IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

Week 09: Advanced Topics

Instructor: Thanh H. Nguyen

Many slides are adapted from https://web.stanford.edu/class/cs224w/

Deep Generative Models for Graphs

Motivation for Graph Generation

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

Image credit: Medium

Social Networks

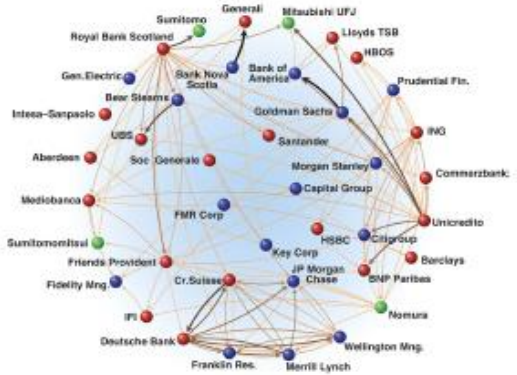


Image credit: Science

Economic Networks

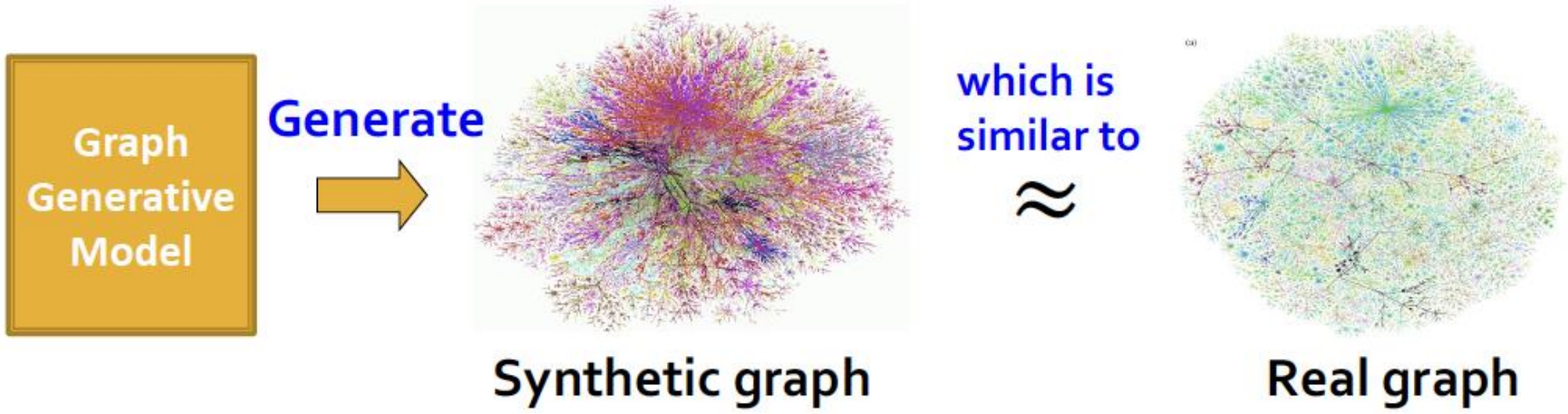
Image credit: Lumen Learning

Communication Networks

- But how are these graphs generated?

The Problem: Graph Generation

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



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

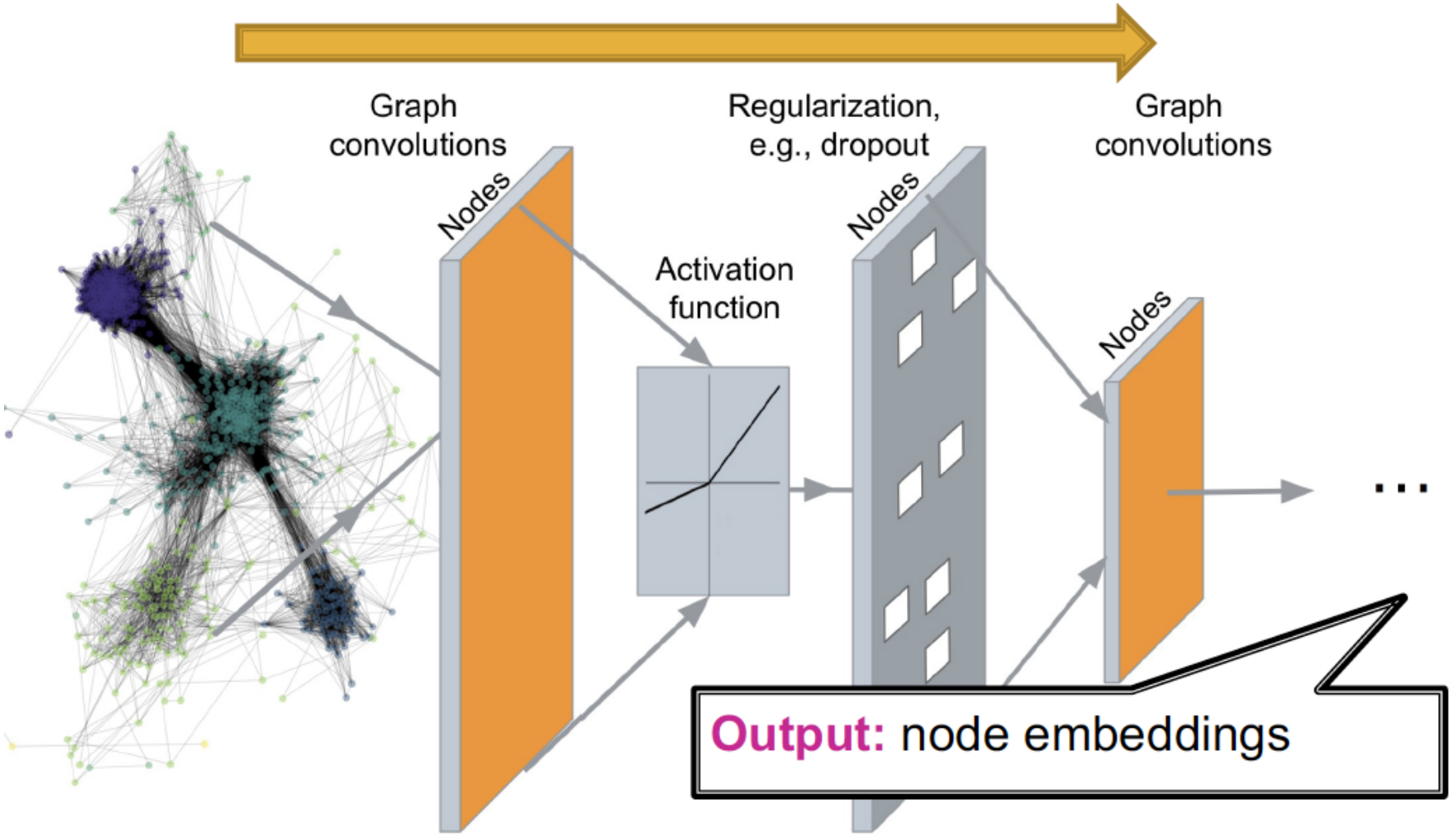
Why do We Study Graph Generation?

- 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

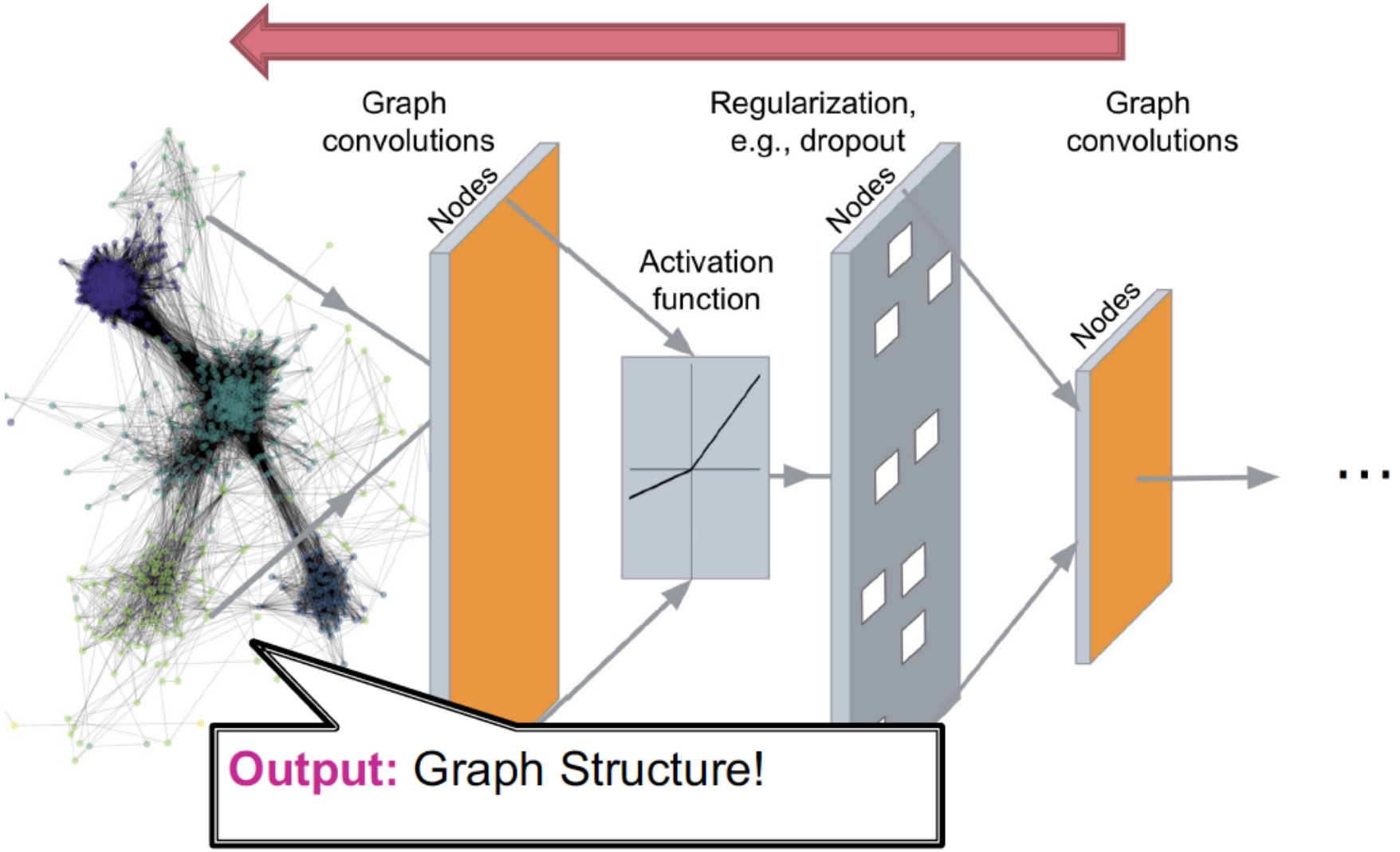
History of Graph Generation

- 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



Today: Deep Graph Decoders



Machine Learning for Graph Generation

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 $p_{model}(G)$

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 $p_{model}(x;\theta)$ 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

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

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

Generative Models: Basics

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

- 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 $f(\cdot)$: $\boldsymbol{x}_i = f(z_i;\ \theta)$
  - Then $\boldsymbol{x}_i$ follows a complex distribution

- Q: How to design $\boldsymbol{f}(\cdot)$?
- A: Use Deep Neural Networks, and train it using the data we have!

Deep Generative Models

Auto-regressive models:

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

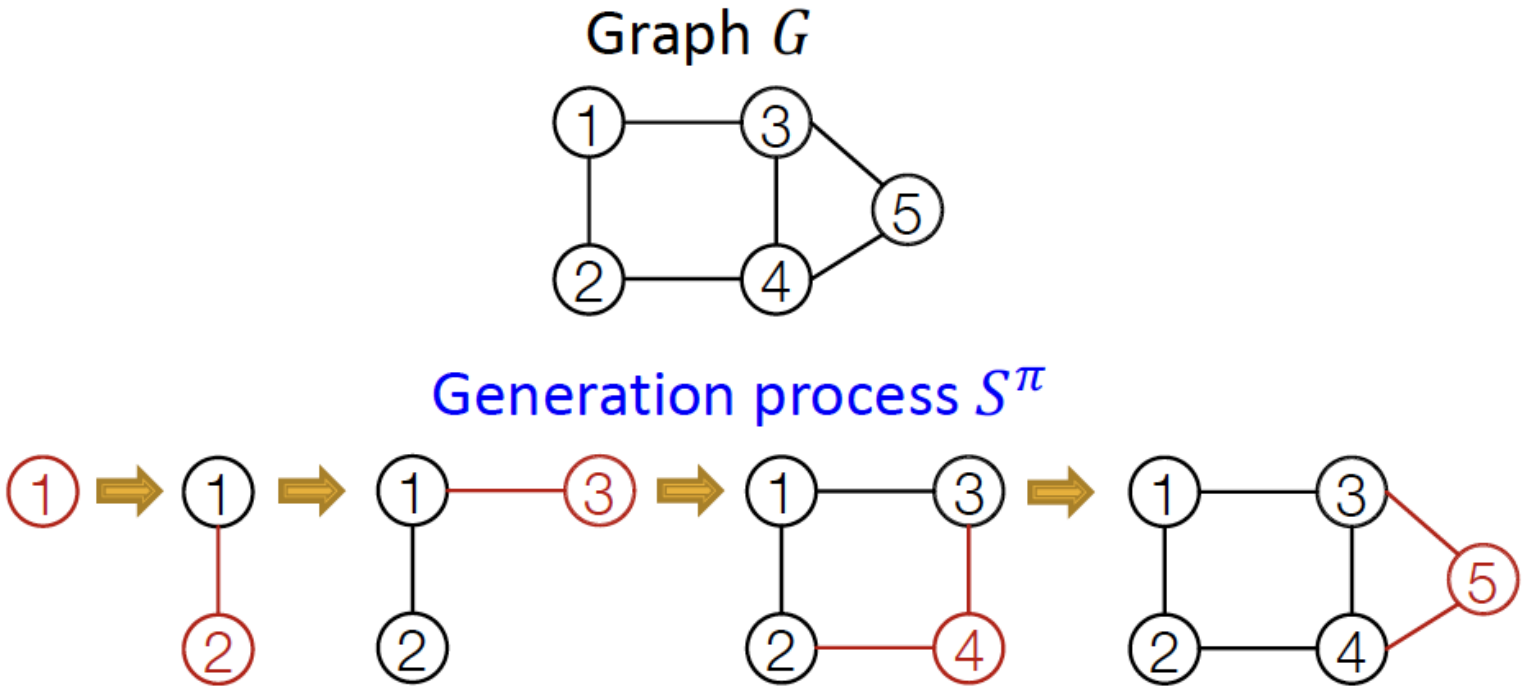
$$p_{model}(\boldsymbol{x};\theta) = \prod_{t=1}^{n} p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$$

  - 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)

GraphRNN: Generating Realistic Graphs

GraphRNN Idea

- Generating graphs via sequentially adding nodes and edges

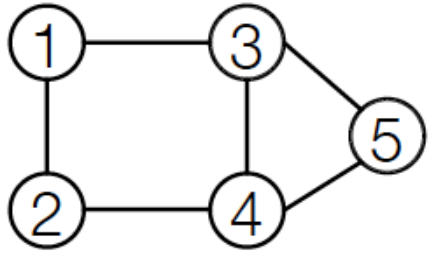


GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. J. You, R. Ying, X. Ren, W. L. Hamilton, J. Leskovec. International Conference on Machine Learning (ICML), 2018.

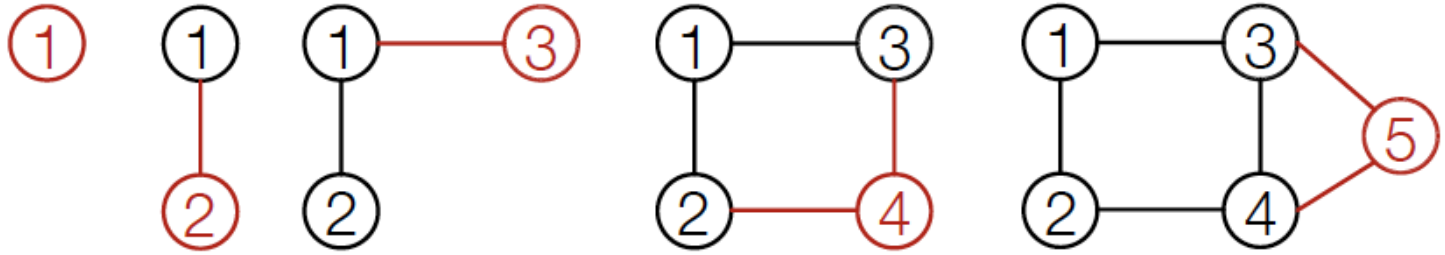
Model Graphs as Sequences

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:



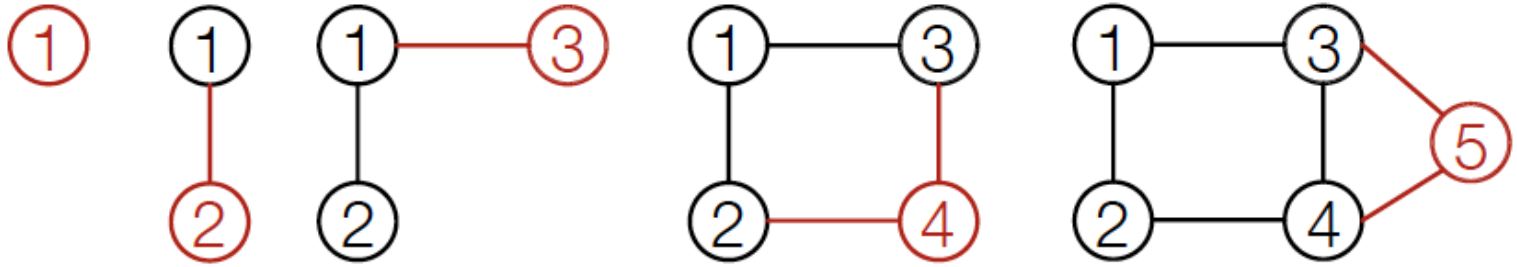
Sequence S^{π}:



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

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

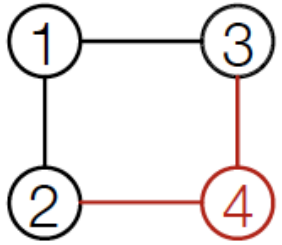


$$S^\pi = (S_1^\pi , S_2^\pi , S_3^\pi , S_4^\pi , S_5^\pi)$$

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

Model Graphs as Sequences

- The sequence S^π has two levels (S is a sequence of sequences):
  - Each Node-level step is an edge-level sequence
  - Edge-level: At each step, add a new edge



S_4^π

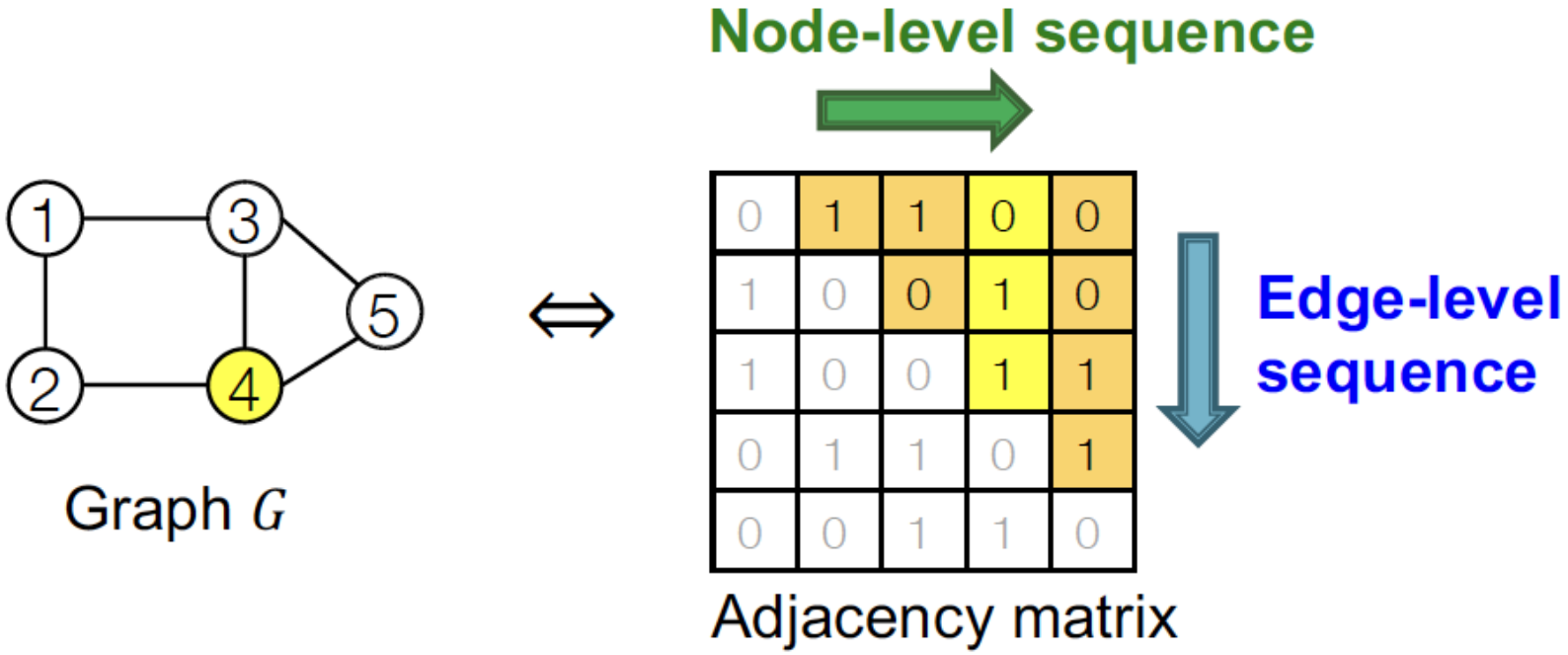
$$S_4^\pi = (\; S_{4,1}^\pi \; , \; S_{4,2}^\pi \; , \; S_{4,3}^\pi \;)$$

"Not connect 4, 1" "Connect 4, 2" "Connect 4, 3"

0 1 1

Model Graphs as Sequences

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



Model Graphs as Sequences

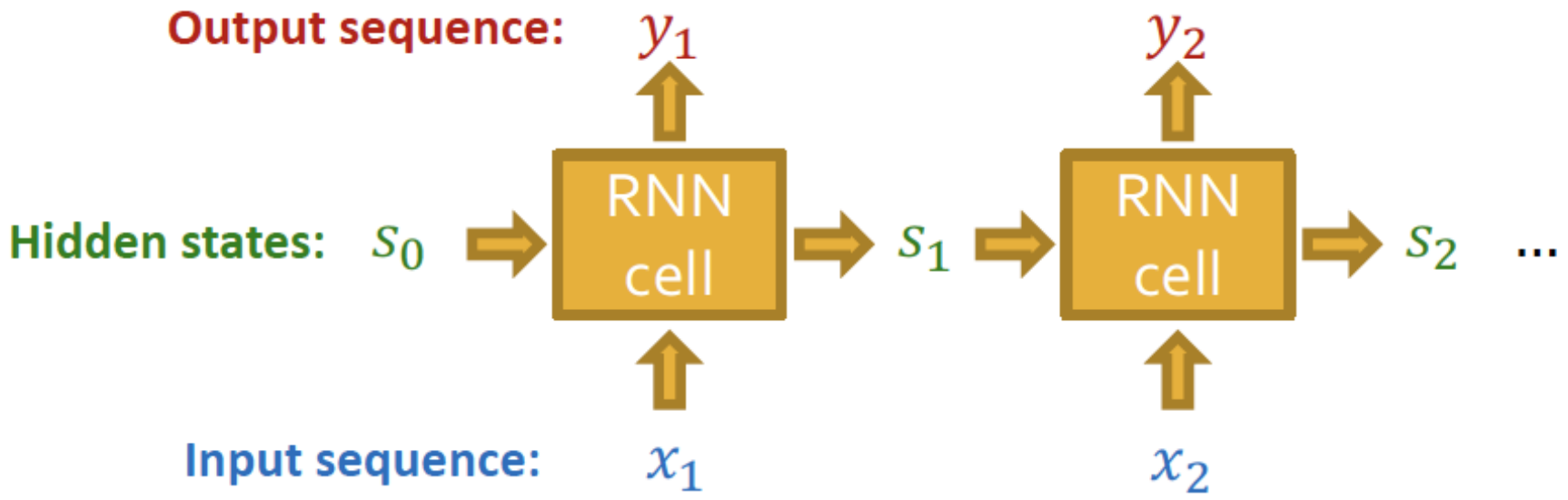
- We have transformed graph generation problem into a sequence generation problem

- Need to model two processes:
  - 1) Generate a state for a new node (Node-level sequence)
  - 2) Generate edges for the new node based on its state (Edge-level sequence)

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

Background: Recurrent NNs

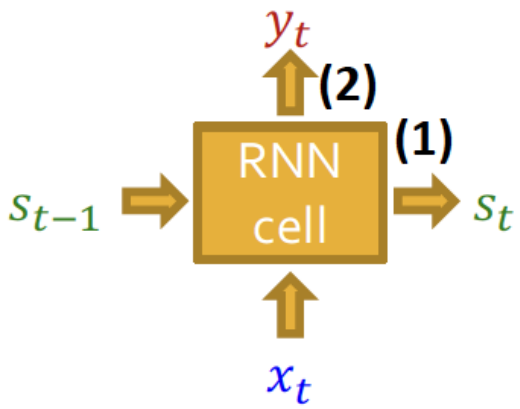
- RNNs are designed for sequential data
  - RNN sequentially takes input sequence to update its hidden states
  - The hidden states summarize all the information input to RNN
  - The update is conducted via RNN cells



Background: Recurrent NNs

- s_t: State of RNN after step t
- 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.

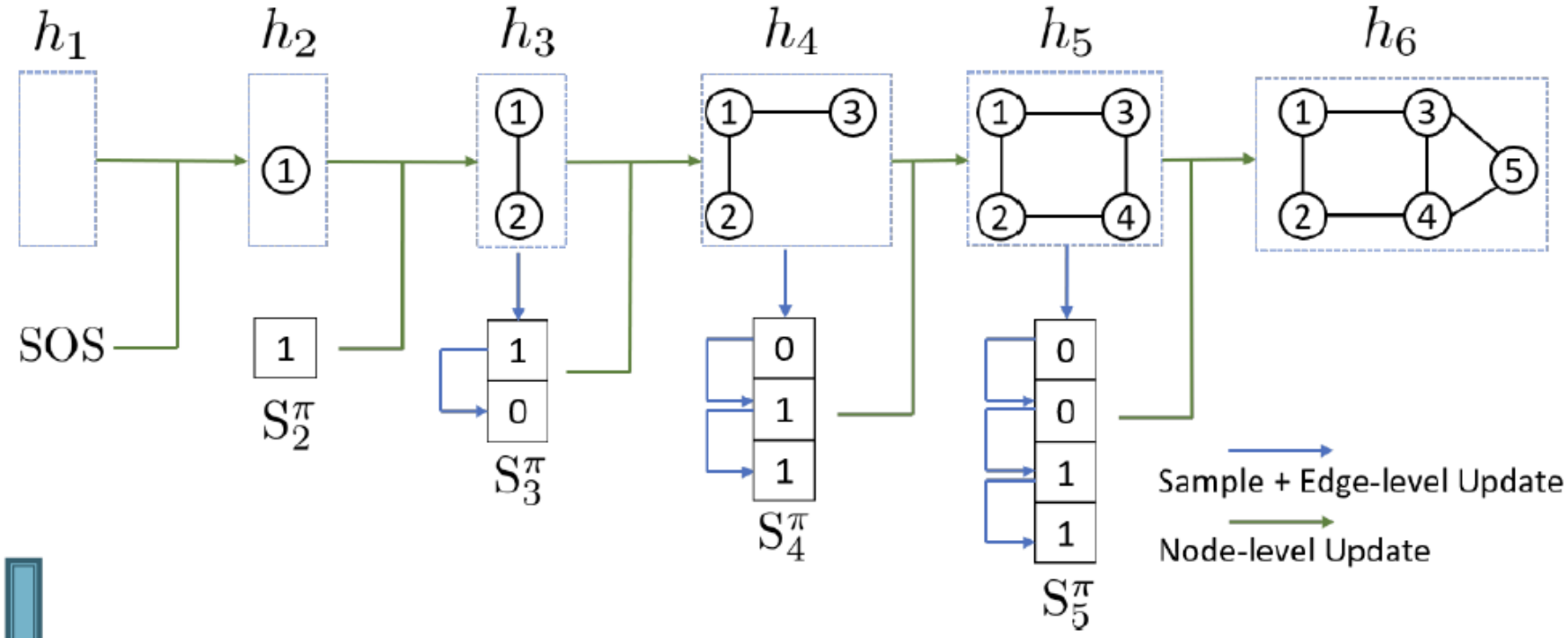
GraphRNN: Two Levels of RNN

- 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

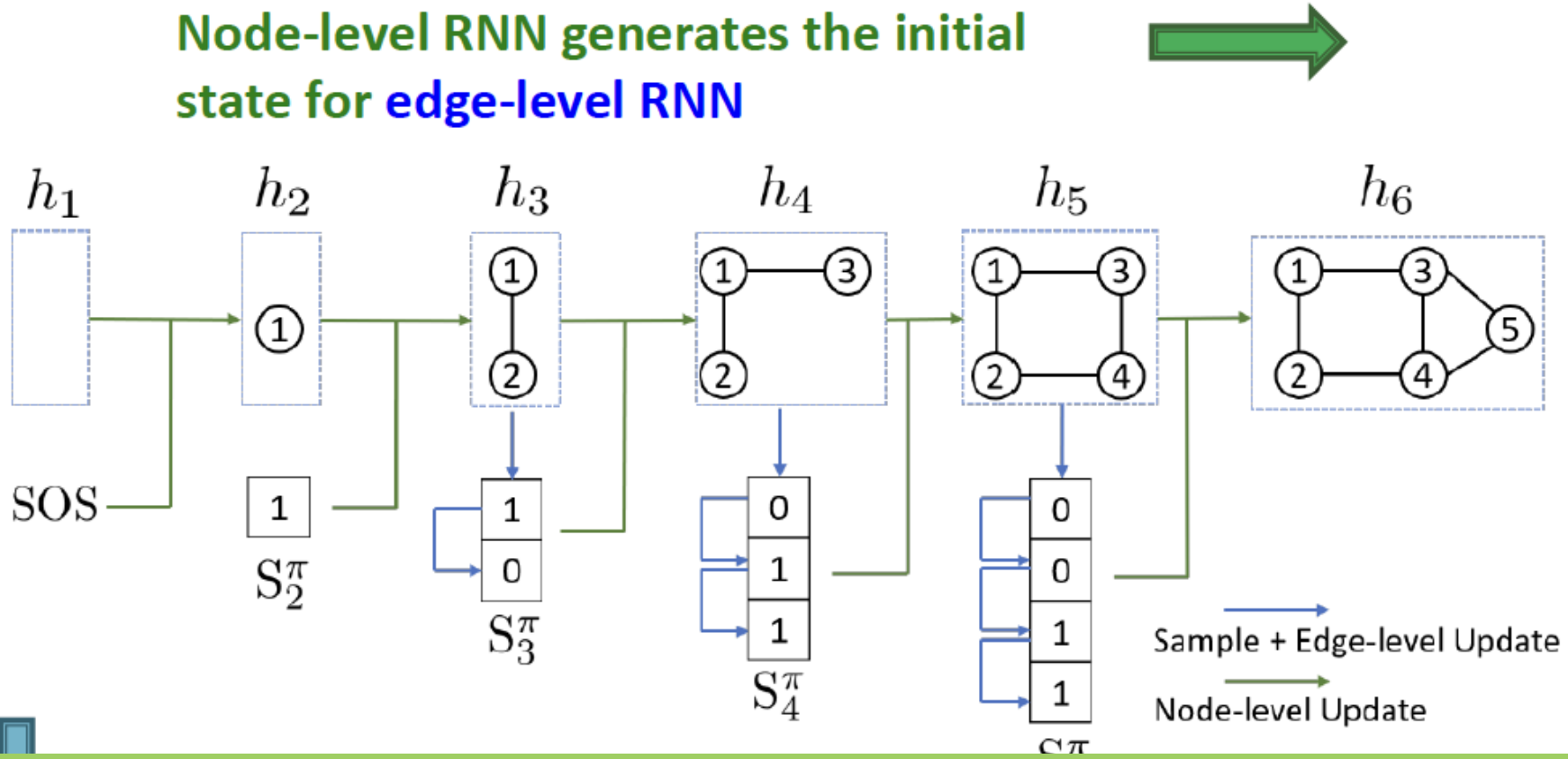
GraphRNN: Two Levels of RNN

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

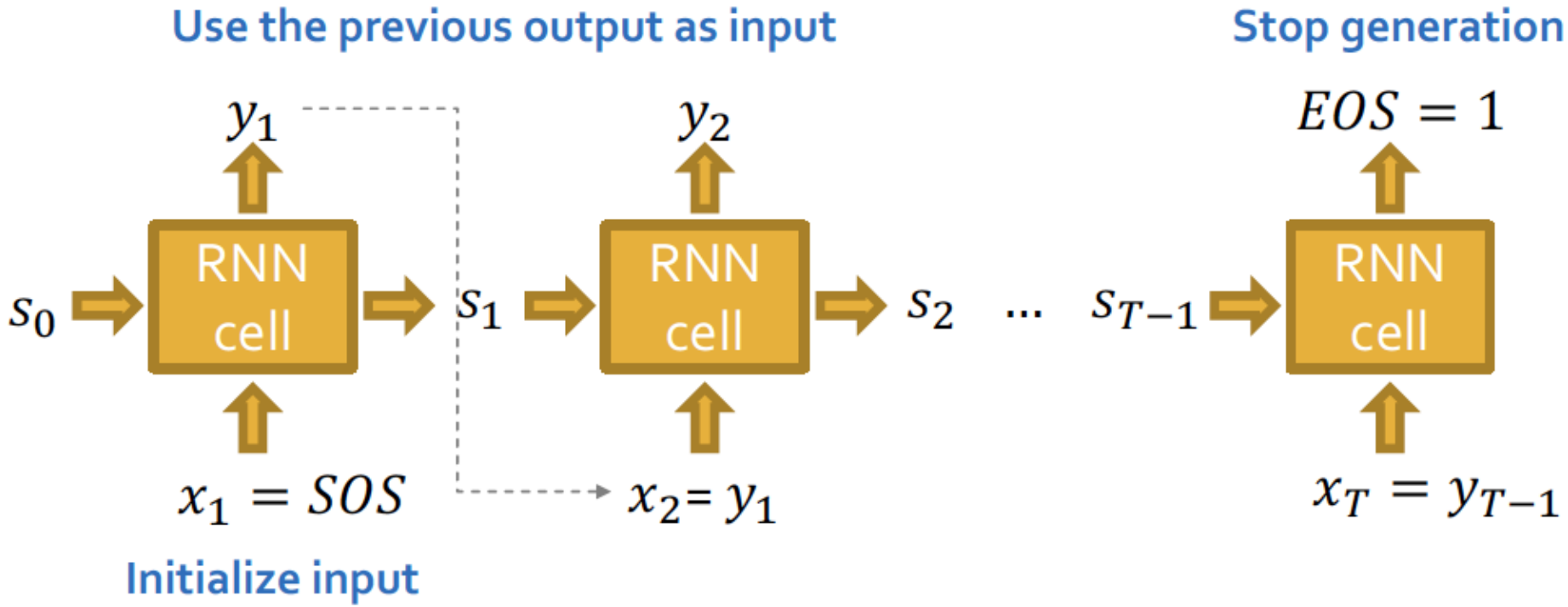


Next: How to generate a sequence with RNN?

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

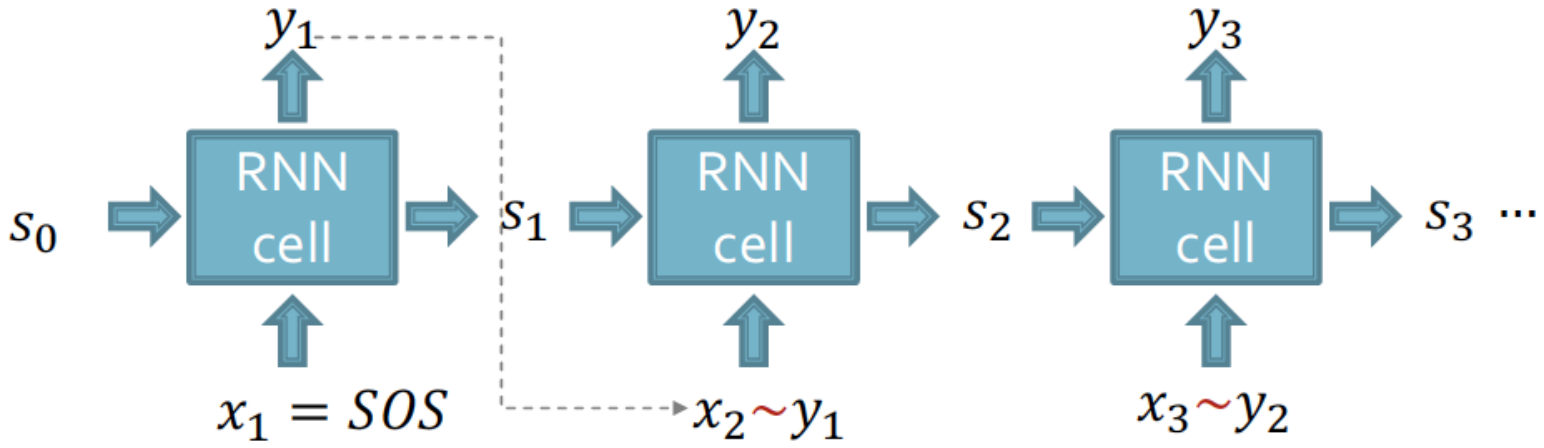
RNN for Sequence Generation



This is good, but this model is deterministic

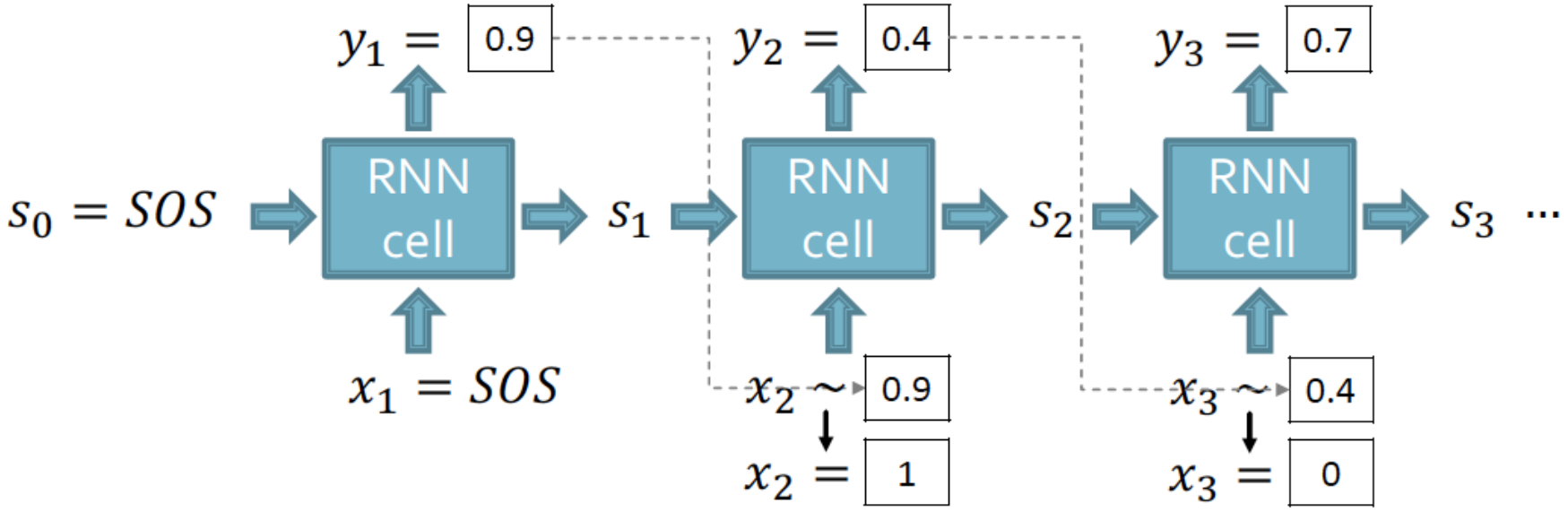
Towards Edge-Level RNN

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



Towards Edge-Level RNN

- Suppose we already have trained the edge-level RNN
  - y_t is a scalar, following a Bernoulli distribution
  - $\boxed{p}$ means value 1 has prob. p, value 0 has prob. $1 - p$

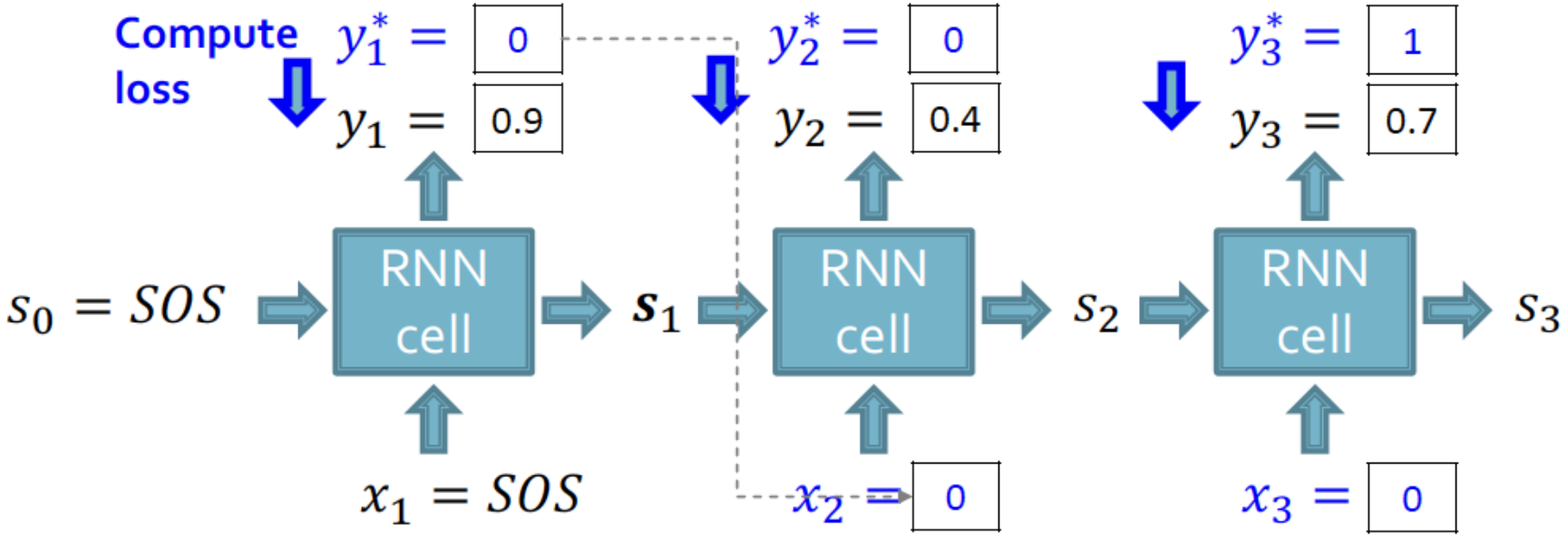


  - How do we use training data $\boldsymbol{x_1, x_2, \ldots, x_n}$?

Edge-Level RNN at Training Time

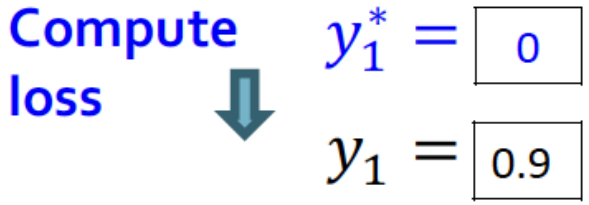
Training the model:

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



- If $y_1^* = 1$, we minimize $-\log(y_1)$, making y- 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!

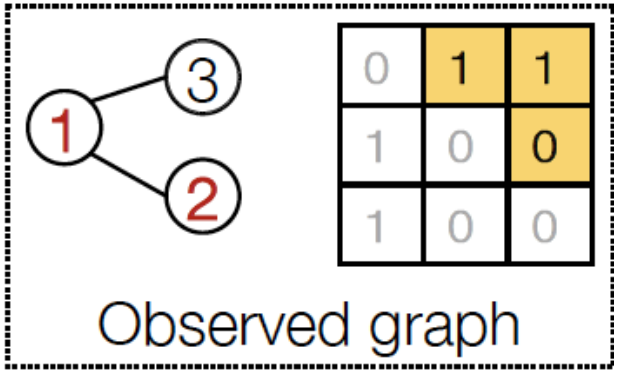
Putting Things Together

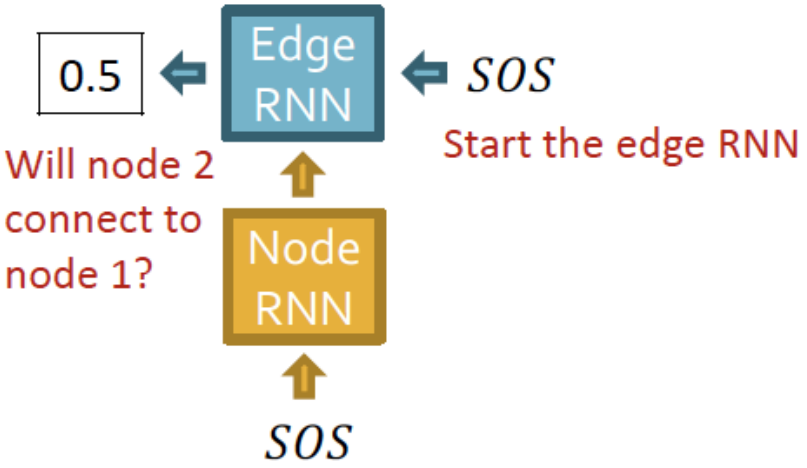
Our Plan:

- (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.

Putting Things Together: Training

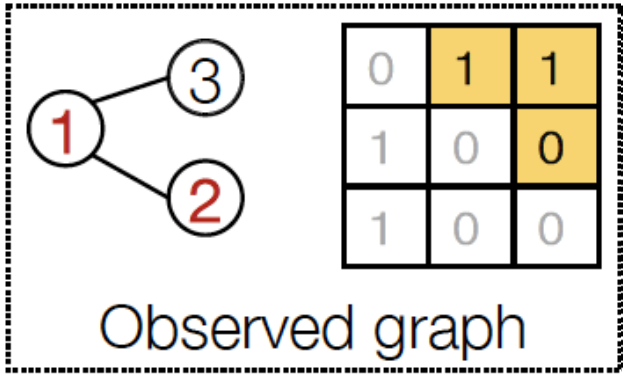
Edge RNN predicts how
Node 2 connects to **Node 1**

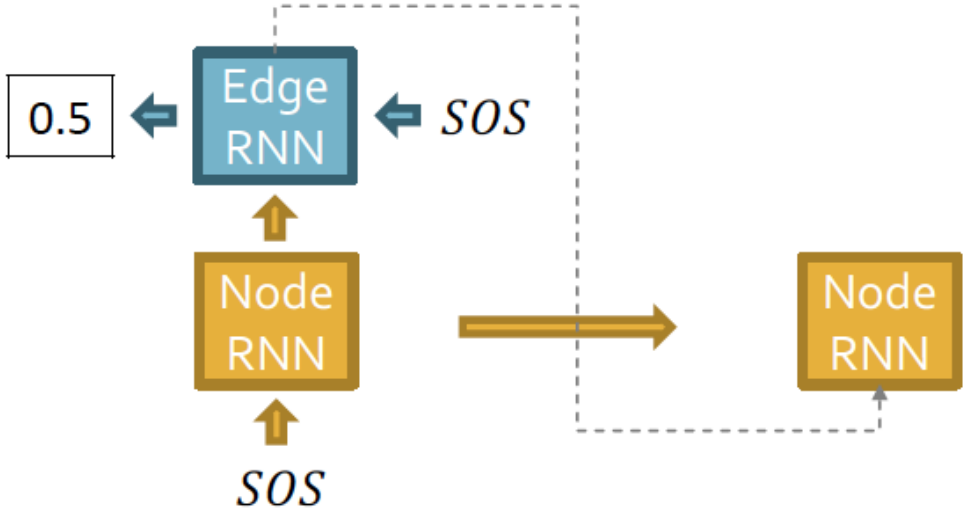




Putting Things Together: Training

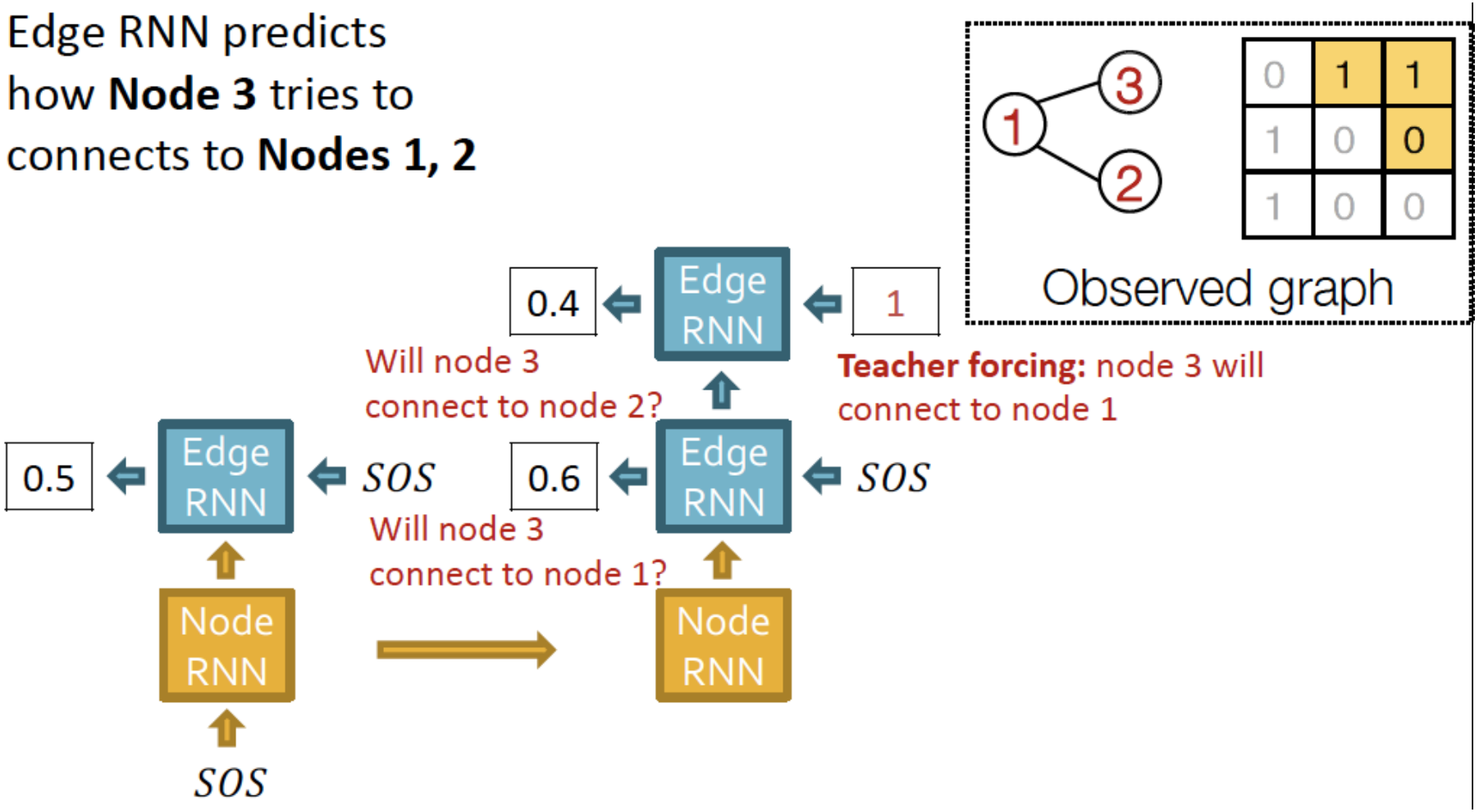
Update Node RNN using
Edge RNN's hidden state





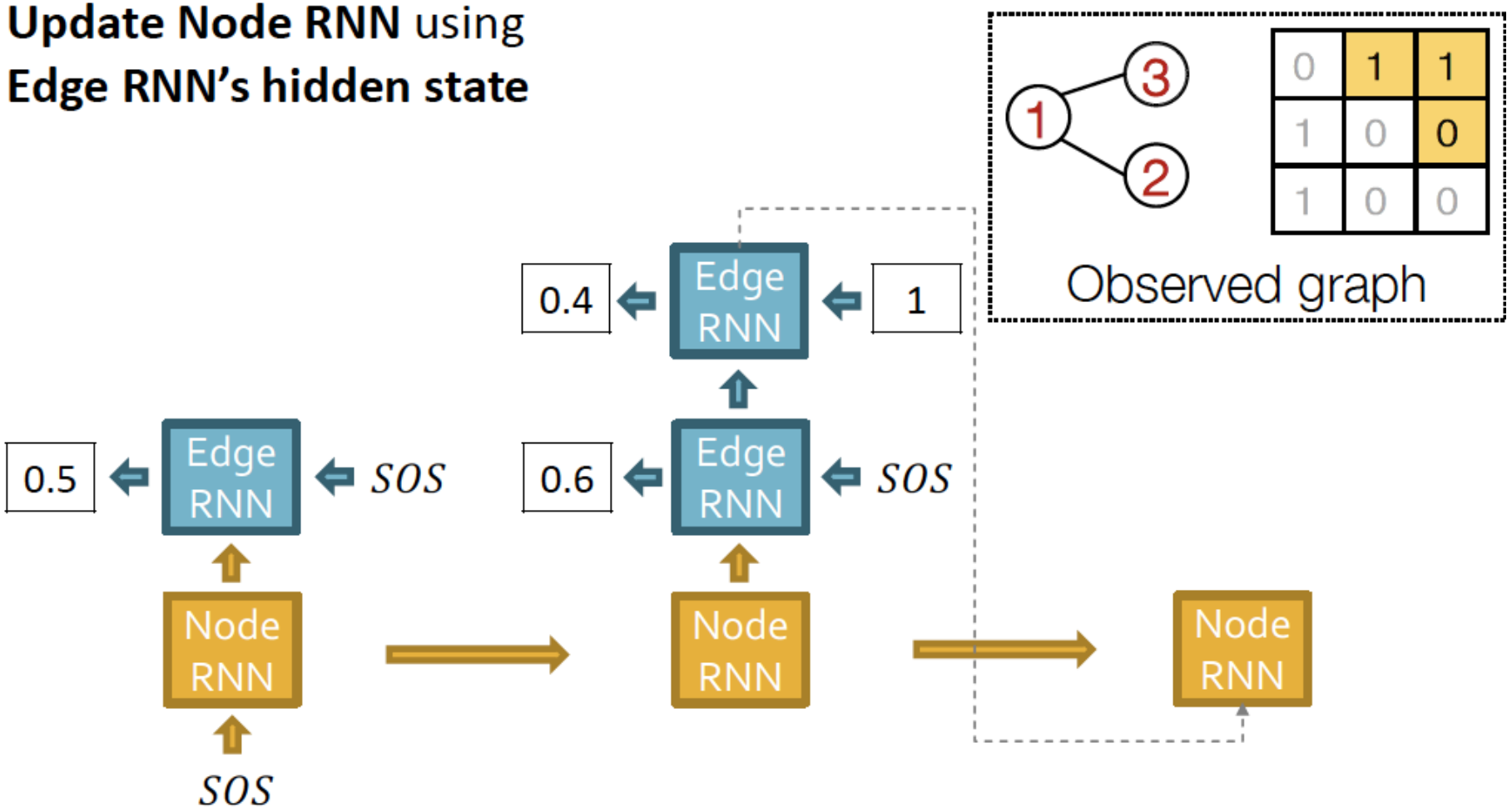
Putting Things Together: Training

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



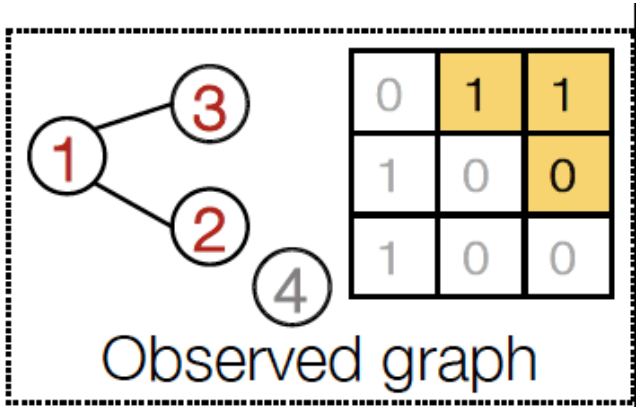
Putting Things Together: Training

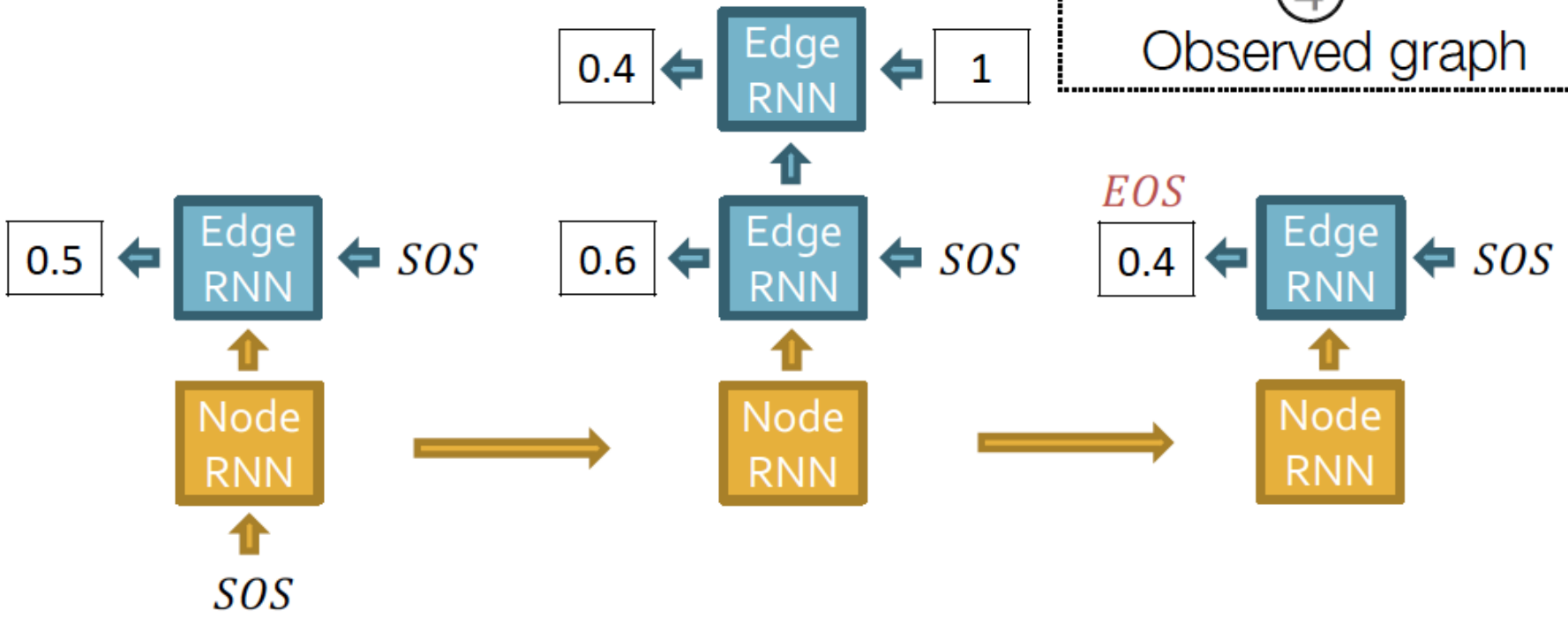
Update Node RNN using
Edge RNN's hidden state



Putting Things Together: Training

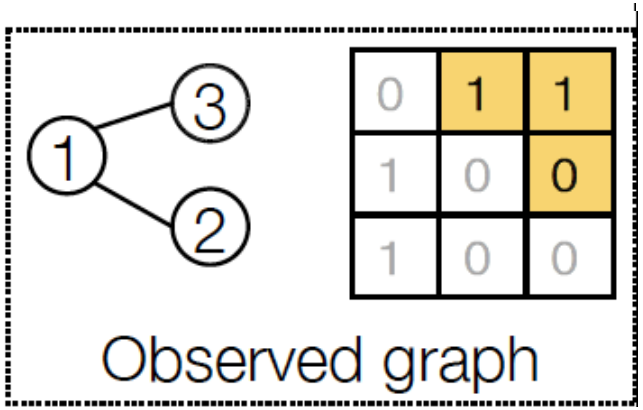
Stop generation since we know node 4 won't connect to any nodes

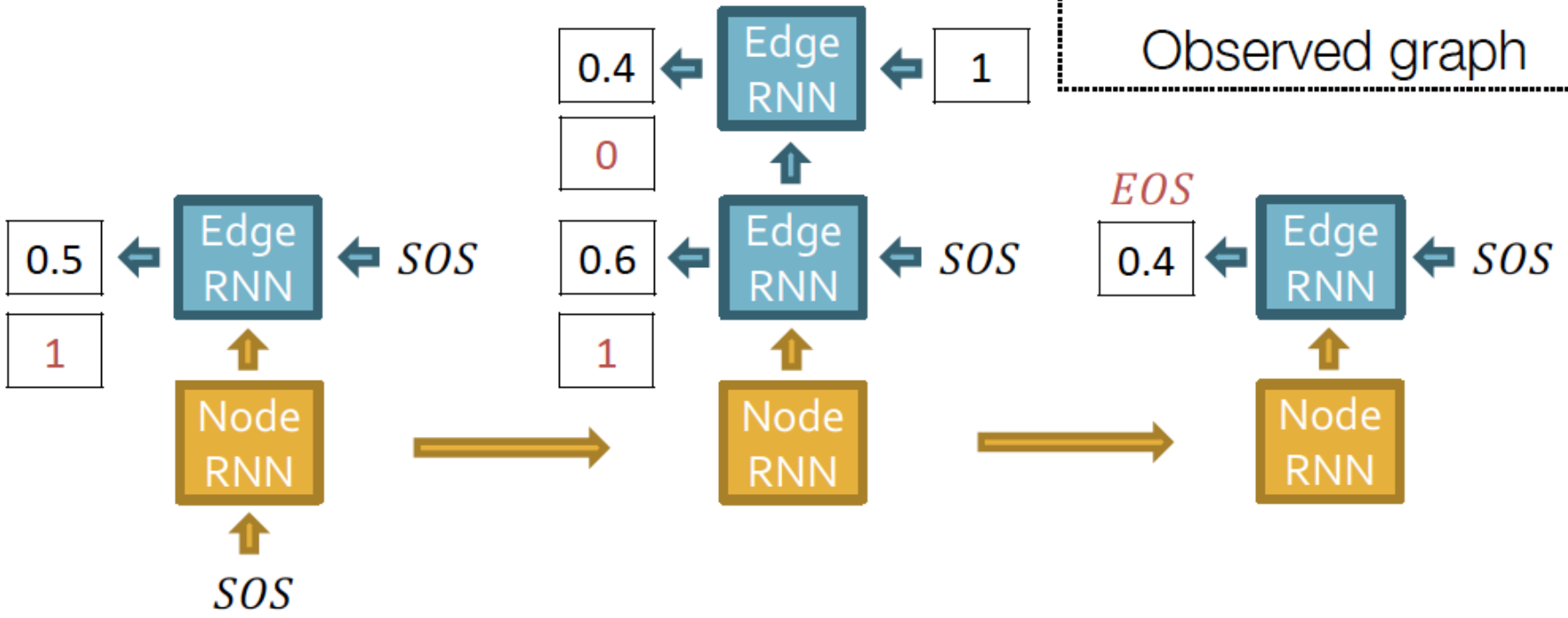




Putting Things Together: Training

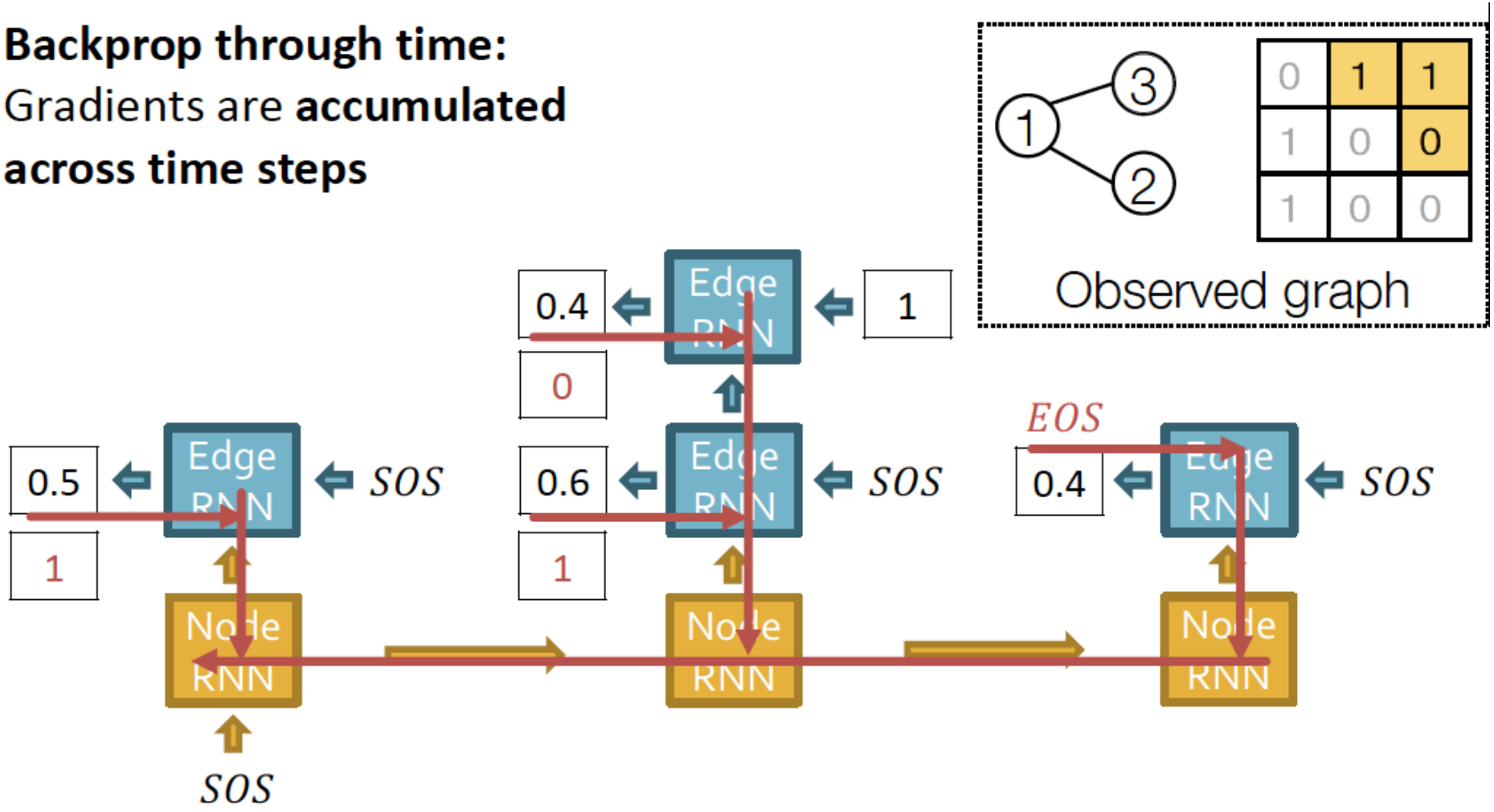
For each prediction, we **get supervision from the ground truth**





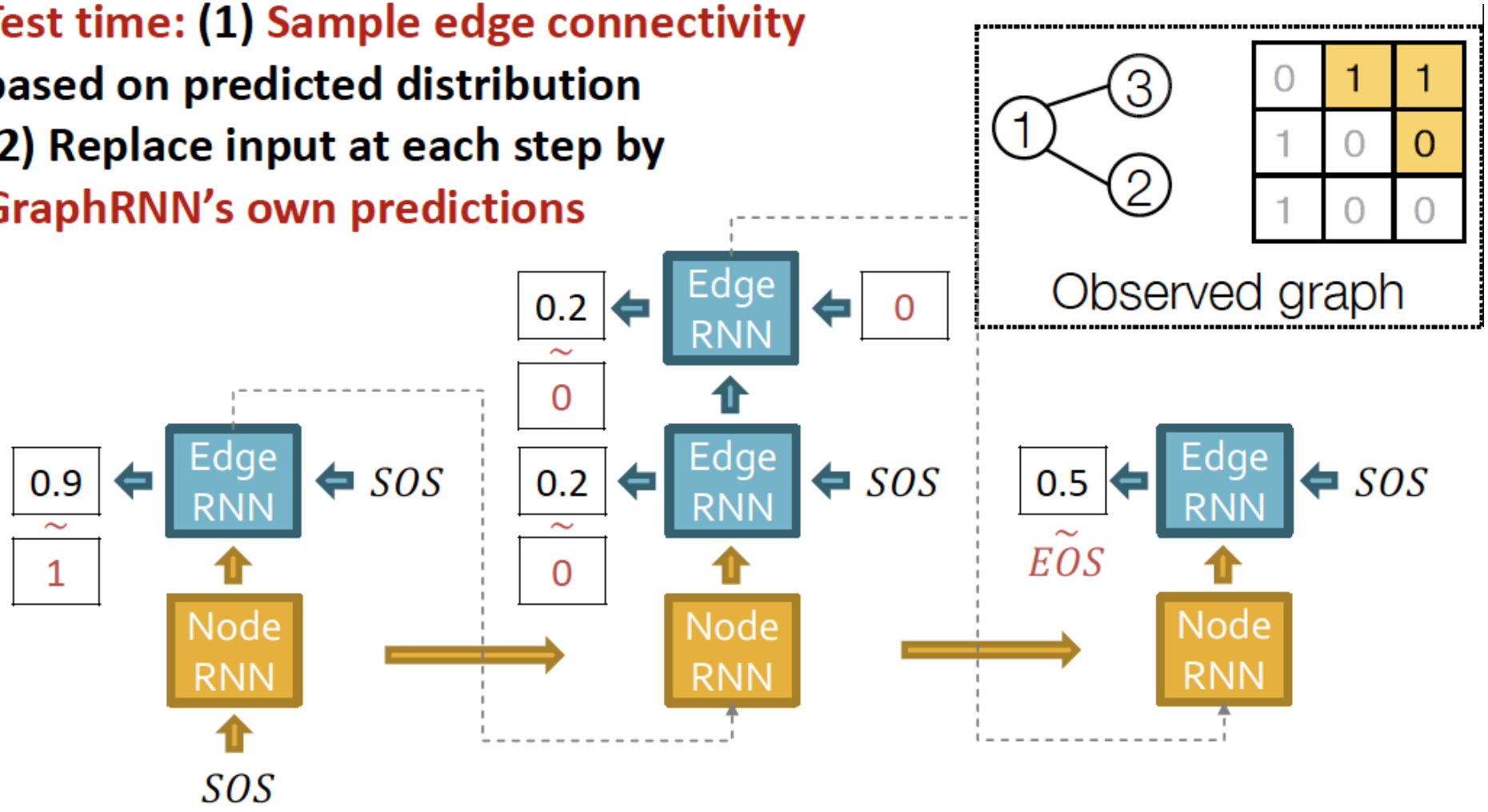
Putting Things Together: Training

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



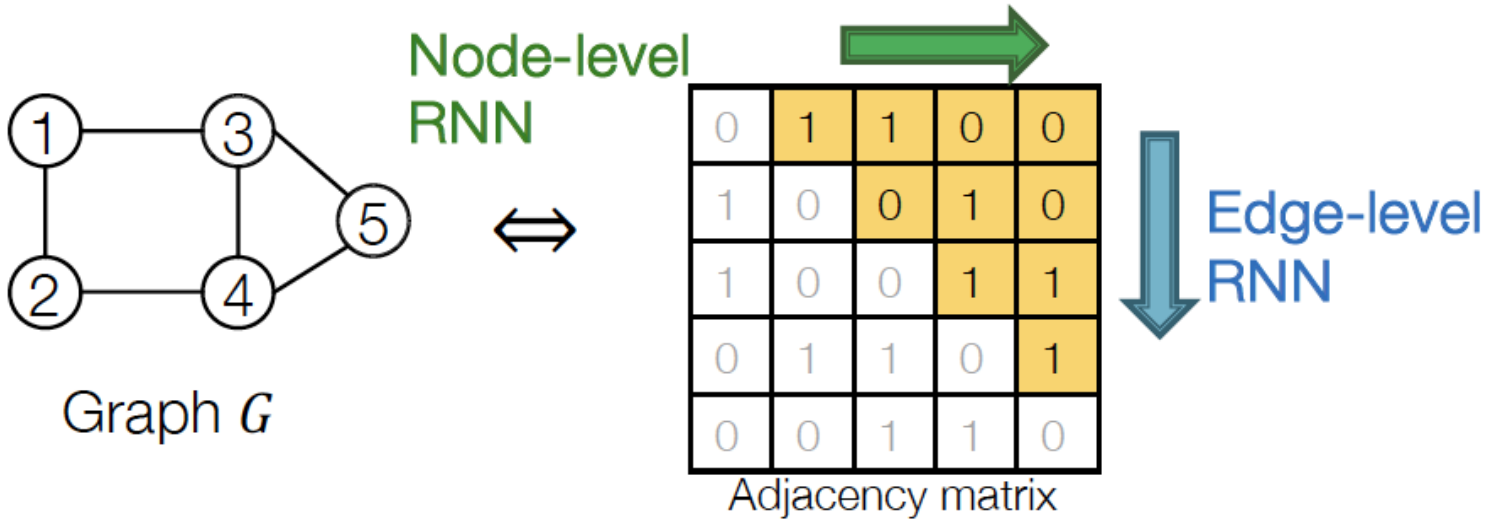
Putting Things Together: Test

Test time: (1) Sample edge connectivity based on predicted distribution (2) Replace input at each step by GraphRNN's own predictions



GraphGNN: Two Levels of RNN

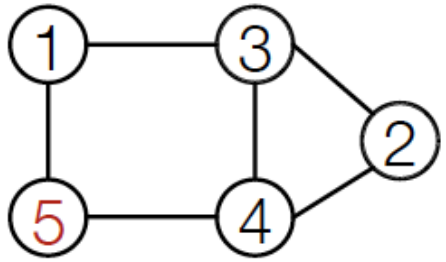
- Quick Summary of GraphRNN:
  - Generate a graph by generating a two-level sequence
  - Use RNN to generate the sequences
- Next: Making GraphRNN tractable, proper evaluation



Scaling Up and Evaluating Graph Generation

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

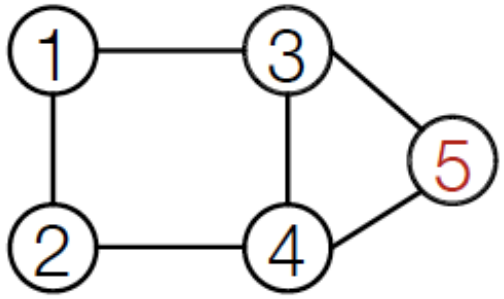
"Recipe" to generate the left graph:

- 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

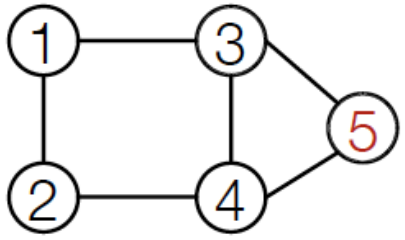
"Recipe" to generate the left graph:

- 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



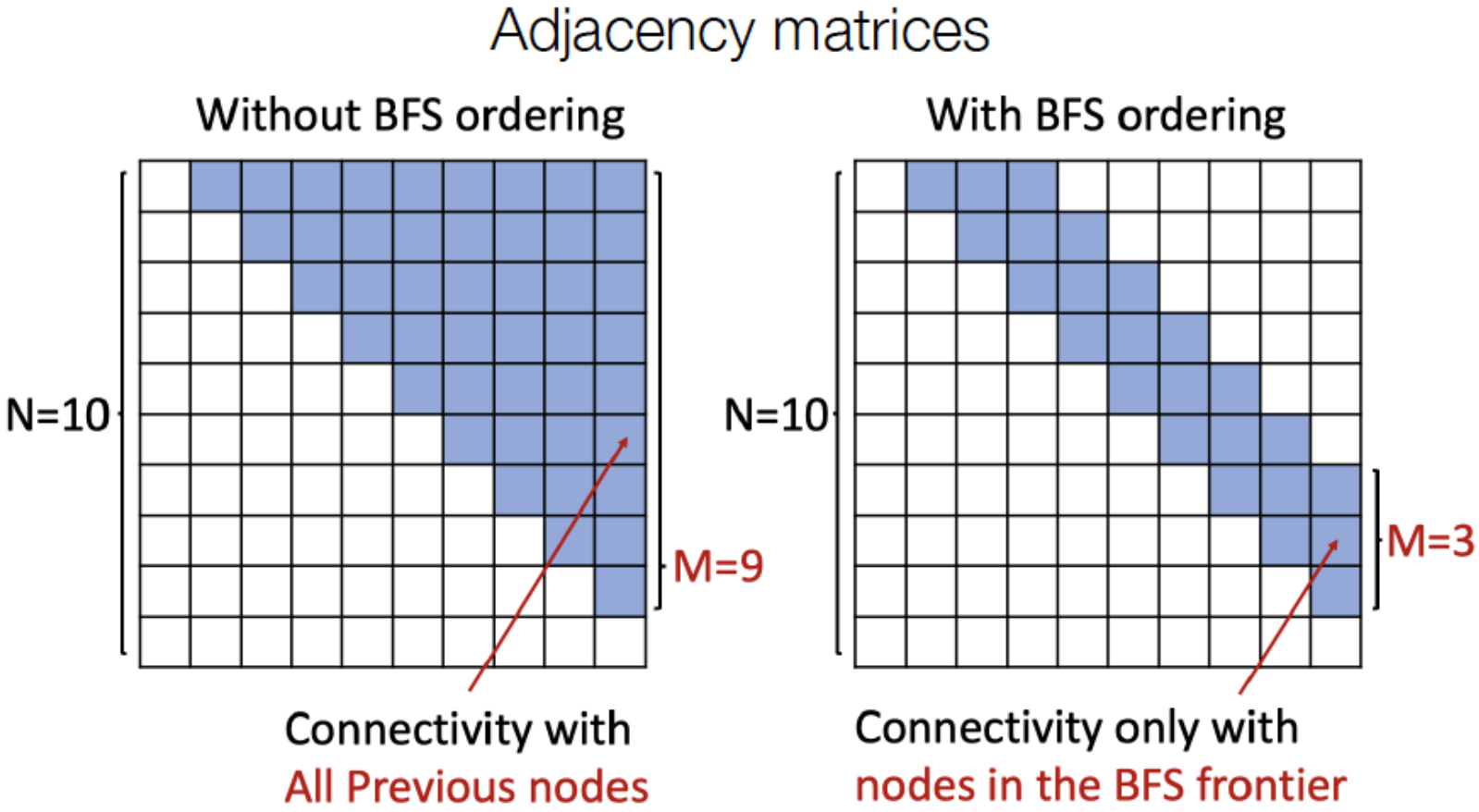
BFS ordering

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

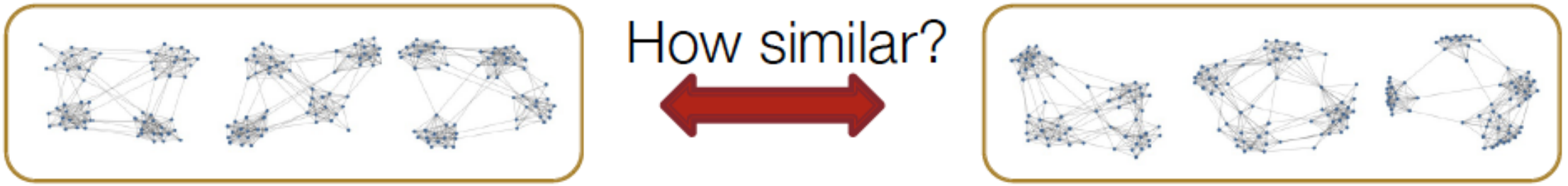
Solution: Tractability via BFS

- BFS reduces the number of steps for edge generation



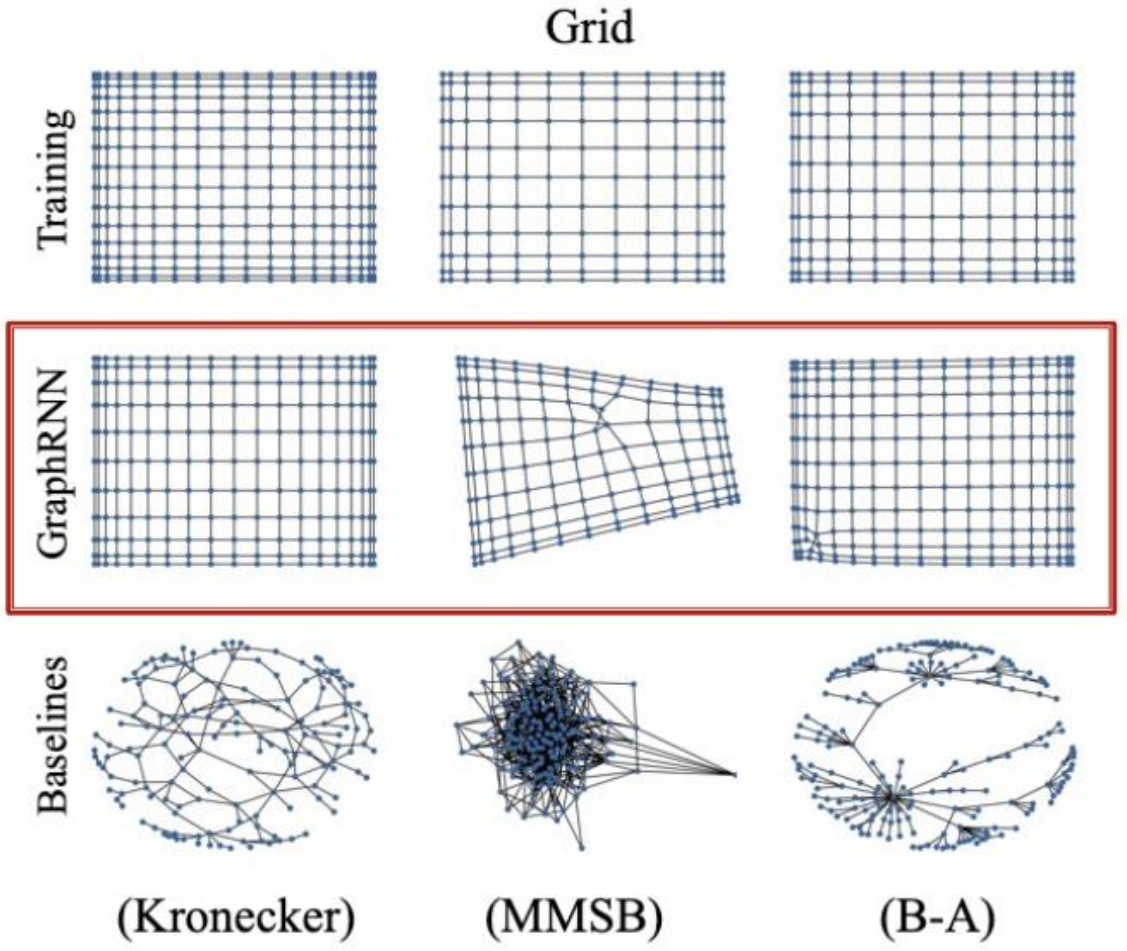
Evaluating Generated Graphs

- Task: Compare two sets of graphs

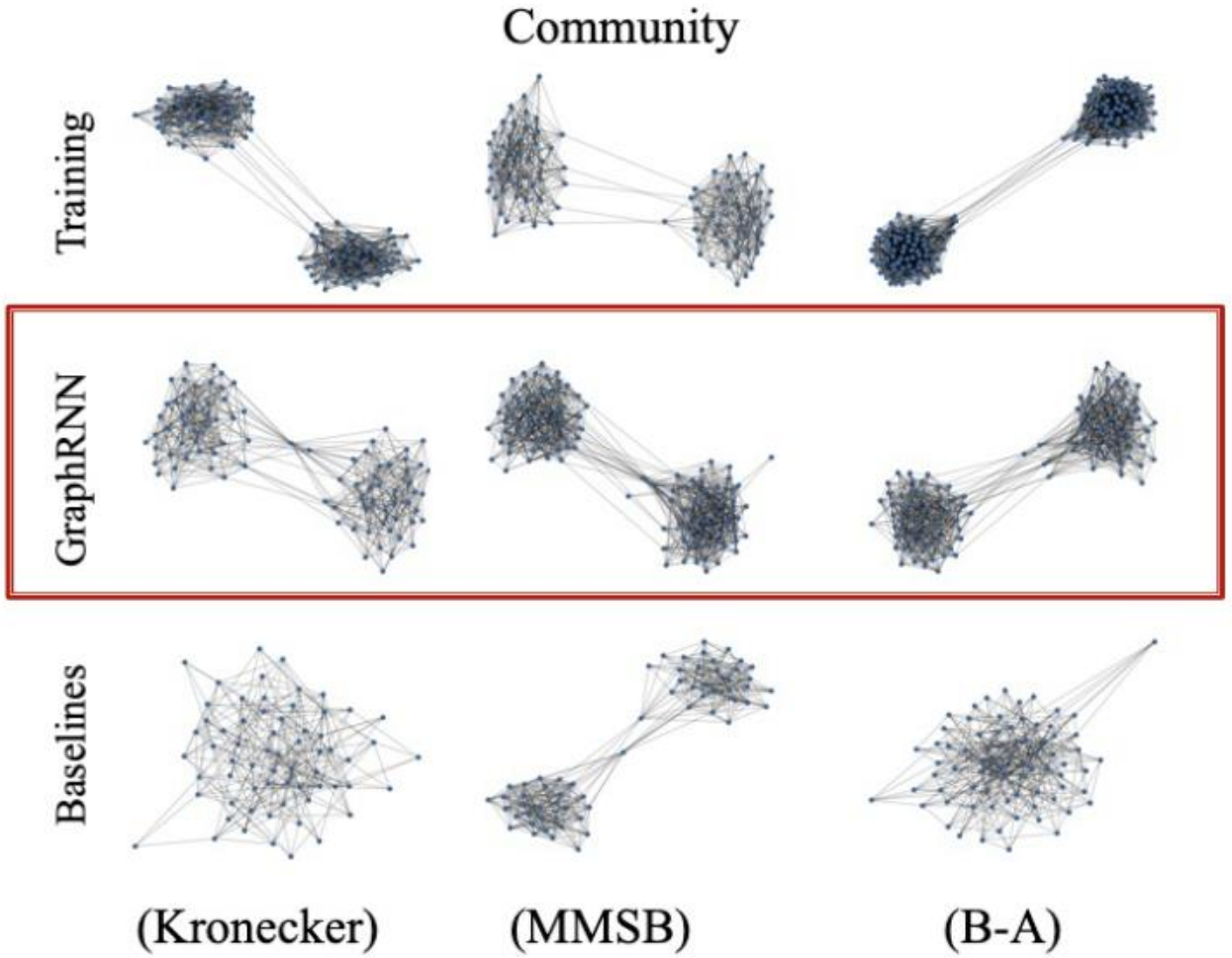


- Goal: Define similarity metrics for graphs
- Solution
  - (1) Visual similarity
  - (2) Graph statistics similarity

(1) Visual Similarity



(1) Visual Similarity

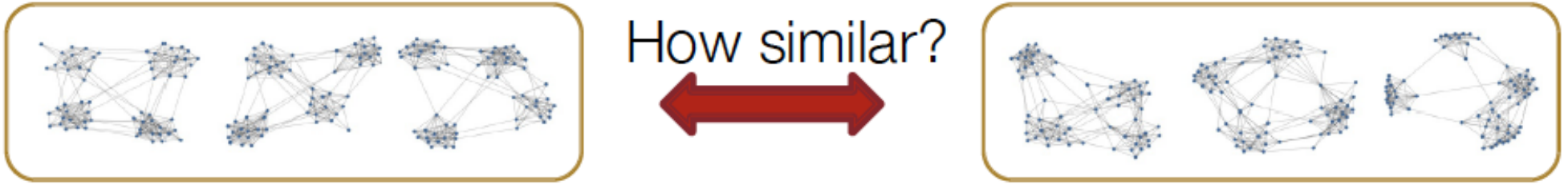


(2) Graph Statistics Similarity

- Can we do more rigorous comparison?
- Issue: Direct comparison between two graphs is hard (isomorphism test 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 distributions x_i and y_j, that minimizes the overall cost. d_{ij} is the ground distance between x_i and y_j.

(2) Graph Statistics Similarity

- Step 2: Maximum Mean Discrepancy (MMD)
  - Idea of representing distances between distributions as distances between mean embeddings of feature

$$\mathrm{MMD}^2(p||q) = \mathbb{E}_{x,y\sim p}[k(x,y)] + \mathbb{E}_{x,y\sim q}[k(x,y)] - 2\mathbb{E}_{x\sim p,y\sim q}[k(x,y)].$$

{2,3,5} High MMD {8,2,3} {2,3,5} Low MMD {5,4,2}

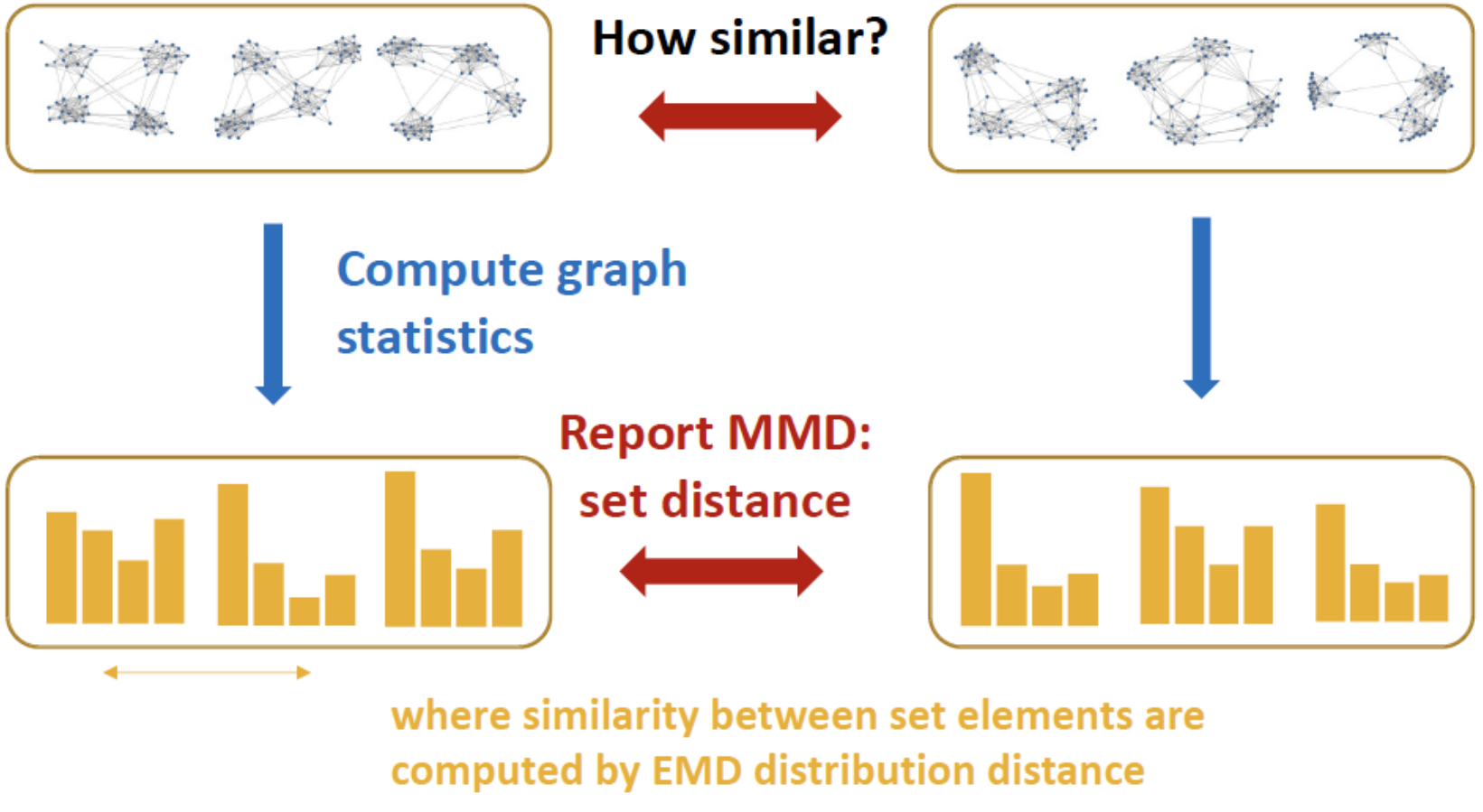
Similarity between set elements: L2 distance
(Each element is a scalar)

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

Similarity between set elements: EMD
(Each element is a distribution)

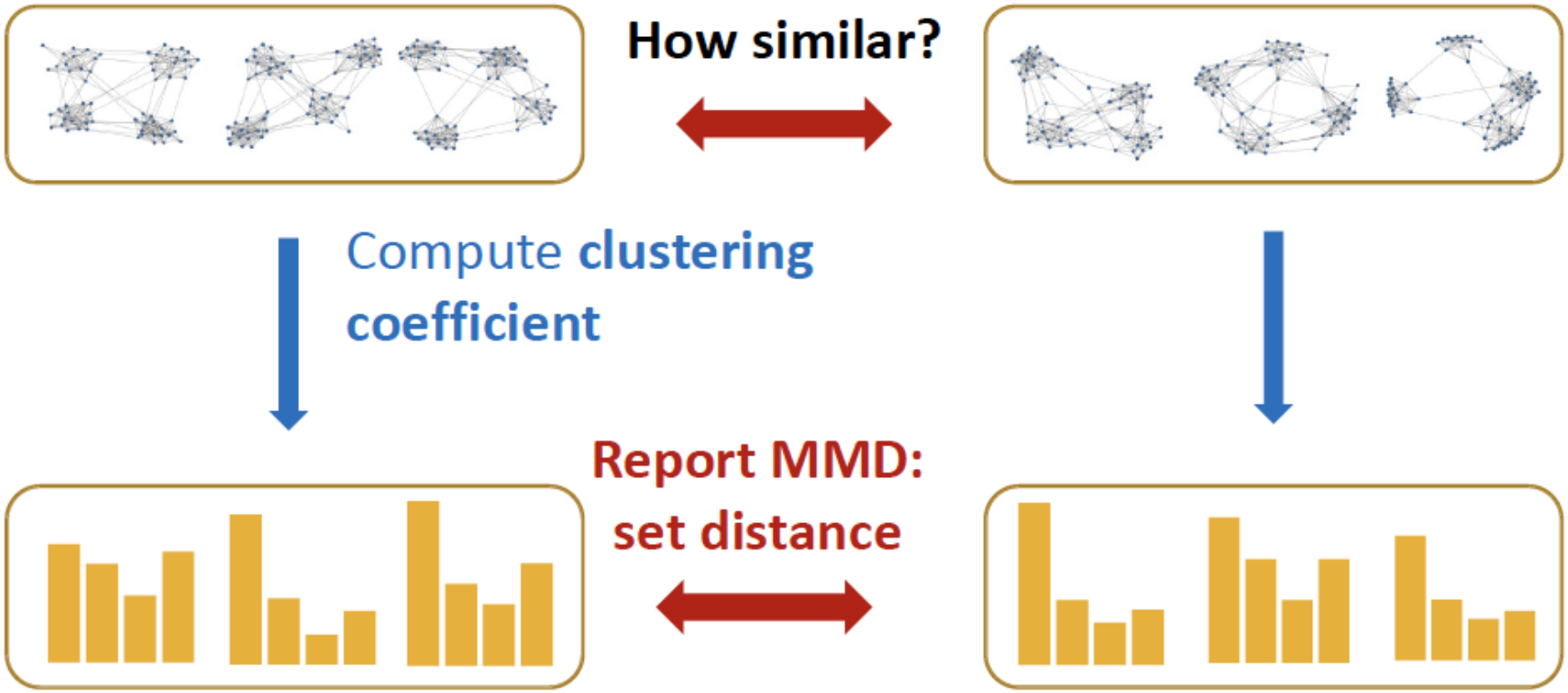
(2) Graph Statistics Similarity

- Putting things together



(2) Graph Statistics Similarity

- Exam

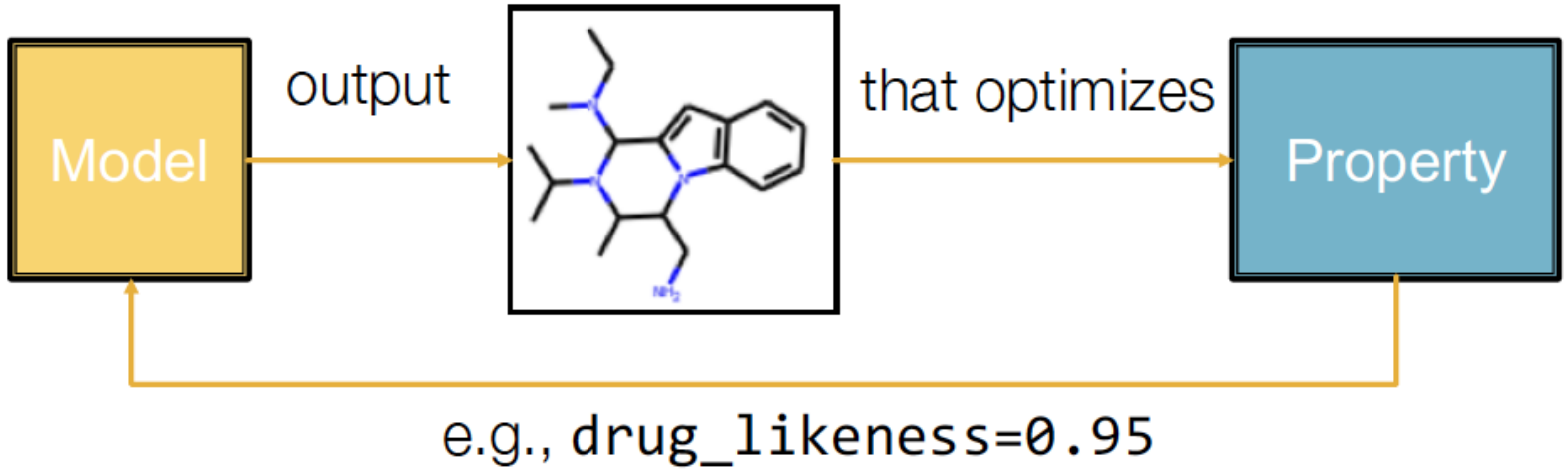


	Erdos-Renyi	Kronecker	GraphRNN
MMD score	1.24	1.67	0.002

Applications of Deep Graph Generative Models to Molecule Generation

Application: Drug Discovery

- 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)

Including a "Black-box" to Graph Generation:

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

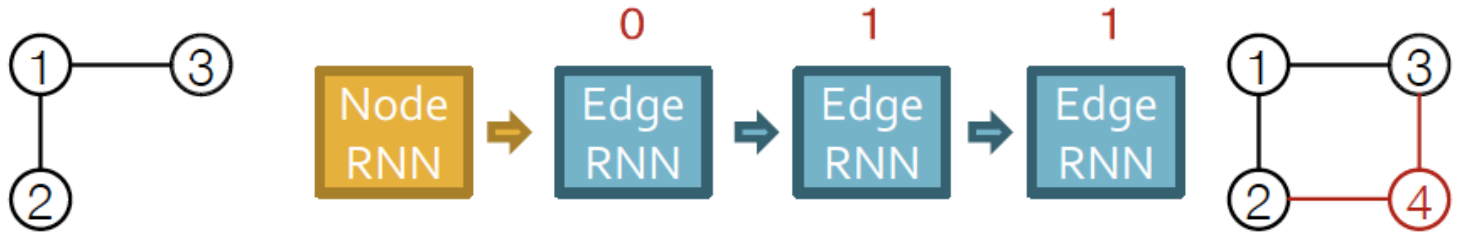
GCPN versus GraphRNN

- 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

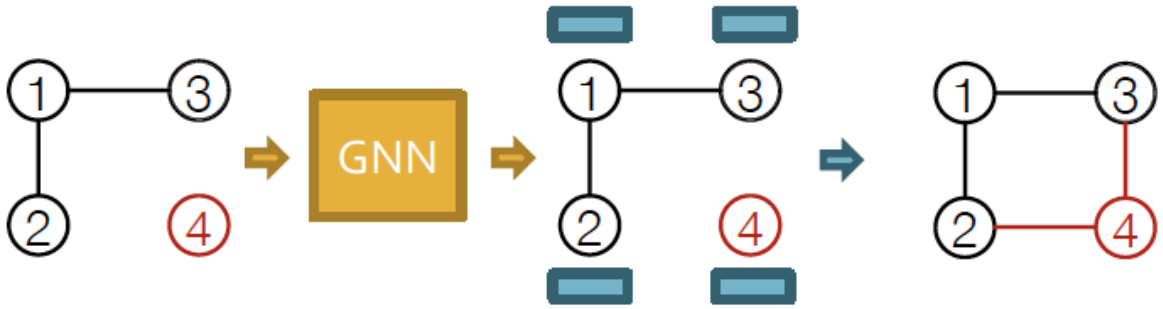
GCPN versus GraphRNN

- **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**



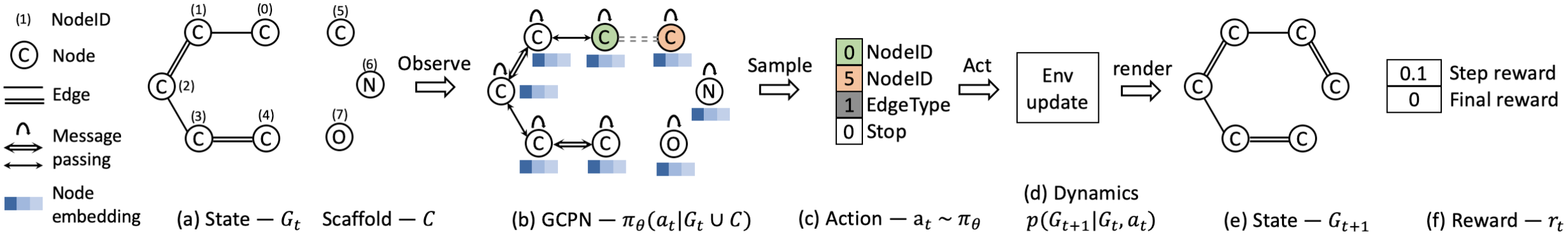
Node embeddings

Predict potential links using node embeddings

Recall the link prediction head:

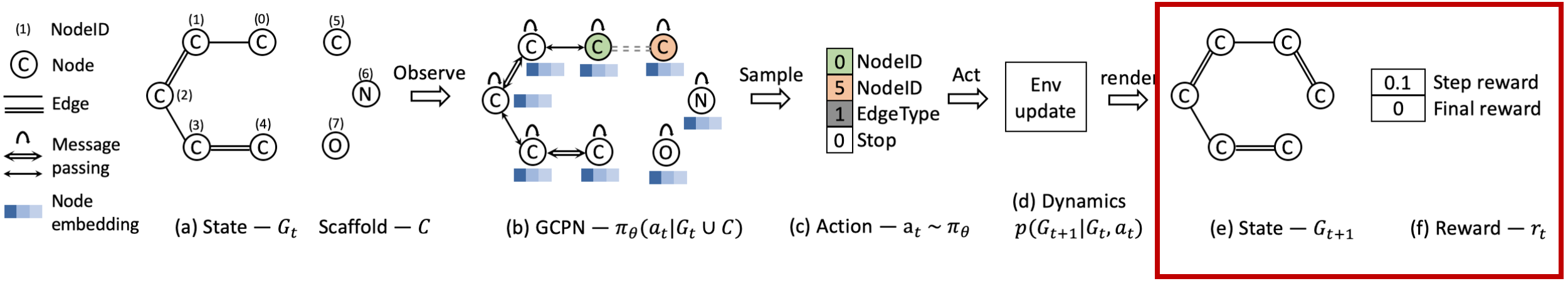
$\text{Head}_{edge}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}) :=$
$\text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$

Overview of GCPN



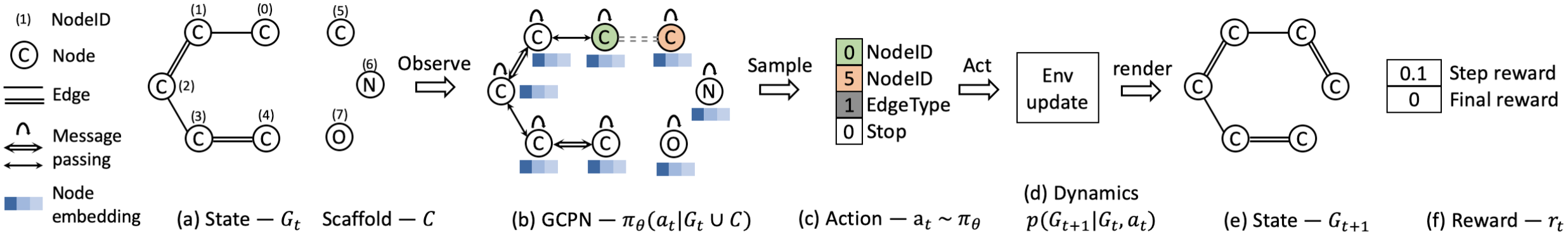
- (a) Insert nodes
- (b,c) Use GNN to predict which nodes to connect
- (d) Take an action (check chemical validity)
- (e, f) Compute reward

How do We Set the Rewards?



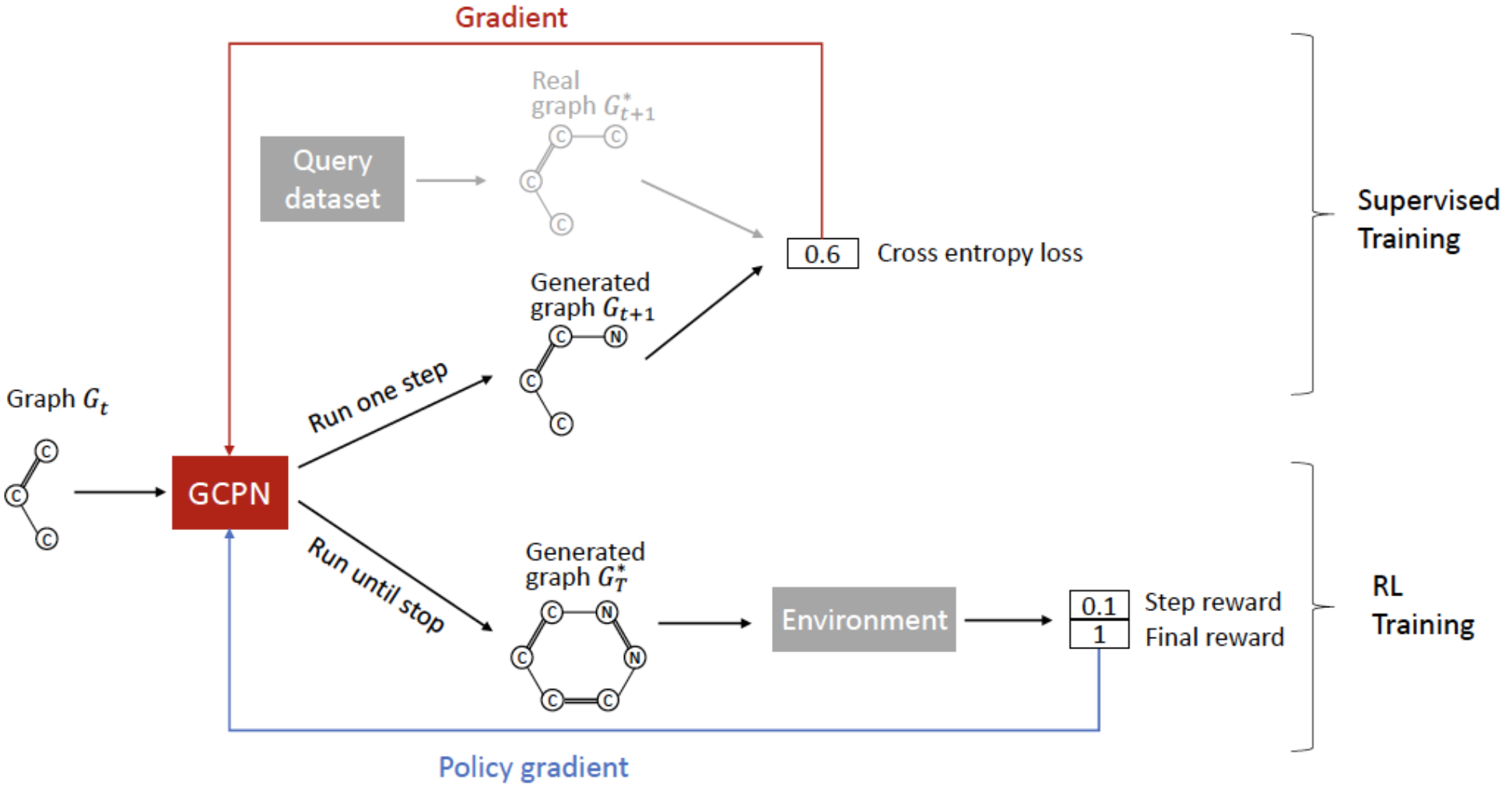
- Step reward: Learn to take valid action
  - At each step, assign small positive reward for valid action
- Final reward: Optimize desired properties
  - At the end, assign positive reward for high desired property
- Reward = Final reward + Step reward

How do We Train?



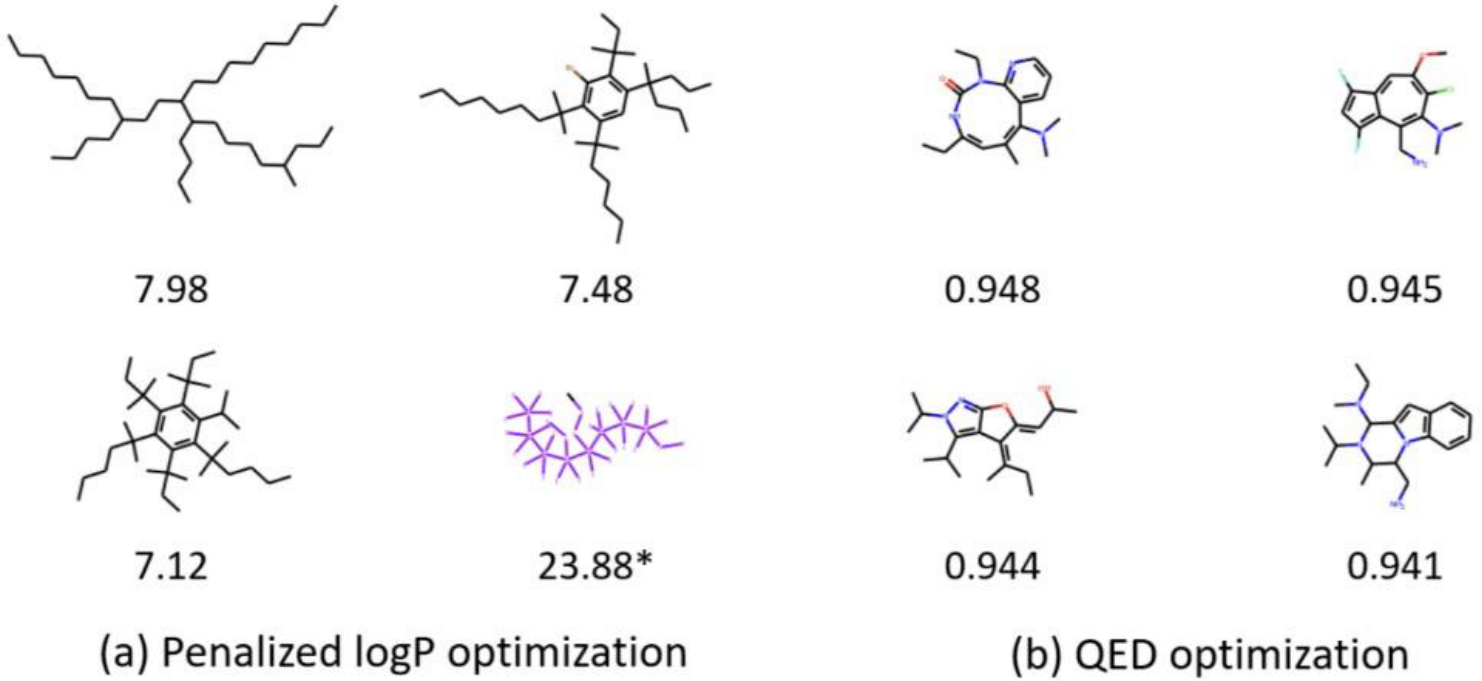
- Two parts:
- (1) Supervised training: Train policy by imitating the action given by real observed graphs. Use gradient.
  - We have covered this idea in GraphRNN
- (2) RL training: Train policy to optimize rewards. Use standard policy gradient algorithm.

Training GCPN



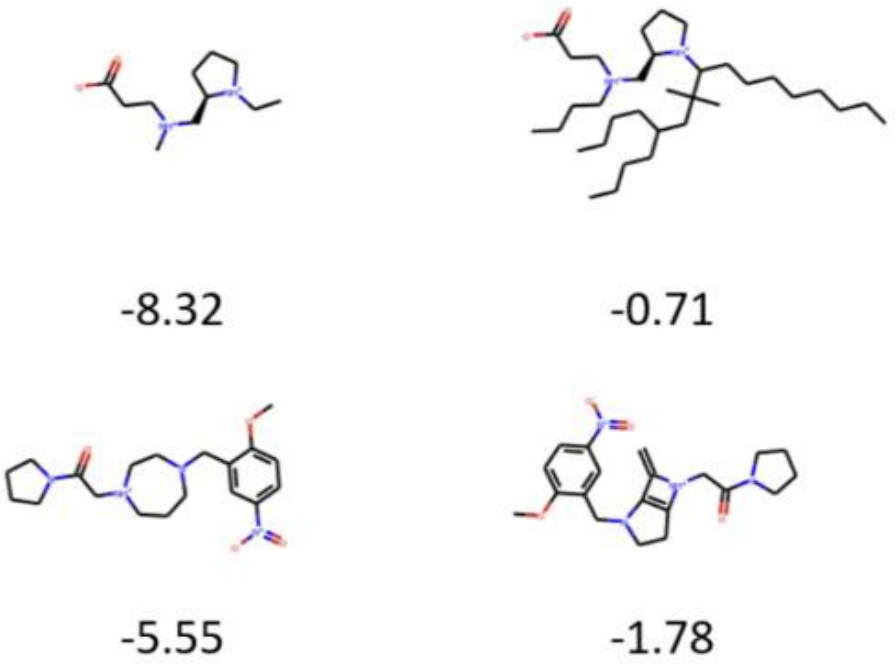
Qualitative Results

- Visualization of GCPN graphs:
- Property optimization Generate molecules with high specified property score



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

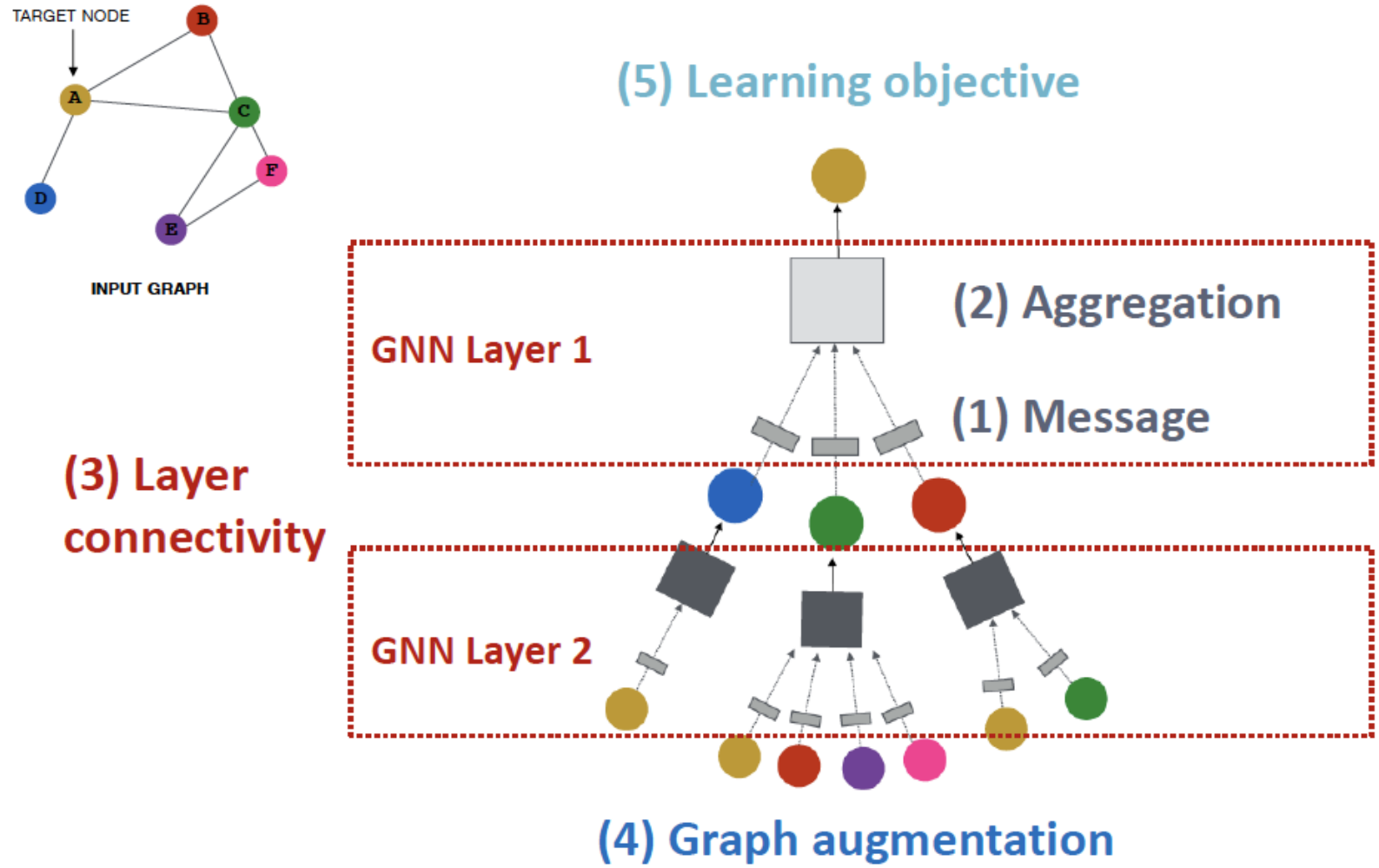
- Complex graphs can be successfully generated via sequential generation using deep learning

- 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

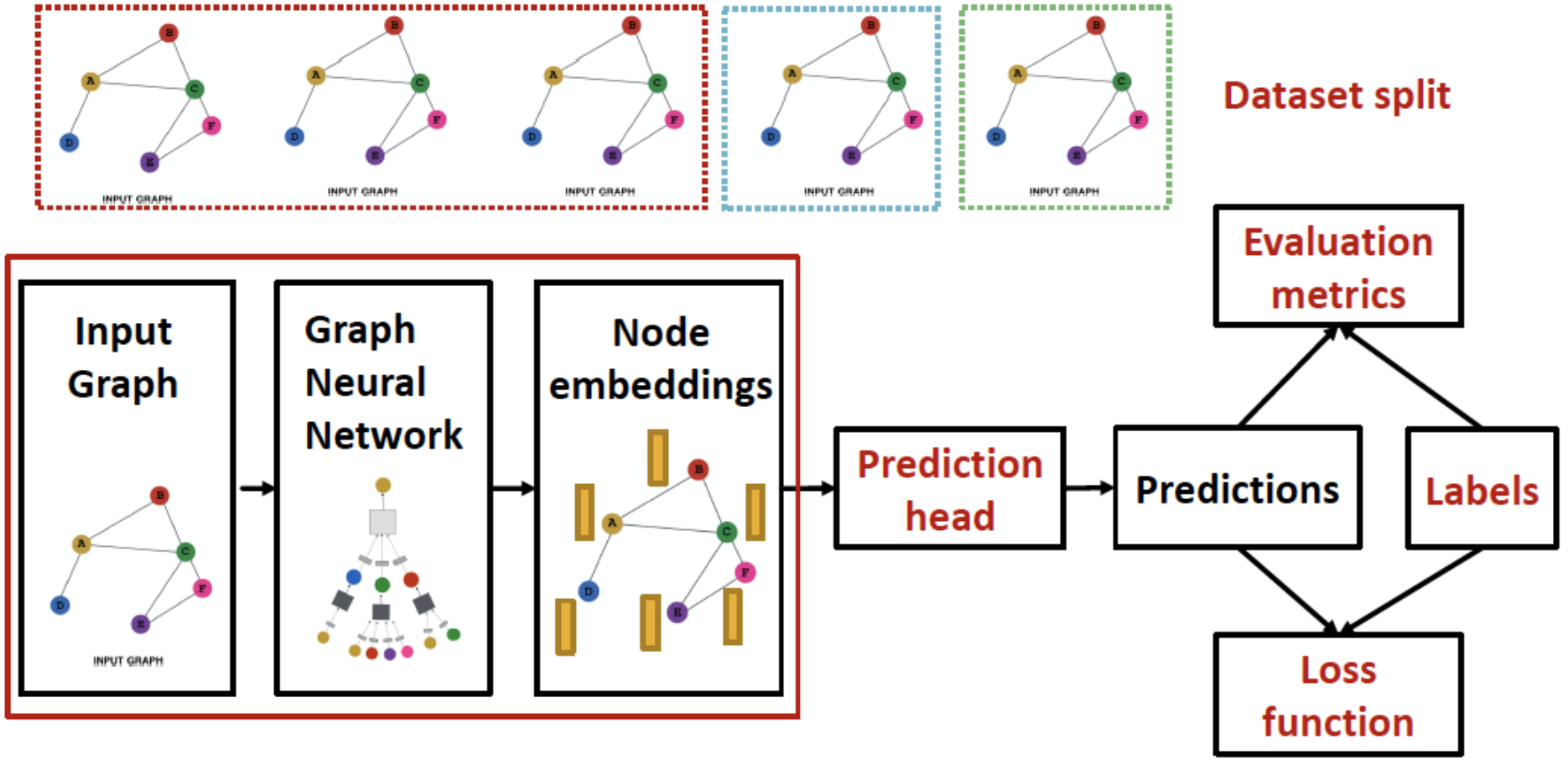
Other Advanced Topics

Recap: A General GNN Framework



J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020c

Recap: GNN Training Pipeline

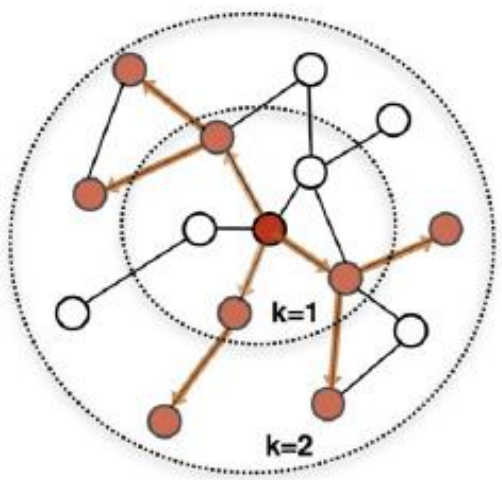


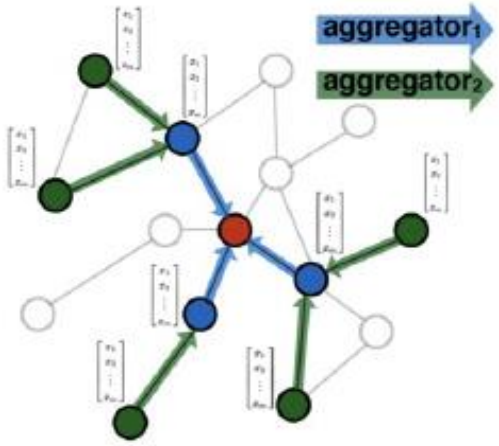
Today's lecture: Can we make GNN representation more expressive?

Limitations of Graph Neural Networks

A Perfect GNN Model

- A thought experiment: What should a perfect GNN do?
  - A k-layer GNN embeds a node based on the K-hop neighborhood structure

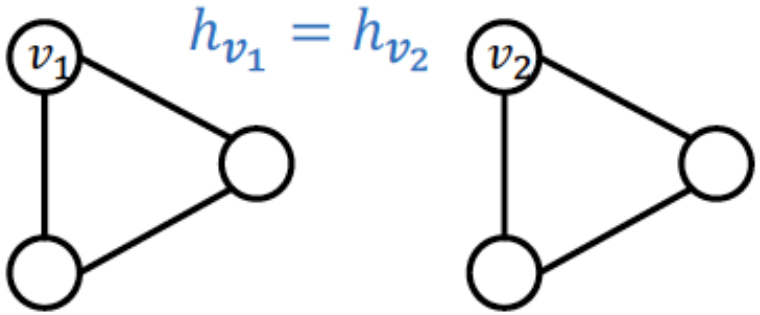




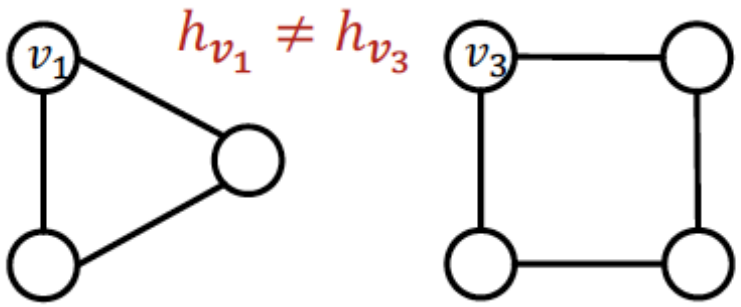
  - A perfect GNN should build an injective function between neighborhood structure (regardless of hops) and node embeddings

A Perfect GNN Model

- Therefore, for a perfect GNN:
  - Observation 1: If two nodes have the same neighborhood structure, they must have the same embedding

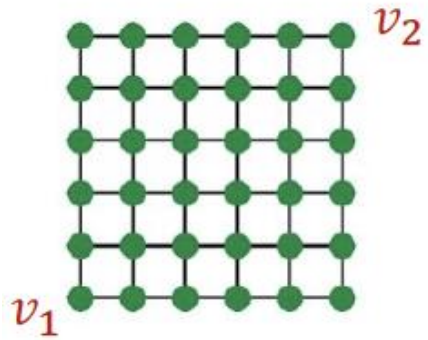


  - Observation 2: If two nodes have different neighborhood structure, they must have different embeddings



Imperfections of Existing GNNs

- However, Observations 1 & 2 are imperfect
- Observation 1 could have issues:
  - Even though two nodes may have the same neighborhood structure, we may want to assign different embeddings to them
  - Because these nodes appear in different positions in the graph
  - We call these tasks Position-aware tasks
  - Even a perfect GNN will fail for these tasks:



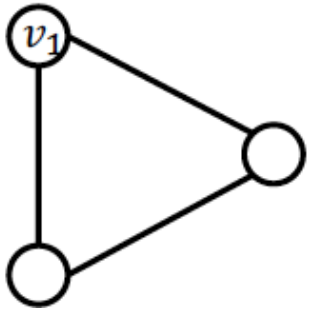
A grid graph

NYC road network

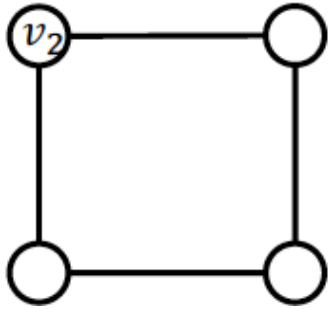
Imperfections of Existing GNNs

- Observation 2 often cannot be satisfied:
  - The GNNs we have introduced so far are not perfect
  - In previous lecture, we discussed that their expressive power is upper bounded by the WL test
  - For example, message passing GNNs cannot count the cycle length:

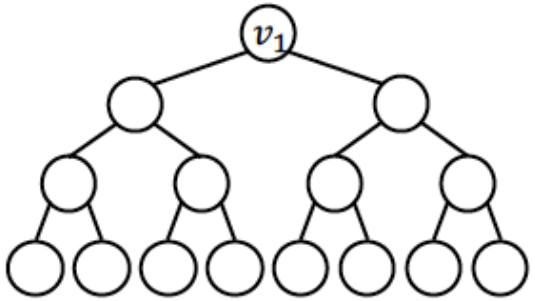
v_1 resides in a cycle with length 3



v_2 resides in a cycle with length 4



The computational graphs for nodes v_1 and v_2 are always the same

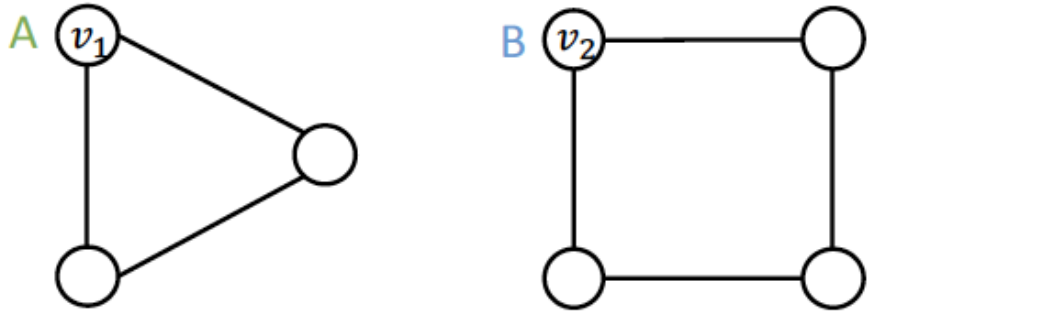


Outline

- We will resolve both issues by building more expressive GNNs
- Fix issues in Observation 1:
  - Create node embeddings based on their positions in the graph
  - Example method: Position-aware GNNs
- Fix issues in Observation 2:
  - Build message passing GNNs that are more expressive than WL test
  - Example method: Identity-aware GNNs

Our Approach

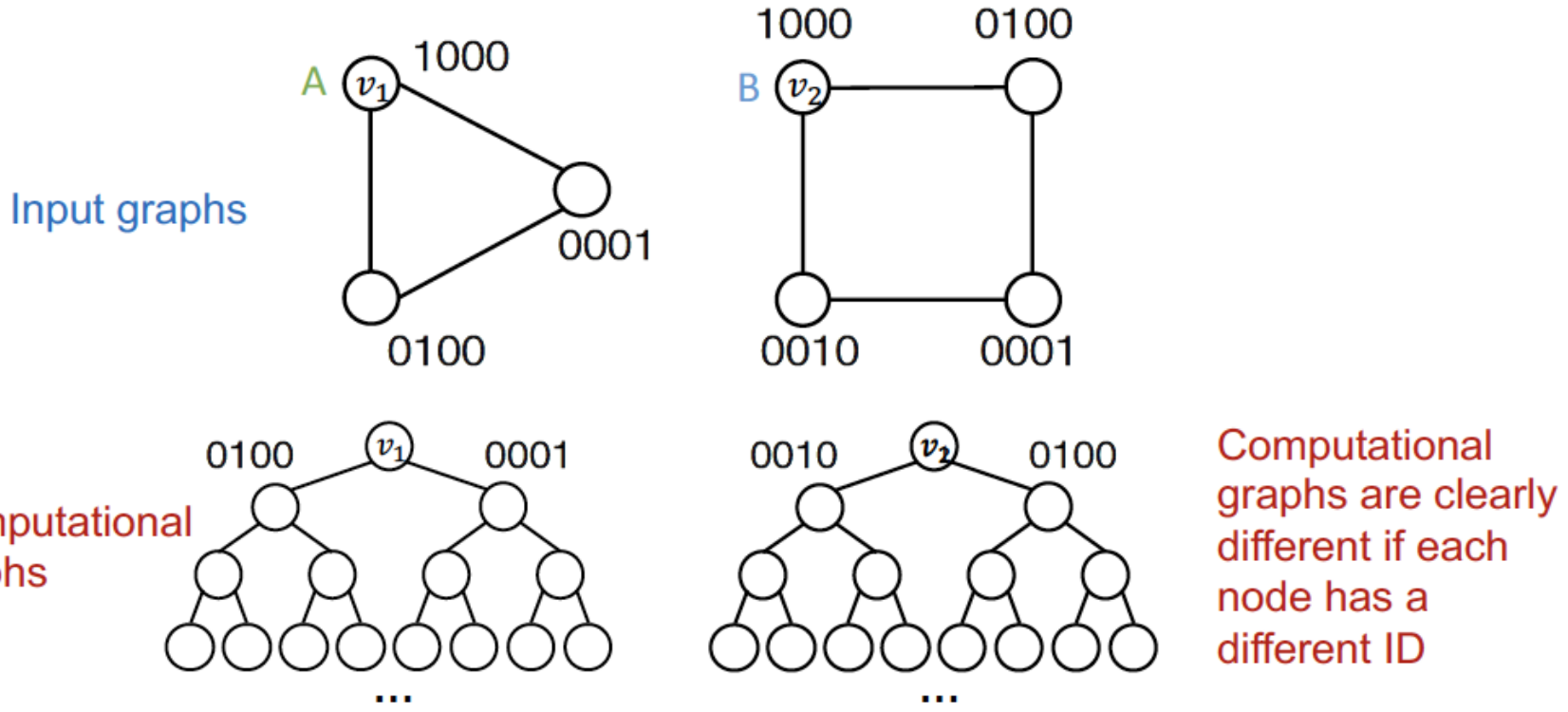
- We use the following thinking:
  - Two different inputs (nodes, edges, graphs) are labeled differently
  - A "failed" model will always assign the same embedding to them
  - A "successful" model will assign different embeddings to them
  - Embeddings are determined by GNN computational graphs:



Two inputs: nodes v_1 and v_2
Different labels: A and B
Goal: assign different embeddings to v_1 and v_2

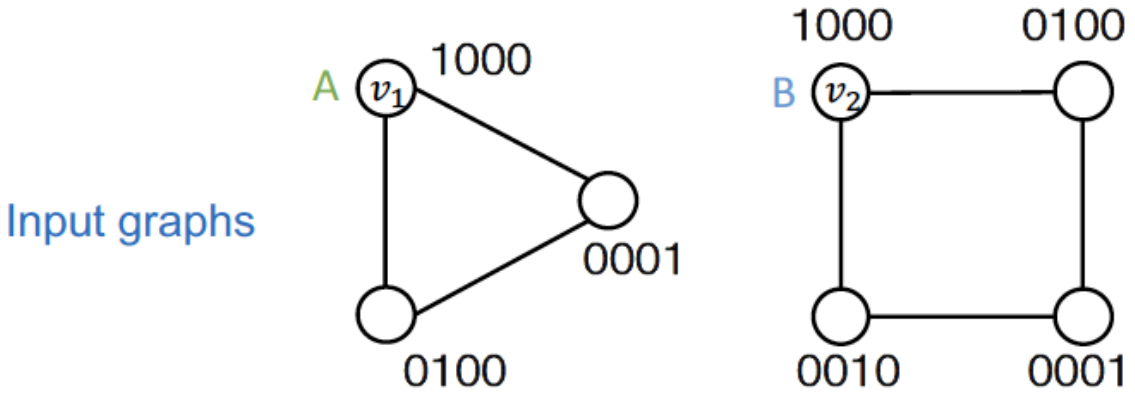
Naïve Solution is not Desirable

- A naïve solution: One-hot encoding
  - Encode each node with a different ID, then we can always differentiate different nodes/edges/graphs



Naïve Solution is not Desirable

- A naïve solution: One-hot encoding
  - Encode each node with a different ID, then we can always differentiate different nodes/edges/graphs

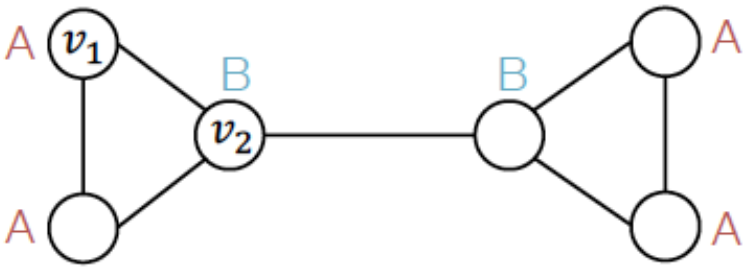


  - Issues:
    - Not scalable: Need $O(N)$ feature dimensions (N is the number of nodes)
    - Not inductive: Cannot generalize to new nodes/graphs

Position-aware Graph Neural Networks

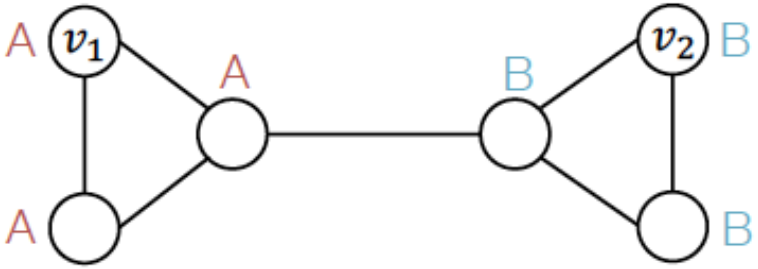
Two Types of Tasks on Graphs

Structure-aware task



- Nodes are labeled by their structural roles in the graph

Position-aware task

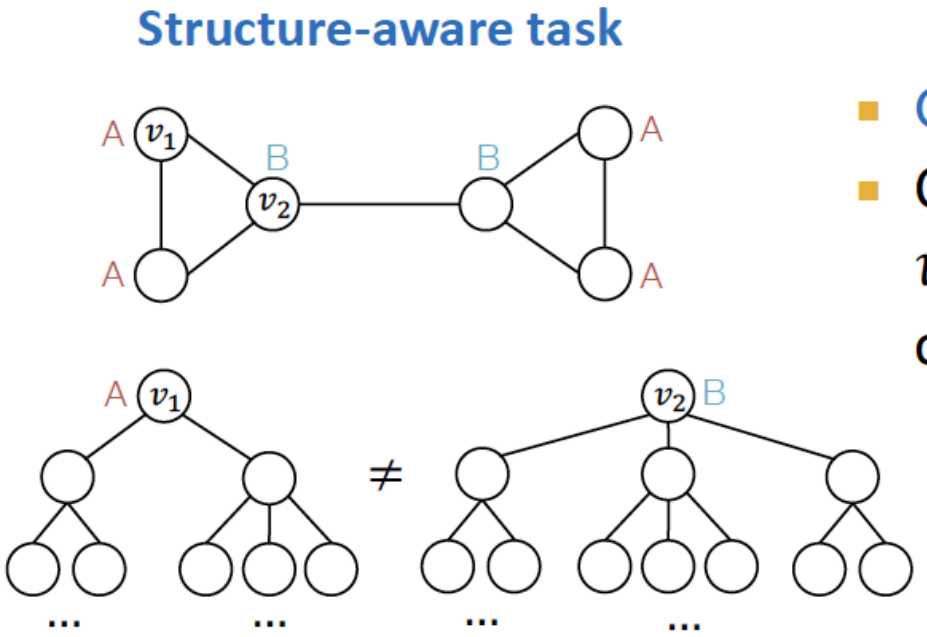


- Nodes are labeled by their positions in the graph

J. You, R. Ying, J. Leskovec. Postion-aware Graph Neural Networks, ICML 2019

Structure-Aware Tasks

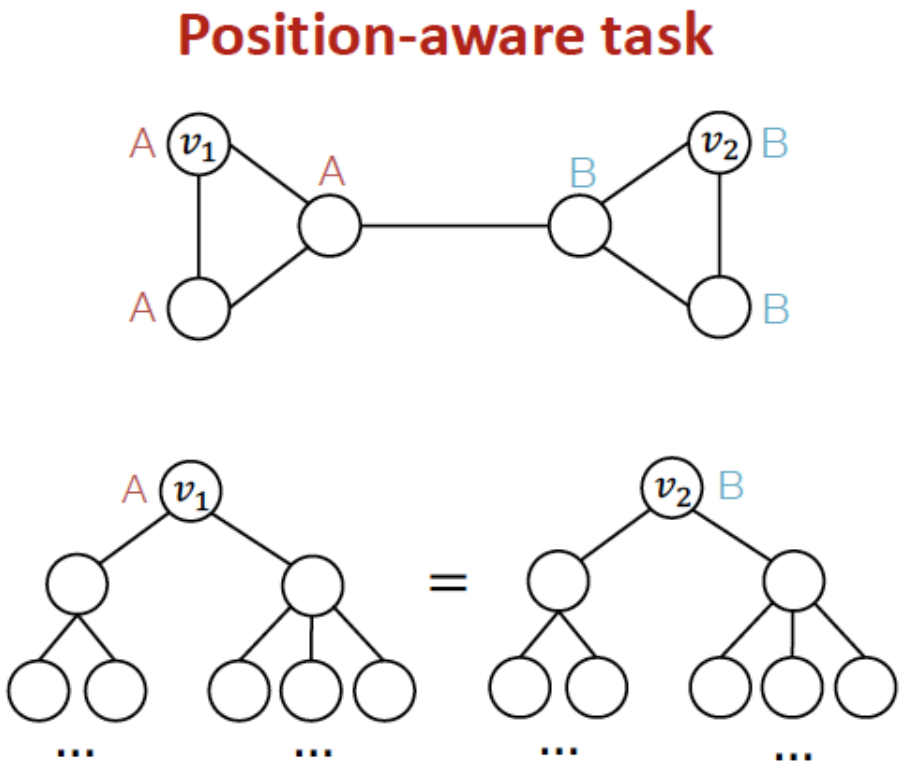
- GNNs often work well for structure-aware tasks



- GNNs work ☺
- Can differentiate v_1 and v_2 by using different computational graphs

Position-Aware Tasks

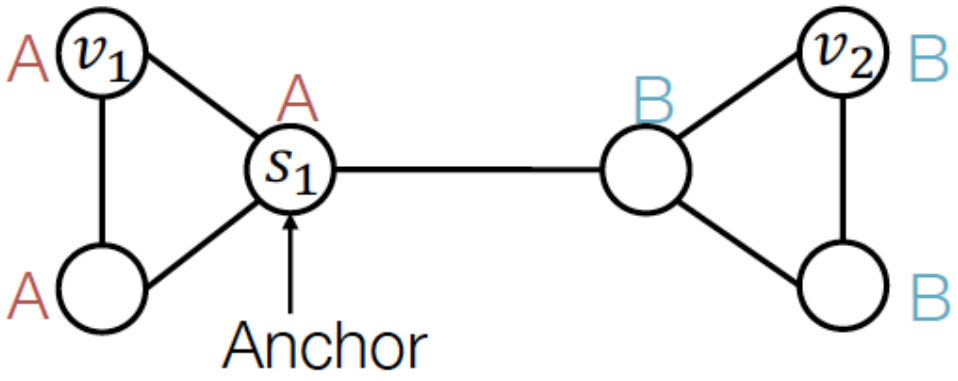
- GNNs will always fail for position-aware tasks



- GNNs fail ☹
- v_1 and v_2 will always have the same computational graph, **due to structure symmetry**
- **Can we define deep learning methods that are position-aware?**

Power of Anchor

- Randomly pick a node s_1 as an anchor node
- Represent v_1 and v_2 via their relative distances w.r.t. the anchor s_1, which are different
- An anchor node serves as a coordinate axis
  - Which can be used to locate nodes in the graph

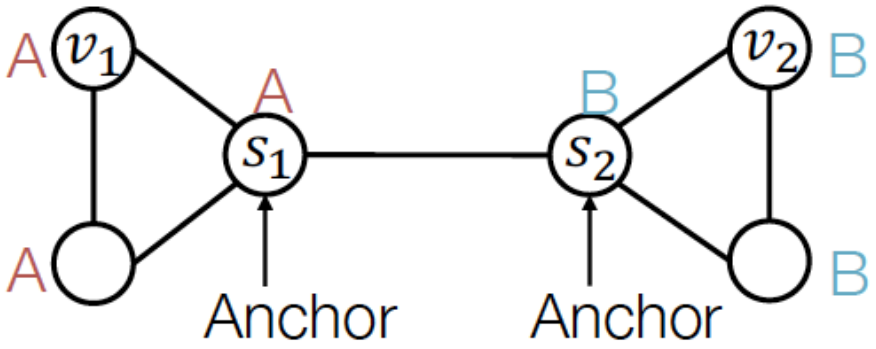


Relative Distances

	s_1
v_1	1
v_2	2

Power of Anchors

- Pick more nodes s_1, s_2 as anchor nodes
- Observation: More anchors can better characterize node position in different regions of the graph
- Many anchors –> Many coordinate axes

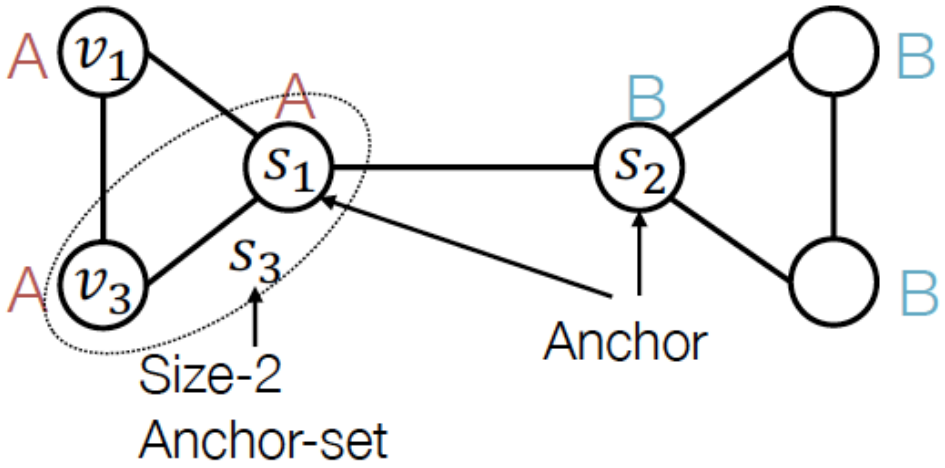


Relative Distances

	s_1	s_2
v_1	1	2
v_2	2	1

Power of Anchor Sets

- Generalize anchor from a single node to a set of nodes
  - We define distance to an anchor-set as the minimum distance to all the nodes in the ancho-set
- Observation: Large anchor-sets can sometimes provide more precise position estimate
  - We can save the total number of anchors



Relative Distances

	s_1	s_2	s_3
v_1	1	2	1
v_3	1	2	0

Anchor s_1, s_2 cannot differentiate node v_1, v_3, but anchor-set s_3 can

Anchor Set: Theory

- Goal: Embed the metric space (V, d) into the Euclidian space $\mathbb{R}^k$ such that the original distance metric is preserved.
  - For every node pairs $u, v \in V$, the Euclidian embedding distance $\|z_u - z_v\|_2$ is close to the original distance metric $d(u, v)$.

Anchor Set: Theory

- Bourgain Theorem [Informal] [Bourgain 1985]
  - Consider the following embedding function of node $v \in V$.

$$f(v) = \left(d_{\min}(v, S_{1,1}), d_{\min}(v, S_{1,2}), \dots, d_{\min}(v, S_{\log n, c\log n})\right) \in \mathbb{R}^{c \log^2 n}$$

  - Where
    - c is a constant
    - $S_{i,j} \subset V$ is chosen by including each node in V independently with probability $\frac{1}{2^i}$
    - $d_{\min}(v, S_{i,j}) = \min_{u \in S_{i,j}} d(v, u)$

  - The embedding distance produced by $\boldsymbol{f}$ is provably close to the original distance metric ($\boldsymbol{V}$, $\boldsymbol{d}$) .

Anchor Set: Theory

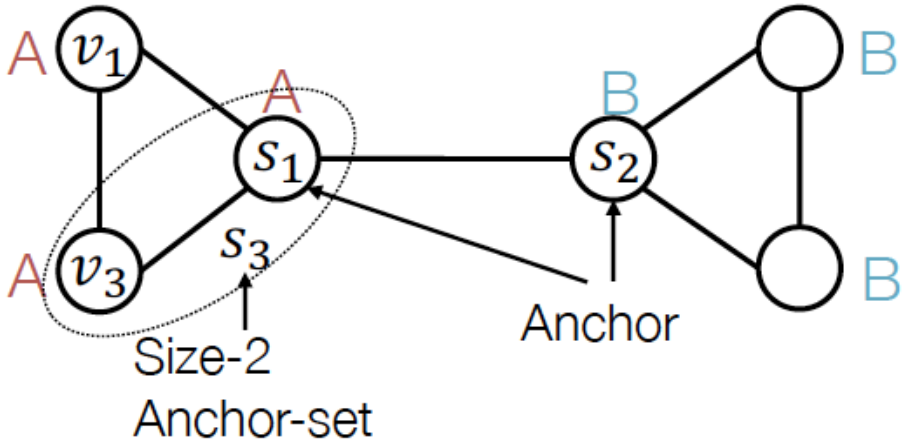
- P-GNN follows the theory of Bourgain theorem
  - First samples $O(\log^2 n)$ anchor sets $S_{i,j}$.
  - Embed each node v via

$$\left(d_{\min}(v, S_{1,1}), d_{\min}(v, S_{1,2}), \dots, d_{\min}(v, S_{\log n, c\log n})\right) \in \mathbb{R}^{c \log^2 n}$$

- P-GNN maintains the inductive capability
  - During training, new anchor sets are re-sampled every time.
  - P-GNN is learned to operate over the new anchor sets.
  - At test time, given a new unseen graph, new anchor sets are sampled.

Position Information: Summary

- Position encoding for graphs: Represent a node's position by its distance to randomly selected anchor-sets
  - Each dimension of the position encoding is tied to an anchor-set



	s_1	s_2	s_3
v_1	1	2	1
v_3	1	2	0

v_1's Position encoding

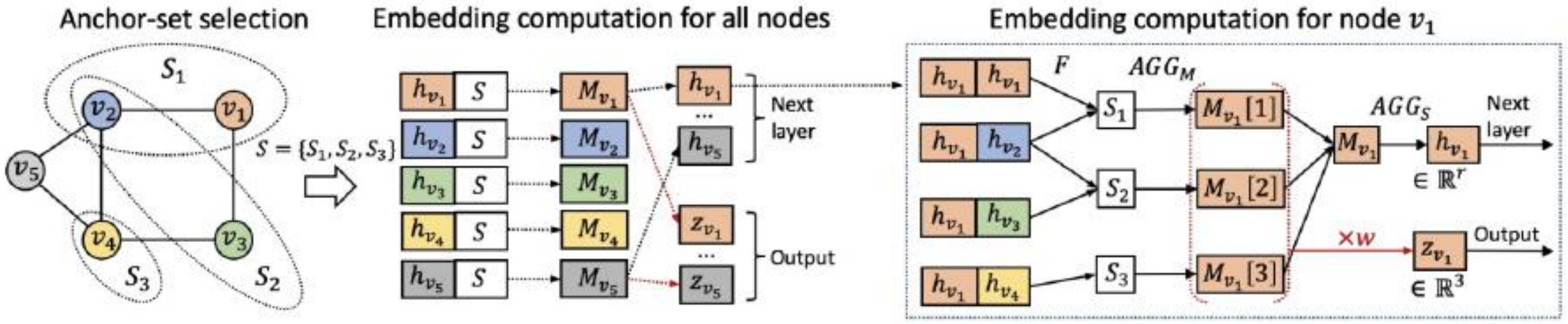
v_3's Position encoding

How to Use Position Information

- The simple way: Use position encoding as an augmented node feature (works well in practice)
  - Issue: Since each dimension of position encoding is tied to a random anchor set, dimensions of positional encoding can be randomly permuted, without changing its meaning
  - Imagine you permute the input dimensions of a normal NN, the output will surely change

How to Use Position Information

- The rigorous solution: Requires a special NN that can maintain the permutation invariant property of position encoding
  - Permuting the input feature dimension will only result in the permutation of the output dimension, the value in each dimension won't change
  - Position-aware GNN paper has more details



Identity-Aware Graph Neural Networks

More Failure Cases for GNNs

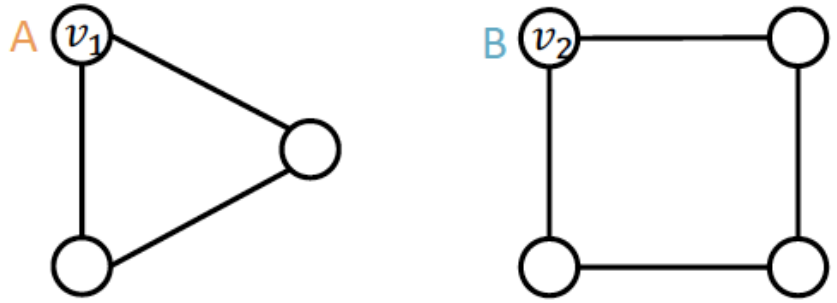
- We learned that GNNs would fail for position-aware tasks
- But can GNN perform perfectly in structure-aware tasks?
  - Unfortunately, NO.
- GNNs exhibit three levels of failure cases in structure-aware tasks:
  - Node level
  - Edge level
  - Graph level

J. You, J. Gomes-Selman, R. Ying, J. Leskovec. Identity-aware Graph Neural Networks, AAAI 2021

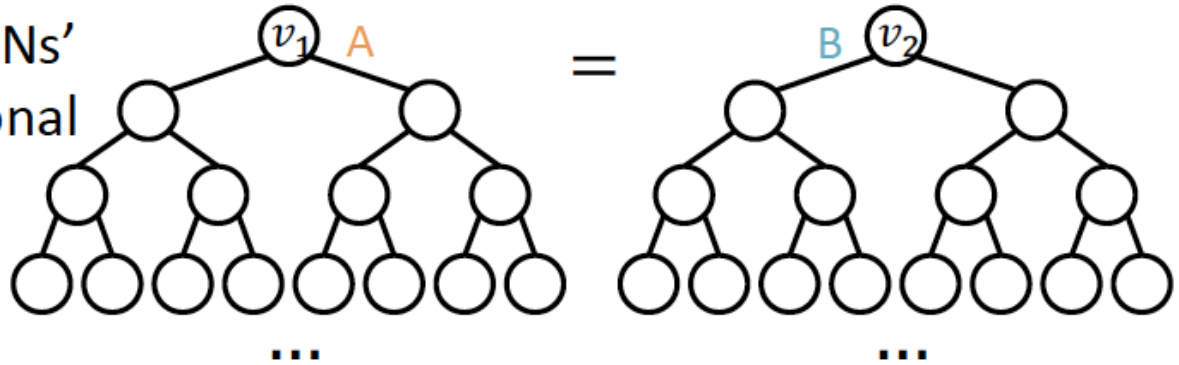
GNN Failure 1: Node-Level Tasks

- Different Inputs but the same computational graph → GNN fails

Example input graphs



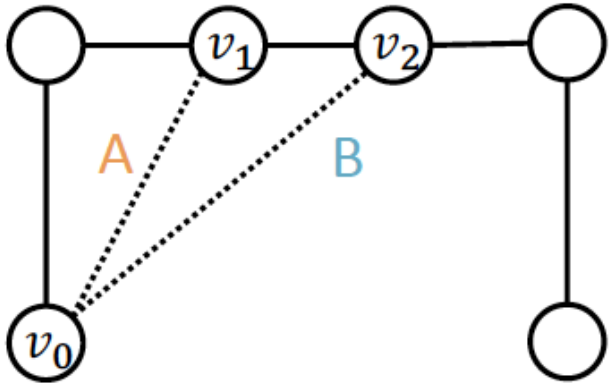
Existing GNNs' computational graphs



GNN Failure 2: Edge-Level Tasks

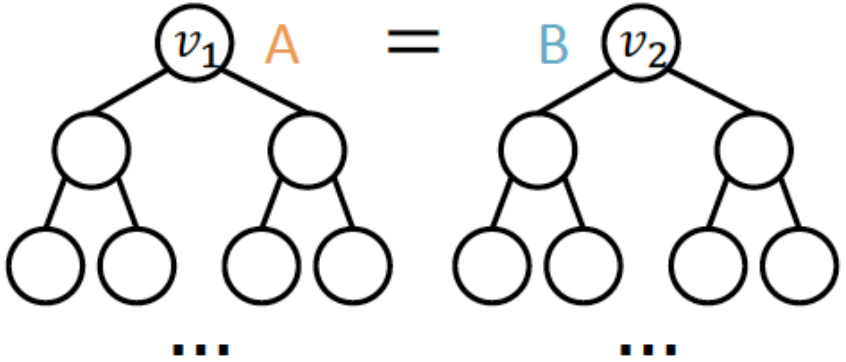
- Different Inputs but the same computational graph → GNN fails

Example input graphs



Edge A and B share node v_0
We look at embeddings for v_1 and v_2

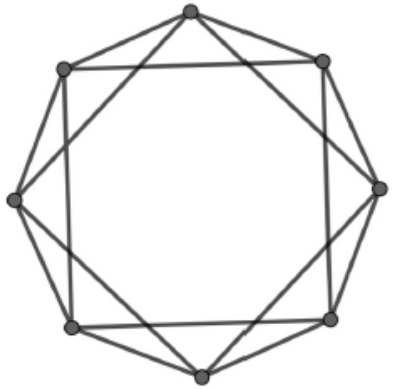
Existing GNNs' computational graphs



GNN Failure 3: Graph-Level Tasks

- Different Inputs but the same computational graph → GNN fails

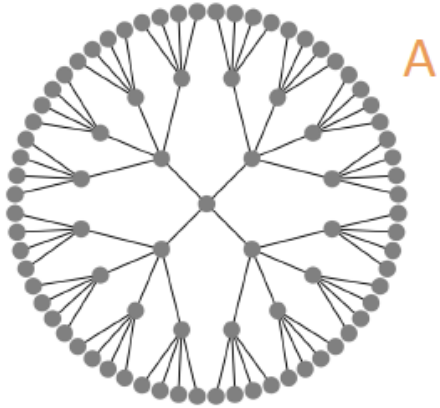
Example input graphs



We look at embeddings for each node

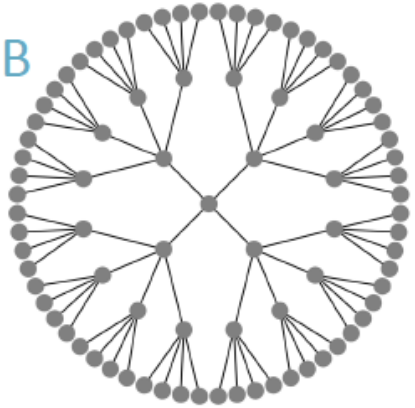
Existing GNNs' computational graphs

For each node:



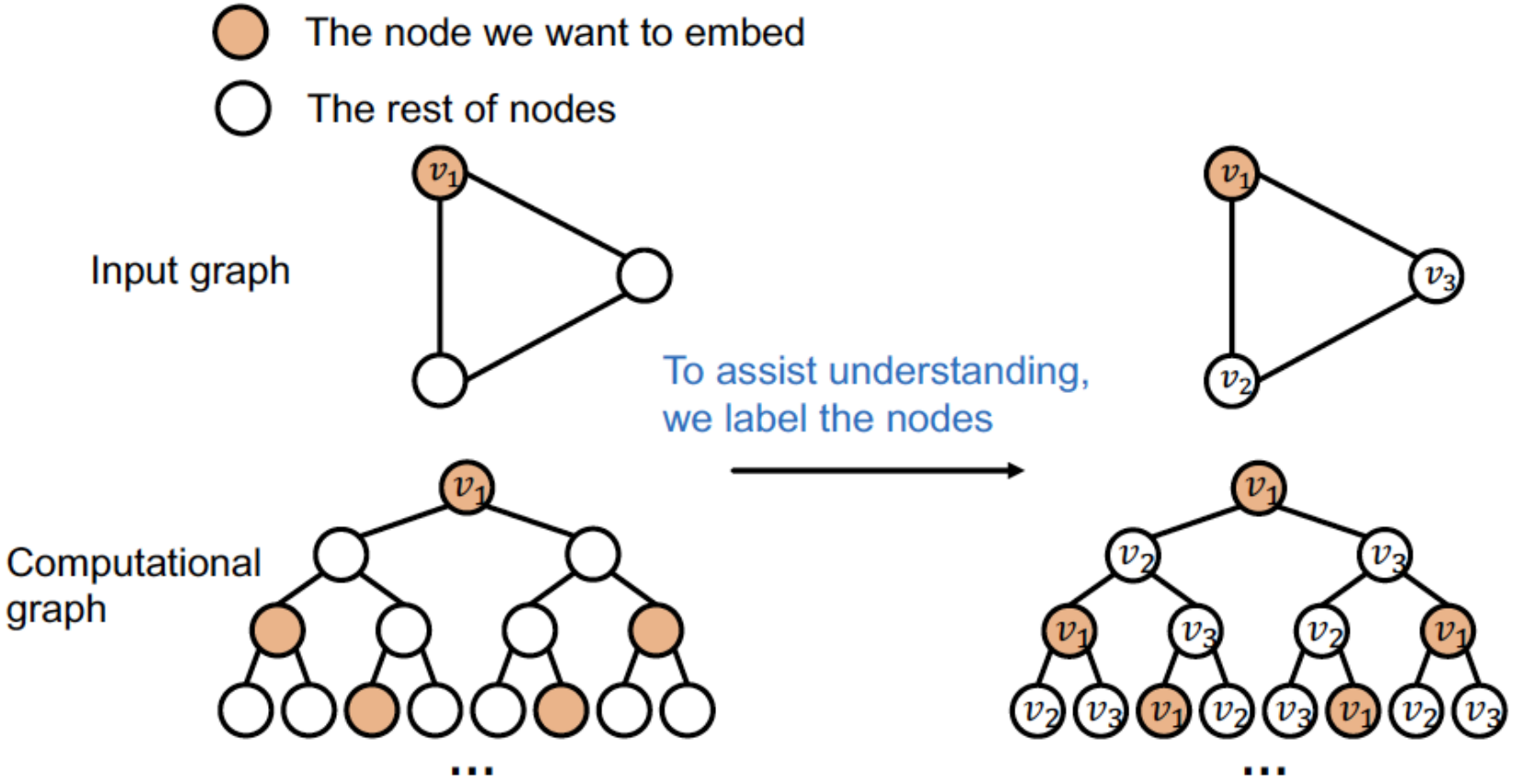
=

For each node:



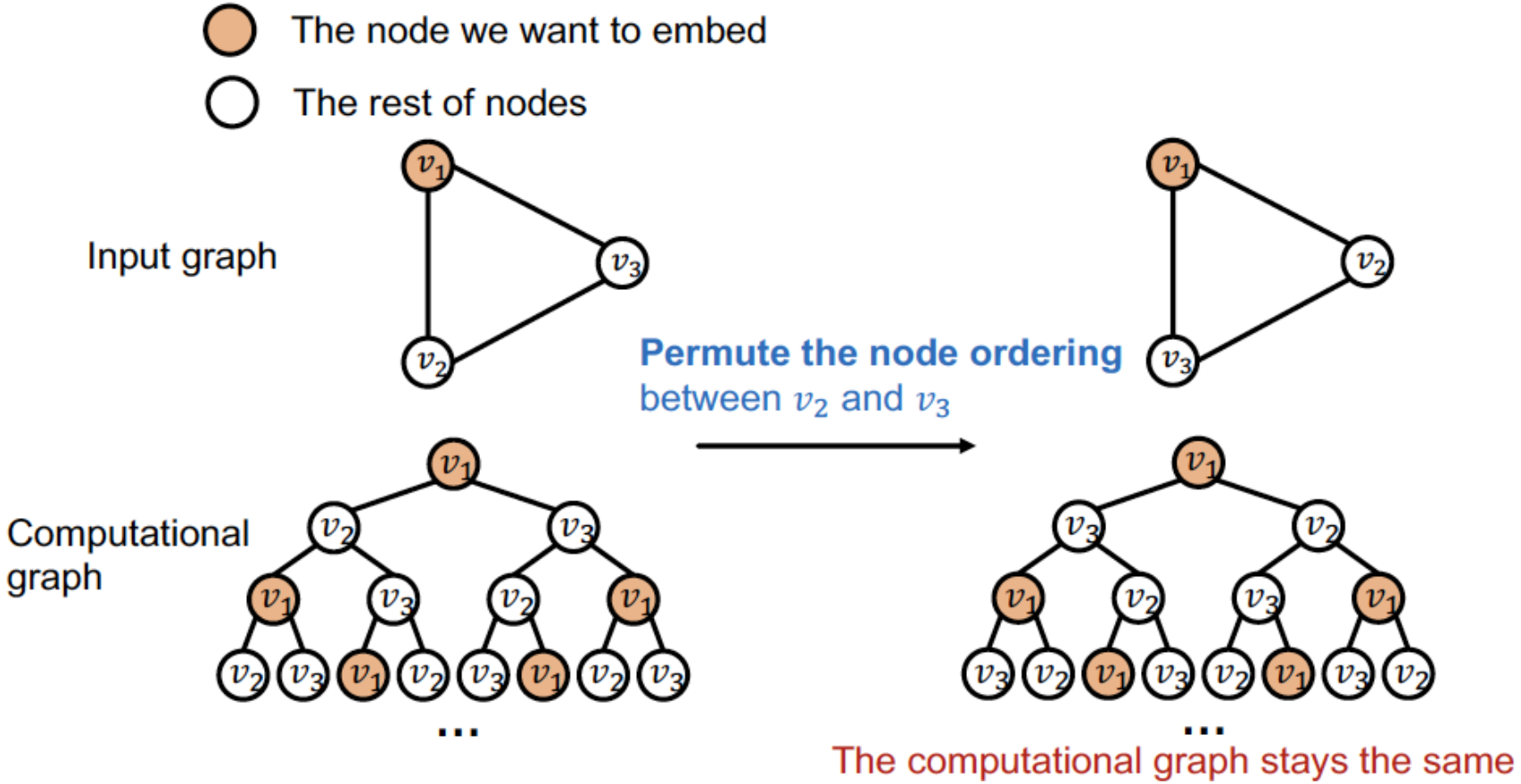
Idea: Inductive Node Coloring

- Idea: We can assign a color to the node we want to embed



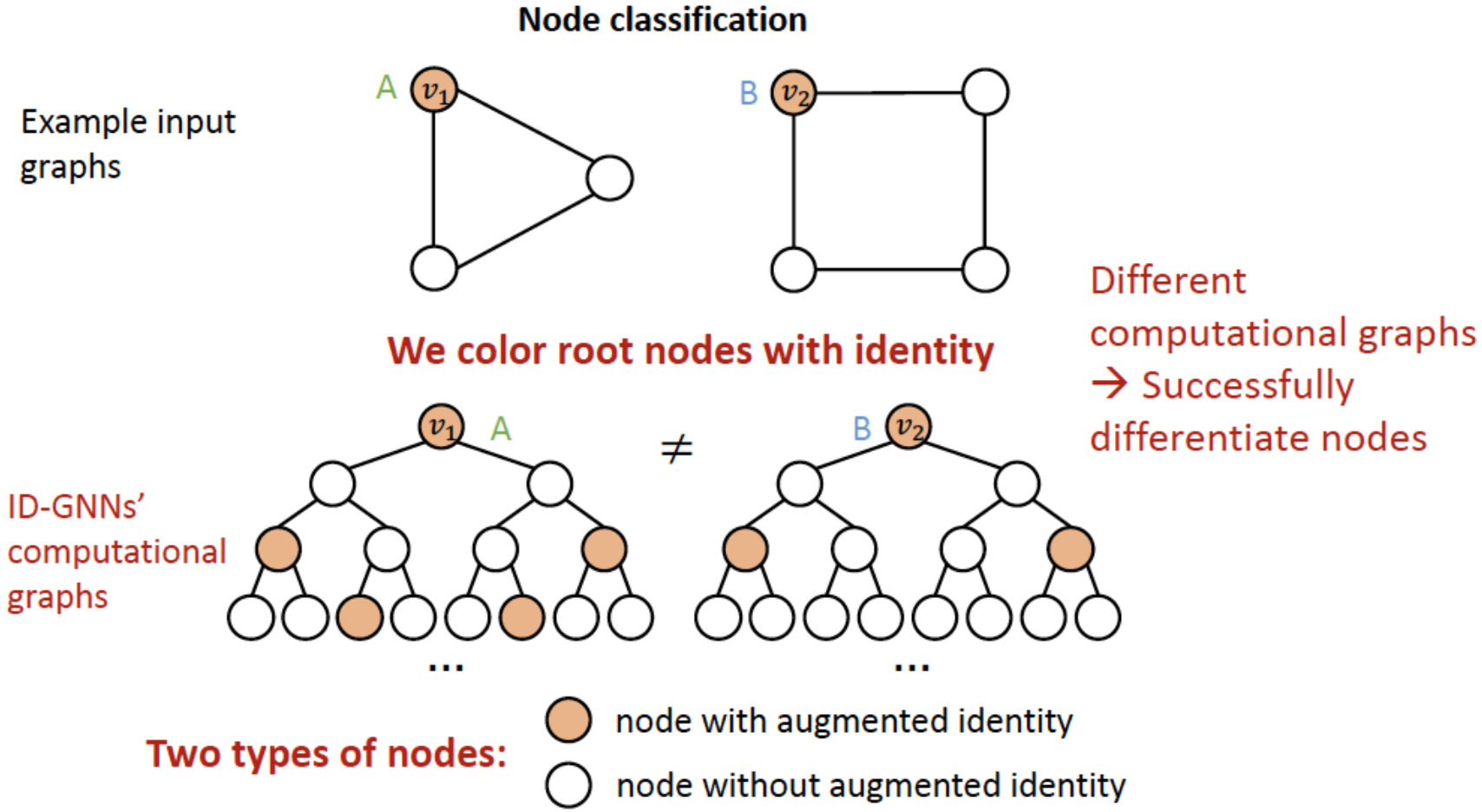
Idea: Inductive Node Coloring

- This coloring is inductive:
  - It is invariant to node ordering/identities



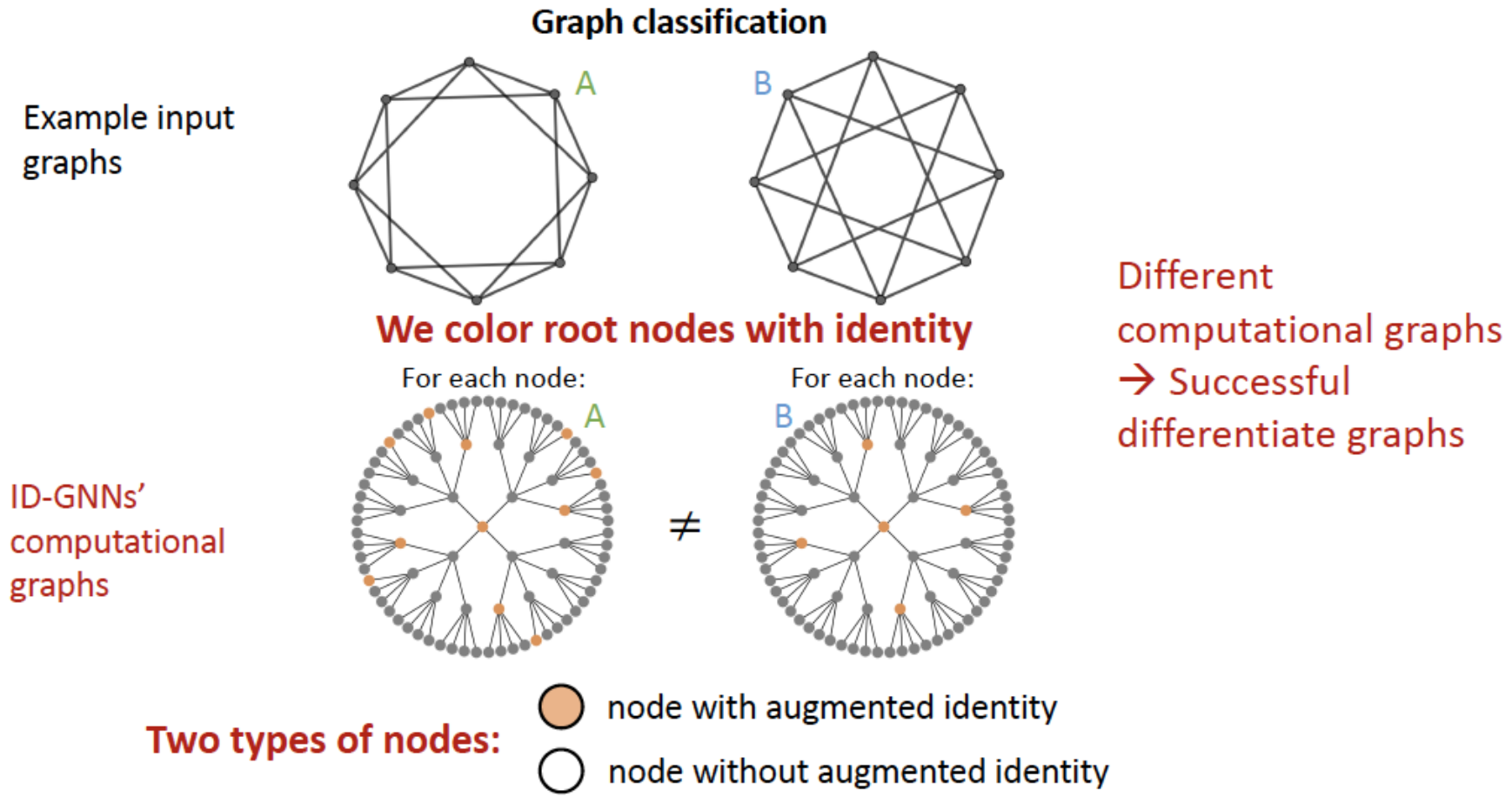
Inductive Node Coloring - Node Level

- Inductive node coloring can help node classification



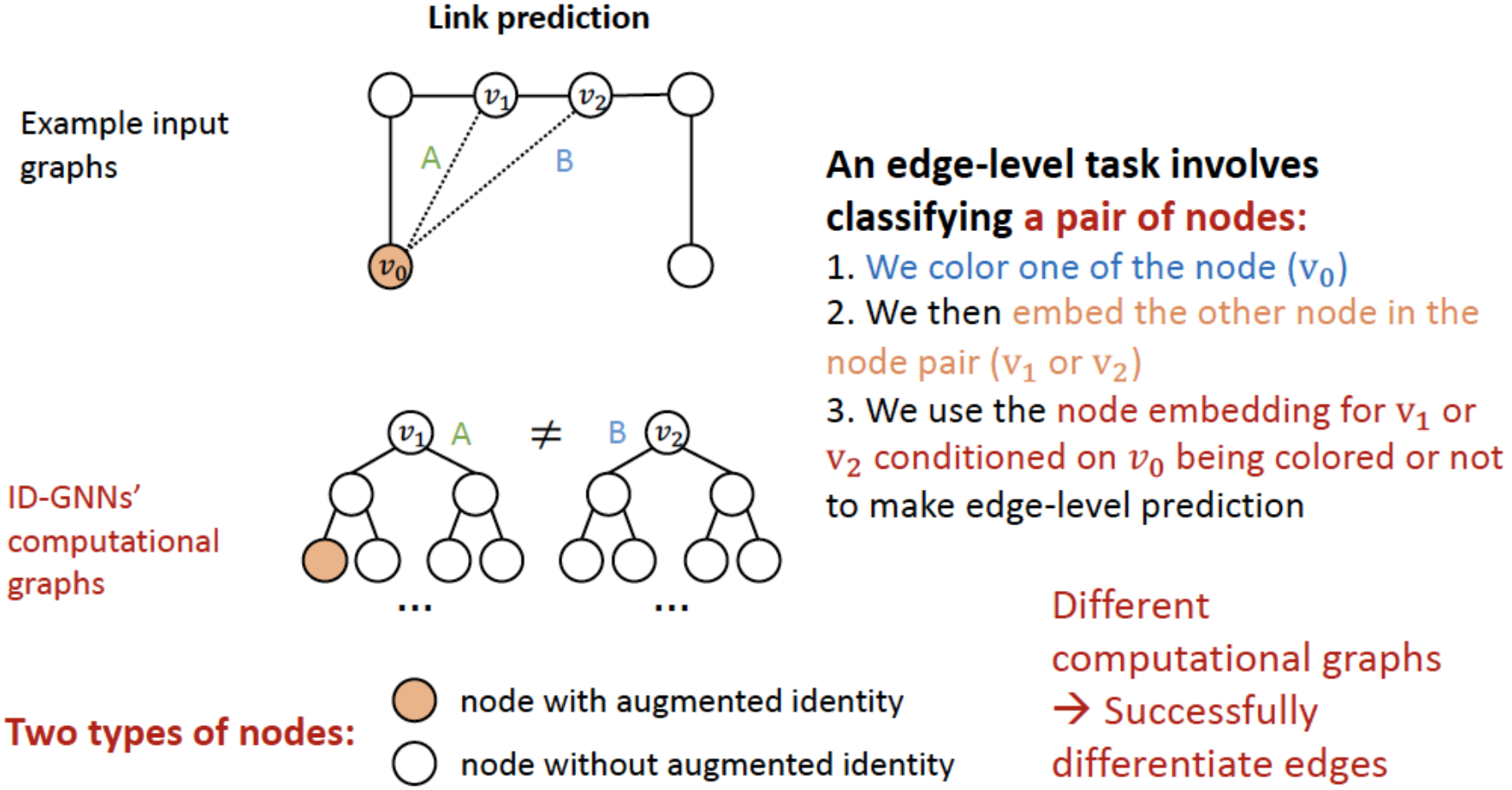
Inductive Node Coloring – Graph Level

- Inductive node coloring can help graph classification



Inductive Node Coloring – Edge Level

- Inductive node coloring can help link prediction



An edge-level task involves classifying a pair of nodes:

1. We color one of the node (v_0)
2. We then embed the other node in the node pair (v_1 or v_2)
3. We use the node embedding for v_1 or v_2 conditioned on v_0 being colored or not to make edge-level prediction

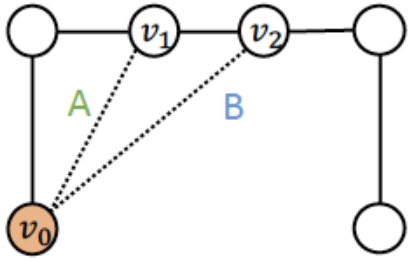
Different computational graphs → Successfully differentiate edges

Inductive Node Coloring – Edge Level

- Inductive node coloring can help link prediction

Link prediction

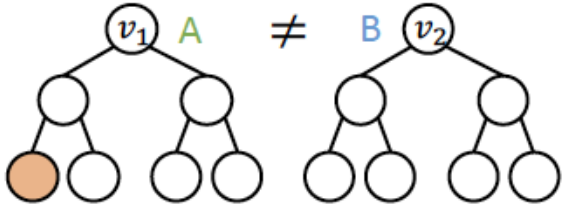
Example input graphs



An edge-level task involves classifying a pair of nodes:

1. We color one of the node (v_0)
2. We then embed the other node in the node pair (v_1 or v_2)
3. We use the node embedding for v_1 or v_2 conditioned on v_0 being colored or not to make edge-level prediction

ID-GNNs' computational graphs

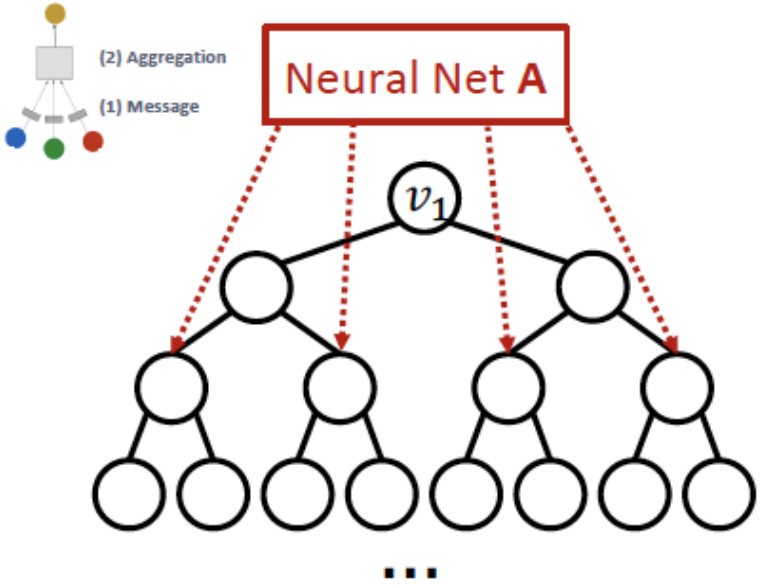


Two typ

How to build a GNN using node coloring?

Identity-Aware GNN

- Utilize inductive node coloring in embedding computation
  - Idea: Heterogenous message passing
  - Normally, a GNN applies the same message/aggregation computation to all the nodes

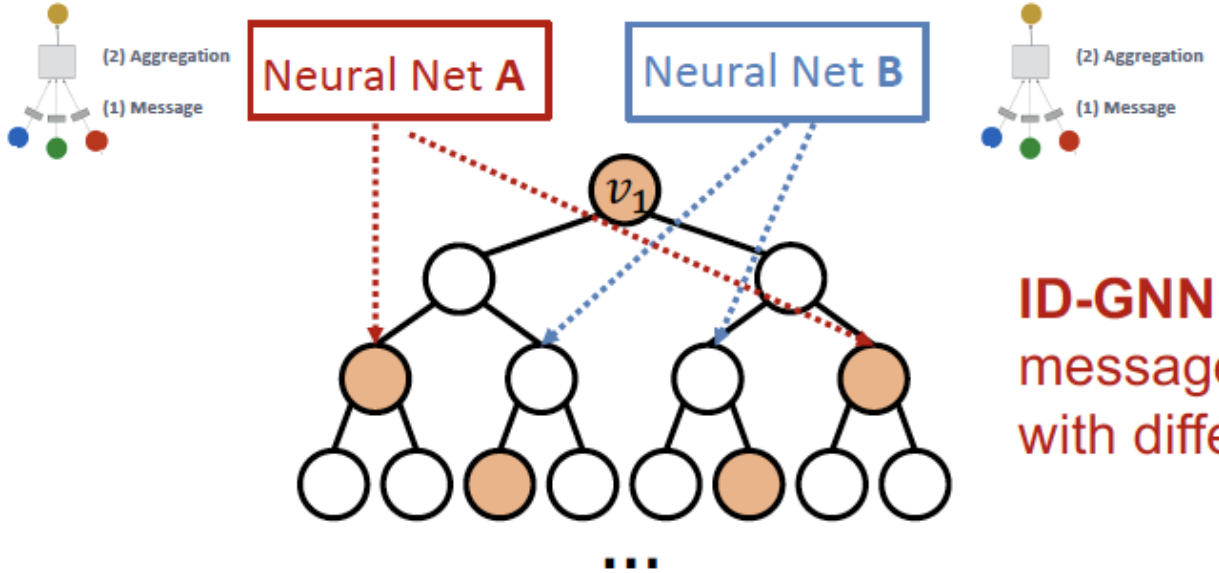


GNN: At a given layer, we apply the same message/aggregation to each node

J. You, J. Gomes-Selman, R. Ying, J. Leskovec. Identity-aware Graph Neural Networks, AAAI 2021

Identity-Aware GNN

- Idea: Heterogenous message passing
  - Heterogenous: different types of message passing is applied to different nodes
  - An ID-GNN applies different message/aggregation to nodes with different colorings



ID-GNN: At a given layer, different message/aggregation to nodes with different colorings

Identity-Aware GNN

- Output: Node embedding $h_v^{(K)}$ for $v \in V$
- Step 1: Extract ego-network
  - $\mathcal{G}_v^{(K)}$: K-hop neighborhood graph around v
  - Set the initial node feature
    - For $u \in \mathcal{G}_v^{(K)}, h_u^{(0)} \leftarrow x_u$ (input node feature)

Identity-Aware GNN

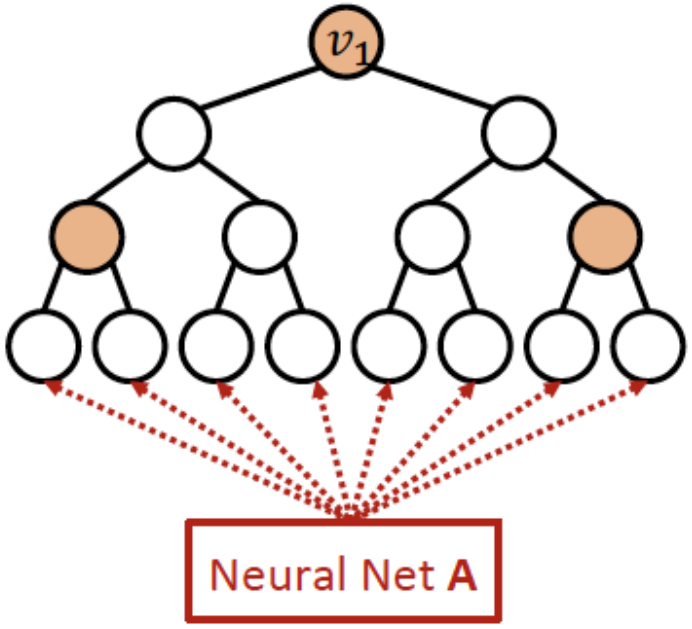
- Step 2: Heterogeneous message passing
  - For $k = 1, \cdots, K$ do
    - For $u \in \mathcal{G}_v^{(K)}$, do

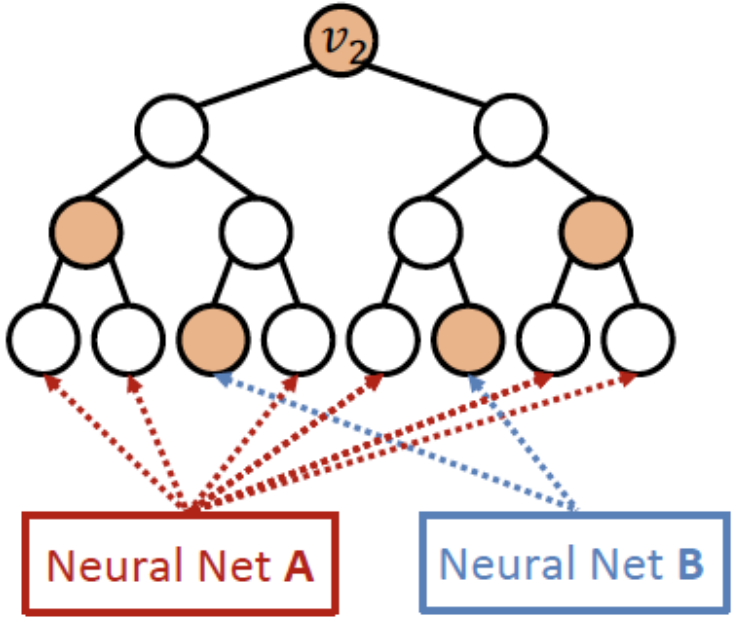
$$\boldsymbol{h}_u^{(k)} \leftarrow AGG^{(k)}\left(\left\{\mathrm{MSG}^{(k)}_{\mathbf{1}[s=v]}\left(\boldsymbol{h}_s^{(k-1)}\right), s \in N(u)\right\}, \boldsymbol{h}_u^{(k-1)}\right)$$

Depending on whether $s = v$ (s is the center node v) or not, we use different neural network functions to transform $\boldsymbol{h}_s^{(k-1)}$.

Identity-Aware GNN

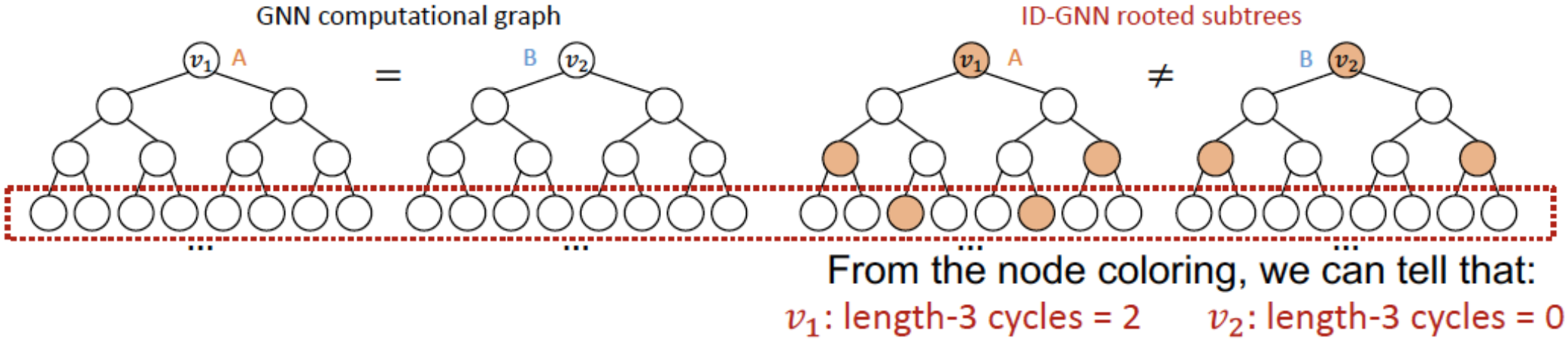
- Why does heterogenous message passing work:
  - Suppose two nodes v_1, v_2 have the same computational graph structure, but have different node colorings
  - Since we will apply different neural network for embedding computation, their embeddings will be different





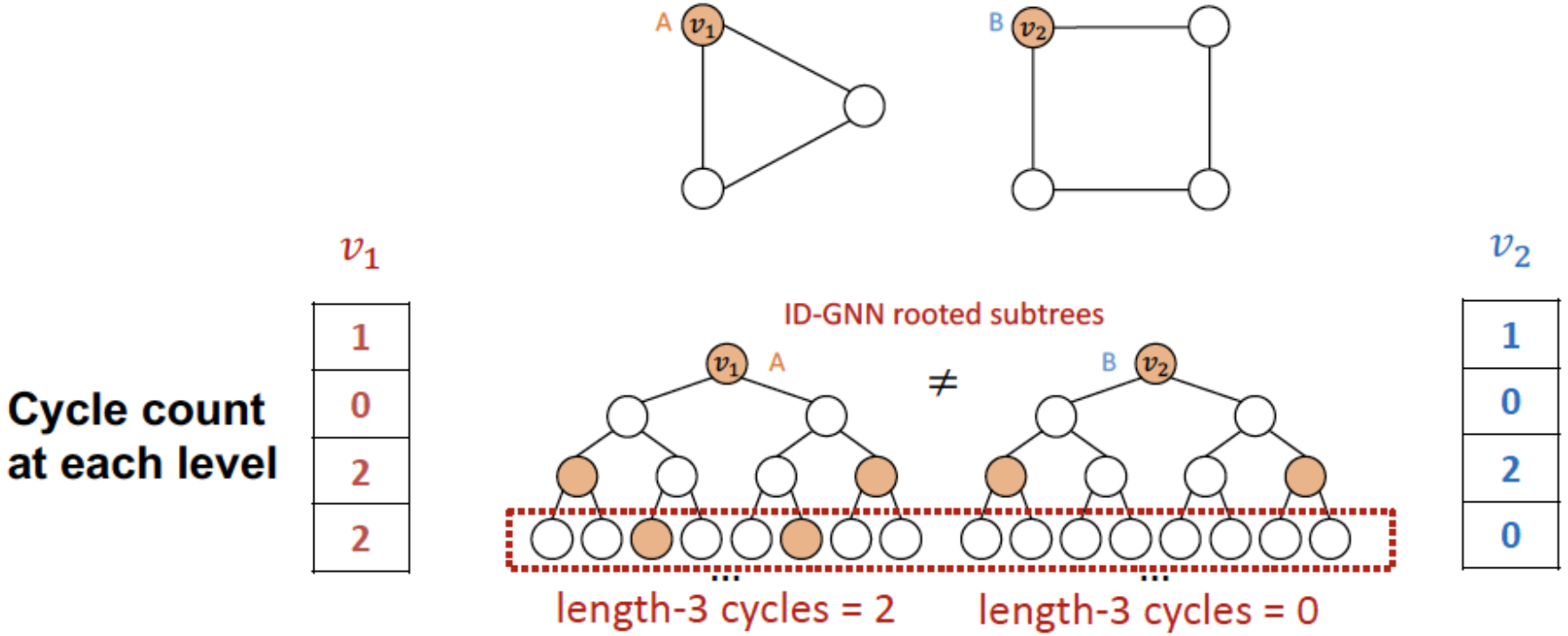
GNN vs ID-GNN

Goal: classify v_1 and v_2



- Why does ID-GNN work better than GNN?
- Intuition: ID-GNN can count cycles originating from a given node, but GNN cannot

Simplified Version: ID-GNN-Fast



- Based on the intuition, we present a simplified version ID-GNN-Fast
  - Include identity information as an augmented node feature (no need to do heterogenous message passing)
  - Use cycle counts in each layer as an augmented node feature. Also can be used together with any GNN

Identity-Aware GNN

- Summary of ID-GNN: A general and powerful extension to GNN framework
  - We can apply ID-GNN on any message passing GNNs (GCN, GraphSAGE, GIN, …)
    - ID-GNN provides consistent performance gain in node/edge/graph level tasks
  - ID-GNN is more expressive than their GNN counterparts. ID-GNN is the first message passing GNN that is more expressive than 1-WL test
  - We can easily implement ID-GNN using popular GNN tools (PyG, DGL, …)

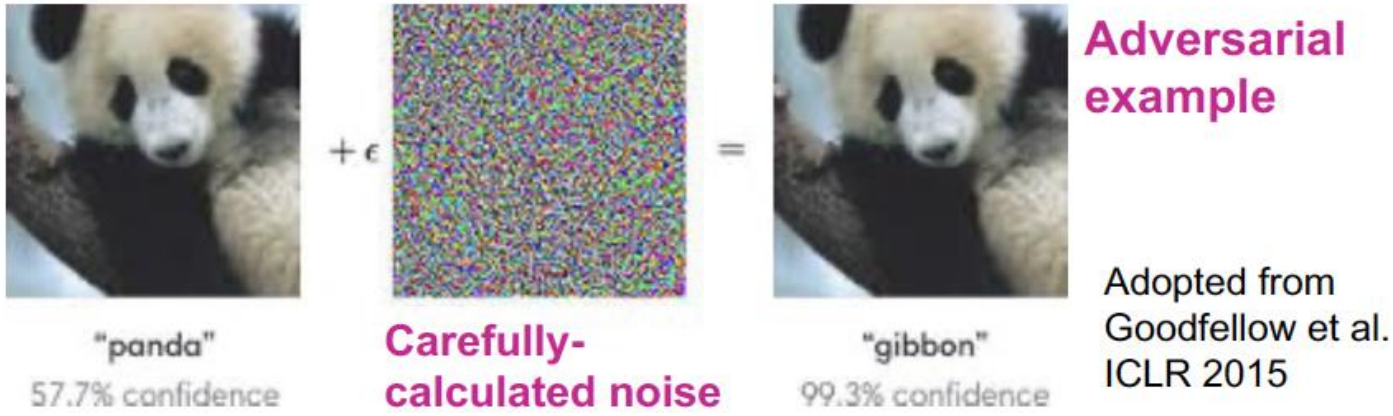
Robustness of Graph Neural Networks

Deep Learning Performance

- Recent years have seen impressive performance of deep learning models in a variety of applications.
  - Example: In computer vision, deep convolutional networks have achieved human-level performance on ImageNet (image category classification)
- Are these models ready to be deployed in real world?

Adversarial Examples

- Deep convolutional neural networks are vulnerable to adversarial attacks:
  - Imperceptible noise changes the prediction.



- Adversarial examples are also reported in natural language processing [Jia & Liang et al. EMNLP 2017] and audio processing [Carlini et al. 2018] domains.

Implications of Adversarial Examples

- The existence of adversarial examples prevents the reliable deployment of deep learning models to the real world.
  - Adversaries may try to actively hack the deep learning models.
  - The model performance can become much worse than we expect.

- Deep learning models are often not robust.
  - In fact, it is an active area of research to make these models robust against adversarial examples

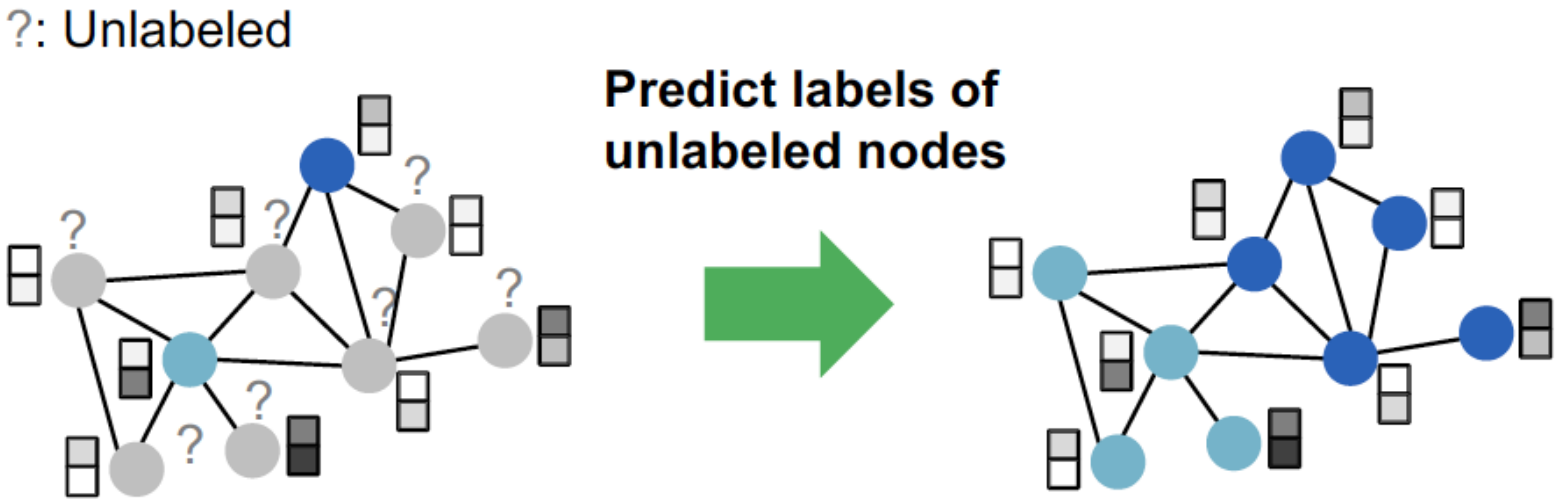
Robustness of GNNs

- How about GNNs? Are they robust to adversarial examples?
- Premise: Common applications of GNNs involve public platforms and monetary interests.
  - Recommender systems
  - Social networks
  - Search engines
- Adversaries have the incentive to manipulate input graphs and hack GNNs' predictions.

Zügner et al. Adversarial Attacks on Neural Networks for Graph Data, KDD 2018

Setting to Study GNN Robustness

- To study the robustness of GNNs, we specifically consider the following setting:
  - Task: Semi-supervised node classification
  - Model: GCN [Kipf & Welling ICLR 2017]

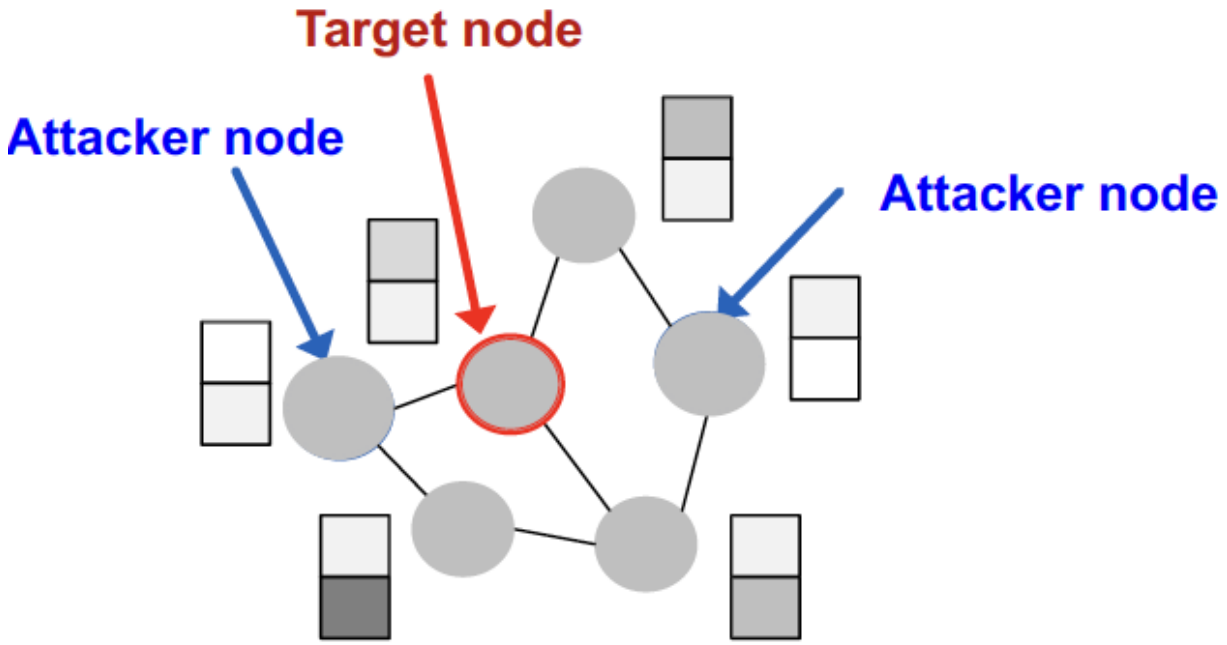


Roadmap

- We first describe several real-world adversarial attack possibilities.
- We then review the GCN model that we are going to attack (knowing the opponent).
- We mathematically formalize the the attack problem as an optimization problem.
- We empirically see how vulnerable GCN's prediction is to the adversarial attack.

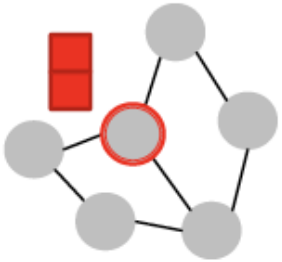
Attack Possibilities

- What are the attack possibilities in real world?
  - Target node $t \in V$: node whose label prediction we want to change
  - Attacker nodes $S \subset V$: nodes the attacker can modify

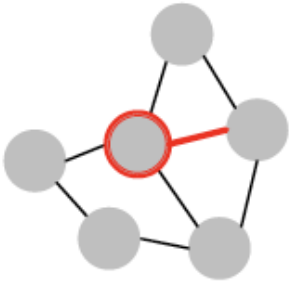


Attack Possibilities: Direct Attack

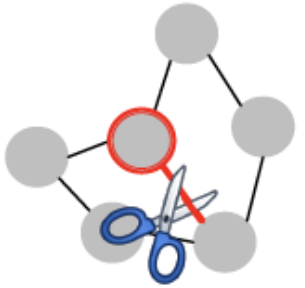
- Direct Attack: Attacker node is the target node: $s = t$
- Modify target node feature
  - Ex) Change website content



- Add connections to target
  - Ex) Buy likes/followers

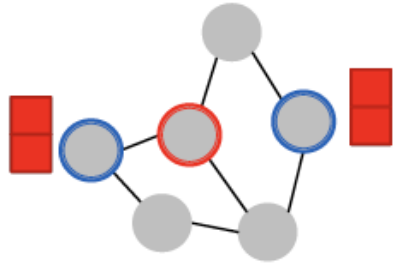


- Remove connections from target
  - Ex) Unfollow users

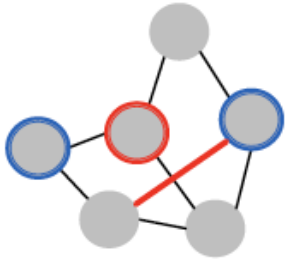


Attack Possibilities: Indirect Attack

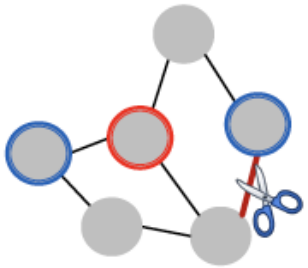
- Indirect Attack: The target node is not in the attacker nodes: $t \notin S$
- Modify attacker node features
  - Ex) Hijack friends of targets



- Add connections to attackers
  - Ex) Create a link, link farm



- Remove connections from attackers
  - Ex) Delete undesirable link

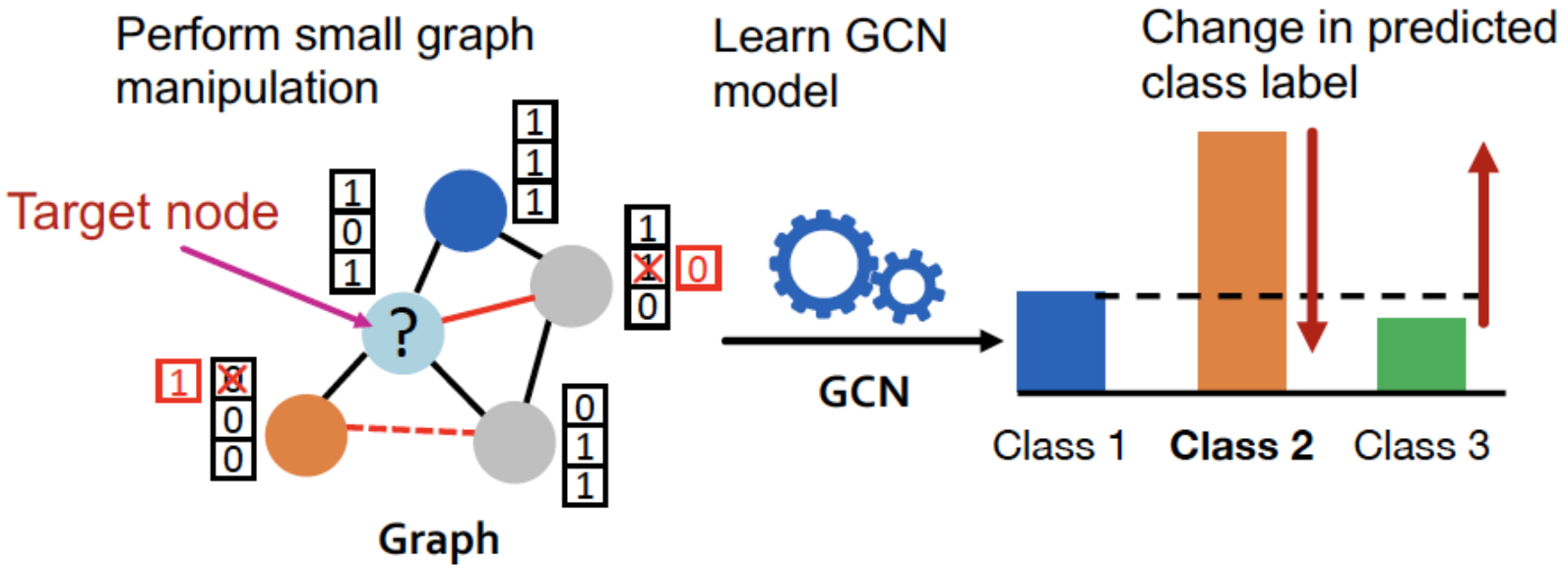


Formalizing Adversarial Attacks

- **Objective for the attacker**:

Maximize (change of target node label prediction)
Subject to (graph manipulation is small)

If graph manipulation is too large, it will easily be detected. Successful attacks should change the target prediction with "unnoticeably-small" graph manipulation.



Mathematical Formulation (1)

- Original graph:
  - $\boldsymbol{A}$: adjacency matrix, $\boldsymbol{X}$: feature matrix
- Manipulated graph (after adding noise):
  - $\boldsymbol{A}$': adjacency matrix, $\boldsymbol{X}$': feature matrix
- Assumption: $(\boldsymbol{A}', \boldsymbol{X}') \approx (\boldsymbol{A}, \boldsymbol{X})$
  - Graph manipulation is unnoticeably small.
  - Preserving basic graph statistics (e.g,. Degree distribution) and feature statistics.
  - Graph manipulation is either direct (changing the feature/connection of target nodes) or indirect.

Mathematical Formulation (2)

- Overview of the attack framework
  - Original adjacency matrix $\boldsymbol{A}$, node features $\boldsymbol{X}$, node labels $\boldsymbol{Y}$.
  - $\boldsymbol{\theta}^*$: Model parameter learned over $\boldsymbol{A}$, $\boldsymbol{X}$, $\boldsymbol{Y}$.
    - c_v^*: class label of node v predicted by GCN with $\boldsymbol{\theta}^*$
  - An attacker has access to $\boldsymbol{A}$,$\boldsymbol{X}$, $\boldsymbol{Y}$, and the learning algorithm.
  - The attacker modifies ($\boldsymbol{A}$, $\boldsymbol{X}$) into ($\boldsymbol{A}$', $\boldsymbol{X}$').
  - $\boldsymbol{\theta}^{*\prime}$: Model parameter learned over $\boldsymbol{A}$', $\boldsymbol{X}$', $\boldsymbol{Y}$.
    - $c_v^{*\prime}$: class label of node v predicted by GCN with $\boldsymbol{\theta}^{*\prime}$
  - The goal of the attacker is to make $c_v^{*\prime} \neq c_v^*$.

Mathematical Formulation (3)

- Target node: $v \in V$
- GCN learned over the original graph

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} \mathcal{L}_{train}(\boldsymbol{\theta}; \boldsymbol{A}, \boldsymbol{X})$$

- GCN's original prediction on the target node:

$$c_v^* = \textbf{argmax}_c f_{\boldsymbol{\theta}^*}(\boldsymbol{A}, \boldsymbol{X})_{v,c}$$

Predict the class c_v^* of vertex v that has the highest predicted probability

Mathematical Formulation (4)

- GCN learned over the **manipulated graph**

$$\boldsymbol{\theta}^{*\prime} = \text{argmin}_{\boldsymbol{\theta}} \mathcal{L}_{train}(\boldsymbol{\theta}; \boldsymbol{A}', \boldsymbol{X}')$$

- GCN's prediction on the **target node v**:

$$c_v^{*\prime} = \text{argmax}_c f_{\boldsymbol{\theta}^{*\prime}}(\boldsymbol{A}', \boldsymbol{X}')_{v,c}$$

- **We want the prediction to change after the graph is manipulated:**

$$c_v^{*\prime} \neq c_v^*$$

Mathematical Formulation (5)

- **Change of prediction on target node v:**

$$\Delta(v; \boldsymbol{A}', \boldsymbol{X}') = \log f_{\boldsymbol{\theta}^{*\prime}}(\boldsymbol{A}', \boldsymbol{X}')_{v, c_v^{*\prime}} - \log f_{\boldsymbol{\theta}^{*\prime}}(\boldsymbol{A}', \boldsymbol{X}')_{v, c_v^{*}}$$

Predicted (log) probability of the newly-predicted class $c_v^{*\prime}$

Predicted (log) probability of the originally-predicted class c_v^{*}

↑ Want to increase this term

↓ Want to decrease this term

Mathematical Formulation (6)

- **Final optimization objective:**

$$\text{argmax}_{A',X'} \Delta(v; A', X')$$
$$\text{subject to } (A', X') \approx (A, X)$$

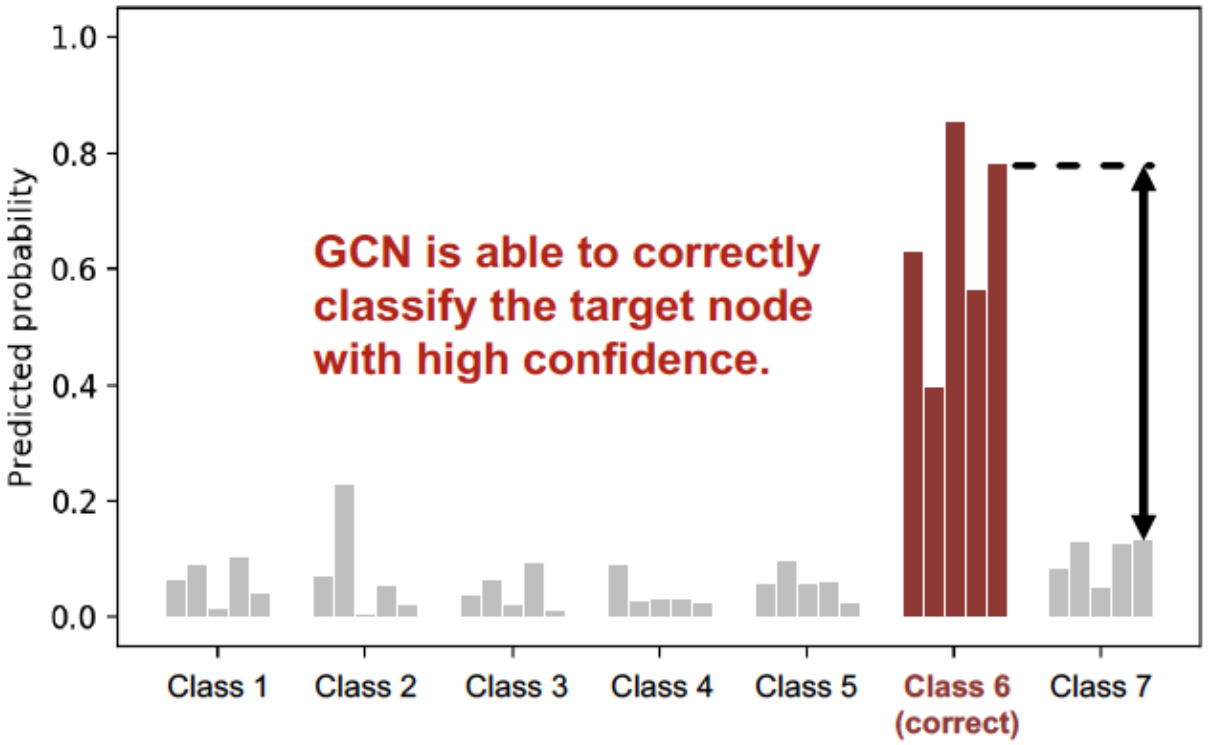
- **Challenges in optimizing the objective**
  - Adjacency matrix A' is a discrete object
  - For every modified graph A' and X', GCN needs to be re-trained: $\theta^{*\prime} = \text{argmin}_{\theta} \mathcal{L}_{train}(\theta; A', X')$
- **Solution [Zügner et al. KDD2018]:**
  - Iteratively follow a locally optimal strategy:
    - Sequentially 'manipulate' the most promising element: an entry from the adjacency matrix or a feature entry
    - Pick the one which obtains the highest difference in the log-probabilites, indicated by the score function.

Experiments: Setting

- Setting: Semi-supervised node classification with GCN
- Graph: Paper citation network (2,800 nodes, 8,000 edges).
- Attack type: Edge modification (addition or deletion of edges)
- Attack budget on node v: $d_v + 2$ modifications (d_v : degree of node v).
  - Intuition: It is harder to attack a node with a larger degree.
- Model is trained and attacked 5 times using different random seeds.

Experiments: Adversarial Attacks

Predicted probabilities of a target node v over 5 re-trainings (each bar represents a single trial)
(without graph manipulation, i.e., clean graph)

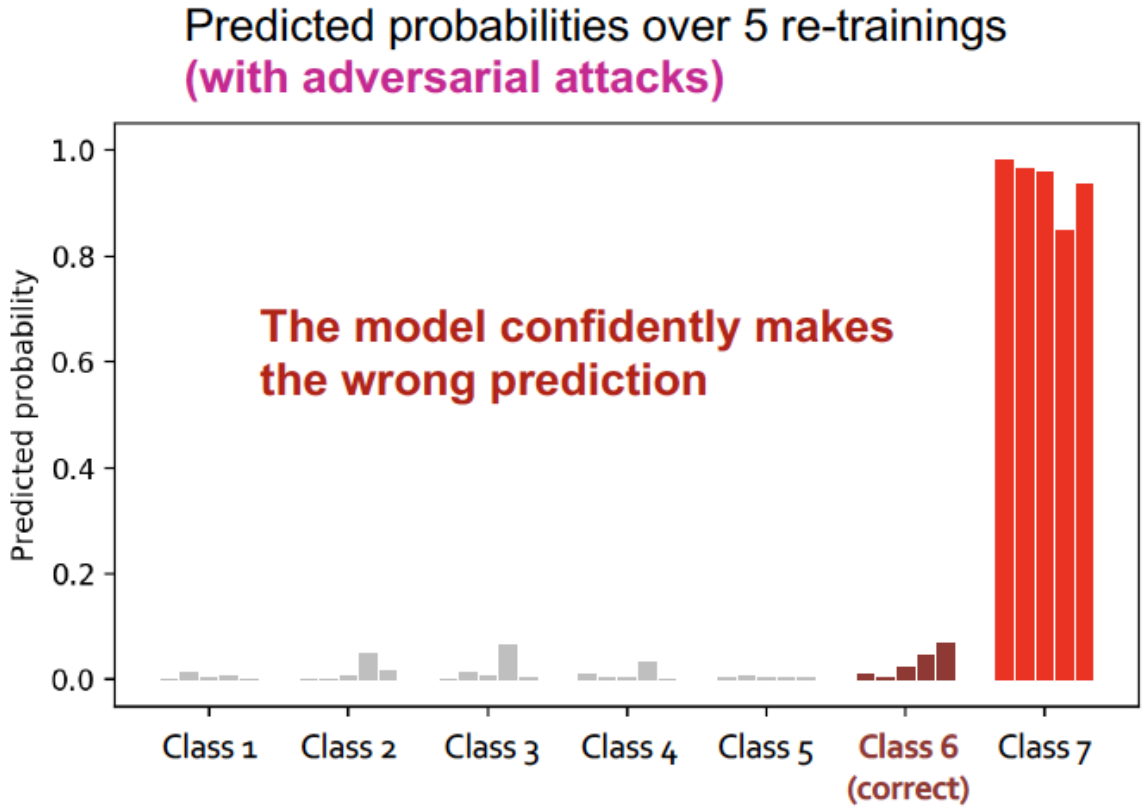


Classification margin
> 0: Correct classification
< 0: Incorrect classification

7-class classification

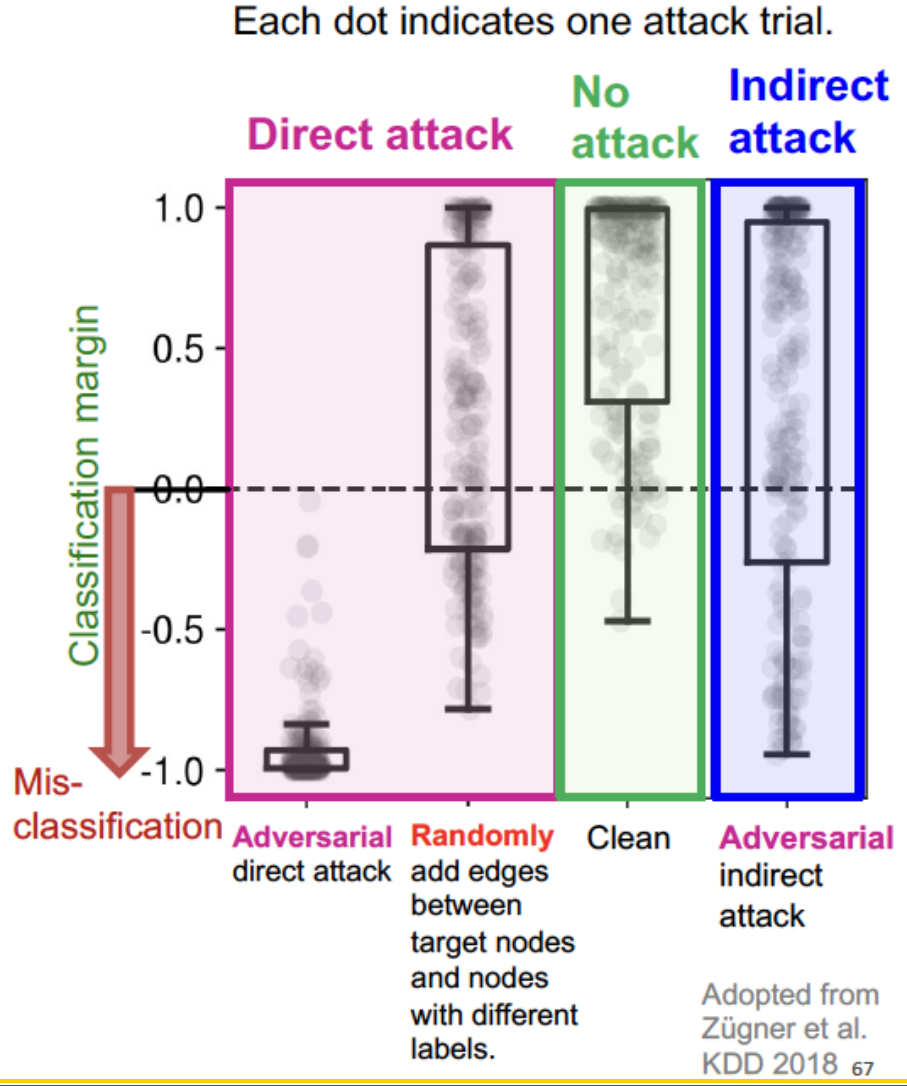
Experiments: Adversarial Attacks

GCN's prediction after modifying 5 edges attached to the target node (direct adversarial attack).



Experiments: Attack Comparison

- Adversarial direct attack is the strongest attack, significantly worsening
- GCN's performance (compared to no attack).
- Random attack is much weaker than adversarial attack.
- Indirect attack is more challenging than direct attack.



Summary

- We study the adversarial robustness of GCN applied to semi-supervised node classification.
- We consider different attack possibilities on graph-structured data.
- We mathematically formulate the adversarial attack as an optimization problem.
- We empirically demonstrate that GCN's prediction performance can be significantly harmed by adversarial attacks.
- GCN is not robust to adversarial attacks but it is somewhat robust to indirect attacks and random noise.