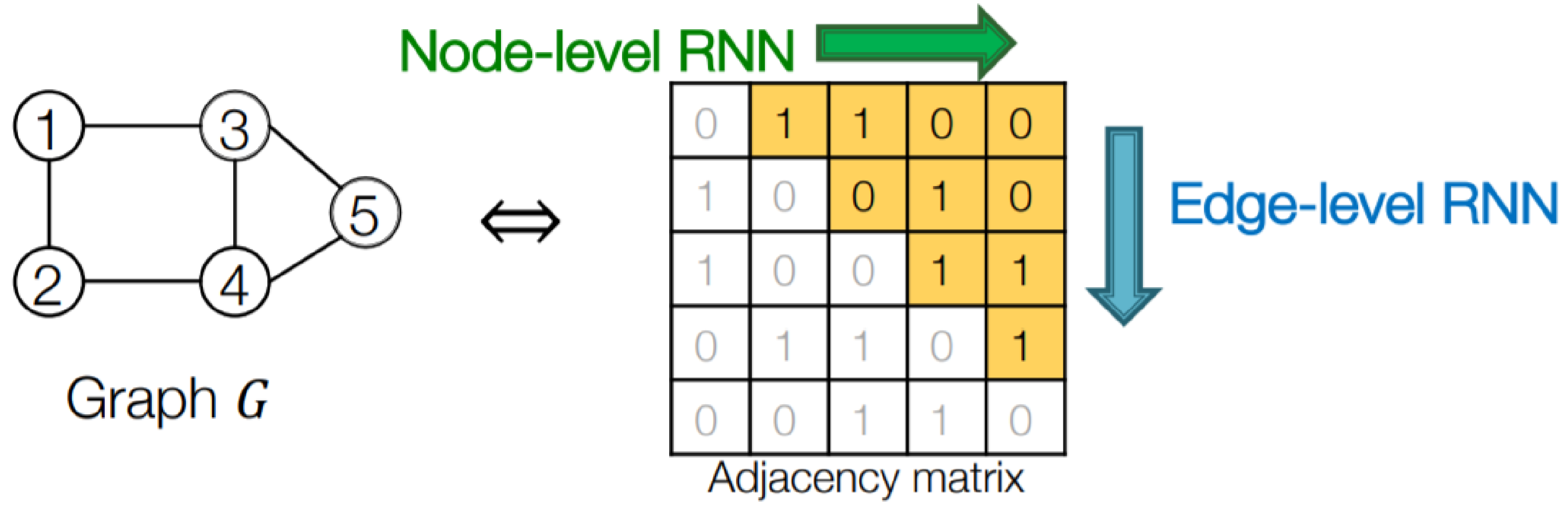
|  |  |  |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |

## **GraphRNN:** Two levels of RNN

**Quick Summary** of GraphRNN:

- Generate a graph by generating a two level sequence
- Use RNN to generate the sequences
- Lack of connection to the encoder

- Next: Making GraphRNN tractable, proper evaluation

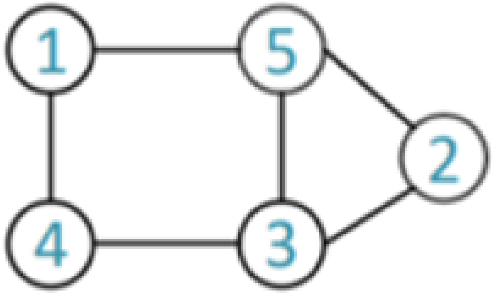


|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |

## GraphRNN:

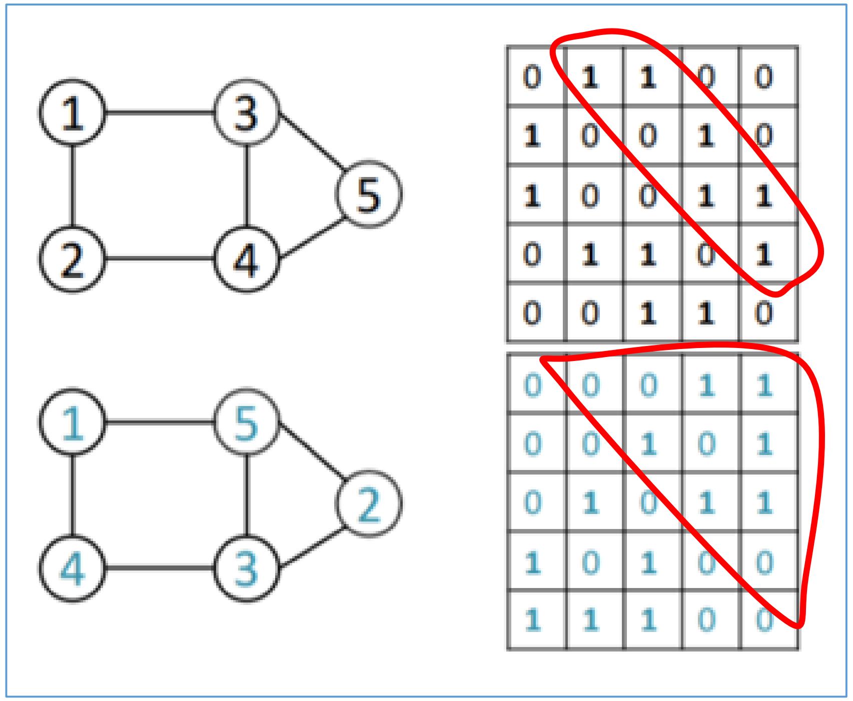
**Tractability**:

- 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
- Add node 1
- Add node 5
- Add node 6
- Add node 4
- ….

→ Sequential edge connection modeling is too complex



| | | | | |
|---|---|---|---|---|
| 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 |

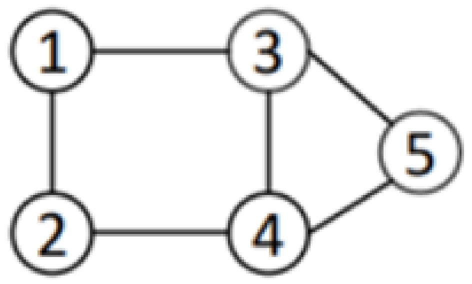
| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |

How do we limit this complexity?

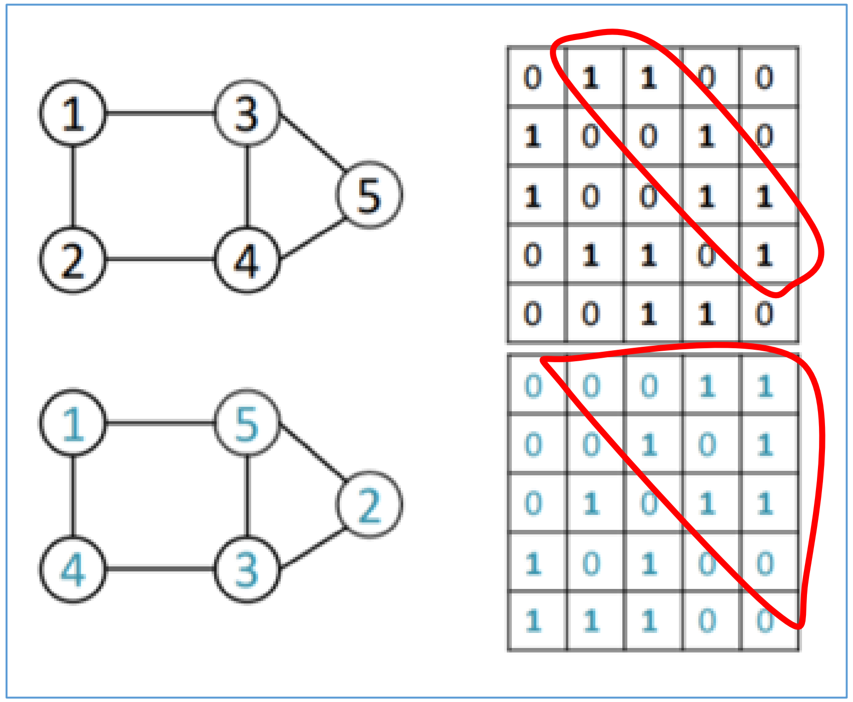
## GraphRNN:

### Tractability via BFS

- Breadth-First Search node ordering



BFS Ordering
- Add node 1
- Add node 2
- Add node 3
- Add node 4
- ….



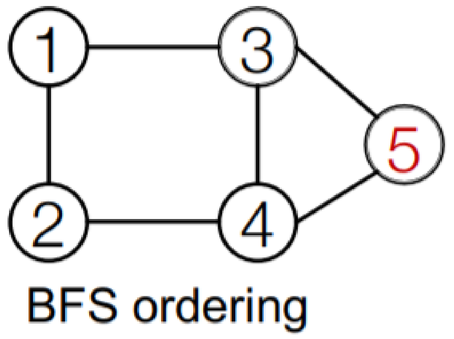
$\rightarrow$ need memory of a couple of “steps” in a hierarchical manner

- BFS node ordering:
  - We know all Node 1’s neighbors have already been traversed
  - Two hop neighbors such as Node 4, 5 need not to connect to Node 1
  - Therefore, multi-hop nodes know the connections to the grandparant nodes
  - We only need memory of a couple of “steps” rather than $n - 1$ steps

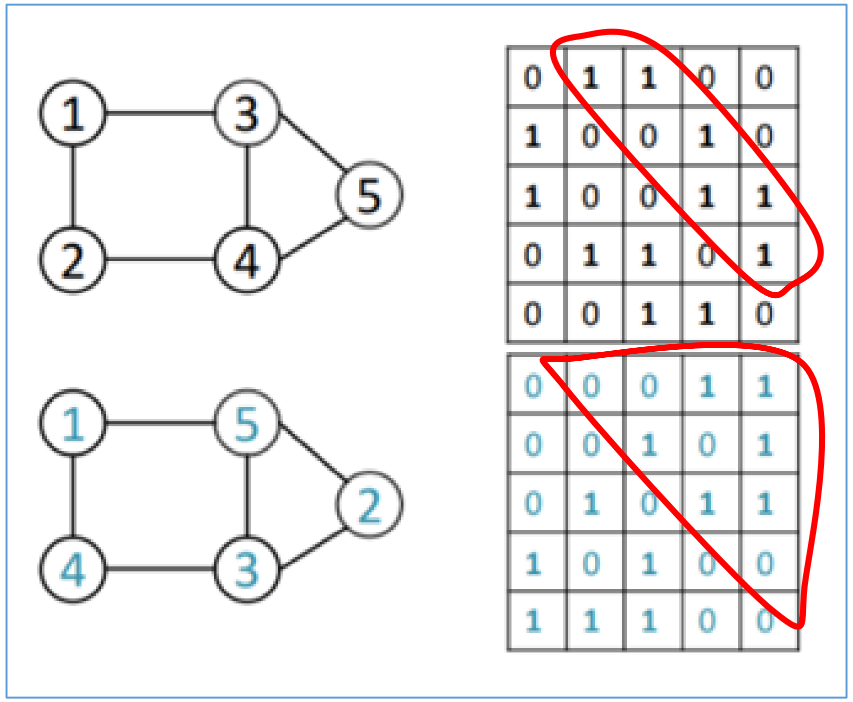
## GraphRNN:

**Tractability via BFS**

- Breadth-First Search node ordering



BFS Ordering
- Add node 1
- Add node 2
- Add node 3
- Add node 4
- ….



|  |  |  |  |  |
|---|---|---|---|---|
| 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 |

|  |  |  |  |  |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |

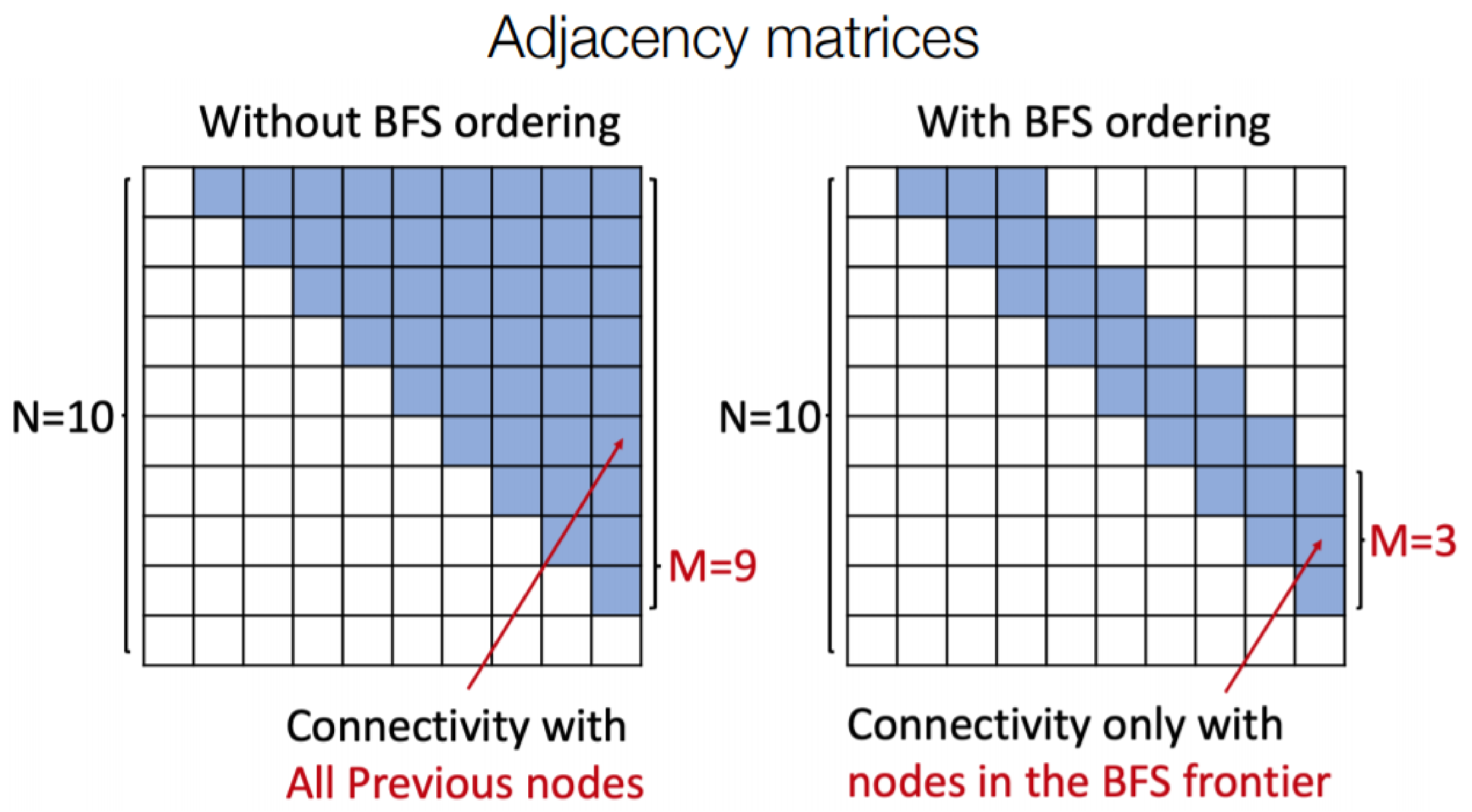
$\rightarrow$ need memory of a couple of “steps” in a hierarchical manner

- 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

## GraphRNN:

**Tractability via BFS**

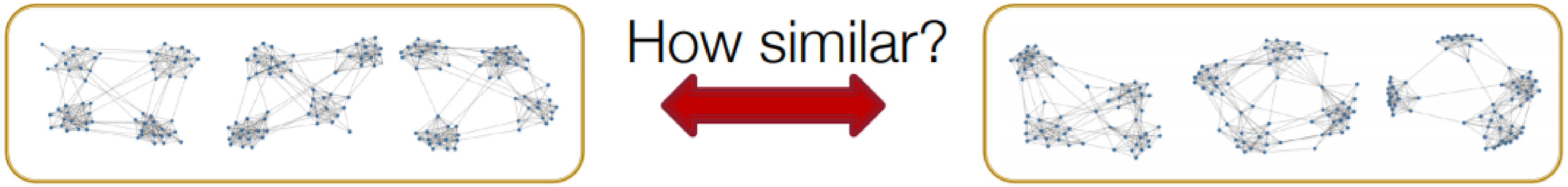
- BFS Reduce the number of steps for edge generation



## GraphRNN:

**Evaluating** Generated Graphs

- Task: Compare two sets of graphs

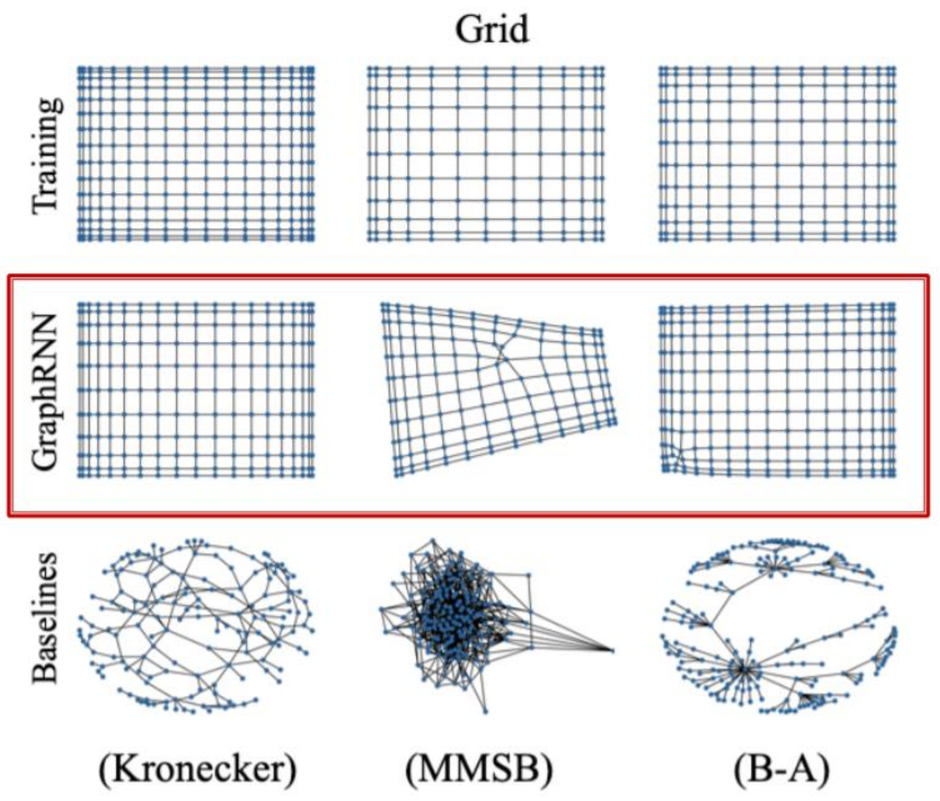


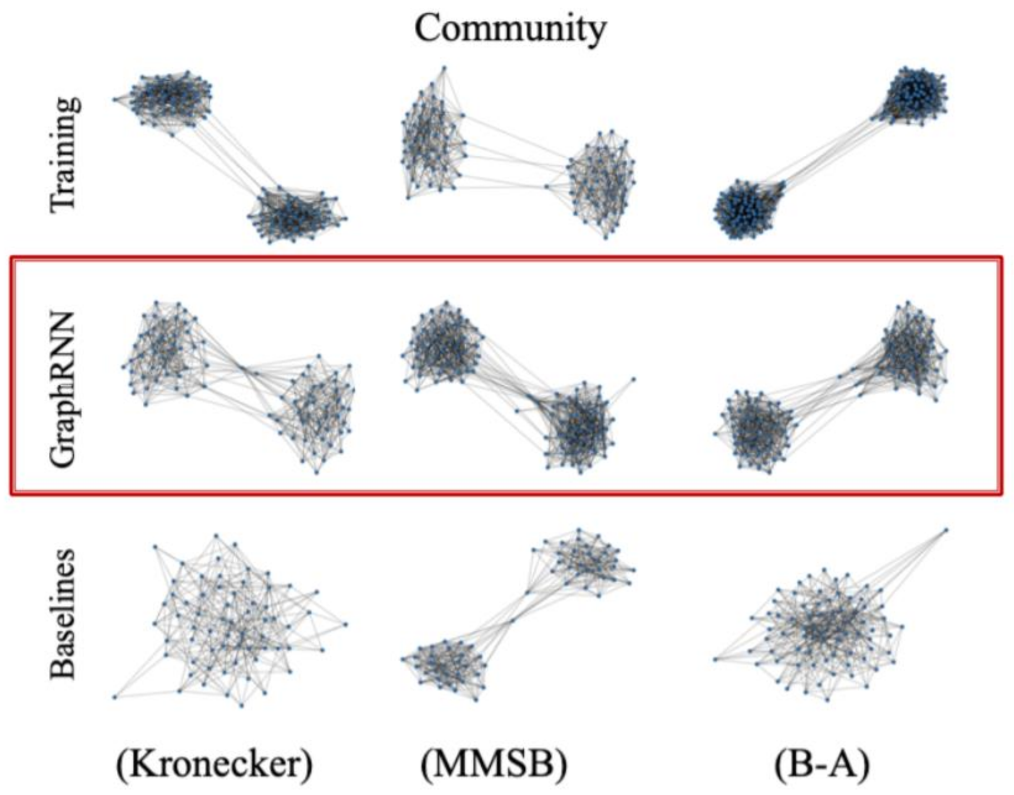
- Goal: Define similarity metrics for graphs
- Challenge: There is no efficient Graph Isomorphism test that can be applied to any class of graphs!
- Solution
  - Visual similarity
  - Graph statistics similarity

Graph similarity scoring and matching
Graph Similarity and Approximate Isomorphism
Modeling and Measuring Graph Similarity: The Case for Centrality Distance

## GraphRNN:

### Visual Similarity





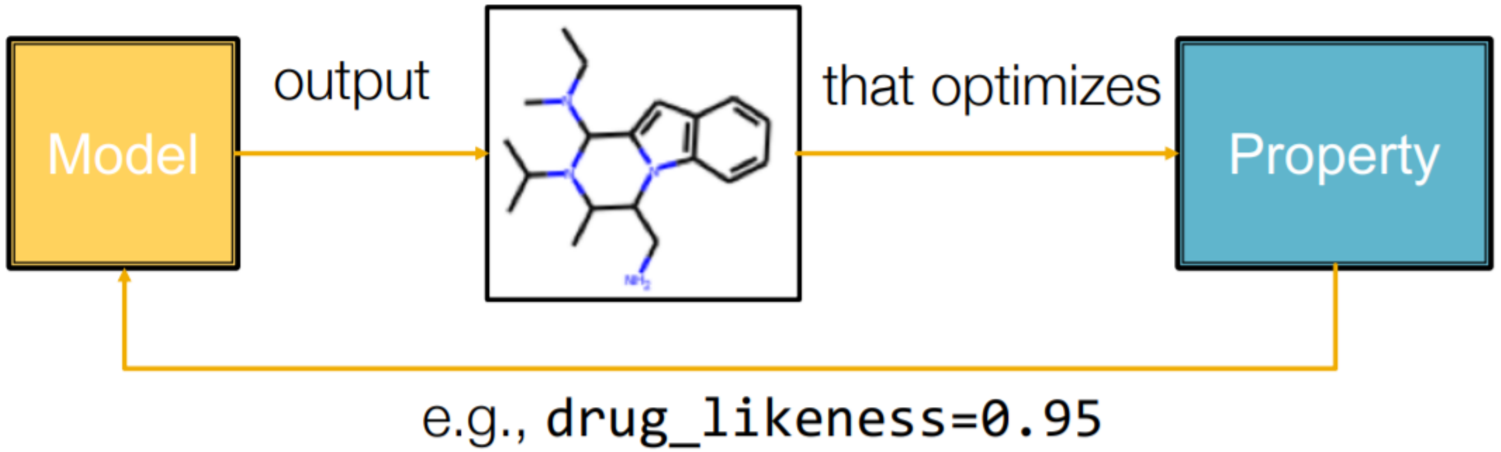
## Deep Graph Decoders

### Outline of Contents

## Applications

Drug Discovery

- **Question**: Can we learn a model that can generate **valid** and **realistic** molecules with high value of a given chemical property?



GCPN: 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.

←Link prediction by Reinforcement learning (Policy Gradient Training)

## Applications

Goal-Directed Graph Generation

- Optimize a given objective (High scores)
  - e.g., drug-likeness (black box)
- Obey underlying rules (Valid)
  - e.g., chemical valency rules
- Are learned from examples (Realistic)
  - e.g., Imitating a molecule graph dataset

GCPN: 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.

←Link prediction by Reinforcement learning (Policy Gradient Training)

## Applications

**Graph Convolutional Policy Network** combines graph representation + RL:

- Graph Neural Network captures complex structural information, and enables validity check in each state transition (Valid)
- Reinforcement learning optimizes intermediate/final rewards (High scores)
- Adversarial training generates samples to imitate examples in given datasets (Realistic)

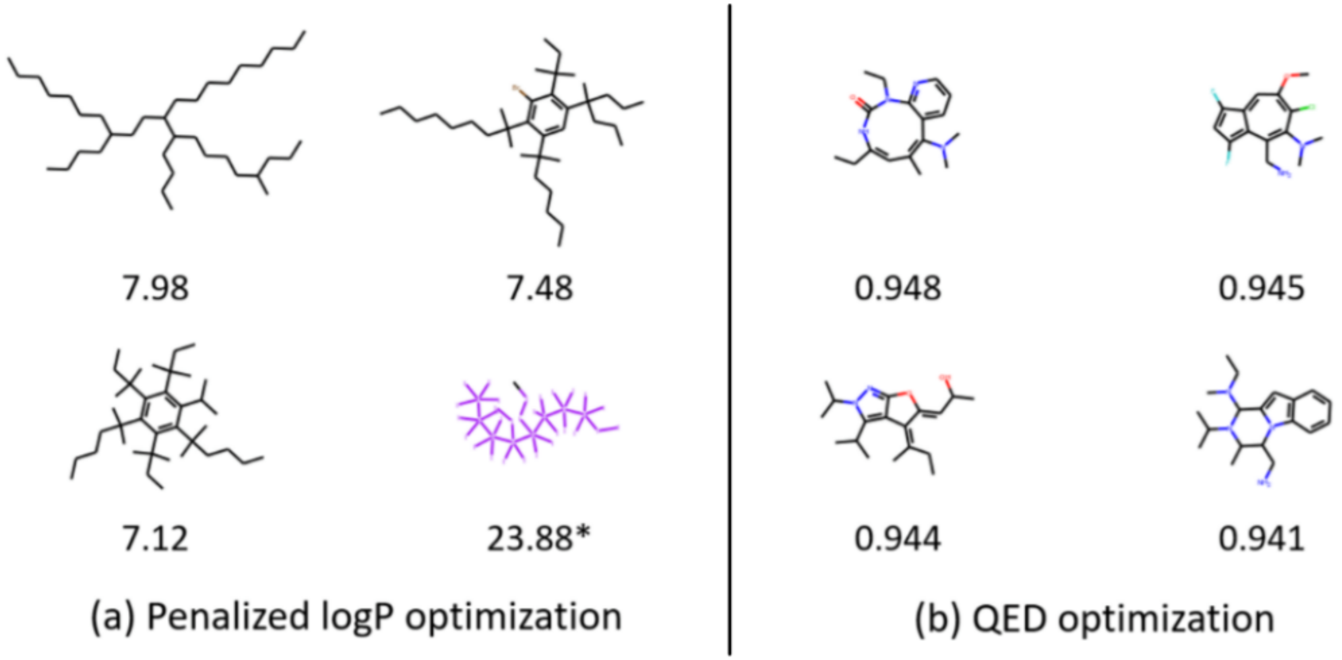
GCPN: Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation. J. You, B. Liu, R. Ying, V. Pande, J. Leskovec. Neural Information Processing Systems (NeurIPS), 2018.

←Link prediction by Reinforcement learning (Policy Gradient Training)

## Applications

**Qualitative Results**:

- Visualization of GCPN graphs:
  - Generate graphs with high property scores

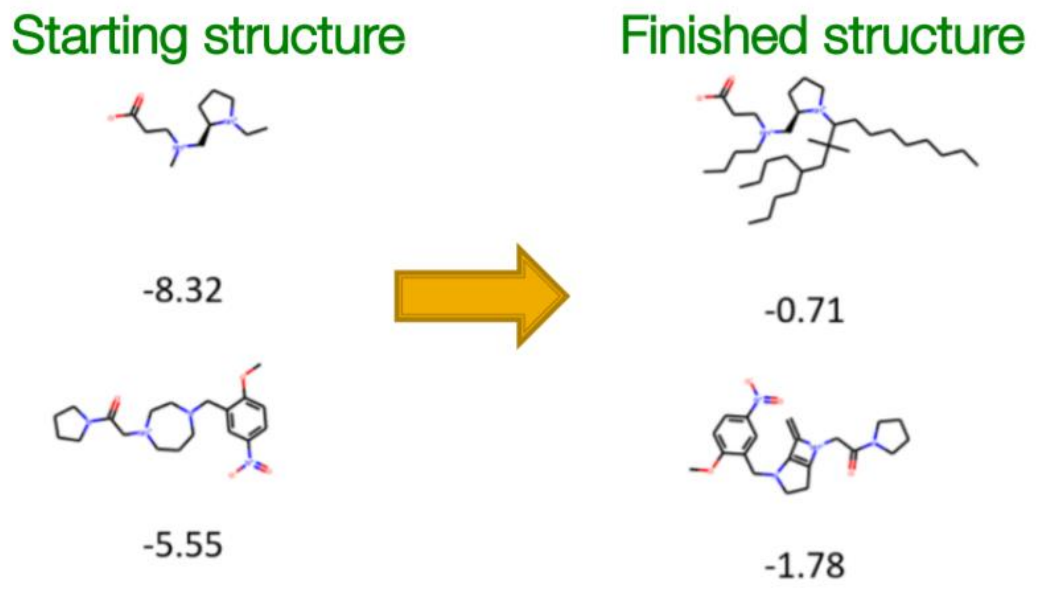


(a) Penalized logP optimization (b) QED optimization

## Applications

**Qualitative Results**:

- Visualization of GCPN graphs:
  - Edit given graph for higher property scores



Constrained optimization of penalized logP

## Open Problems

- Generating graphs in other domains
  - 3D shapes, point clouds, scene graphs, etc.
- Scale up to large graphs
  - Hierarchical action space, allowing high-level action like adding a structure at a time
- Other applications: Anomaly detection (Auto-Encoder, GAN)
  - Use generative models to estimated prob. of real graphs vs. fake graphs

Efficient Graph Generation with Graph Recurrent Attention Networks, NeurIPS 2019 (서성욱 발표)

## Summary Questions of the lecture

- What is the meaning of the output of each RNN cell in **GraphRNN**?
- How can we obtain the input of each RNN cell in **GraphRNN**?
- Explain the training method of **GraphRNN** in the view point of loss and training path.