Stanford CS224W: Generative Models for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Motivation for Graph Generation

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

Image credit: Medium

Social Networks

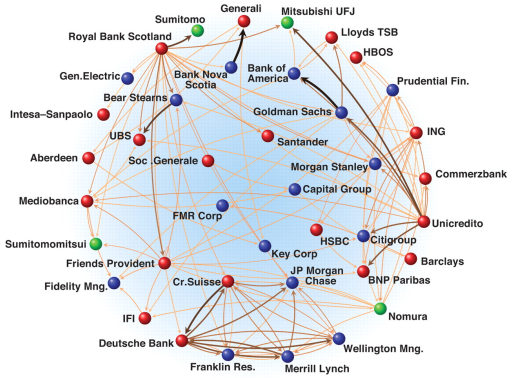


Image credit: Science

Economic Networks

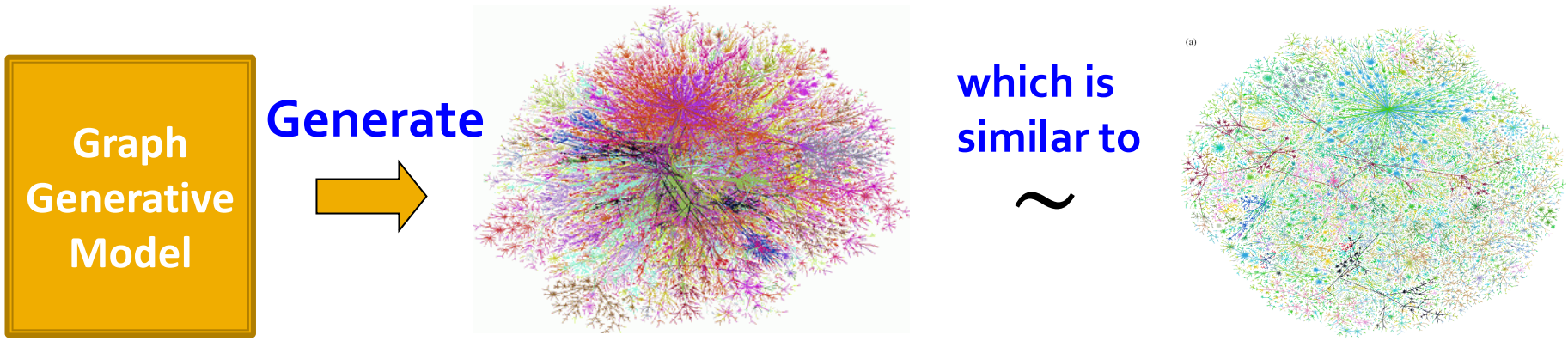
Image credit: Lumen Learning

Communication Networks

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

The Problem: Graph Generation

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



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!**

Stanford CS224W: Properties of Real-world Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Plan: Key Network Properties

We will characterize graphs by:

Degree distribution: $P(k)$

Clustering coefficient: C

Connected components: s

Path length: h

We have introduced these notions in Lecture 1&2. Here we give a quick recap.

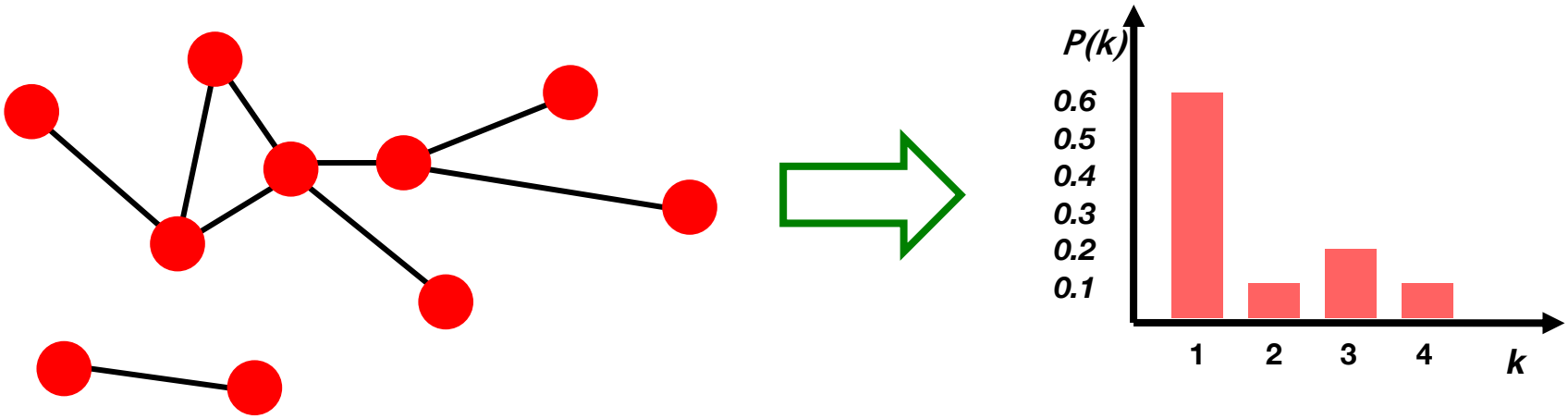
(1) Degree Distribution

- **Degree distribution $P(k)$:** Probability that a randomly chosen node has degree k

  N_k = # nodes with degree k

- Normalized histogram:

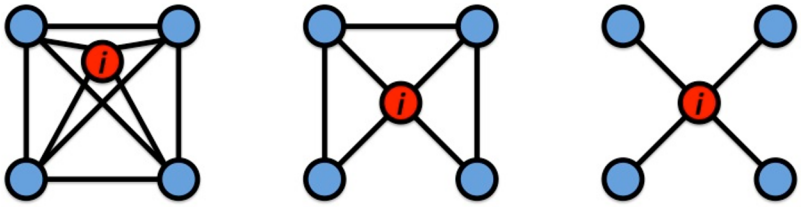
  $P(k) = N_k / N$ → **plot**



(2) Clustering Coefficient

- **Clustering coefficient:**
  - How connected are i's neighbors to each other?
  - Node i with degree k_i
  - $$C_i = \frac{2e_i}{k_i(k_i - 1)}$$

    where e_i is the number of edges between the neighbors of node i



$C_i = 1$ $C_i = 1/2$ $C_i = 0$

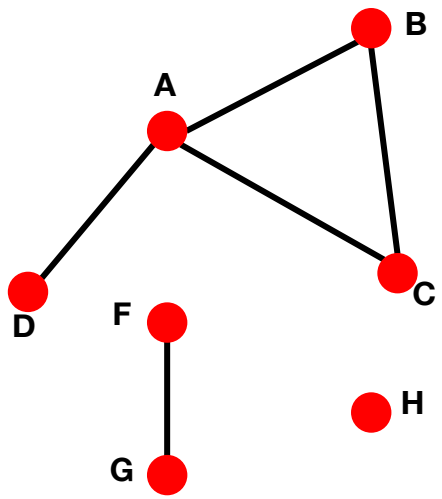
$C_i \in [0,1]$

- **Graph clustering coefficient:**
  - Take average over all the nodes

$$C = \frac{1}{N}\sum_{i}^{N} C_i$$

(3) Connectivity

- **Size of the largest connected component**
  - Largest set where any two vertices can be joined by a path
- **Largest component = Giant component**



How to find connected components:

- Start from random node and perform Breadth First Search (BFS)
- Label the nodes that BFS visits
- If all nodes are visited, the network is connected
- Otherwise find an unvisited node and repeat BFS

(4) Path Length

- **Diameter:** The maximum (shortest path) distance between any pair of nodes in a graph

- **Average path length** for a connected graph or a strongly connected directed graph

$$\bar{h} = \frac{1}{2E_{\max}} \sum_{i,j \neq i} h_{ij}$$

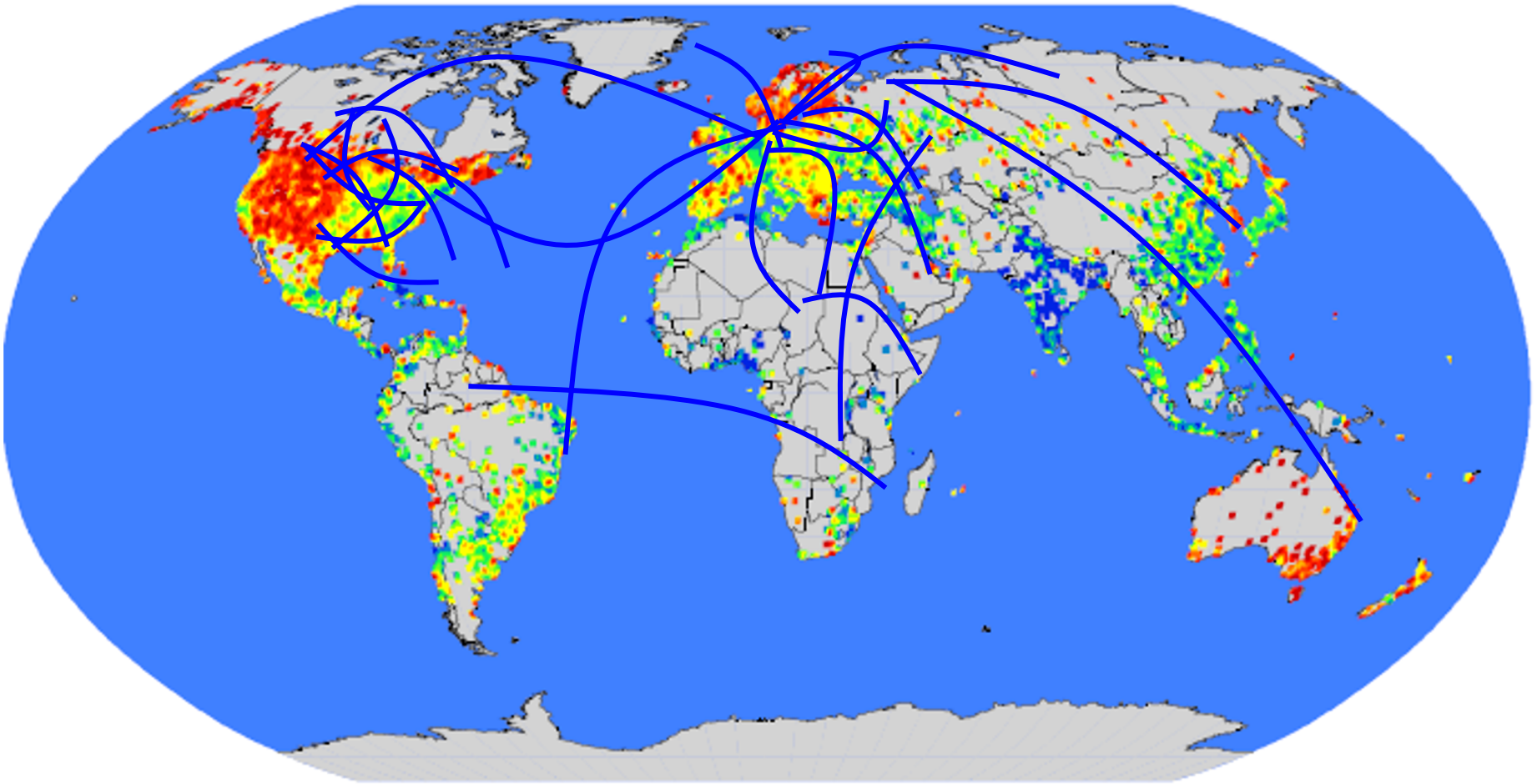
- h_{ij} is the distance from node i to node j
- E_{max} is the max number of edges (total number of node pairs) $= n(n-1)/2$

  - Many times we compute the average only over the connected pairs of nodes (that is, we ignore "infinite" length paths)

Case Study: MSN Graph

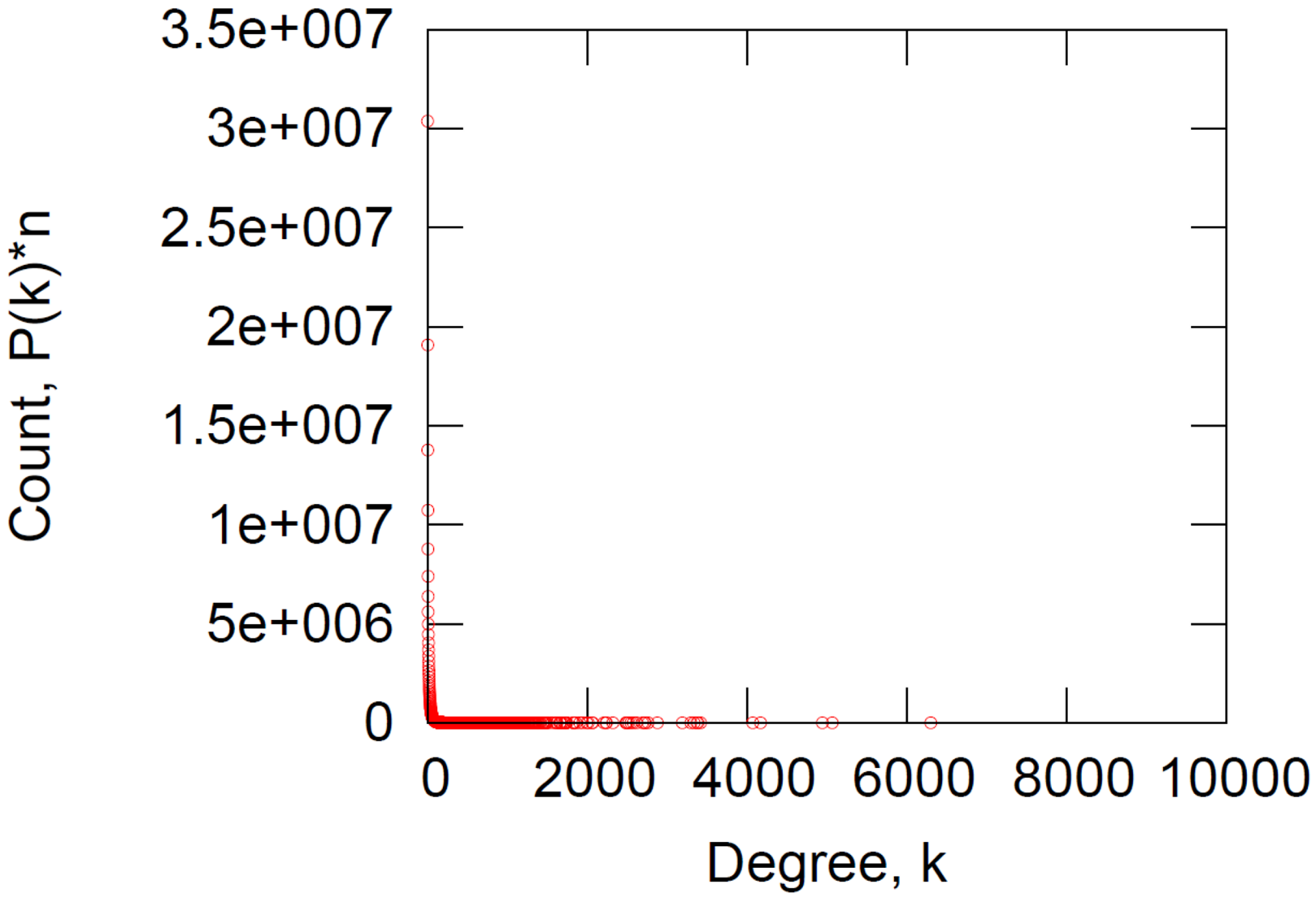
- **MSN Messenger:**
- **1 month of activity**
  - 245 million users logged in
  - 180 million users engaged in conversations
  - More than 30 billion conversations
  - More than 255 billion exchanged messages

Communication Network

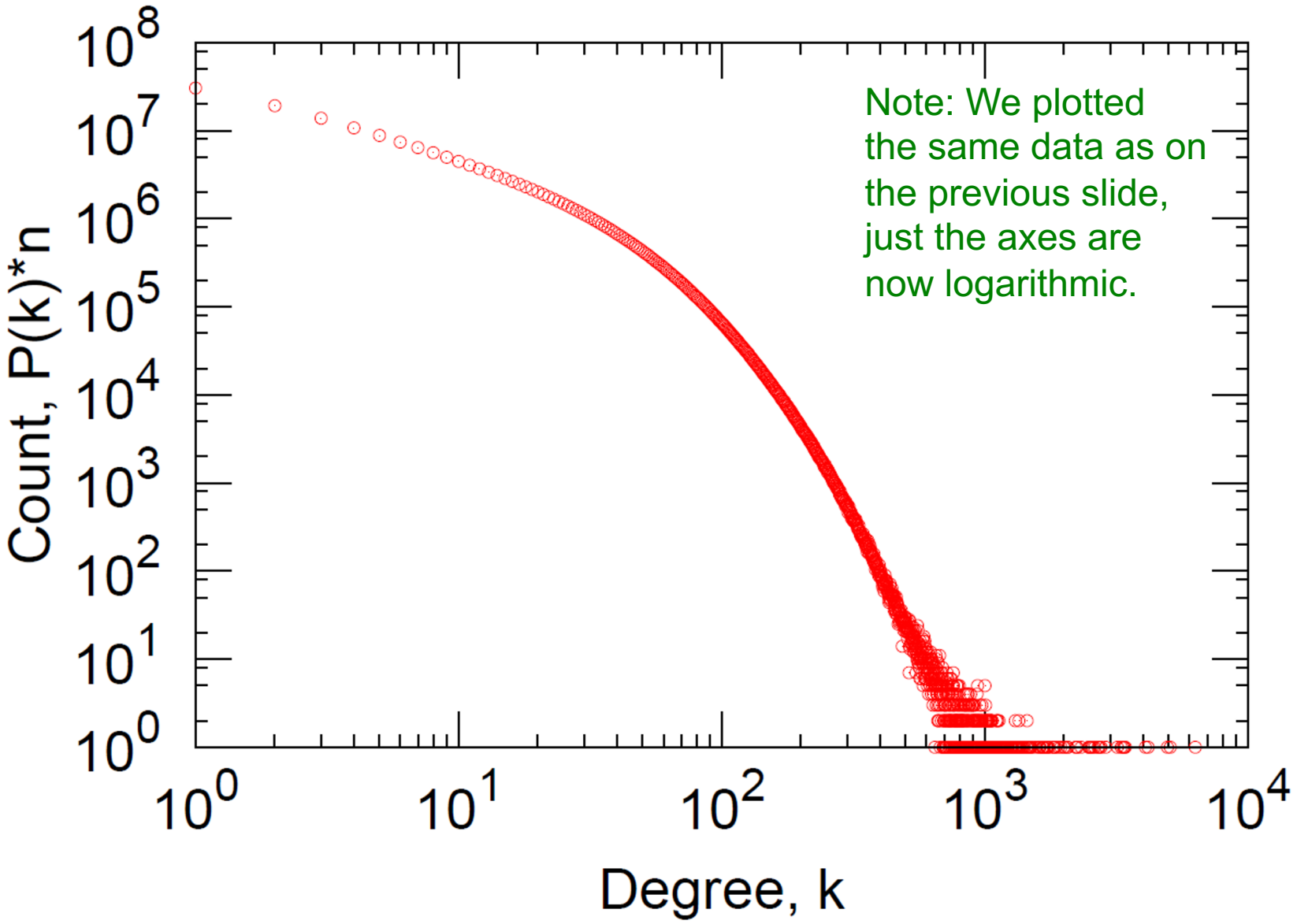


Network: 180M people, 1.3B edges

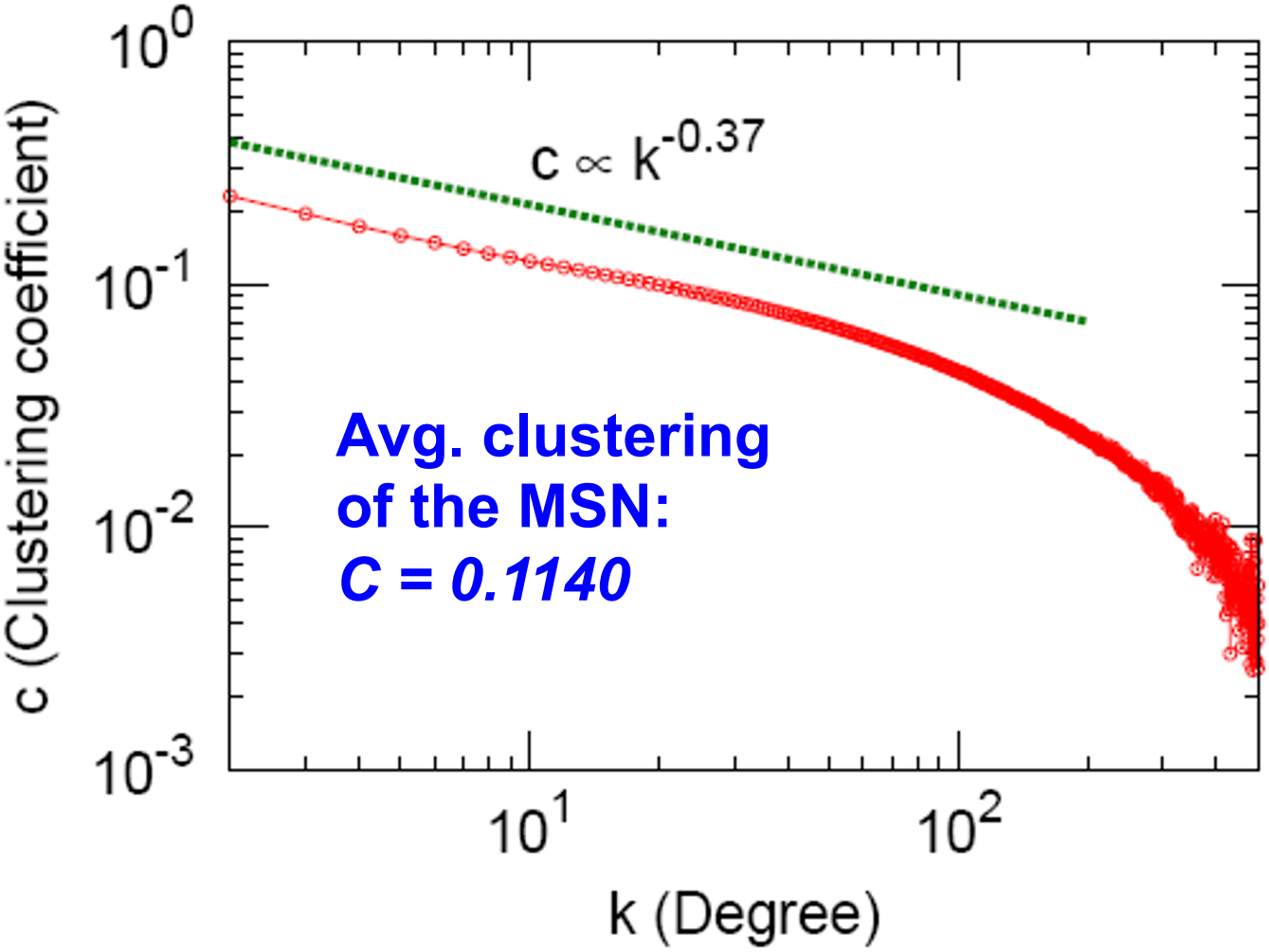
MSN: (1) Degree Distribution



MSN: Log-Log Degree Distribution

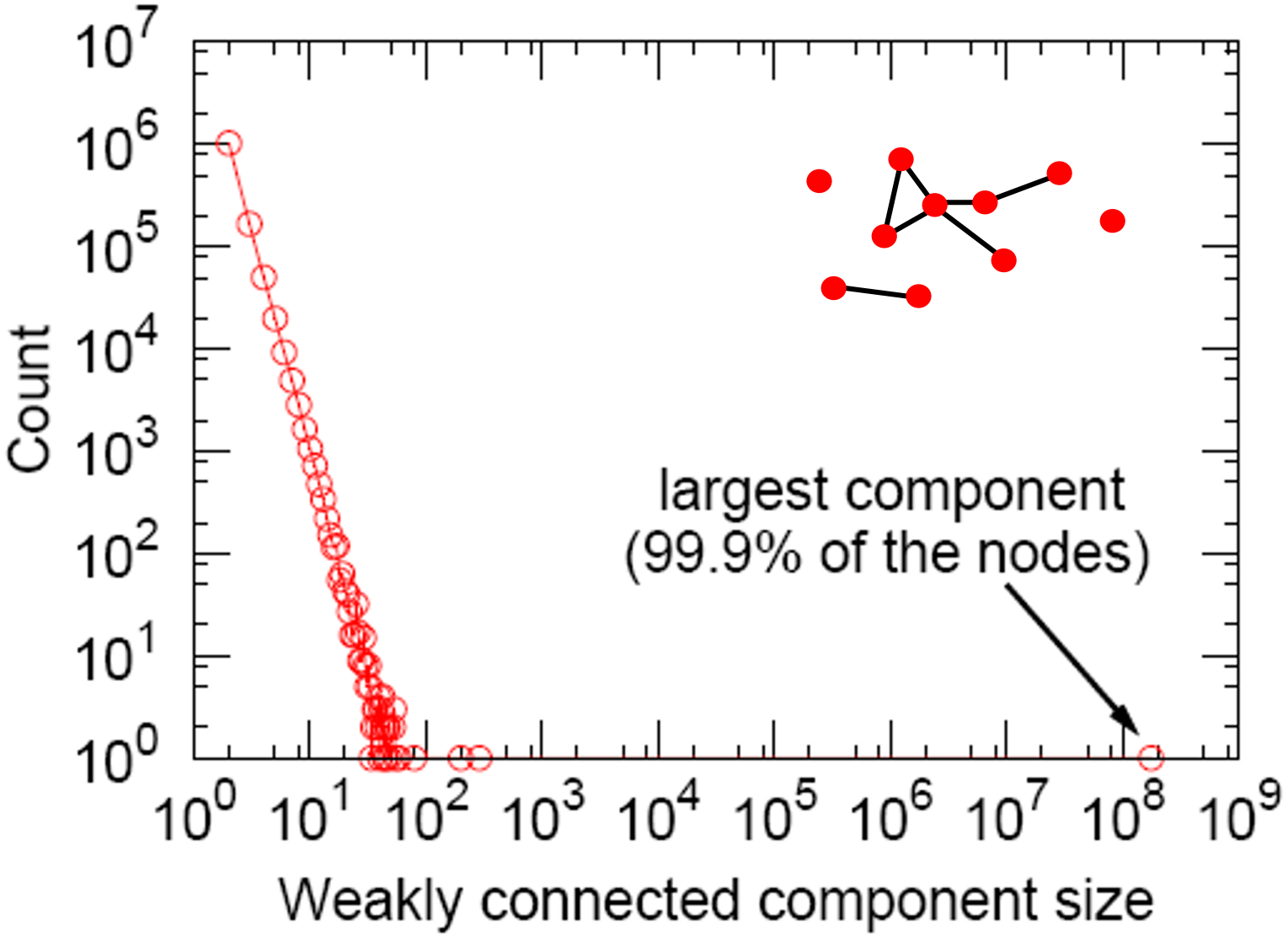


MSN: (2) Clustering

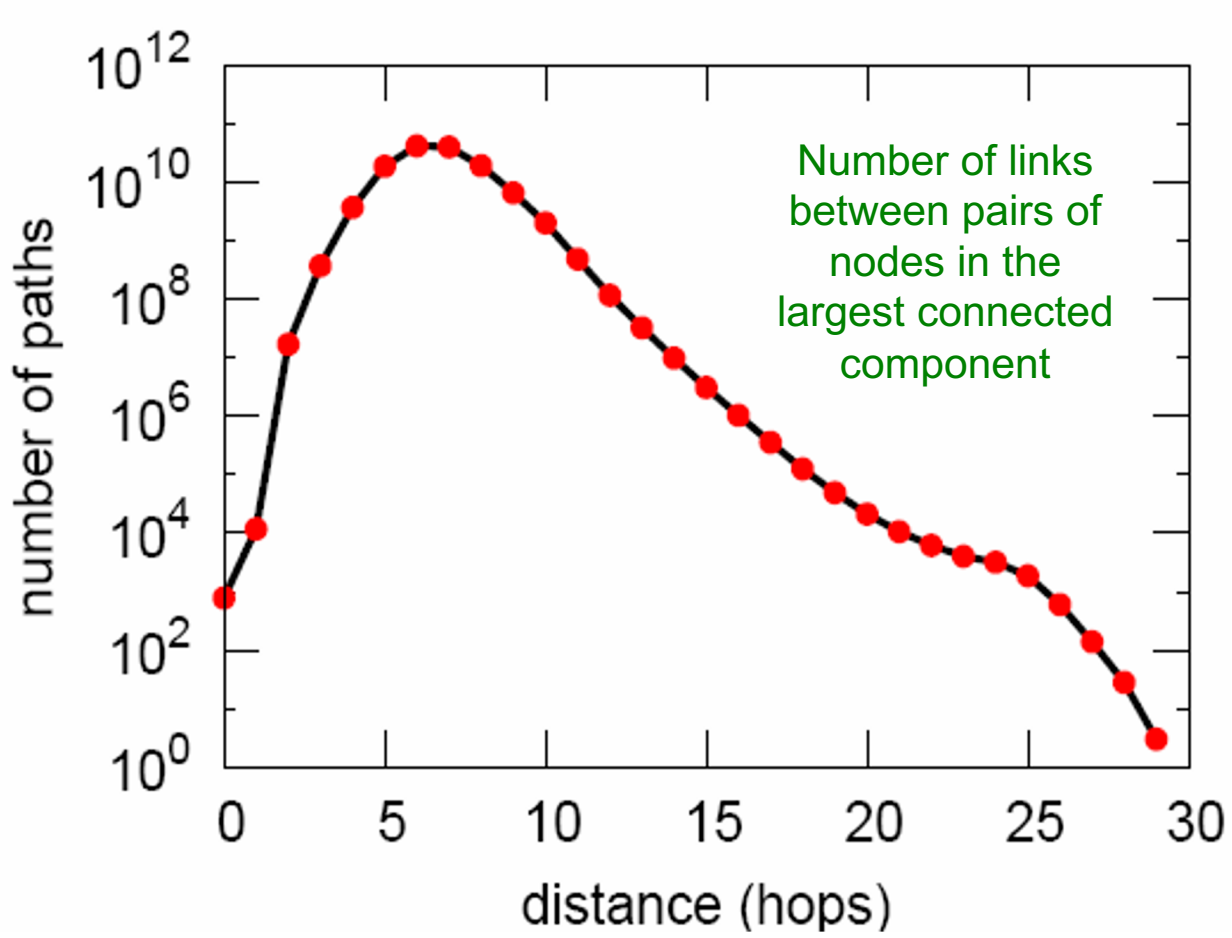


C_k: average C_i of nodes i of degree k: $C_k = \frac{1}{N_k} \sum_{i:k_i=k} C_i$

MSN: (3) Connected Components



MSN: (4) Path Length



Avg. path length **6.6**
90% of the nodes can be reached in < 8 hops

nodes as we do BFS out of a random node

Steps	#Nodes
0	1
1	10
2	78
3	3,96
4	8,648
5	3,299,252
6	28,395,849
7	79,059,497
8	52,995,778
9	10,321,008
10	1,955,007
11	518,410
12	149,945
13	44,616
14	13,740
15	4,476
16	1,542
17	536
18	167
19	71
20	29
21	16
22	10
23	3
24	2
25	3

MSN: Key Network Properties

Degree distribution: *Heavily skewed; avg. degree = 14.4*

Clustering coefficient: *0.11*

Connectivity: *giant component*

Path length: *6.6*

Are these values "expected"?
Are they "surprising"?

To answer this, we need a model!

Stanford CS224W: Erdös-Renyi Random Graphs

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

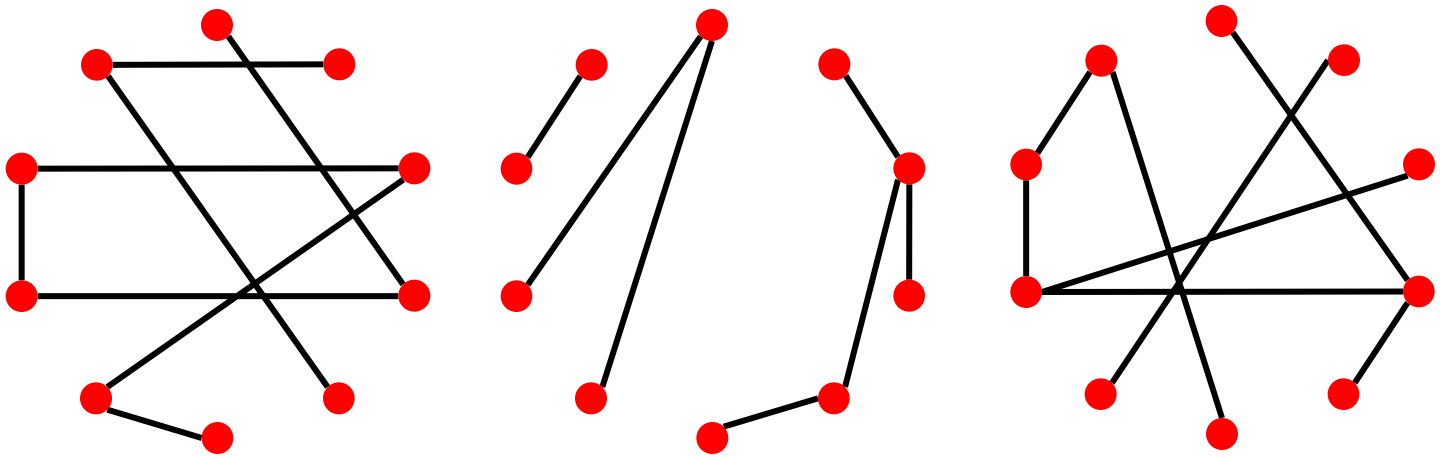
Simplest Model of Graphs

- **Erdös-Renyi Random Graphs** [Erdös-Renyi, '60]
  - **Two variants:**
    - G_{np}: undirected graph on n nodes where each edge (u,v) appears i.i.d. with probability p
    - G_{nm}: undirected graph with n nodes, and m edges picked uniformly at random

What kind of networks do such models produce?

Random Graph Model G_{np}

- ***n* and *p* do not uniquely determine the graph!**
  - The graph is a result of a random process
- We can have many different realizations given the same ***n*** and ***p***



n = 10
p= 1/6

Properties of G_{np}

Degree distribution: $P(k)$

Clustering coefficient: C

Path length: h

What are the values of these properties for G_{np}?

(1) Degree Distribution of G_{np}

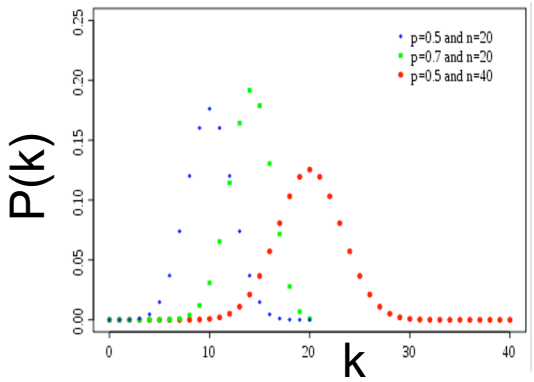
- **Fact:** Degree distribution of G_{np} is binomial.
- Let $P(k)$ denote the fraction of nodes with degree k:

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

Select k nodes out of $n-1$

Probability of having k edges

Probability of missing the rest of the $n-1-k$ edges



Mean, variance of a binomial distribution

$$\bar{k} = p(n-1)$$

$$\sigma^2 = p(1-p)(n-1)$$

(2) Clustering Coefficient of G_{np}

Remember: $C_i = \frac{2e_i}{k_i(k_i-1)}$

Where e_i is the number of edges between i's neighbors

Edges in G_{np} appear i.i.d. with prob. p

So, expected E[e_i] is: $= p\frac{k_i(k_i-1)}{2}$

Each pair is connected with prob. p

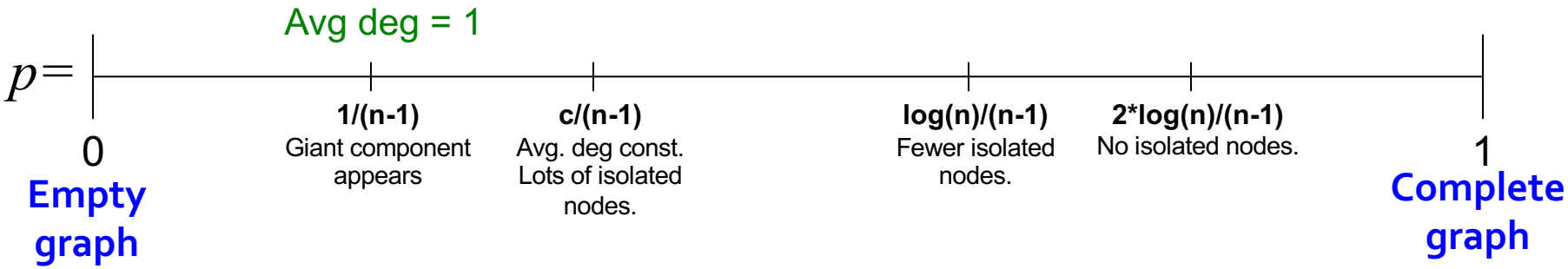
Number of distinct pairs of neighbors of node i of degree k_i

Then $E[C_i]$: $= \frac{p \cdot k_i(k_i-1)}{k_i(k_i-1)} = p = \frac{\bar{k}}{n-1} \approx \frac{\bar{k}}{n}$

Clustering coefficient of a random graph is small.
If we generate bigger and bigger graphs with fixed avg. degree k (that is we set $p = k \cdot 1/n$), then C decreases with the graph size n.

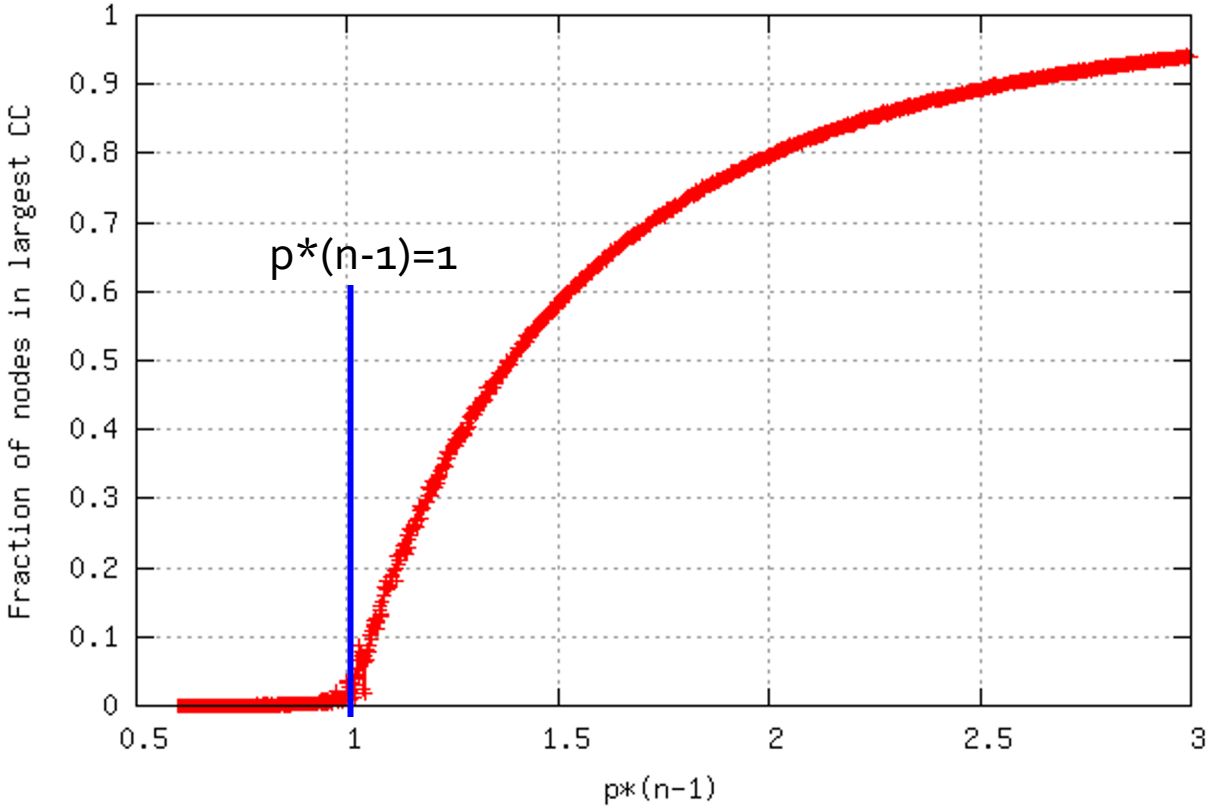
(3) Connected Components of G_{np}

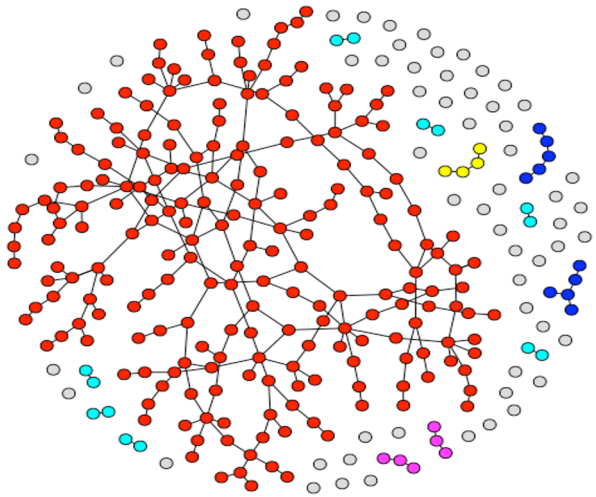
- **Graph structure of G_{np} as p changes:**



- **Emergence of a giant component:**
- avg. degree $k=2E/n$ or $p=k/(n-1)$
  - **Degree $k=1-\varepsilon$:** all components are of size $\Omega(\log n)$
  - **Degree $k=1+\varepsilon$:** 1 component of size $\Omega(n)$, others have size $\Omega(\log n)$
    - Each node has at least one edge in expectation

G_{np} Simulation Experiment





Fraction of nodes in the largest component

G_{np}, n=100,000, $k=p(n-1)$ = 0.5 … 3

Network Properties of G_{np}

Degree distribution: $P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$

Clustering coefficient: $C = p = \bar{k}/n$

Connectivity: GCC exists when k>1.

Path length: ***next!***

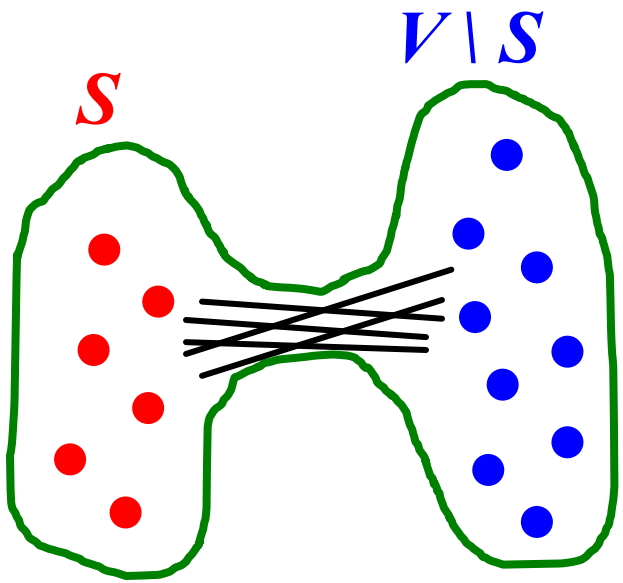
Def: Expansion

- Graph $G(V, E)$ has **expansion α**: if $\forall S \subseteq V$:

  $\#$ of edges leaving $S \geq \alpha \cdot \min(|S|, |V \setminus S|)$

- **Or equivalently:**

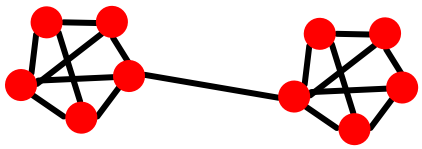
$$\alpha = \min_{S \subseteq V} \frac{\#edges\ leaving\ S}{\min(|S|, |V \setminus S|)}$$



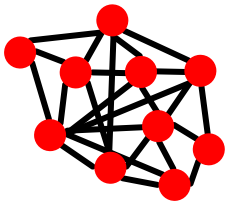
Expansion: Measures Robustness

$$\alpha = \min_{S \subseteq V} \frac{\#\,edges\ leaving\ S}{\min(|S|, |V \setminus S|)}$$

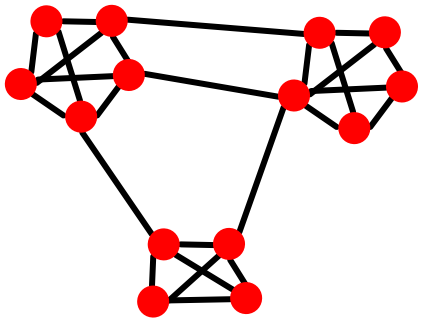
- Expansion is **measure of robustness:**
  - To disconnect l nodes, we need to cut $\geq \alpha \cdot l$ edges
- Low expansion:



- High expansion:

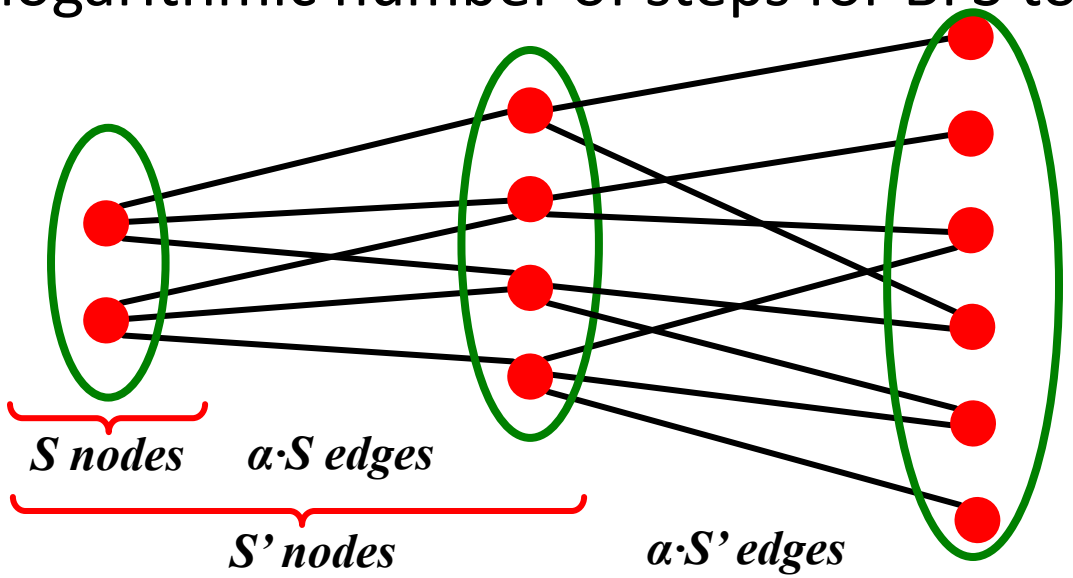


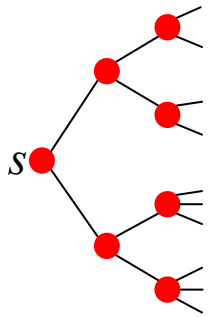
- Social networks:
  - "Communities"



Expansion: Random Graphs

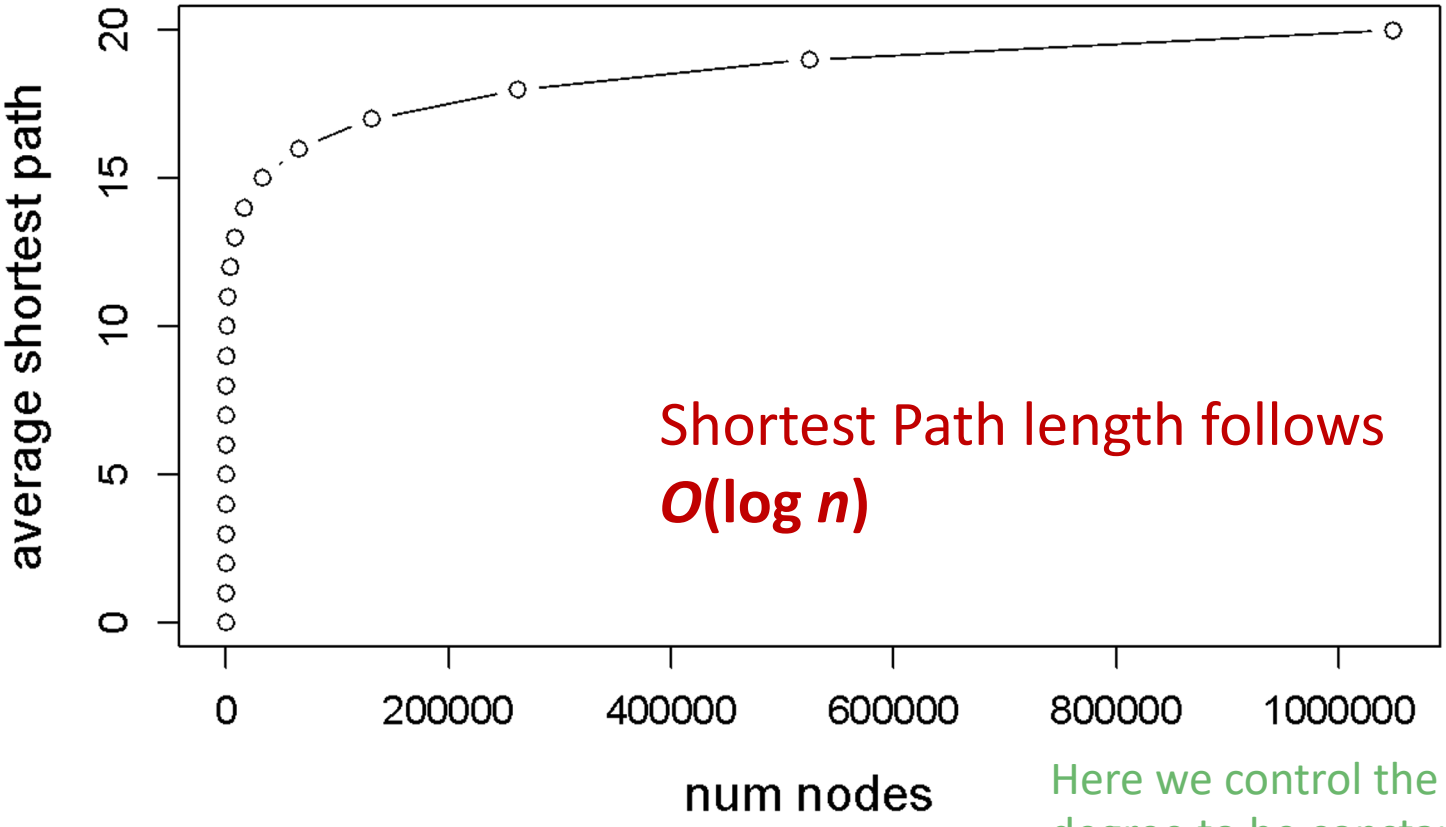
- **Fact:** In a graph on n nodes with expansion α for all pairs of nodes there is a path of lehth $O((\log n)/\alpha)$.
- **Random graph G_{np}:**
  For $\log n > np > c$, diam(G_{np}) = $O(\log n / \log (np))$
  - Random graphs have good expansion so it takes a logarithmic number of steps for BFS to visit all nodes



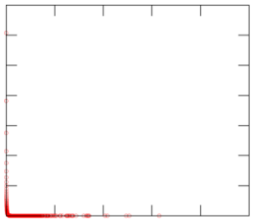
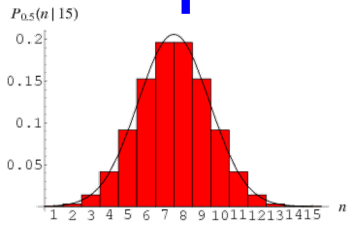


(4) Shortest Path of G_{np}

Erdös-Renyi Random Graph can grow very large but nodes will be just a few hops apart



Back to MSN vs. G_{np}

	MSN	G_{np} n=180M	
Degree distribution:			✗
Avg. path length:	**6.6**	$O(\log n)$ *h ≈ 8.2*	✓
Avg. clustering coef.:	*0.11*	$\bar{k} / n$ *C ≈ 8·10⁻⁸*	✗
Largest Conn. Comp.:	*99%*	GCC exists when $\bar{k}>1$. $\bar{k} \approx 14$.	✓

Real Networks vs. G_{np}

- **Are real networks like random graphs?**
  - Giant connected component: ☺️
  - Average path length: ☺️
  - Clustering Coefficient: ☹️
  - Degree Distribution: ☹️
- **Problems with the random networks model:**
  - Degree distribution differs from that of real networks
  - Giant component in most real networks does NOT emerge through a phase transition
  - No local structure – clustering coefficient is too low
- **Most important: Are real networks random?**
  - The answer is simply: **NO!**

Stanford CS224W: The Small-World Model

CS224W: Machine Learning with Graphs

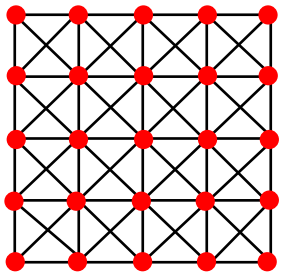
Jure Leskovec, Stanford University

http://cs224w.stanford.edu

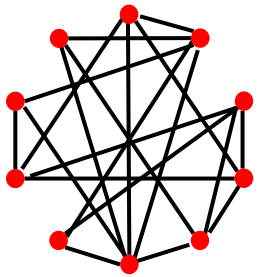
Motivation for Small-World

	MSN	G_{np} n=180M	
Avg. path length:	**6.6**	$O(\log n)$ $h \approx 8.2$	☑
Avg. clustering coef.:	0.11	$\bar{k} / n$ $C \approx 8 \cdot 10^{-8}$	☒

Can we have **high clustering** while also having **short paths**?



Vs.



Regular lattice graph:
High clustering coefficient
High diameter

G_{np} random graph:
Low clustering coefficient
Low diameter

Clustering Implies Edge Locality

- **Real networks have high clustering:**
  - **MSN network has 7 orders of magnitude larger clustering than the corresponding G_{np}!**
- **Other examples:**

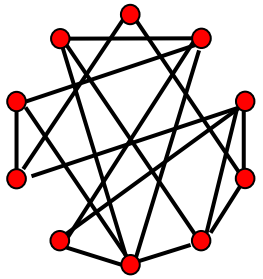
Avg. path length Clustering coefficients

Network	Nodes	Degree	h_{actual}	h_{random}	C_{actual}	C_{random}
Film actor collaborations	225,226	61.00	3.65	2.99	0.79	0.00027
Power Grid	4,941	2.67	18.70	12.40	0.080	0.005
C. elegans	282	14.00	2.65	2.25	0.28	0.05

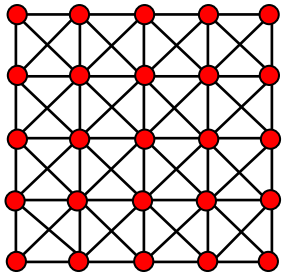
"actual" … real network
"random" … random graph with same avg. degree

The "Controversy"

- **Consequence of expansion:**
  - **Short paths: $O(\log n)$**
    - This is the smallest diameter we can get if we keep the degree constant.
  - But clustering is low!
- **But networks have "local" structure:**
  - **Triadic closure:**
    Friend of a friend is my friend
  - High clustering but diameter is also high
- **How can we have both?**



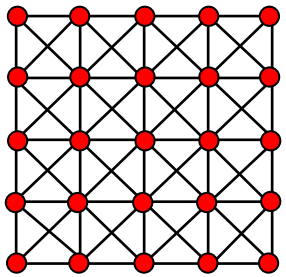
G_{np} random graph:
Low clustering coefficient
Low diameter



Regular lattice graph:
High clustering coefficient
High diameter

Small-world Graphs: Idea

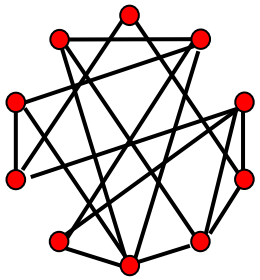
- **Idea:** Interpolate between regular lattice graphs and G_{np} random graph



Regular lattice graph:
High clustering coefficient
High diameter

Interpolate

Small-world graph:
High clustering coefficient
Low diameter



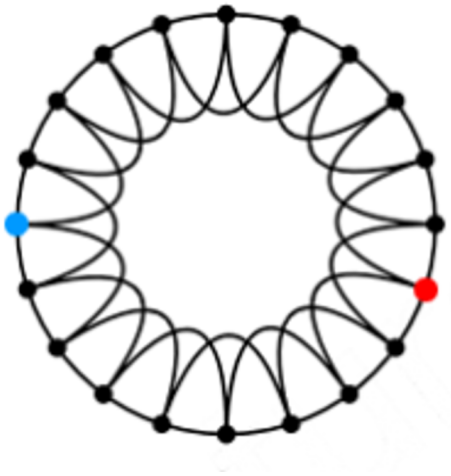
G_{np} random graph:
Low clustering coefficient
Low diameter

- **How do we interpolate between these two graphs?**

[D. Wattts and S. Strogatz, Collective dynamics of 'small-world'networks, Nature]

Solution: The Small-World Model

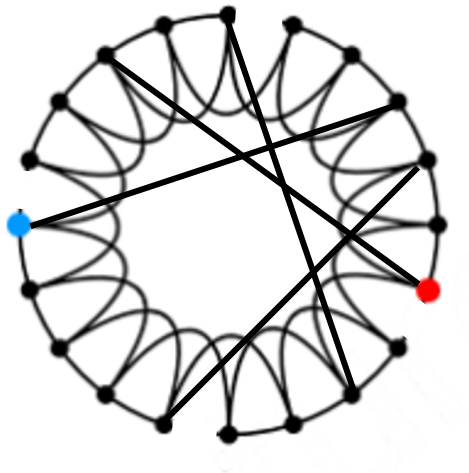
- **Small-World Model**
- Two components to the model:
- **(1) Start with a low-dimensional regular lattice**
  - (In our case we are using a ring as a lattice)
  - Has high clustering coefficient



[D. Wattts and S. Strogatz, Collective dynamics of 'small-world'networks, Nature]

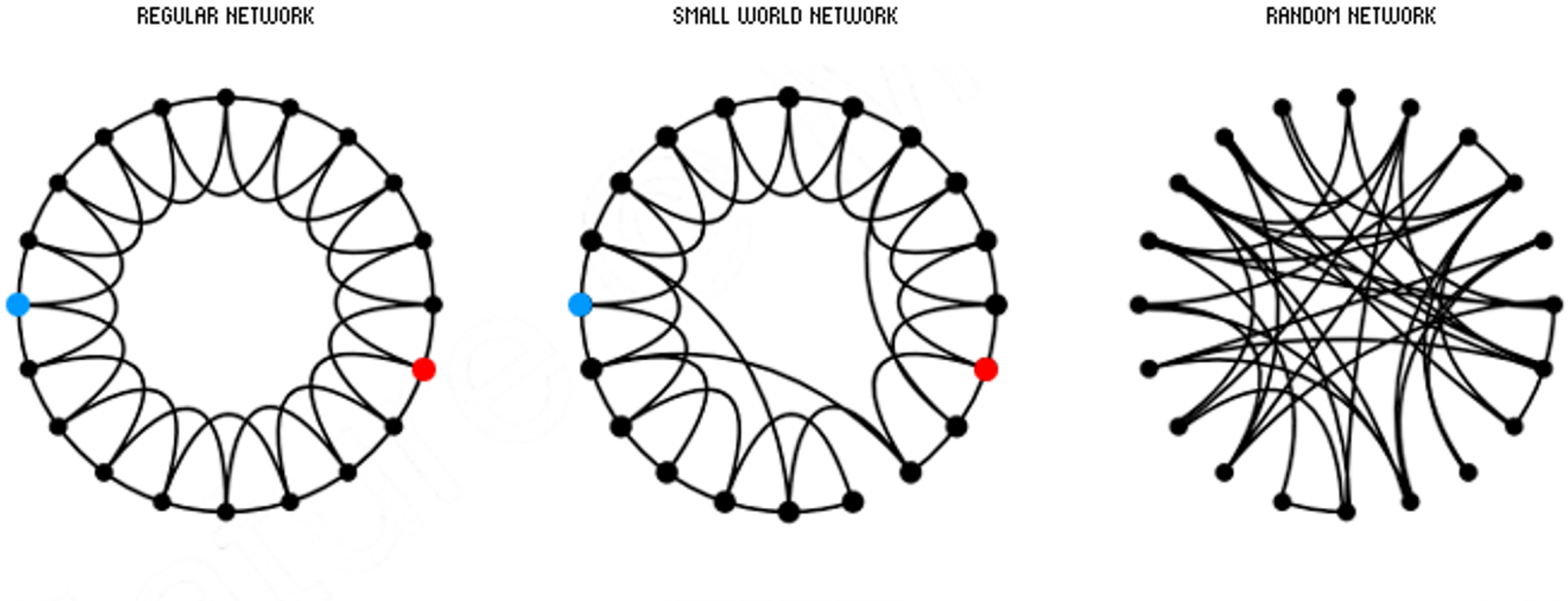
Solution: The Small-World Model

- **Small-World Model**
- Two components to the model:
- **(2) Rewire: Introduce randomness ("shortcuts")**
  - Add/remove edges to create shortcuts to join remote parts of the lattice
  - For each edge, with prob. p, move the other endpoint to a random node



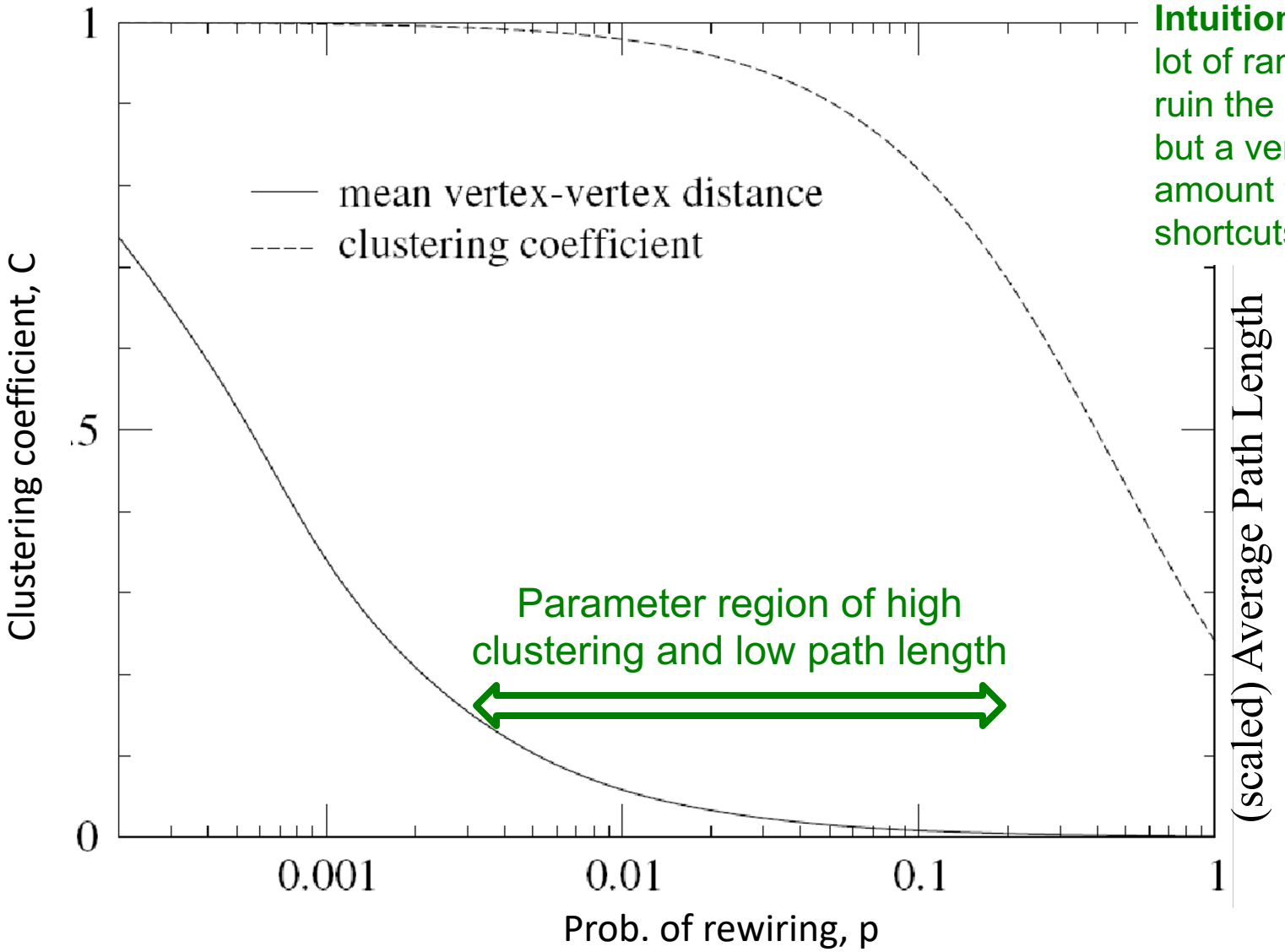
[D. Wattts and S. Strogatz, Collective dynamics of 'small-world'networks, Nature]

The Small-World Model



Rewiring allows us to "interpolate" between a regular lattice and a random graph

The Small-World Model



Intuition: It takes a lot of randomness to ruin the clustering, but a very small amount to create shortcuts.

Small-World: Summary

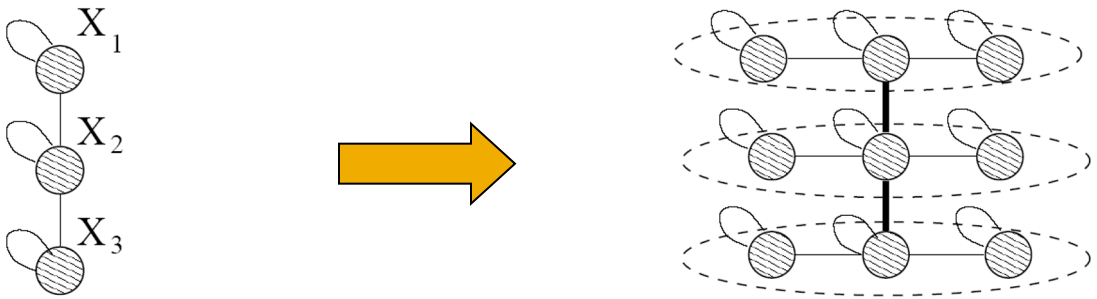
- **Could a network with high clustering be at the same time a small world?**
  - Yes! You don't need more than a few random links
- **The Small-World Model:**
  - Provides insight on the interplay between clustering and the small-world
  - Captures the structure of many realistic networks
  - Accounts for the high clustering of real networks
  - Does not lead to the correct degree distribution

Stanford CS224W: Kronecker Graph Model

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

Idea: Recursive Graph Generation

- **How can we think of network structure recursively? Intuition:** Self-similarity
  - **Object is similar to a part of itself**: the whole has the same shape as one or more of the parts
- **Mimic recursive graph/community growth:**



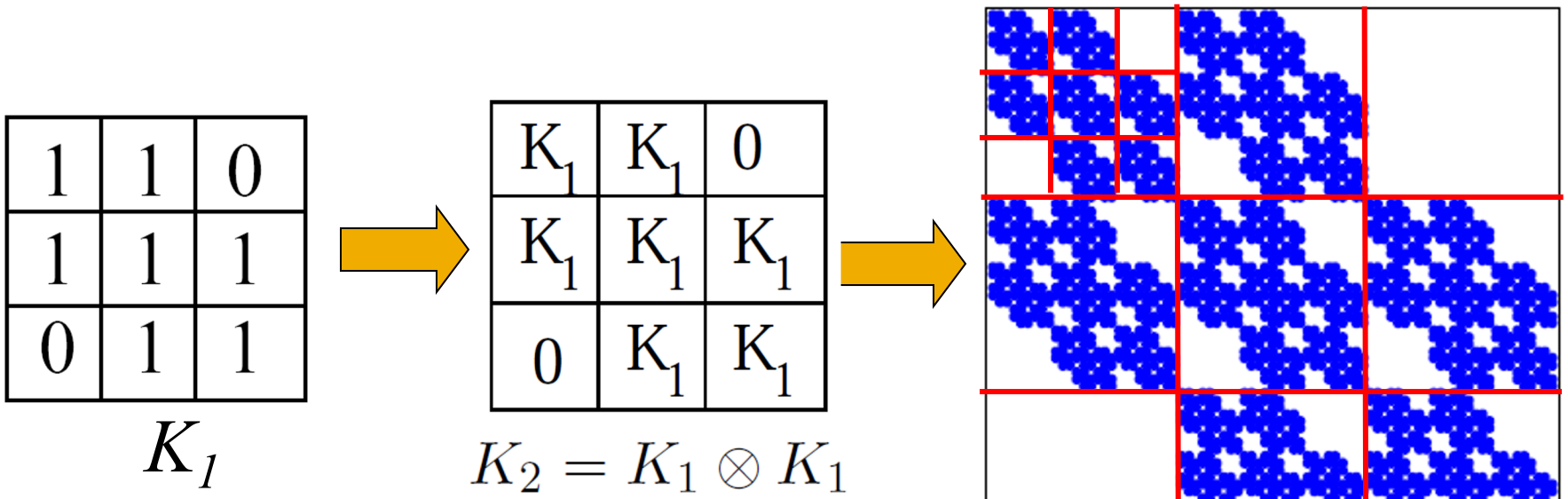
Initial graph | Recursive expansion

- **Kronecker product** is a way of generating self-similar matrices

[PKDD '05]

Kronecker Graph

- **Kronecker graphs:**
  - **A recursive model of network structure**



Kronecker Product: Definition

- **Kronecker product** of matrices *A* and *B* is given by

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \doteq \begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \dots & a_{1,m}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \dots & a_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\mathbf{B} & a_{n,2}\mathbf{B} & \dots & a_{n,m}\mathbf{B} \end{pmatrix}$$

N x M *K x L*

*N*K x M*L*

- Define a Kronecker product of two graphs as a Kronecker product of their **adjacency matrices**

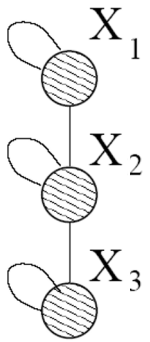
[PKDD '05]

Kronecker Graphs

- **Kronecker graph** is obtained by growing sequence of graphs by iterating the **Kronecker product over the initiator matrix K_1:**

$$K_1^{[m]} = K_m = \underbrace{K_1 \otimes K_1 \otimes \ldots K_1}_{m \text{ times}} = K_{m-1} \otimes K_1$$

1	1	0
1	1	1
0	1	1

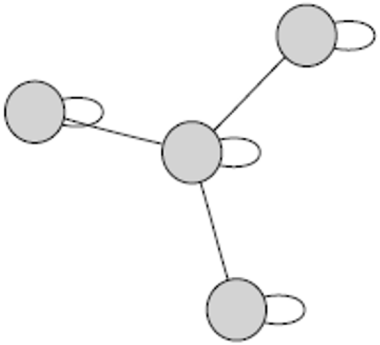


K_1

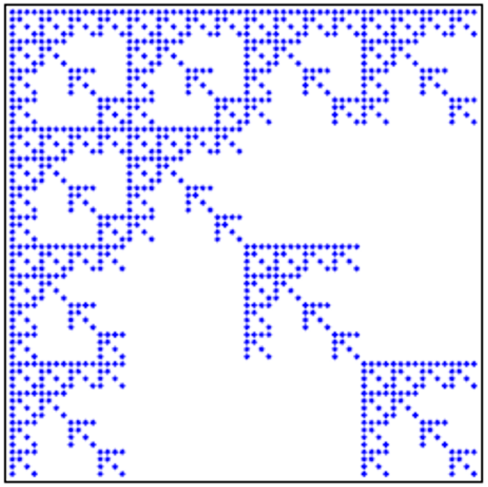
- **Note:** One can easily use multiple initiator matrices (K_1', K_1'', K_1''') (even of different sizes)

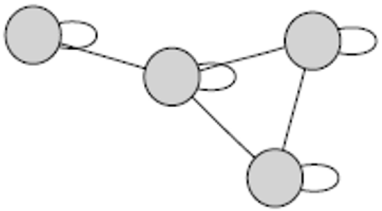
[PKDD '05]

Kronecker Initiator Matrices

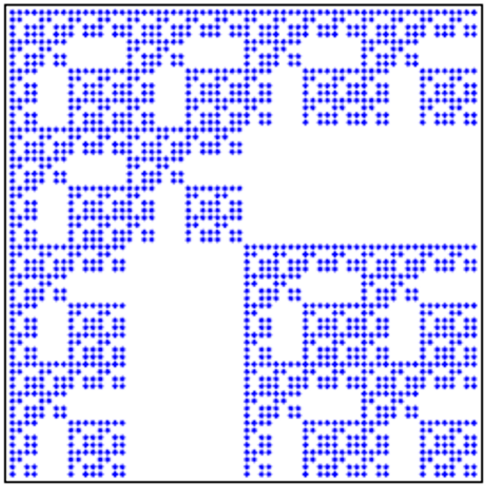


1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1





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

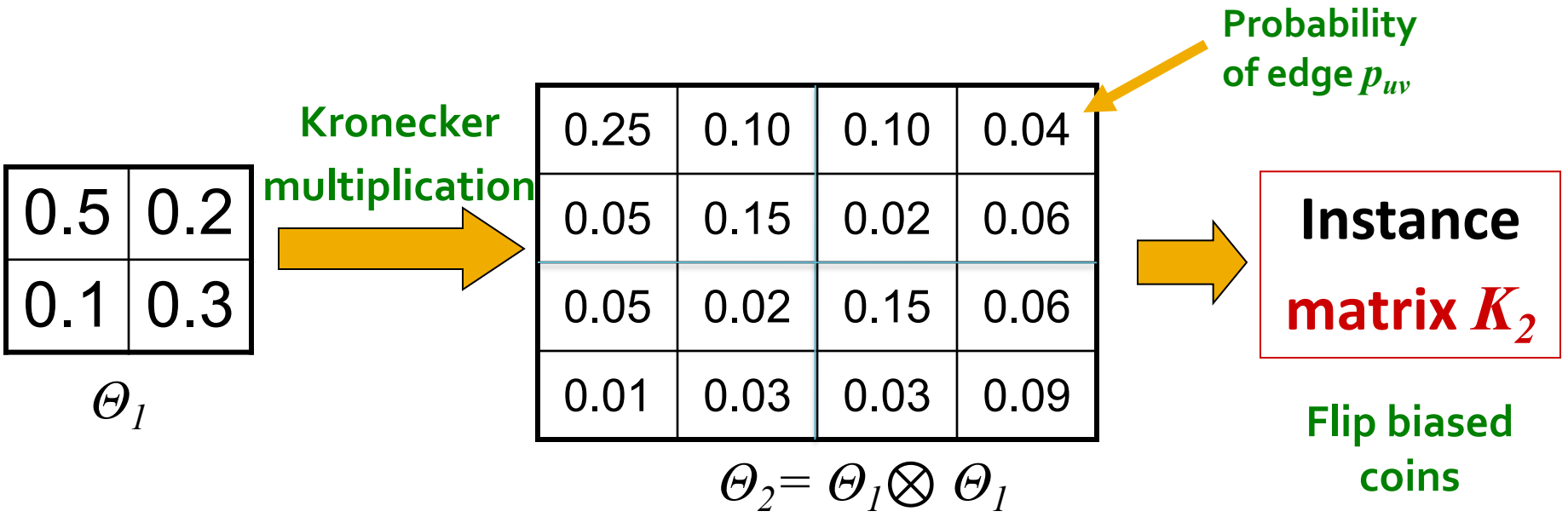


Initiator K_1 | K_1 adjacency matrix | K_3 adjacency matrix

[PKDD '05]

Stochastic Kronecker Graphs

- **Step 1:** Create N_1 x N_1 **probability matrix** Θ_1
- **Step 2:** Compute the k^{th} **Kronecker power** Θ_k
- **Step 3:** For each entry p_{uv} of Θ_k include an edge *(u, v)* in K_k with probability p_{uv}



Generation of Kronecker Graphs

- **How do we generate an instance of a (Directed) stochastic Kronecker graph?**

Probability of edge p_{uv}

0.25	0.10	0.10	0.04
0.05	0.15	0.02	0.06
0.05	0.02	0.15	0.06
0.01	0.03	0.03	0.09

Flip biased coins

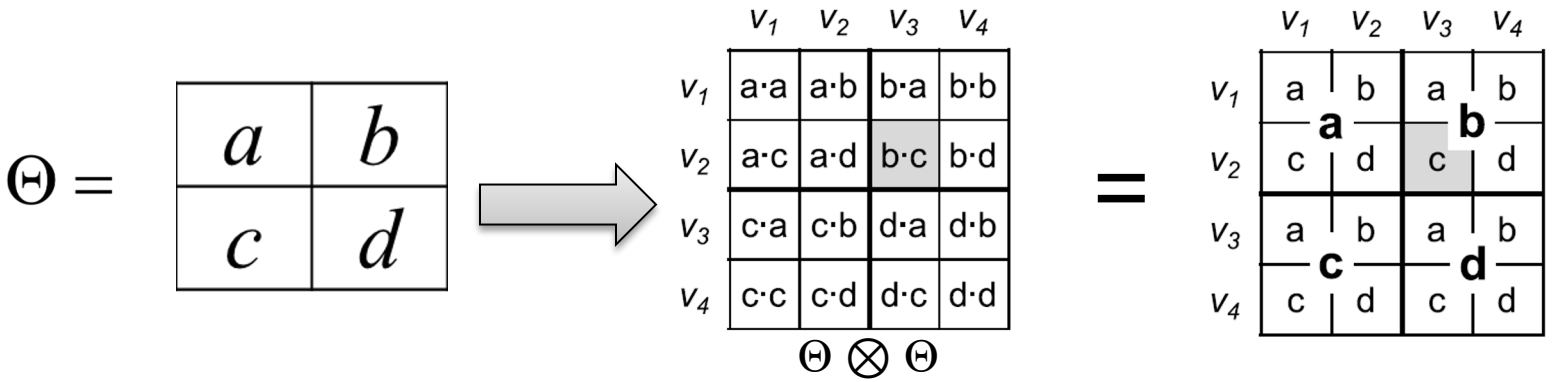
1	1	0	0
0	1	0	1
1	0	1	1
0	1	0	1

Need to flip n^2 coins!! Way too slow!!

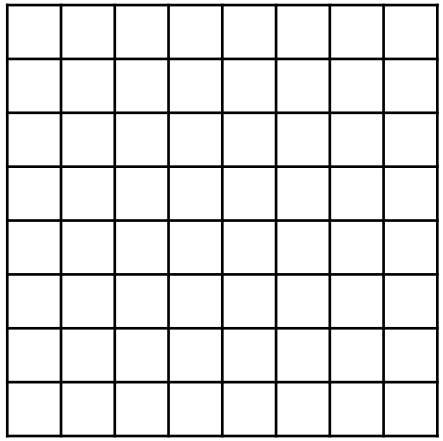
- **Is there a faster way? YES!**
- **Idea:** Exploit the recursive structure of Kronecker graphs
  - "Drop" edges onto the graph one by one

Generation of Kronecker Graphs

- **A faster way to generate Kronecker graph**



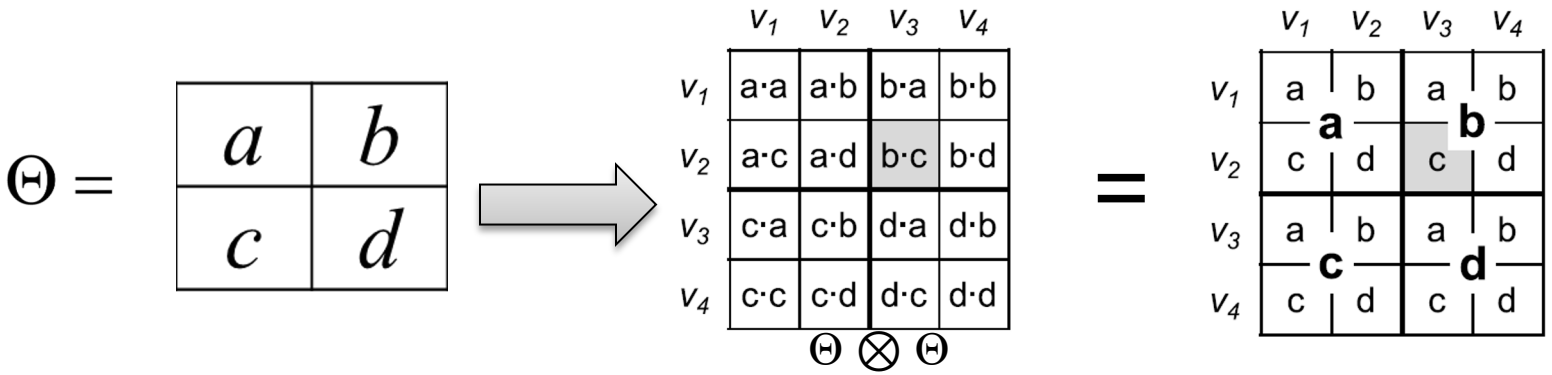
- **How to "drop" an edge into a graph G on $n=2^m$ nodes**



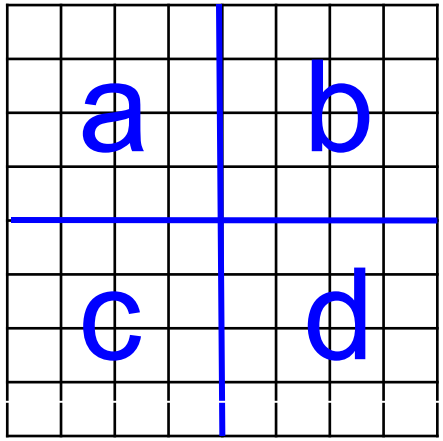
Adjacency matrix **G**

Generation of Kronecker Graphs

- **A faster way to generate Kronecker graph**



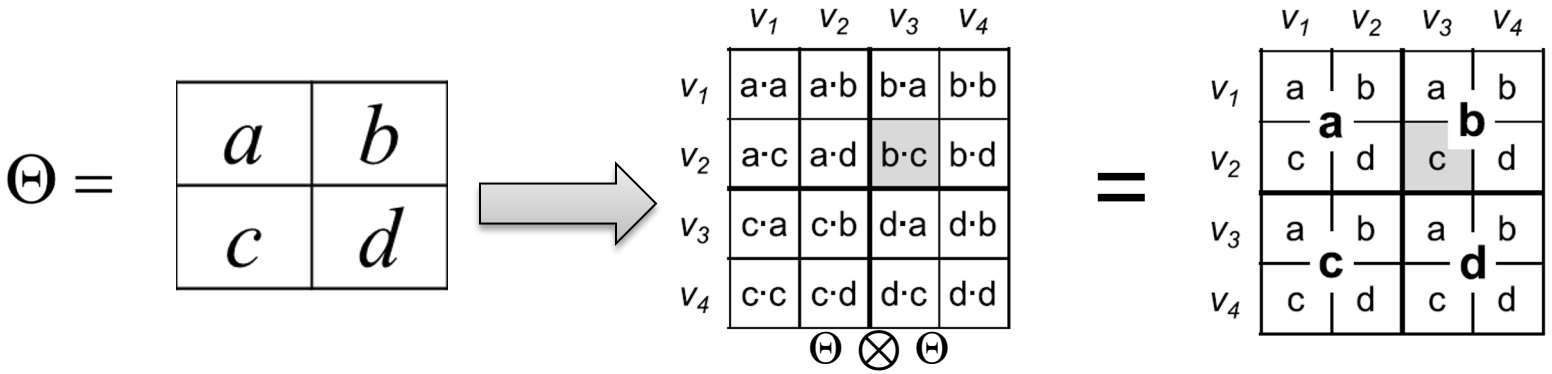
- **How to "drop" an edge into a graph G on $n=2^m$ nodes**



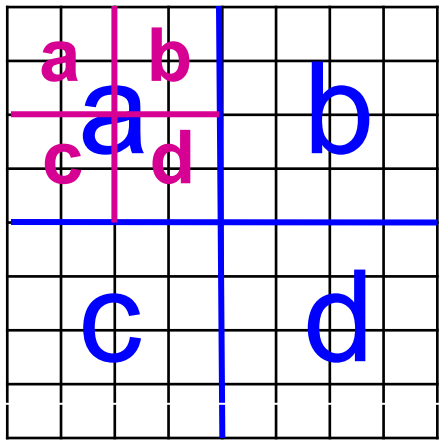
Adjacency matrix **G**

Generation of Kronecker Graphs

- **A faster way to generate Kronecker graph**



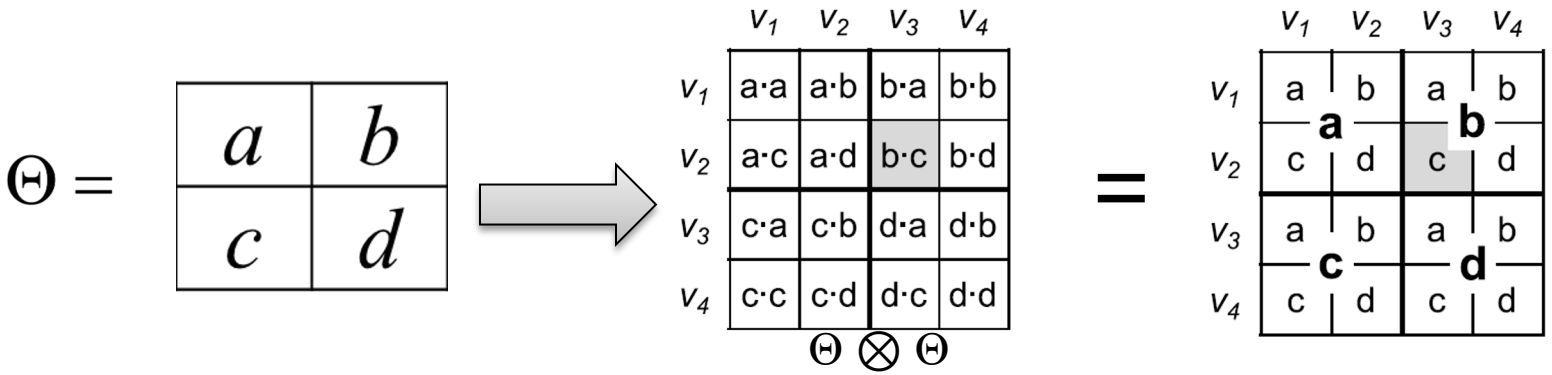
- **How to "drop" an edge into a graph G on $n=2^m$ nodes**



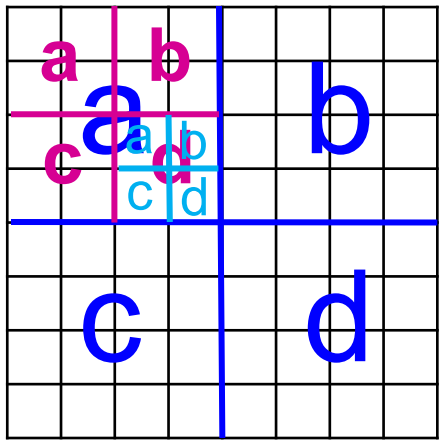
Adjacency matrix **G**

Generation of Kronecker Graphs

- **A faster way to generate Kronecker graph**



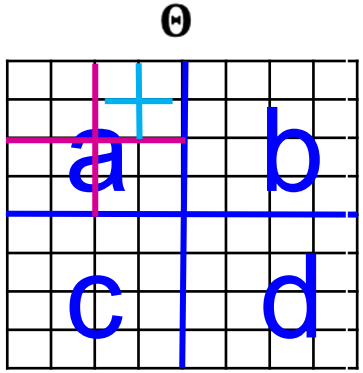
- **How to "drop" an edge into a graph G on $n=2^m$ nodes:**
- We may get a few edges colliding. We simply reinsert them.



Adjacency matrix **G**

Generation of Kronecker Graphs

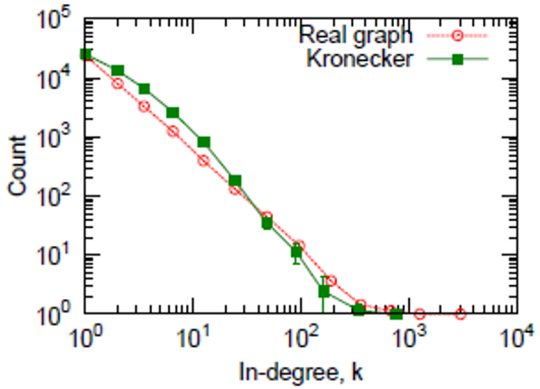
- **Fast Kronecker generator algorithm:**
  - For generating **directed graphs**
- **Insert 1 edge on graph G on $n = 2^m$ nodes:**
  - Create normalized matrix $\boldsymbol{L_{uv} = \Theta_{uv}/(\sum_{op} \Theta_{op})}$
  - **For $\boldsymbol{i = 1 \quad \ldots m}$**
    - Start with $\boldsymbol{x = 0, y = 0}$
    - Pick a row/column $\boldsymbol{(u, v)}$ with prob. $\boldsymbol{L_{uv}}$
    - Descend into quadrant $\boldsymbol{(u, v)}$ at level $\boldsymbol{i}$ of $\boldsymbol{G}$
      - **This means:** $\boldsymbol{x \mathrel{+}= u \cdot 2^{m-i}, y \mathrel{+}= v \cdot 2^{m-i}}$
  - Add an edge $\boldsymbol{(x, y)}$ **to** $\boldsymbol{G}$



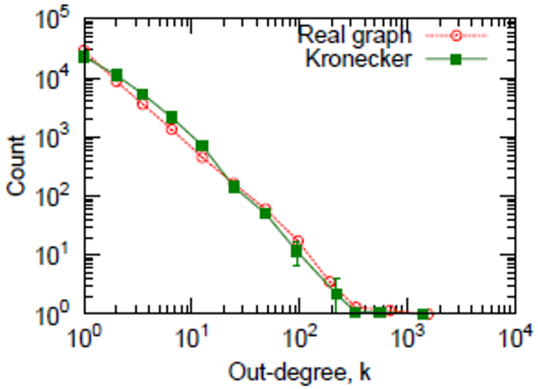
Estimation: Epinions (n=76k, m=510k)

- **Real** and **Kronecker** are very close:

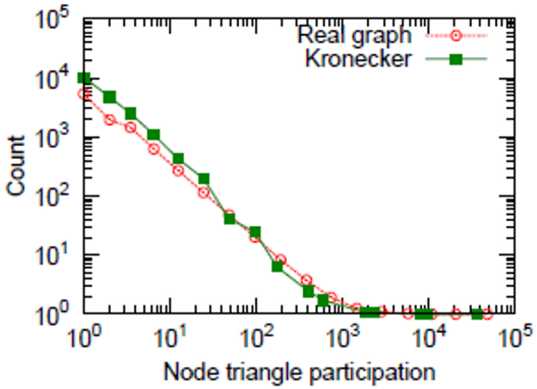
$$\Theta_1 = \begin{bmatrix} 0.99 & 0.54 \\ 0.49 & 0.13 \end{bmatrix}$$



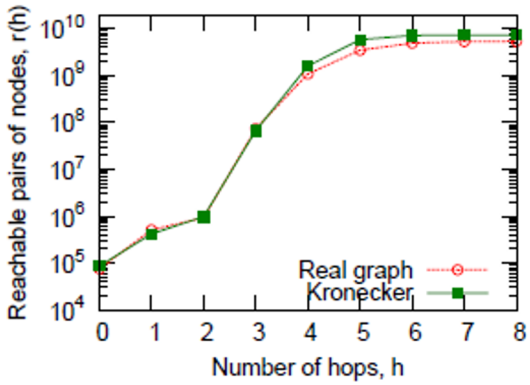
(a) In-Degree



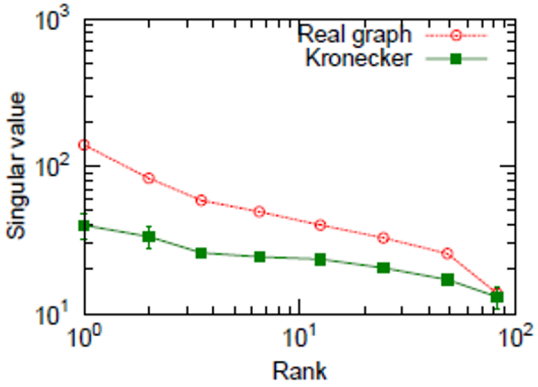
(b) Out-degree



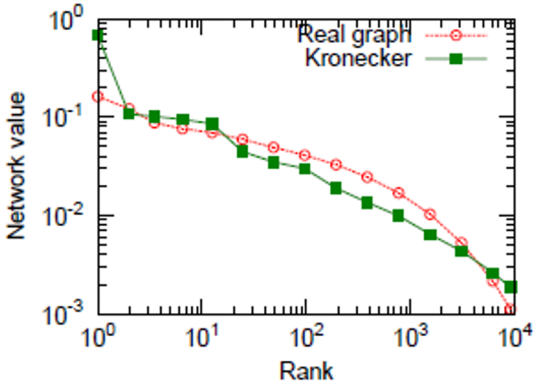
(c) Triangle participation



(d) Hop plot



(e) Scree plot



(f) "Network" value

Summary of the Lecture

- **Today: Traditional graph generative models**
  - Erdös-Renyi graphs
  - Small-world graphs
  - Kronecker graphs
  - All these models have **prior assumption of the graph generation processes**
- **Next: Deep graph generative models**
  - **Learn** the graph generation process **from raw data**

Stanford CS224W: Deep Generative Models for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Motivation for Graph Generation

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

Image credit: Medium

Social Networks

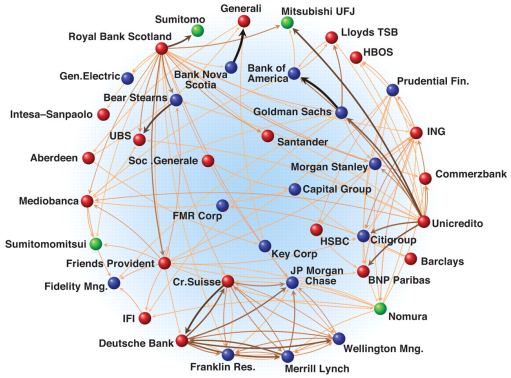


Image credit: Science

Economic Networks

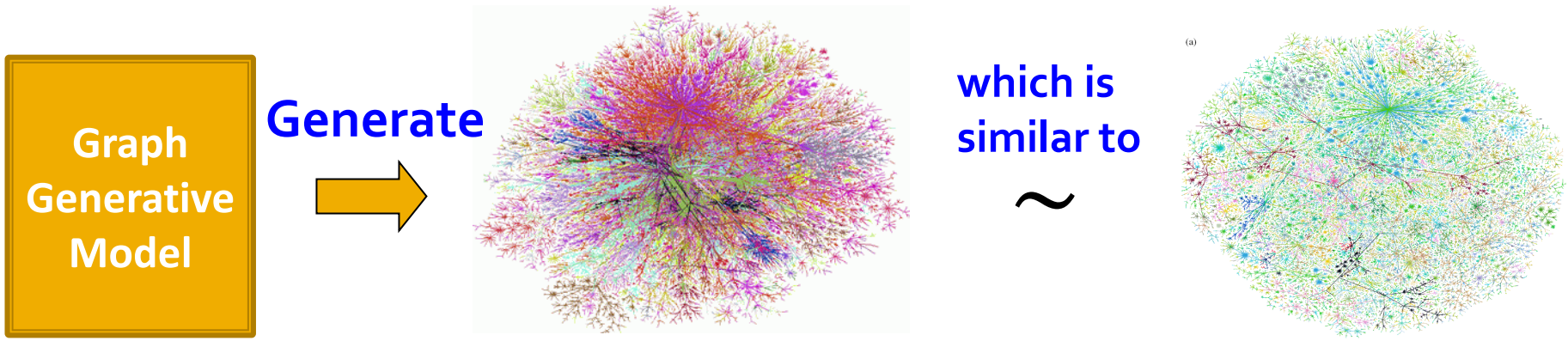
Image credit: Lumen Learning

Communication Networks

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

Recap: Graph Generation Problem

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



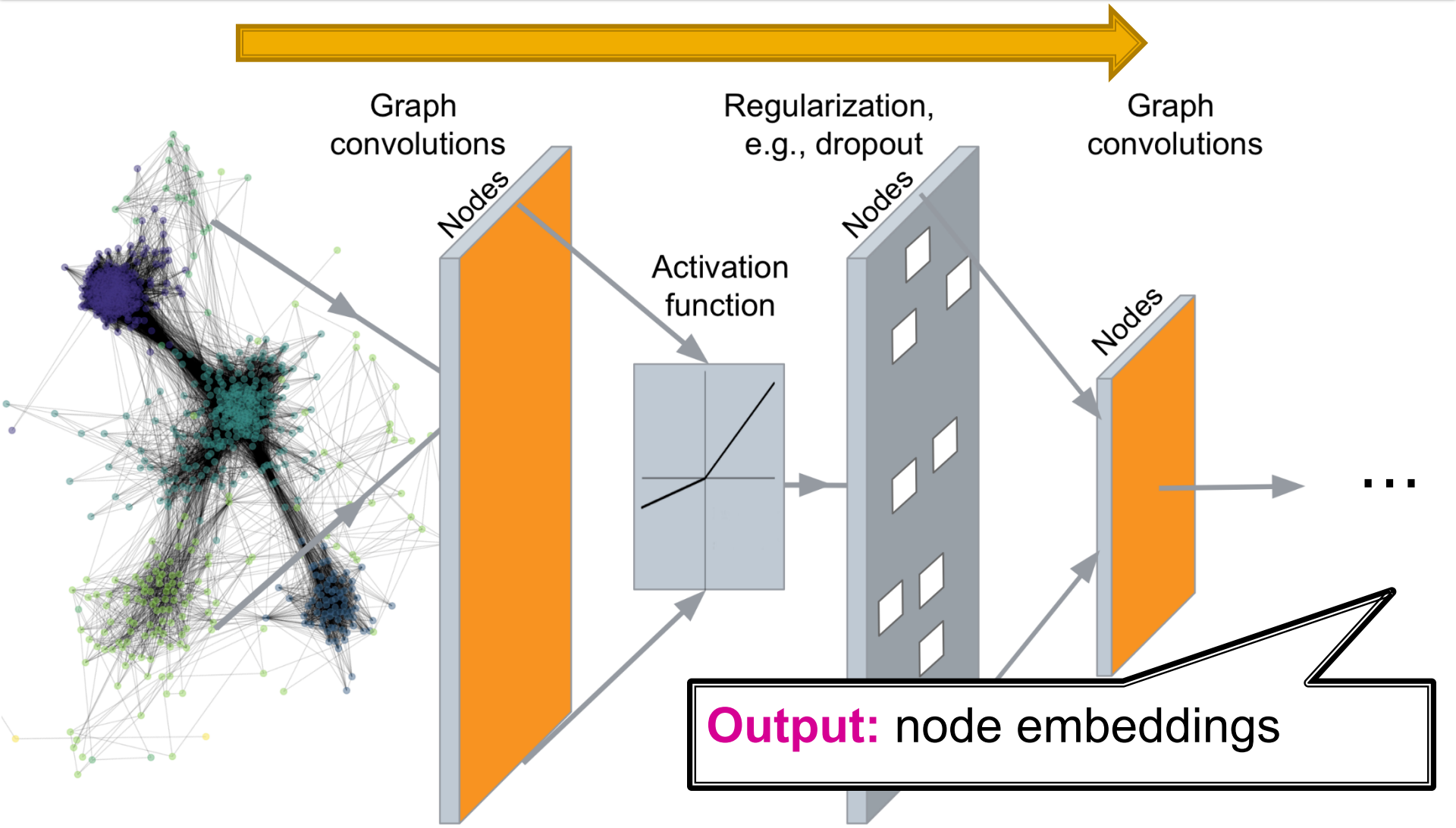
Why Do We Study Graph Generation

- **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

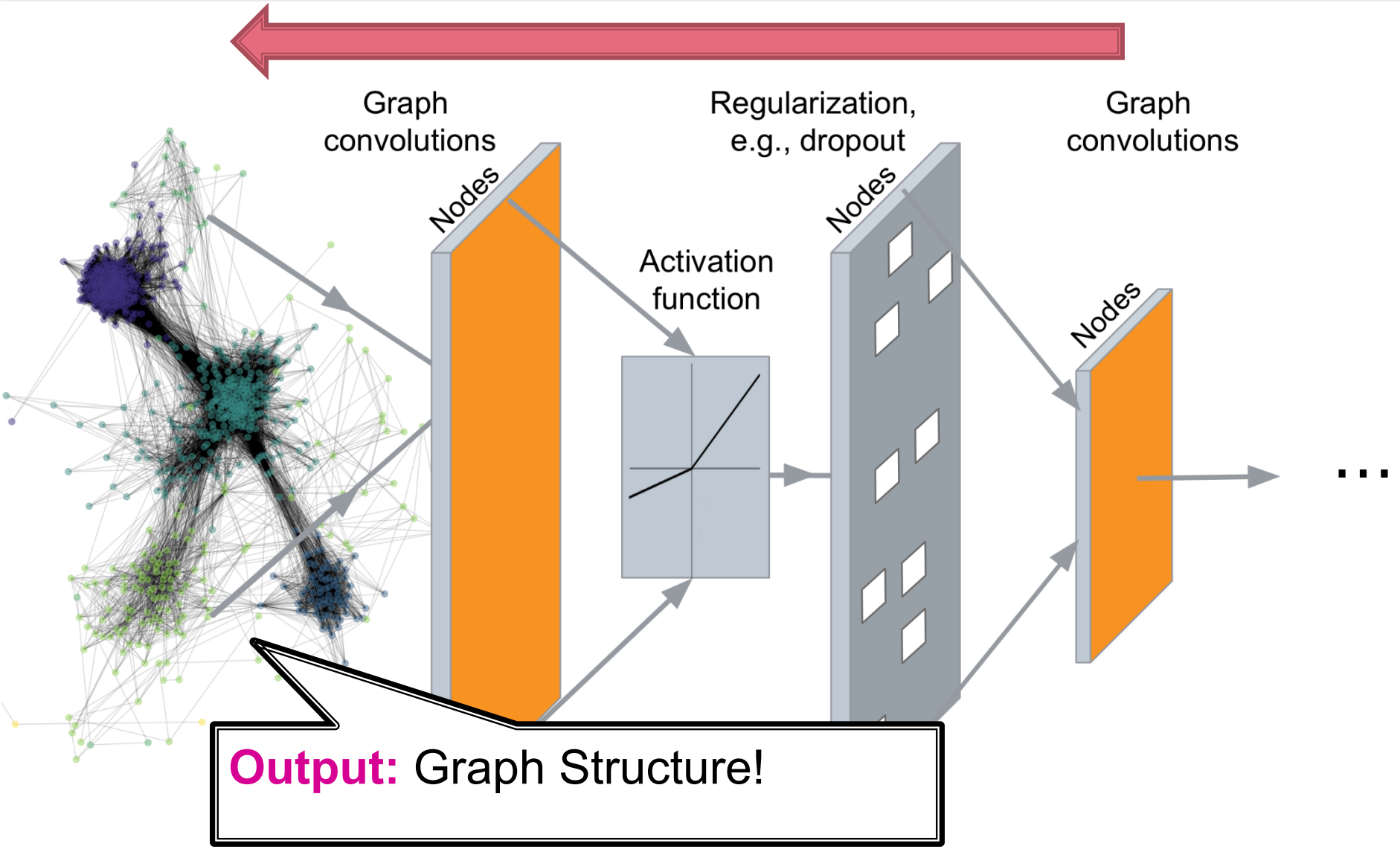
Road Map of Graph Generation

- **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!**

Deep Graph Encoders



Today: Deep Graph Decoders



Stanford CS224W: Machine Learning for Graph Generation

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

Graph Generation Tasks

Task 1: Realistic graph generation

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

Task 2: Goal-directed graph generation

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

Graph Generative Models

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

Generative Models Basics

Setup:

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

Generative Models Basics

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

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

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

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

Generative Models Basics

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

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

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

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

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

  Then x_i follows a complex distribution
- **Q: How to design $f(\cdot)$?**
- **A**: Use Deep Neural Networks, and train it using the data we have!

Deep Generative Models

Auto-regressive models:

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

Stanford CS224W: GraphRNN: Generating Realistic Graphs

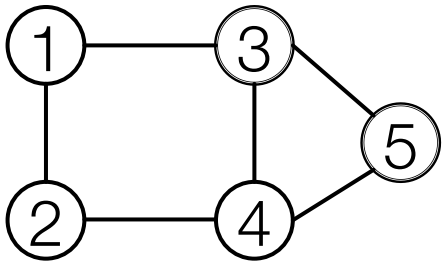
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

[You et al., ICML 2018]

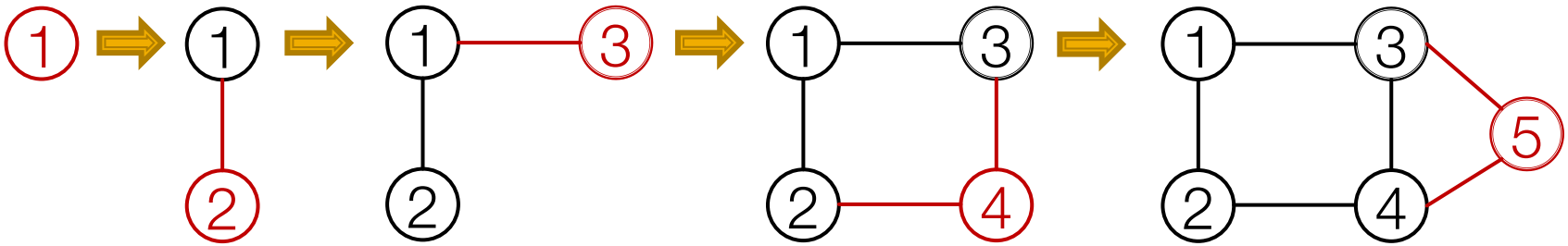
GraphRNN Idea

Generating graphs via sequentially adding nodes and edges

Graph G



Generation process S^{π}

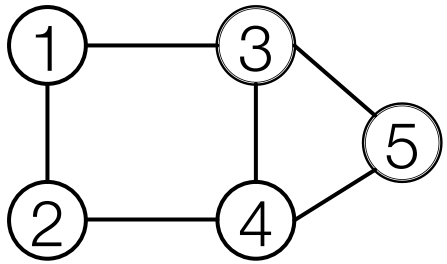


GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. J. You, R. Ying, X. Ren, W. L. Hamilton, J. Leskovec. *International Conference on Machine Learning (ICML)*, 2018.

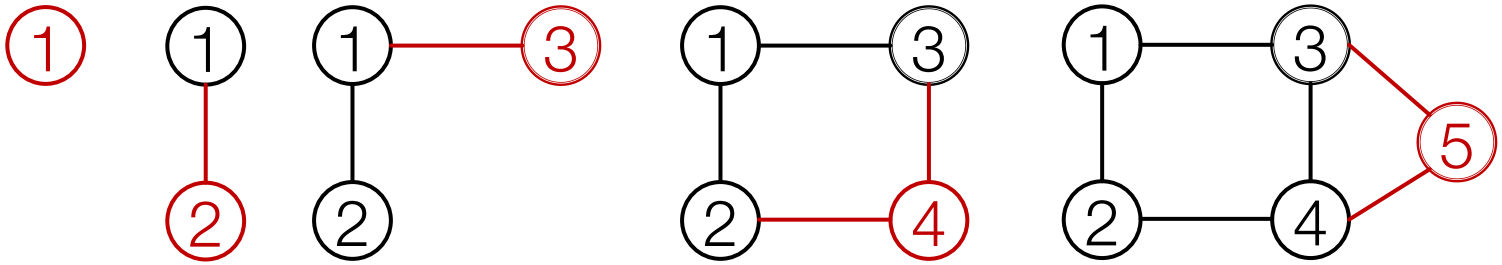
Model Graphs as Sequences

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

Graph G with node ordering π:



Sequence S^{π}:



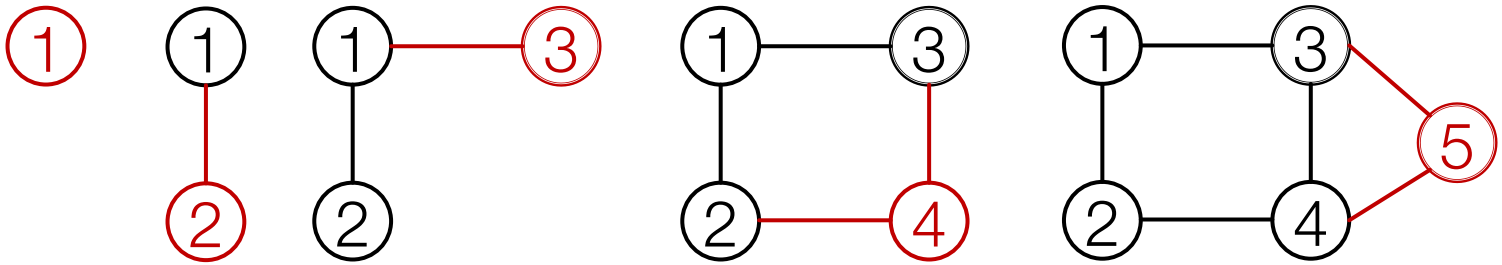
$$S^{\pi} = (S_1^{\pi} , S_2^{\pi} , S_3^{\pi} , S_4^{\pi} , S_5^{\pi})$$

Model Graphs as Sequences

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

- **Node-level:** add nodes, one at a time
- **Edge-level:** add edges between existing nodes

- **Node-level**: At each step, a new node is added



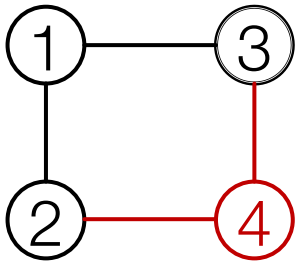
$$S^{\pi} = (\; S_1^{\pi} \; , \; S_2^{\pi} \; , \; S_3^{\pi} \; , \; S_4^{\pi} \; , \; S_5^{\pi} \;)$$

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

Model Graphs as Sequences

The sequence S^{π} has **two levels:**

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



S_4^{π}

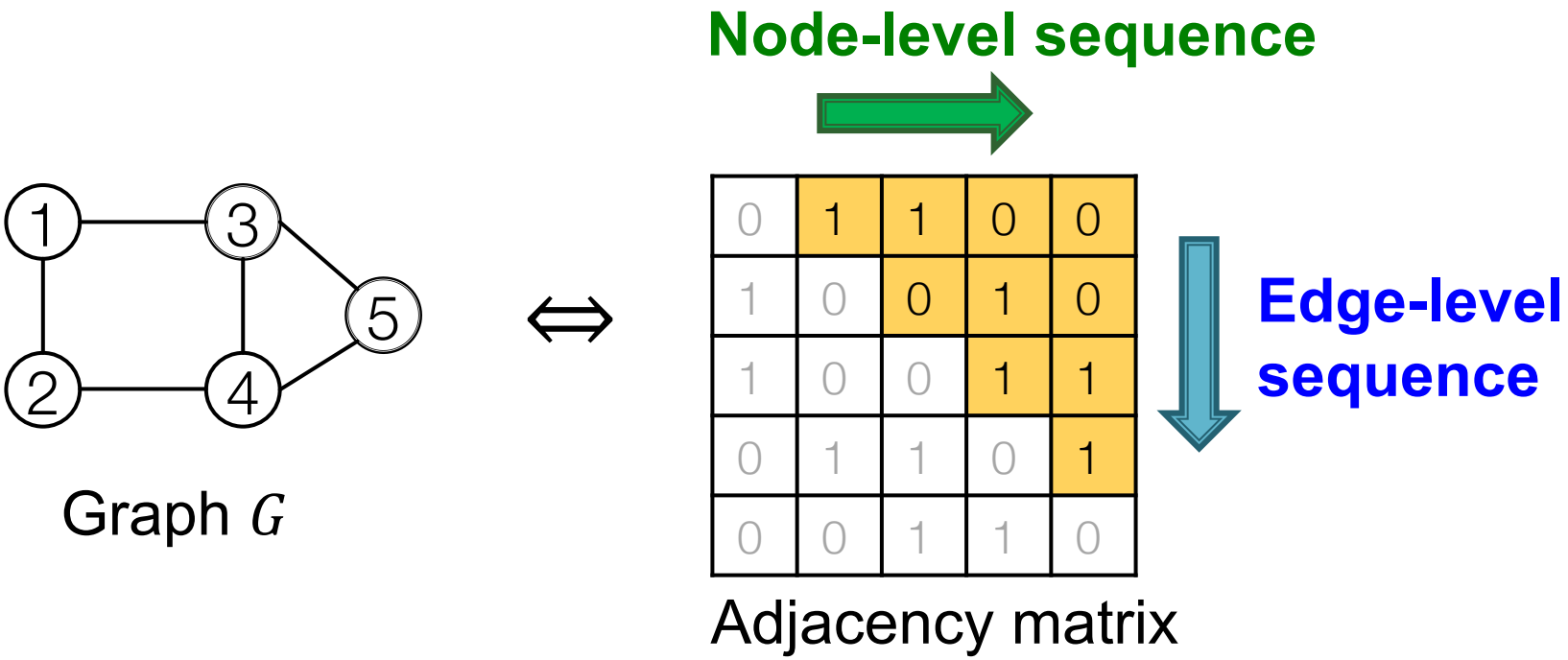
$$S_4^{\pi} = \left(S_{4,1}^{\pi}, S_{4,2}^{\pi}, S_{4,3}^{\pi} \right)$$

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

0 1 1

Model Graphs as Sequences

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

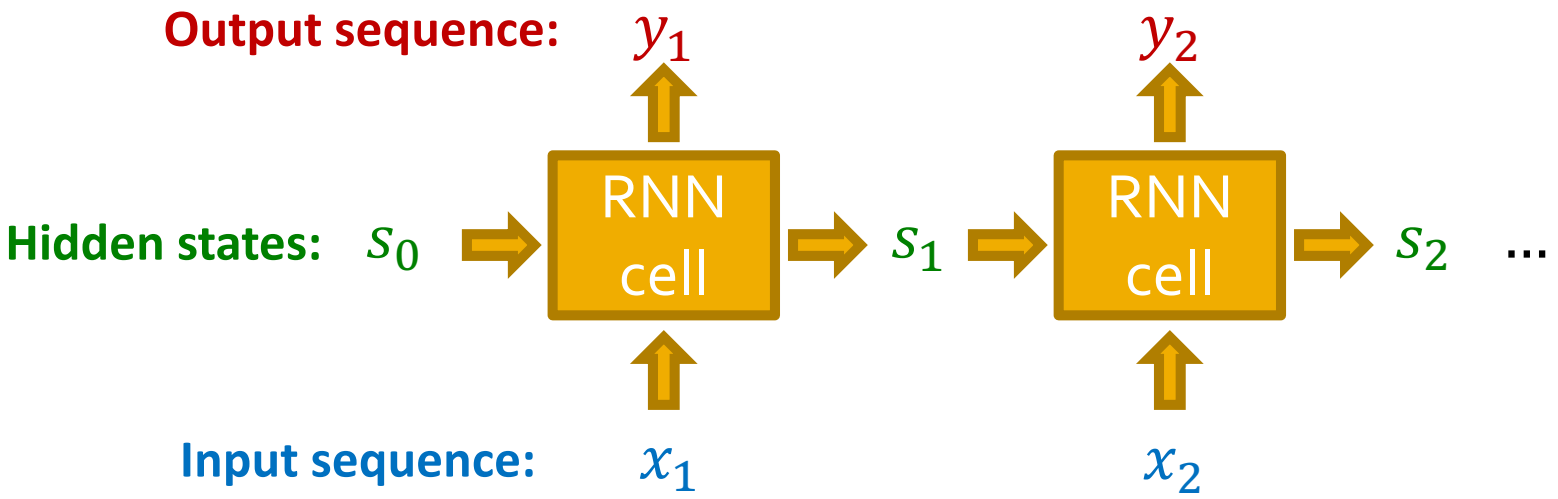


Model Graphs as Sequences

- **We have transformed graph generation problem into a sequence generation problem**
- **Need to model two processes:**
  - **1)** Generate a state for a new node (Node-level sequence)
  - **2)** Generate edges for the new node based on its state (Edge-level sequence)
- **Approach**: Use **Recurrent Neural Networks (RNNs)** to model these processes!

Background: Recurrent NNs

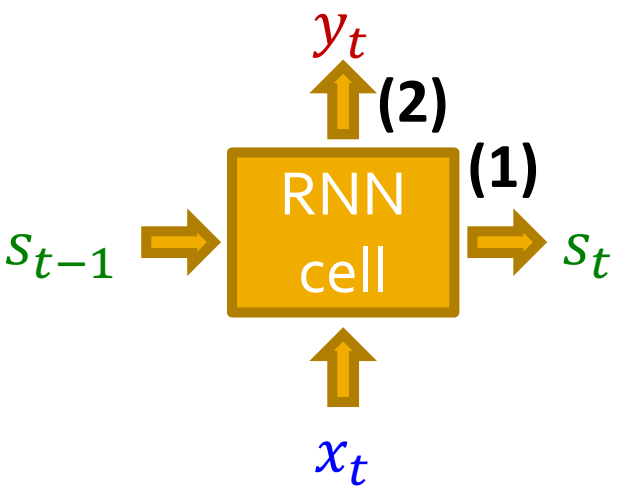
- RNNs are designed for **sequential data**
  - RNN sequentially takes **input sequence** to **update its hidden states**
  - The **hidden states** summarize all the information input to RNN
  - The update is conducted via **RNN cells**



Background: Recurrent NNs

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

In our case s_t, x_t and y_t are scalars



The RNN cell:

(1) Update hidden state:

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

(2) Output prediction:

$$y_t = V \cdot s_t$$

- **More expressive cells:** GRU, LSTM, etc.

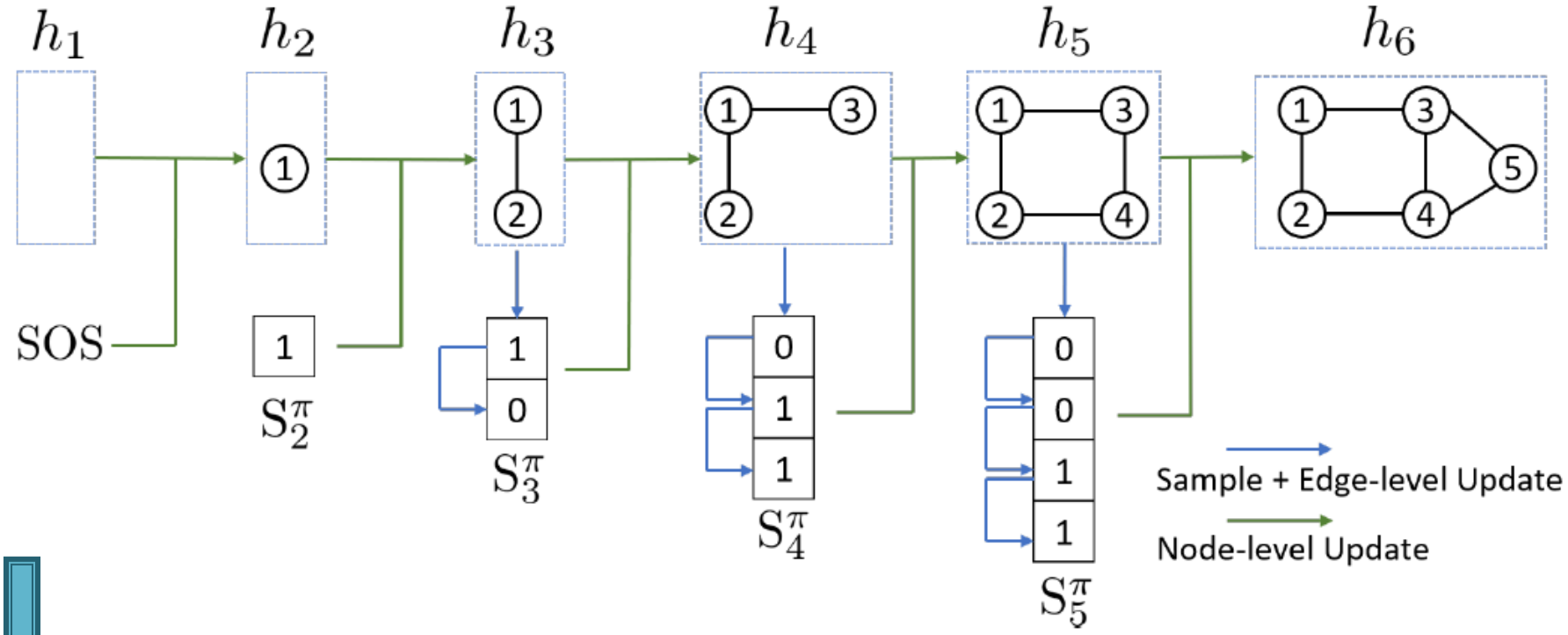
GraphRNN: Two levels of RNN

- **GraphRNN has a node-level RNN and an edge-level RNN**

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

GraphRNN: Two levels of RNN

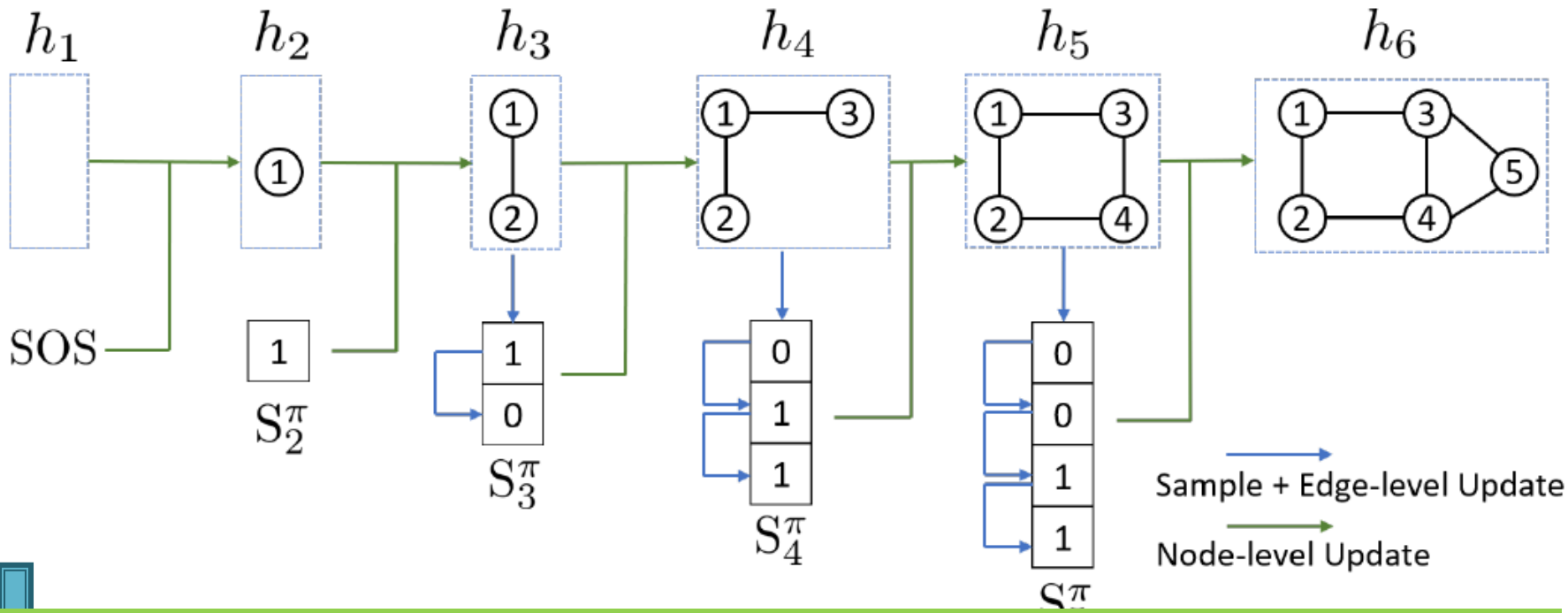
Node-level RNN generates the initial state for edge-level RNN



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

GraphRNN: Two levels of RNN

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



Next: How to generate a sequence with RNN?

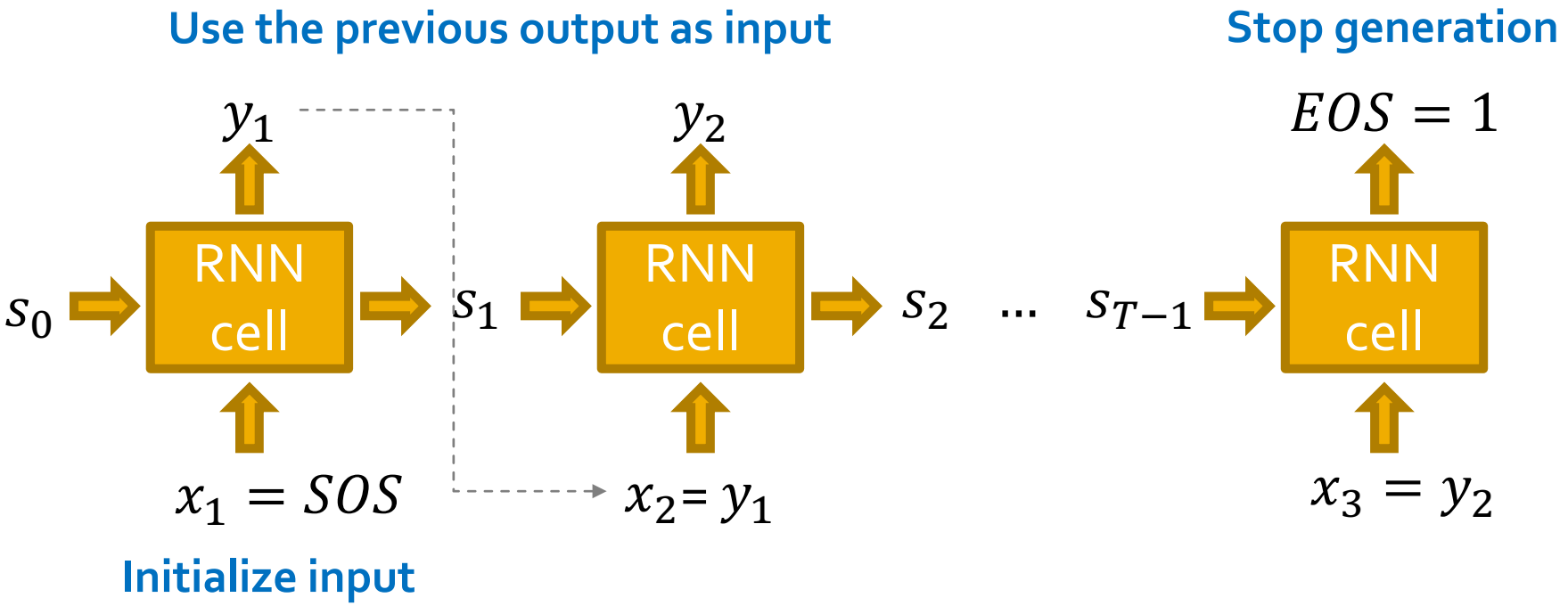
RNN for Sequence Generation

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

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

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

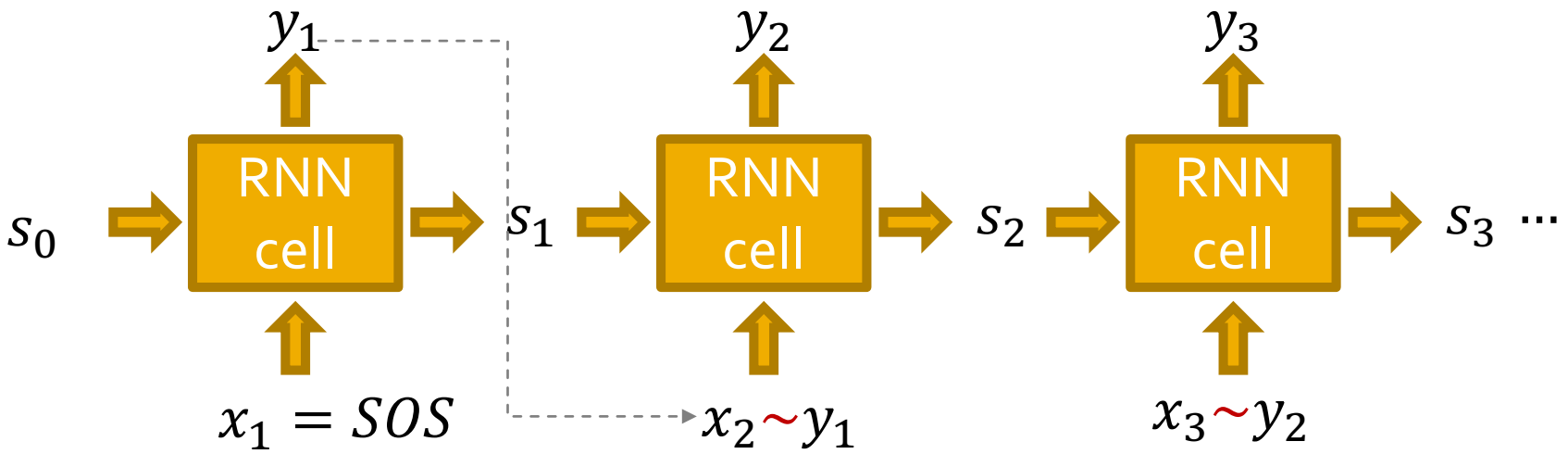
RNN for Sequence Generation



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

RNN for Sequence Generation

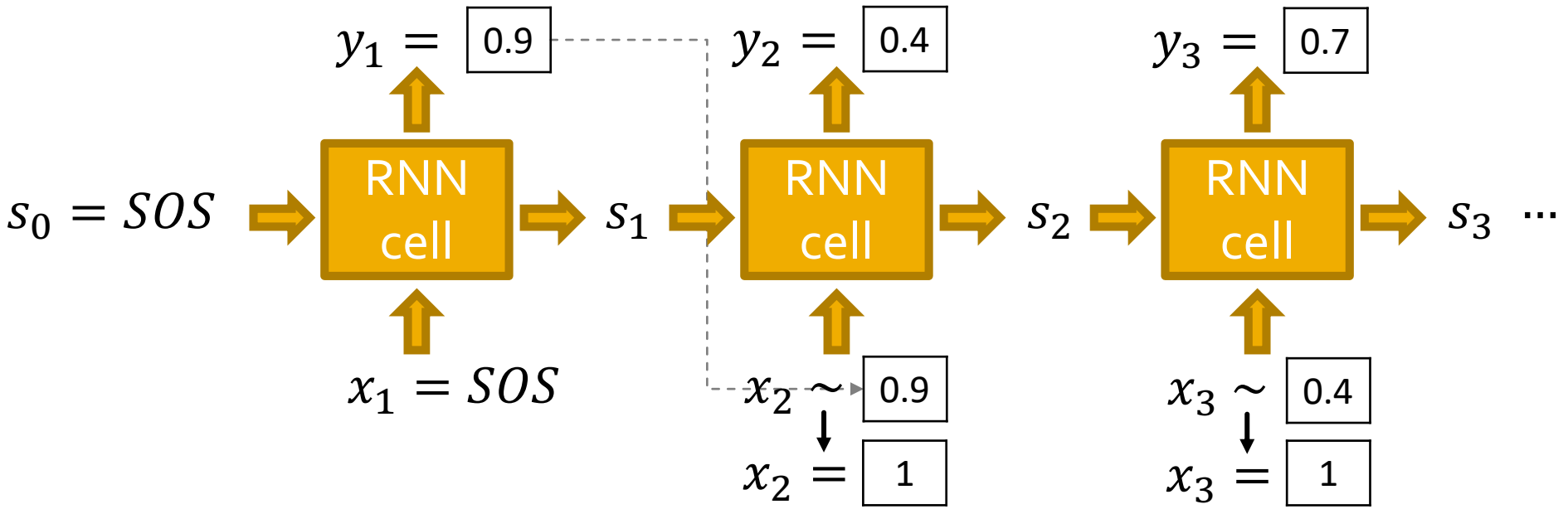
- **Remember our goal**: Use RNN to model $\prod_{k=1}^{n} p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$
- Let $y_t = p_{model}(x_t | x_1, \ldots, x_{t-1}; \theta)$
- Then we need to sample x_{t+1} from y_t: $x_{t+1} \sim y_t$
  - Each step of RNN outputs a **probability of a single edge**
  - We then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



RNN at Test Time

Suppose we already have trained the model

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

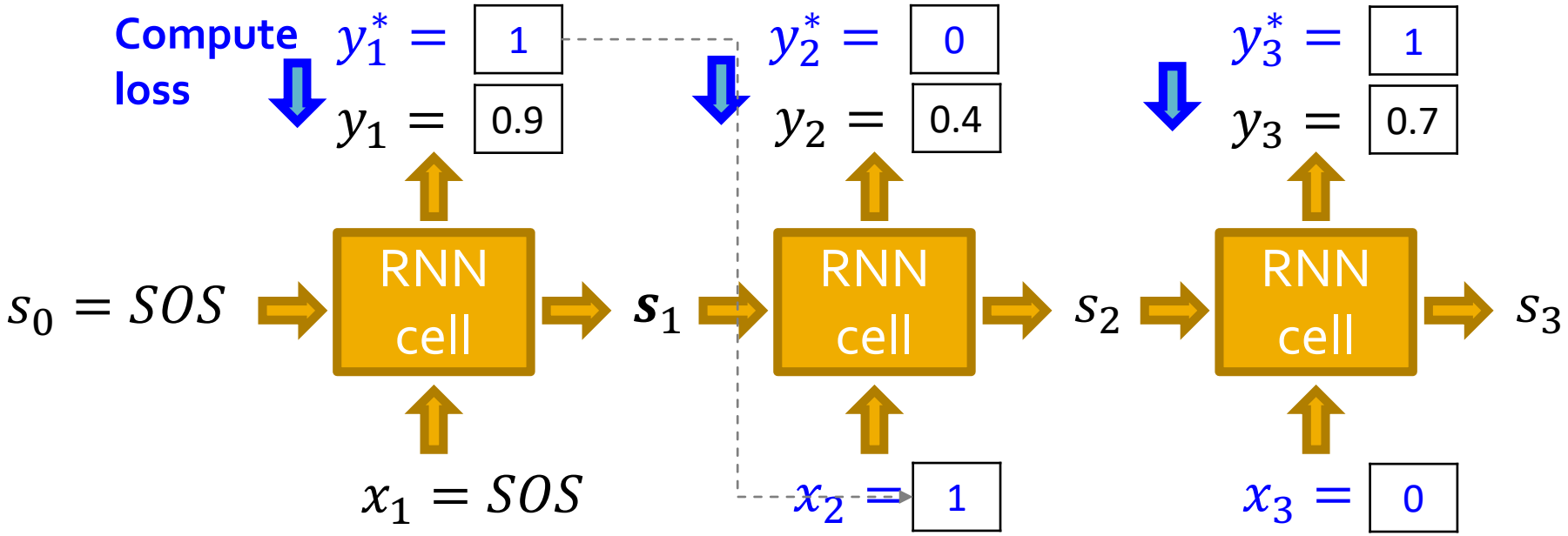


- **How do we use training data $x_1, x_2, \ldots, x_n$?**

RNN at Training Time

Training the model:

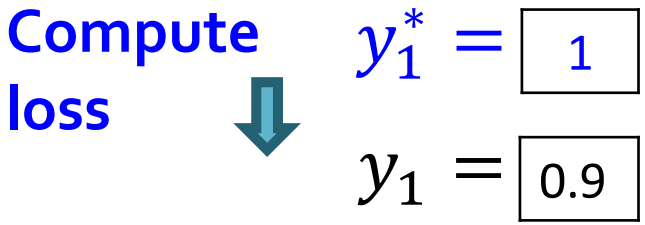
- We observe a sequence y^* of edges [1,0,…]
- **Principle**: **Teacher Forcing** -- Replace input and output by the real sequence



RNN at Training Time

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

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



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

Putting Things Together

Our Plan:

(1) **Add a new node:** We run Node RNN for a step, and use it output to initialize Edge RNN

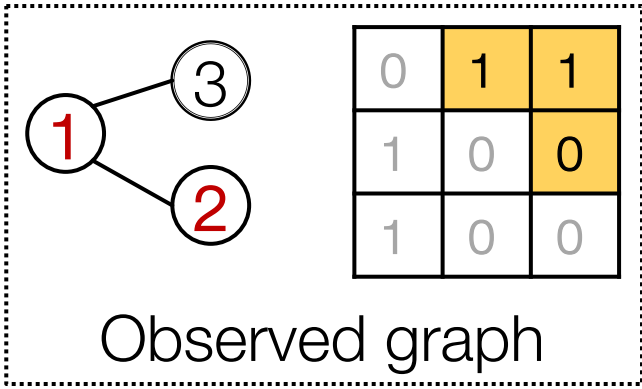
(2) **Add new edges for the new node:** We run Edge RNN to predict if the new node will connect to each of the previous node

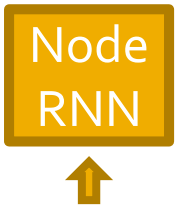
(3) **Add another new node:** We use the last hidden state of Edge RNN to run Node RNN for another step

(4) **Stop graph generation:** If Edge RNN outputs EOS at step 1, we know no edges are connected to the new node. We stop the graph generation.

Put Things Together: Training

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

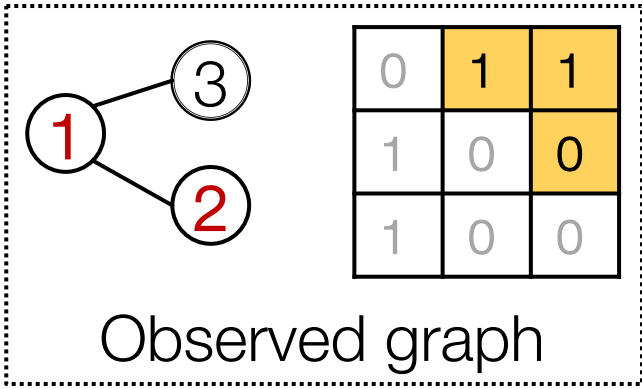


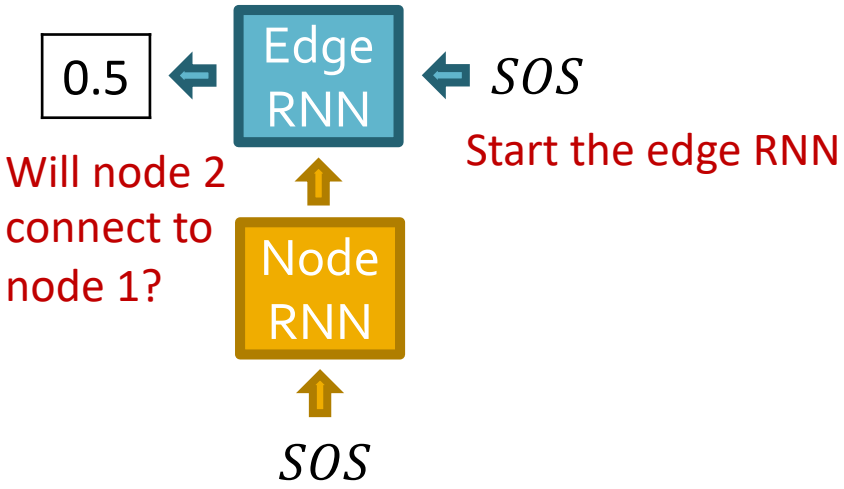


Start the node RNN

Put Things Together: Training

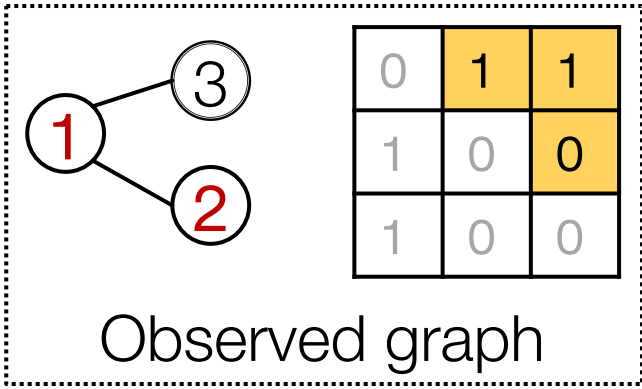
Edge RNN predicts how
Node 2 connects to **Node 1**

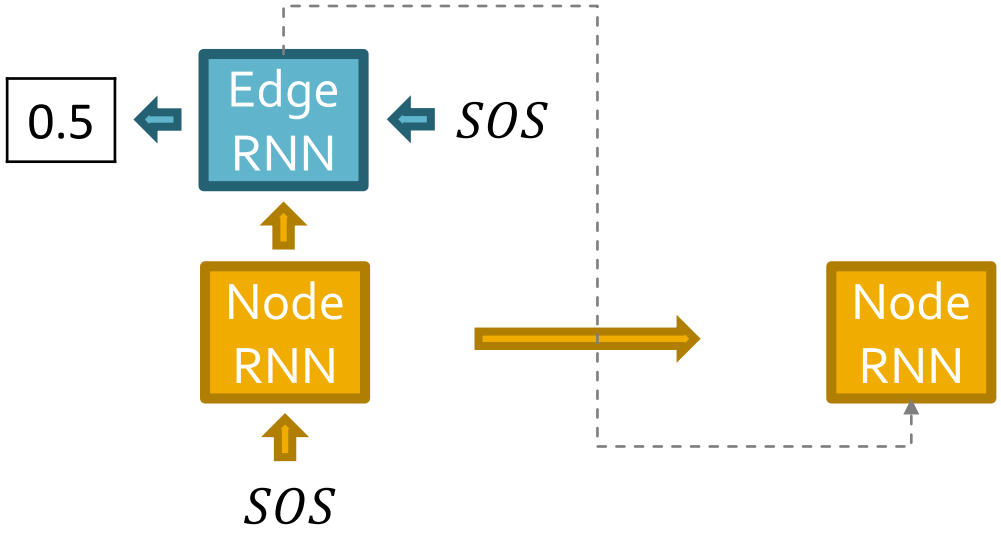




Put Things Together: Training

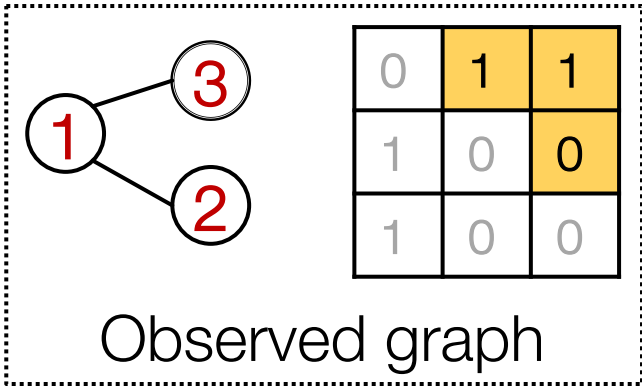
Update Node RNN using
Edge RNN's hidden state

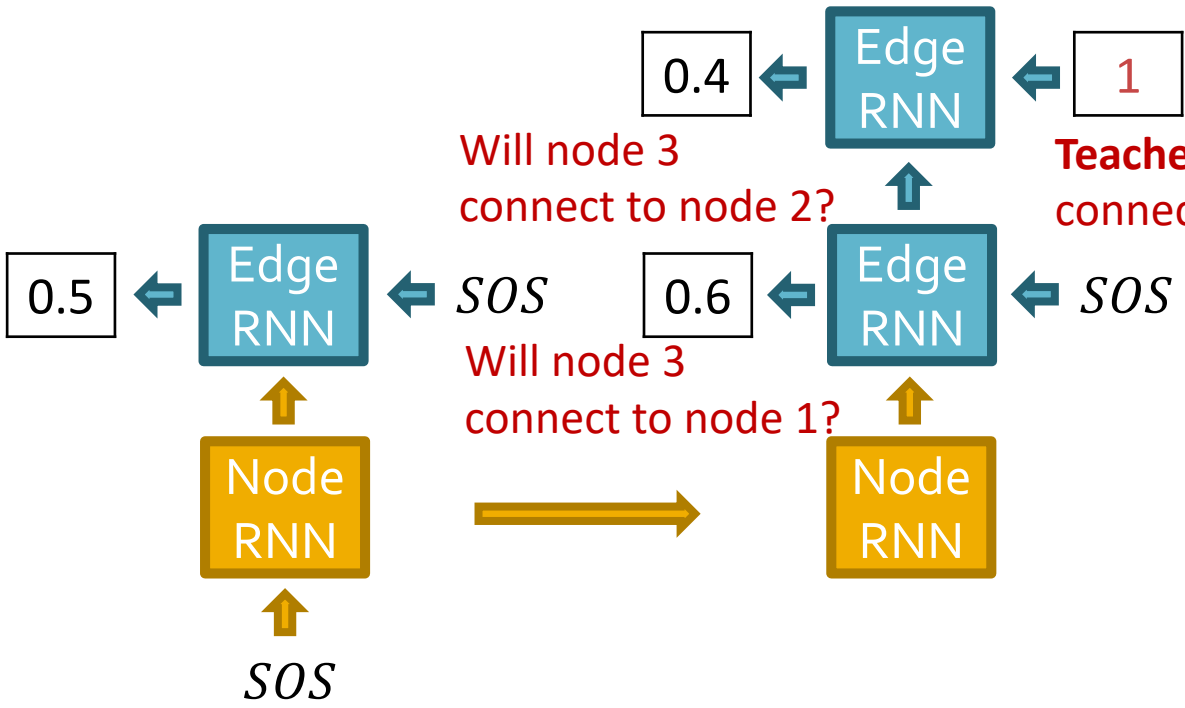




Put Things Together: Training

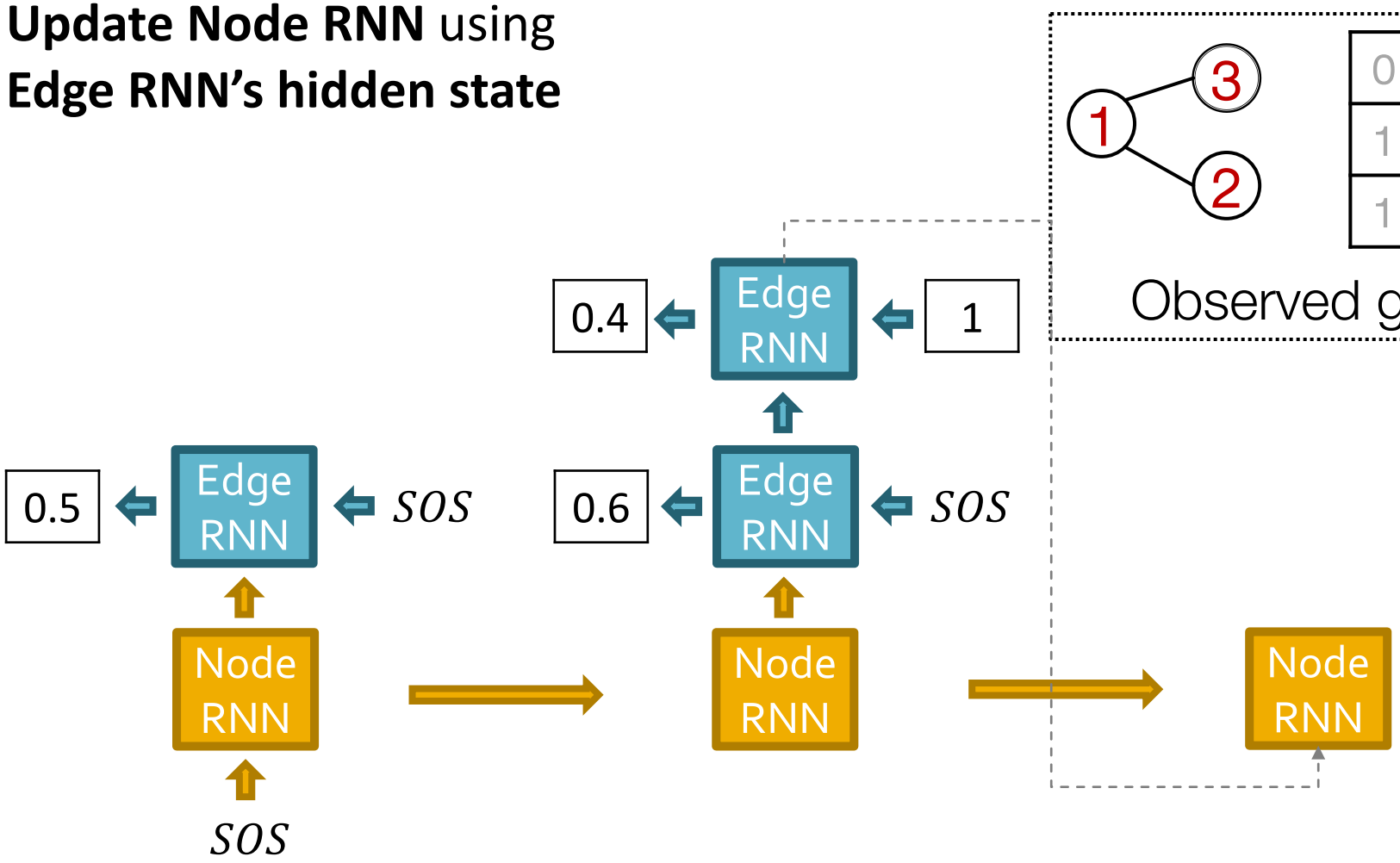
Edge RNN predicts how **Node 3** tries to connects to **Nodes 1, 2**





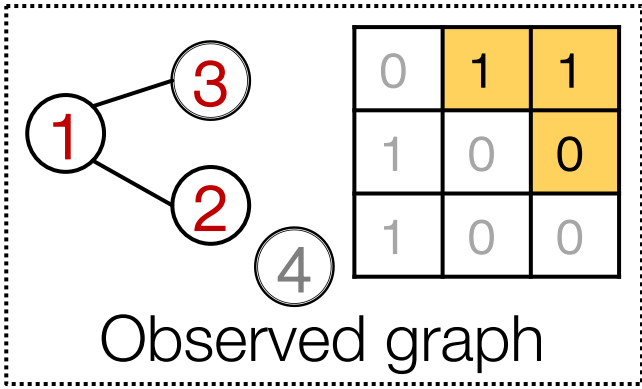
Put Things Together: Training

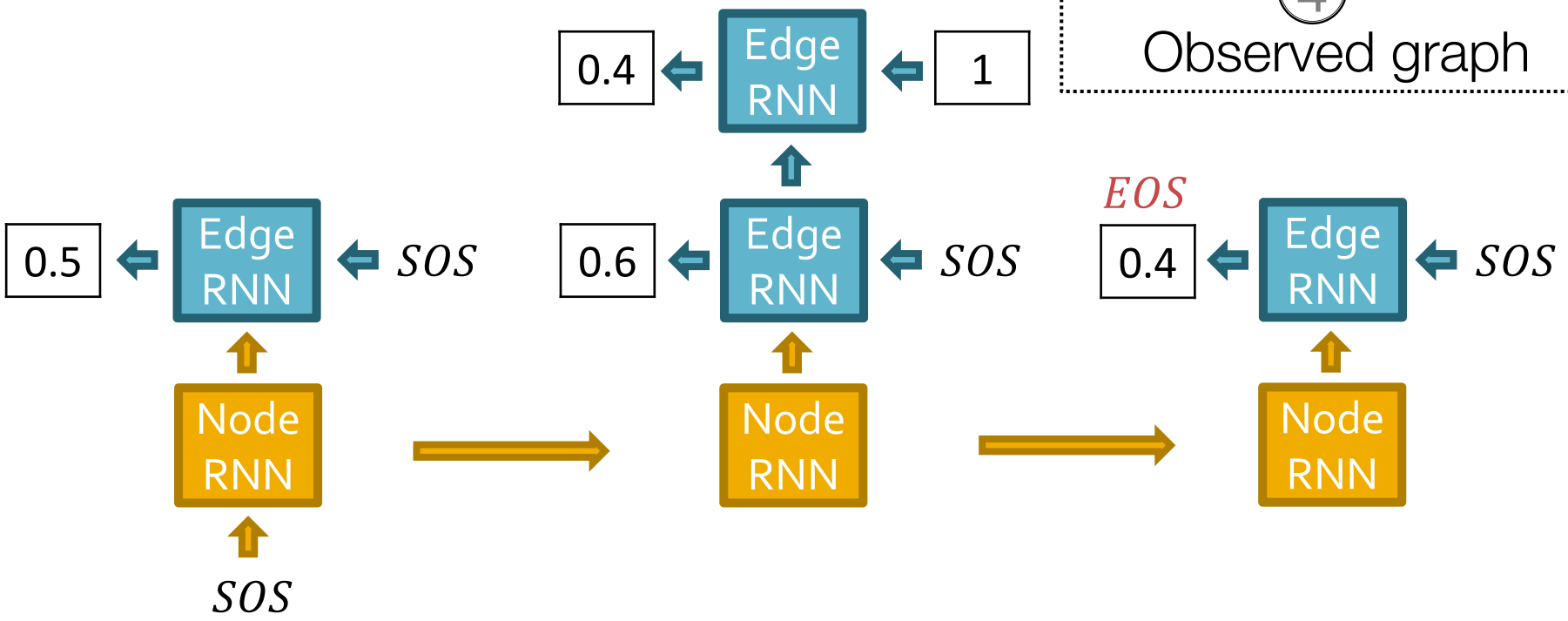
Update Node RNN using
Edge RNN's hidden state



Put Things Together: Training

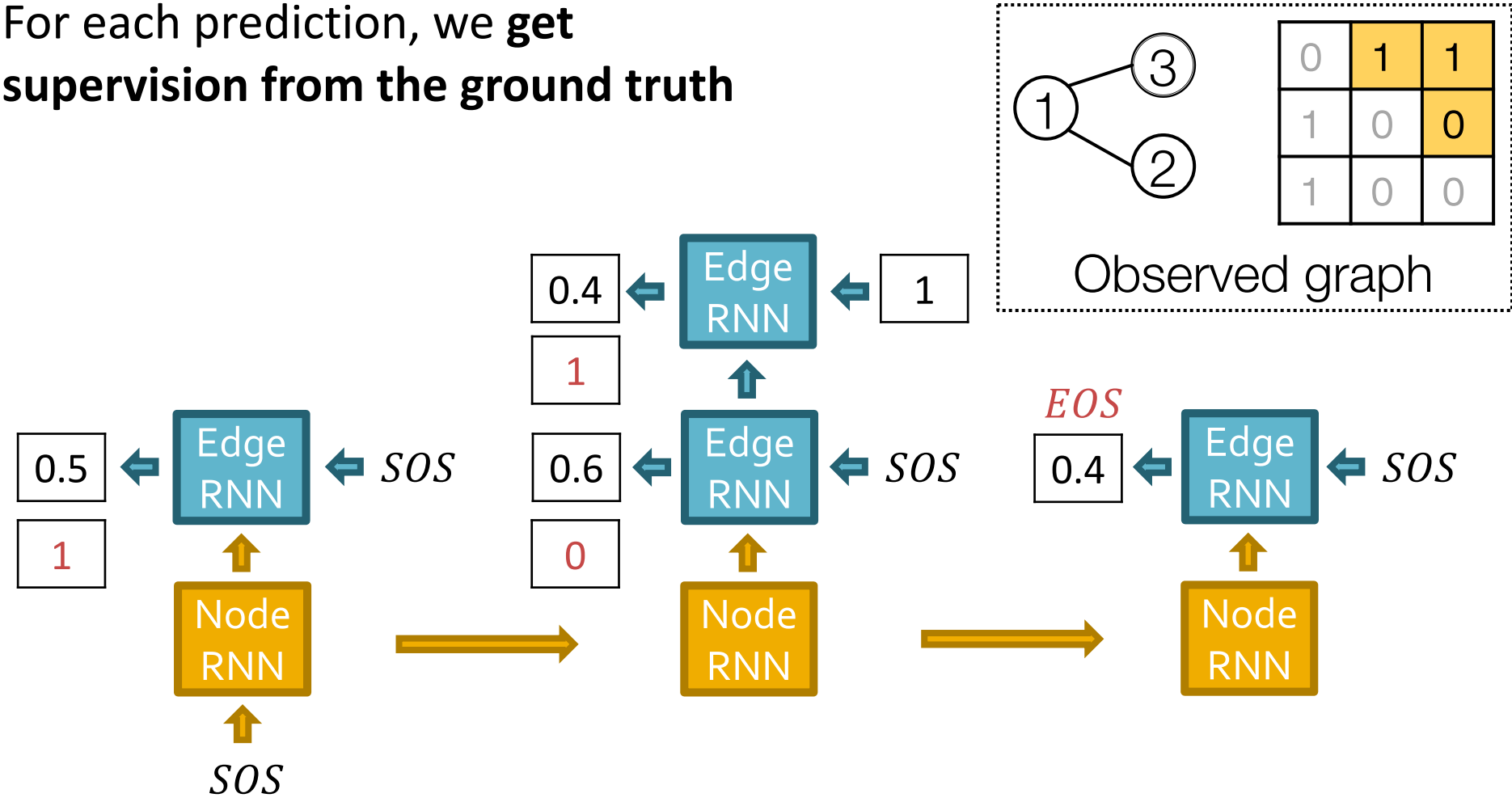
Stop generation since we know node 4 won't connect to any nodes





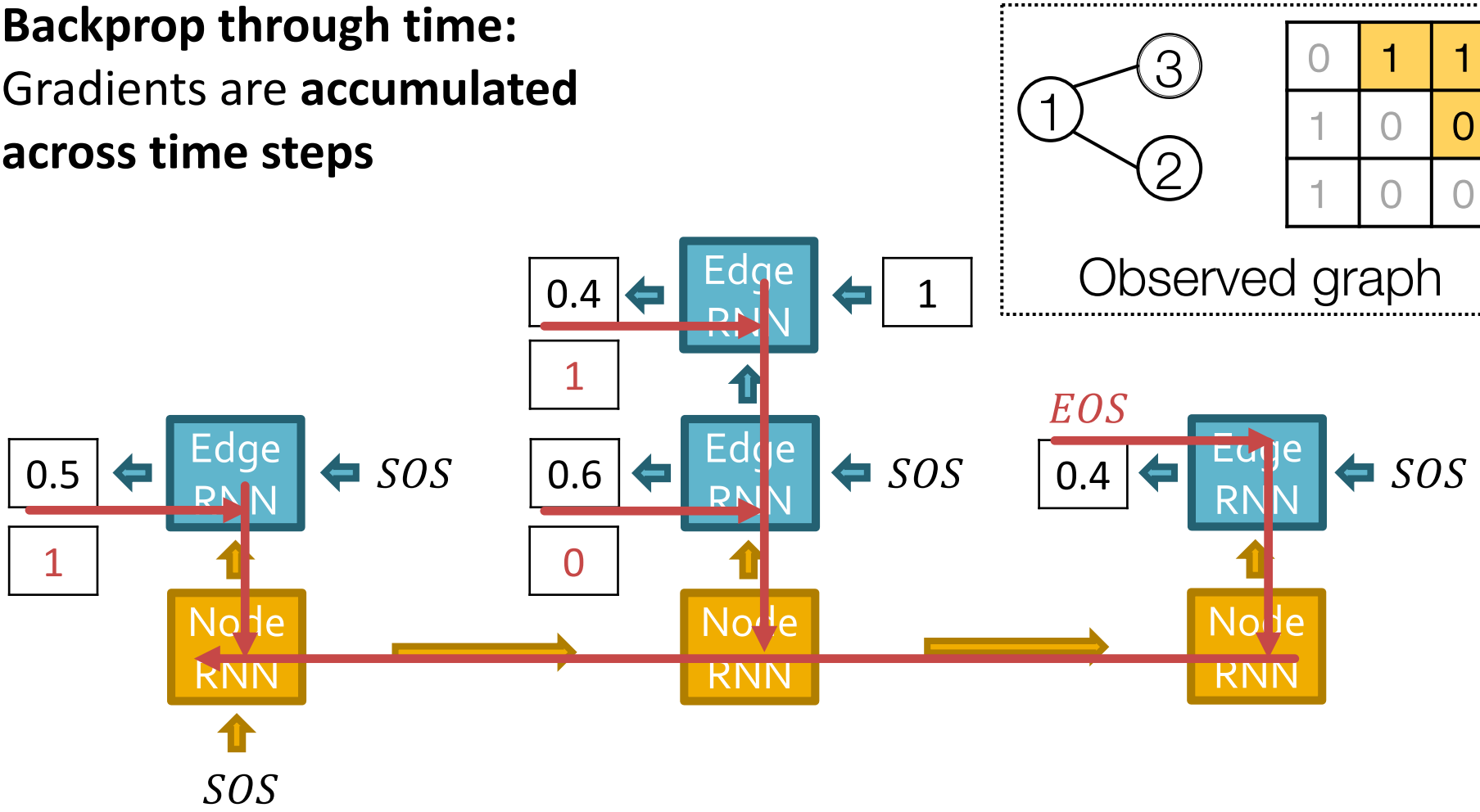
Put Things Together: Training

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



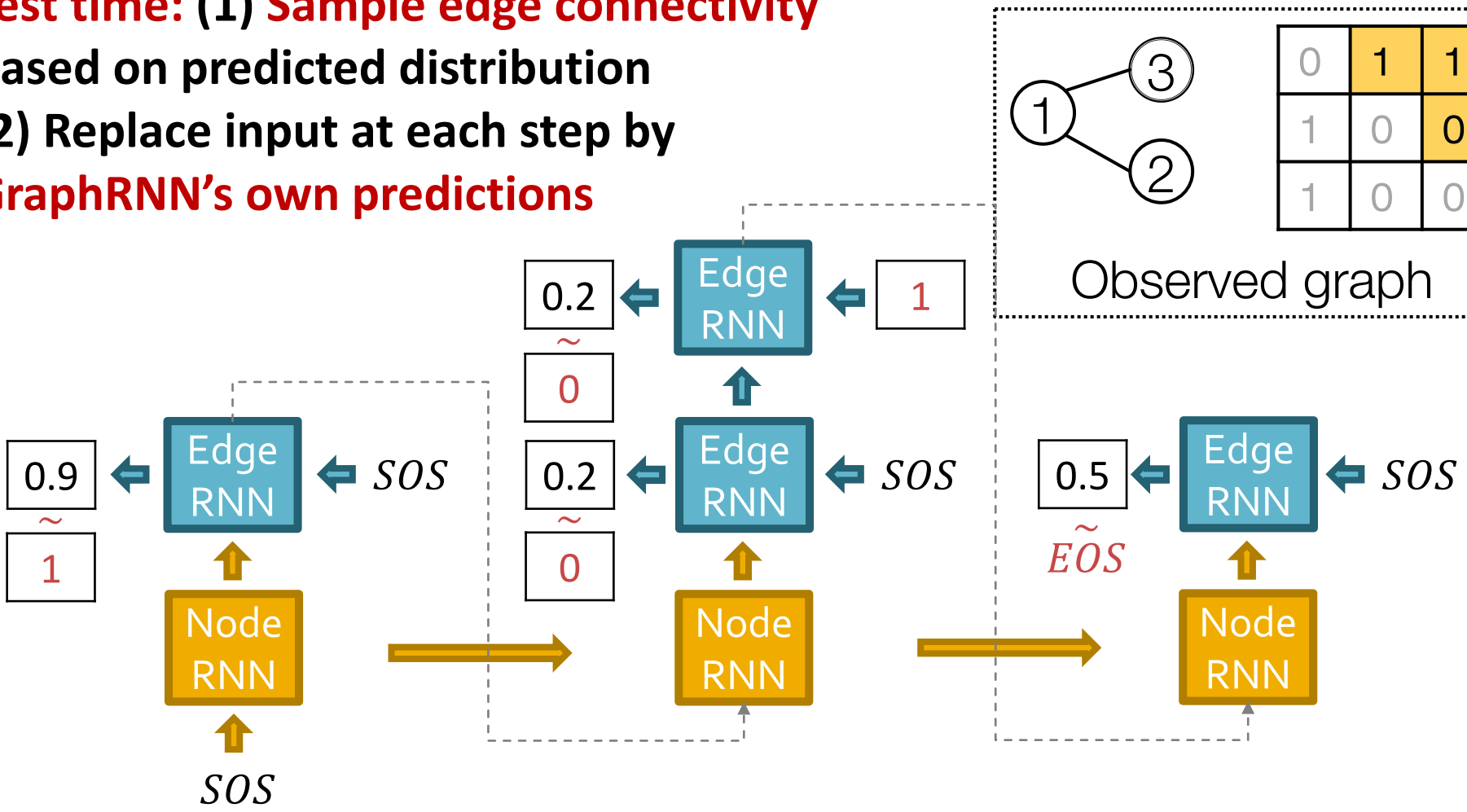
Put Things Together: Training

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



Put Things Together: Test

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

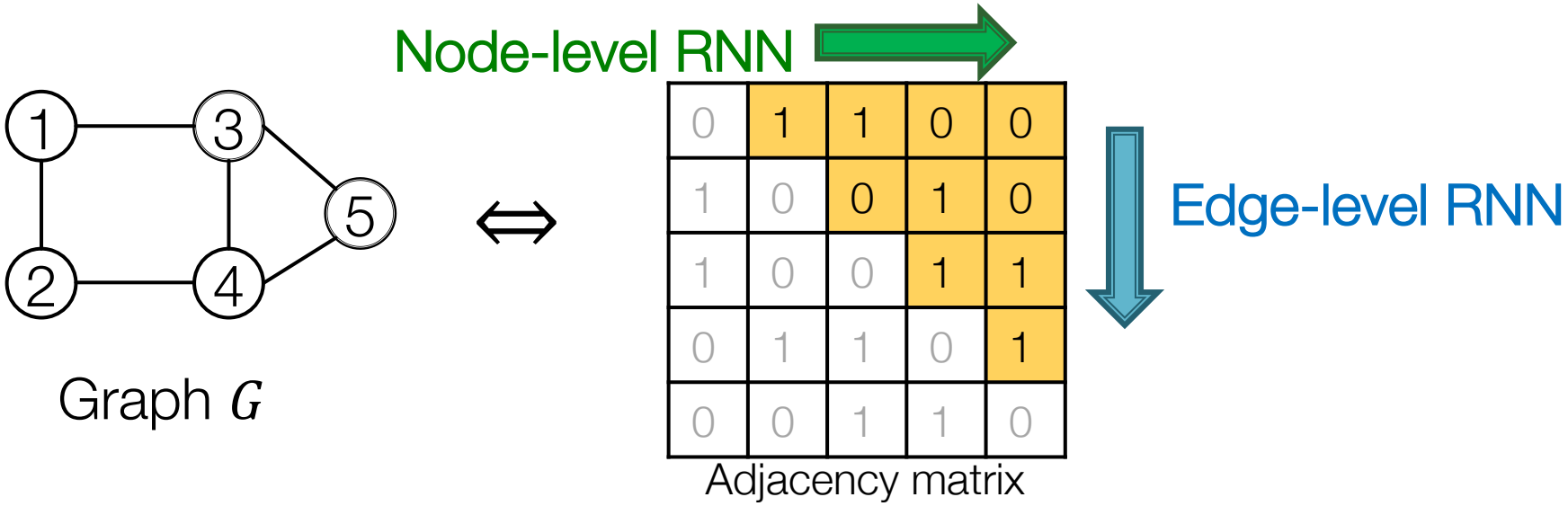


GraphRNN: Two levels of RNN

Quick Summary of GraphRNN:

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

- **Next**: Making GraphRNN tractable, proper evaluation

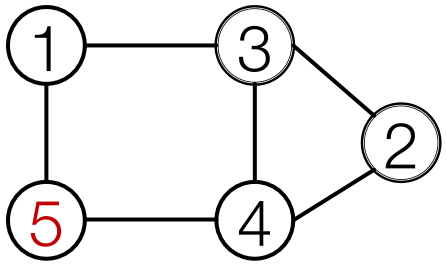


Stanford CS224W: Scaling Up and Evaluating Graph Generation

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

Issue: Tractability

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



Random node ordering:

Node 5 may connect to any/all previous nodes

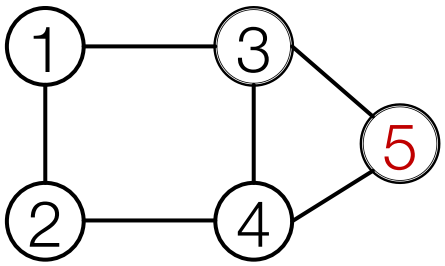
"Recipe" to generate the left graph:

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

How do we limit this complexity?

Solution: Tractability via BFS

- **Breadth-First Search node ordering**



BFS ordering

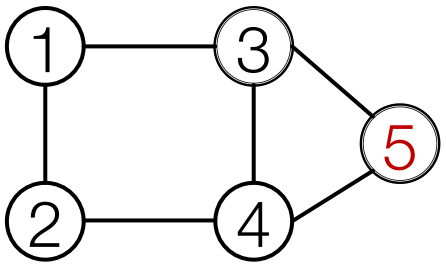
"Recipe" to generate the left graph:

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

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

Solution: Tractability via BFS

- **Breadth-First Search node ordering**



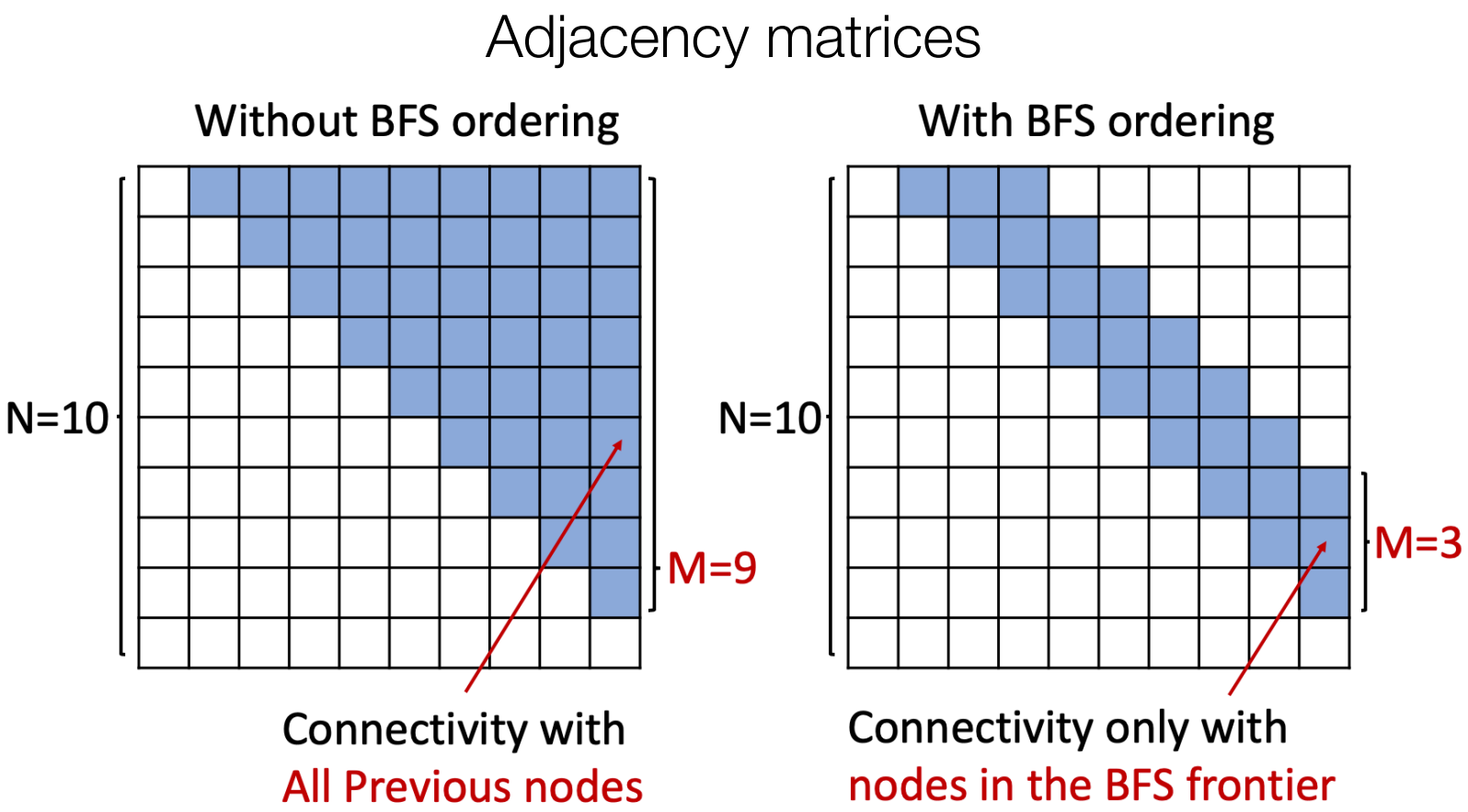
BFS ordering

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

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

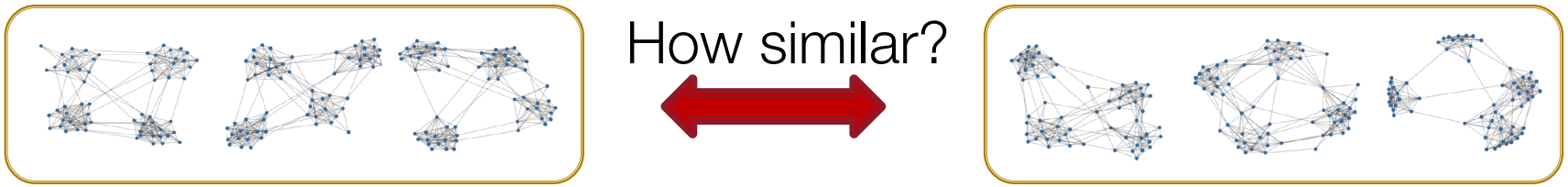
Solution: Tractability via BFS

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



Evaluating Generated Graphs

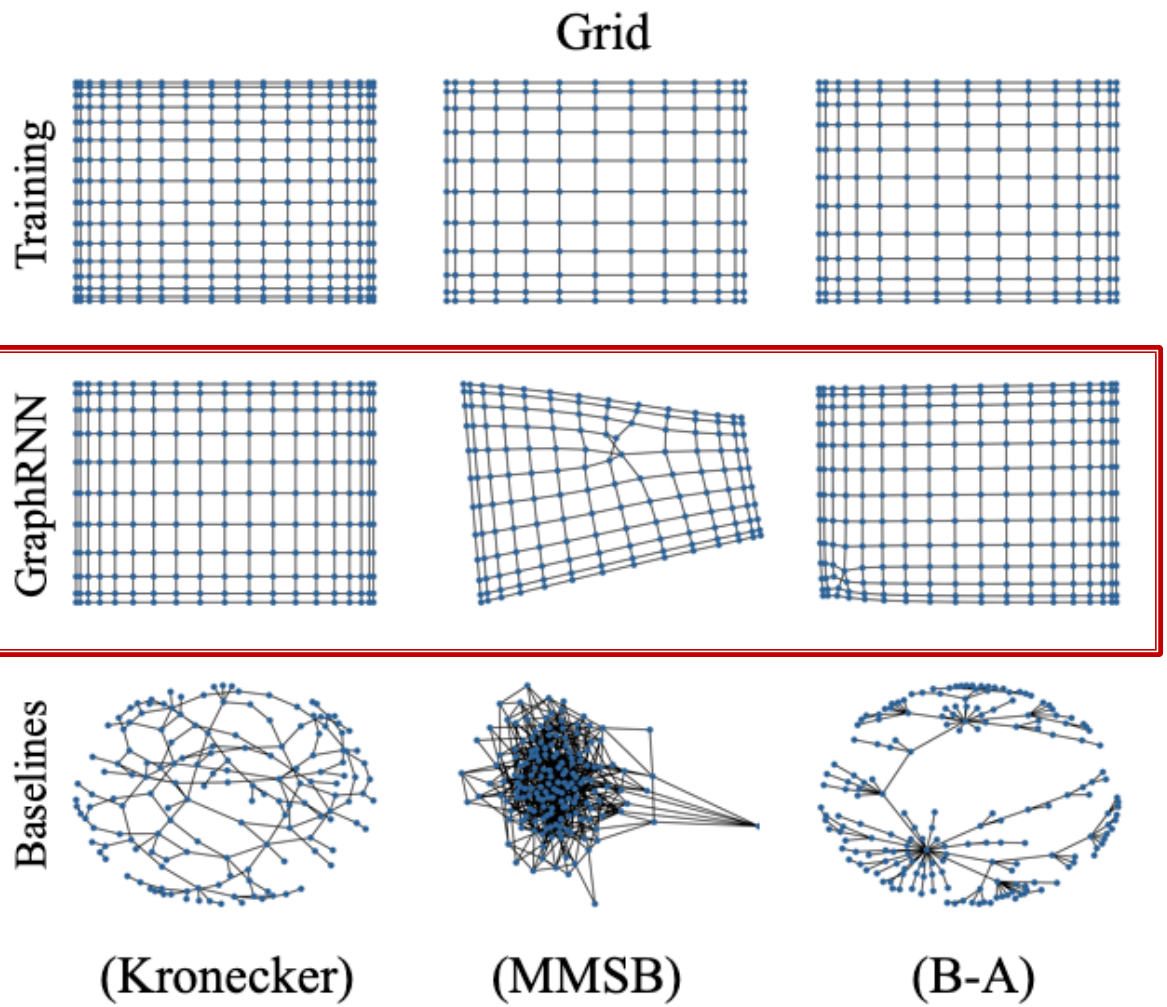
- **Task: Compare two sets of graphs**



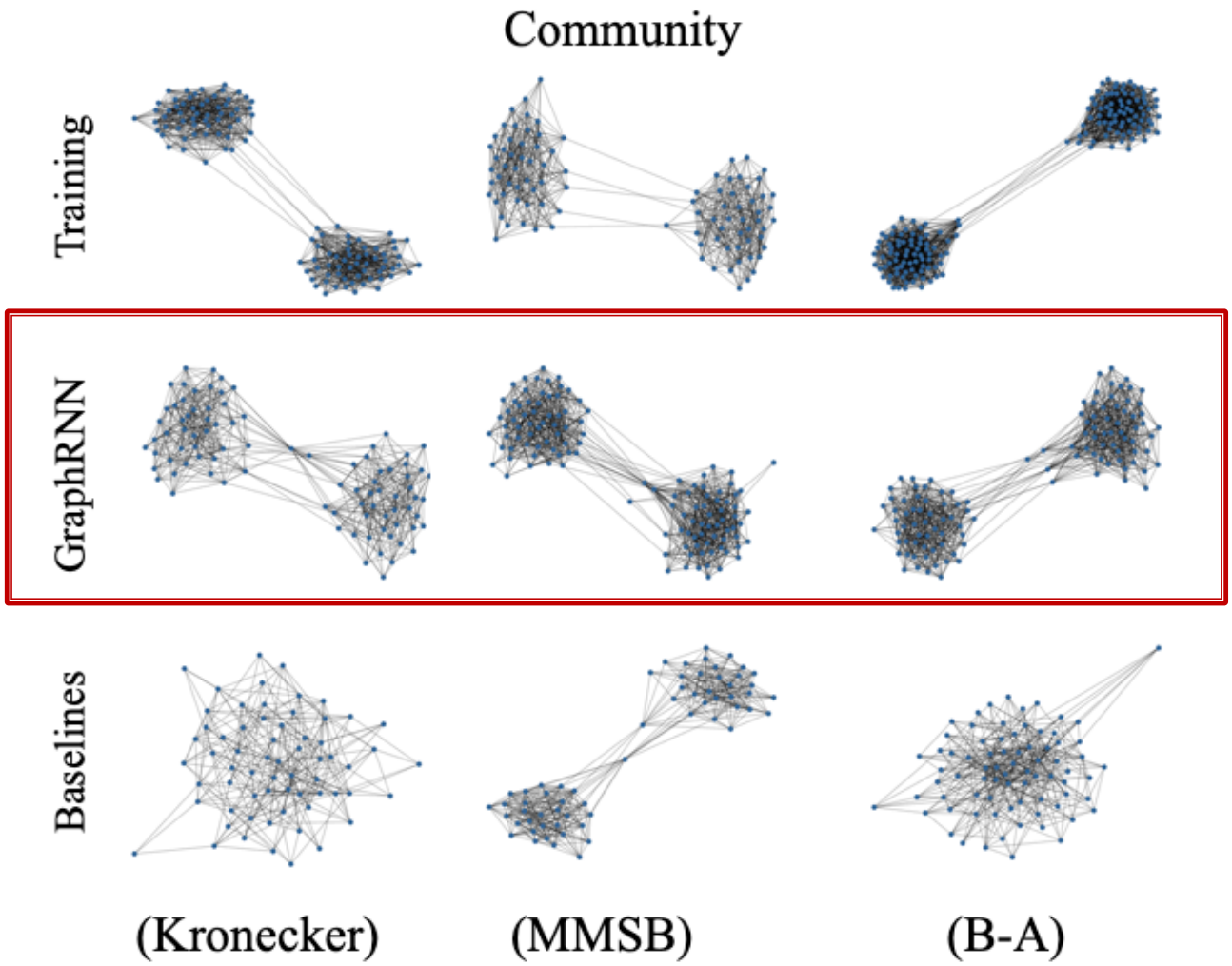
- **Goal:** Define similarity metrics for graphs

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

(1) Visual Similarity



(1) Visual Similarity

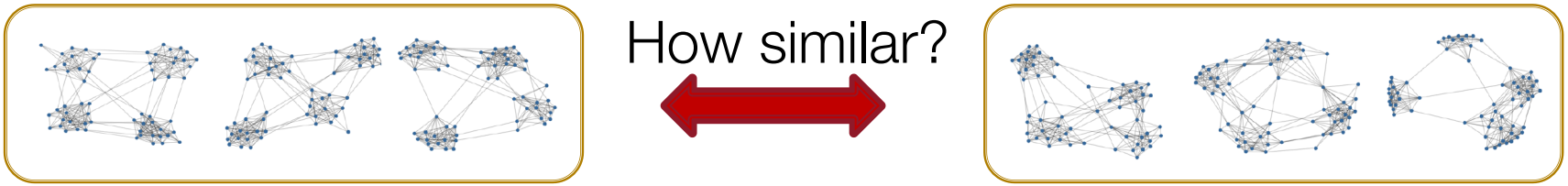


(2) Graph statistics similarity

- **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

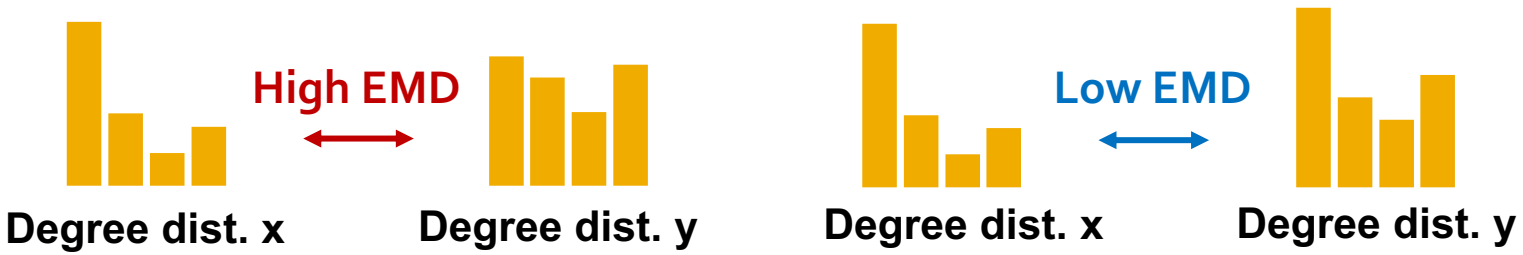
- **Issue:** want to compare sets of training graph statistics and generated graph statistics



- **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.

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

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** $\{5,4,2\}$

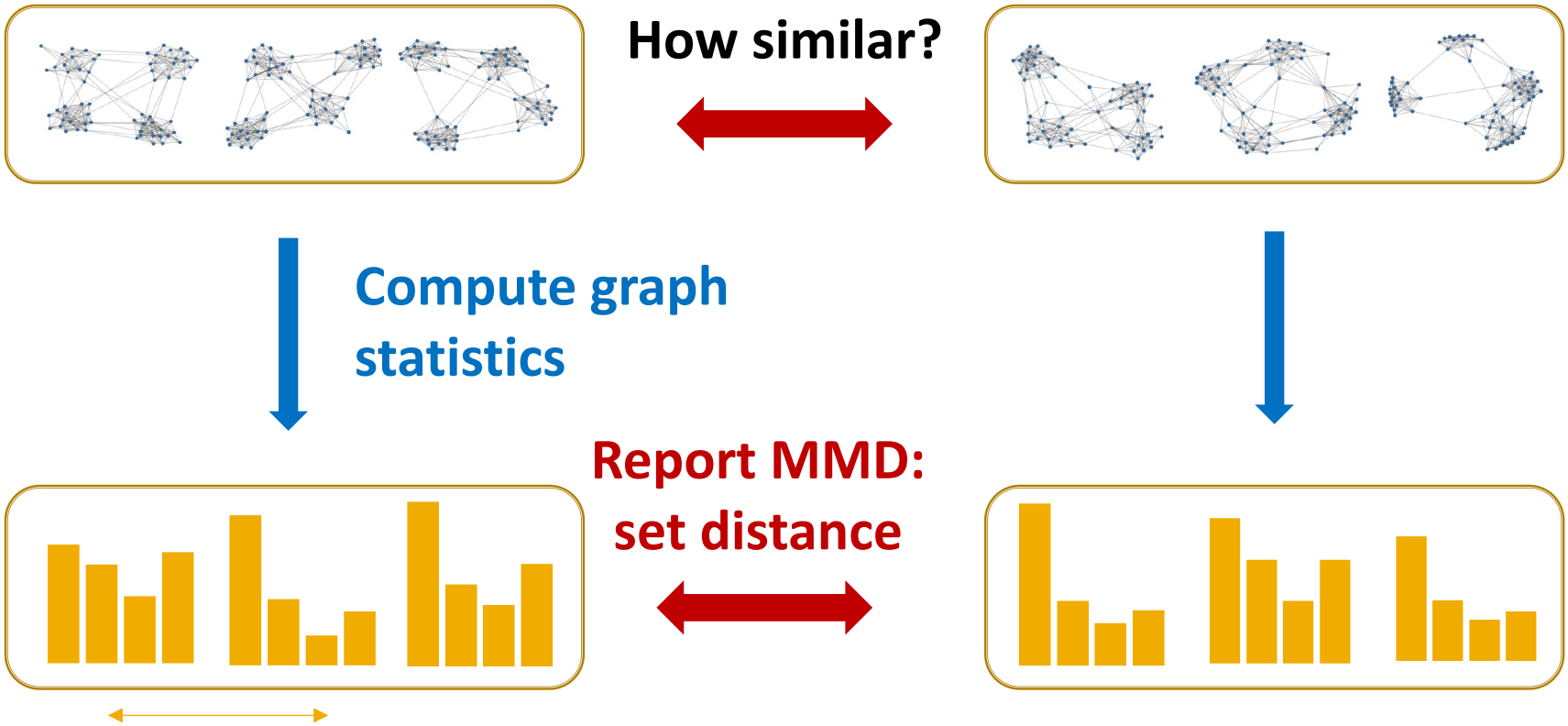
Similarity between set elements: L2 distance
(Each element is a scalar)

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

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

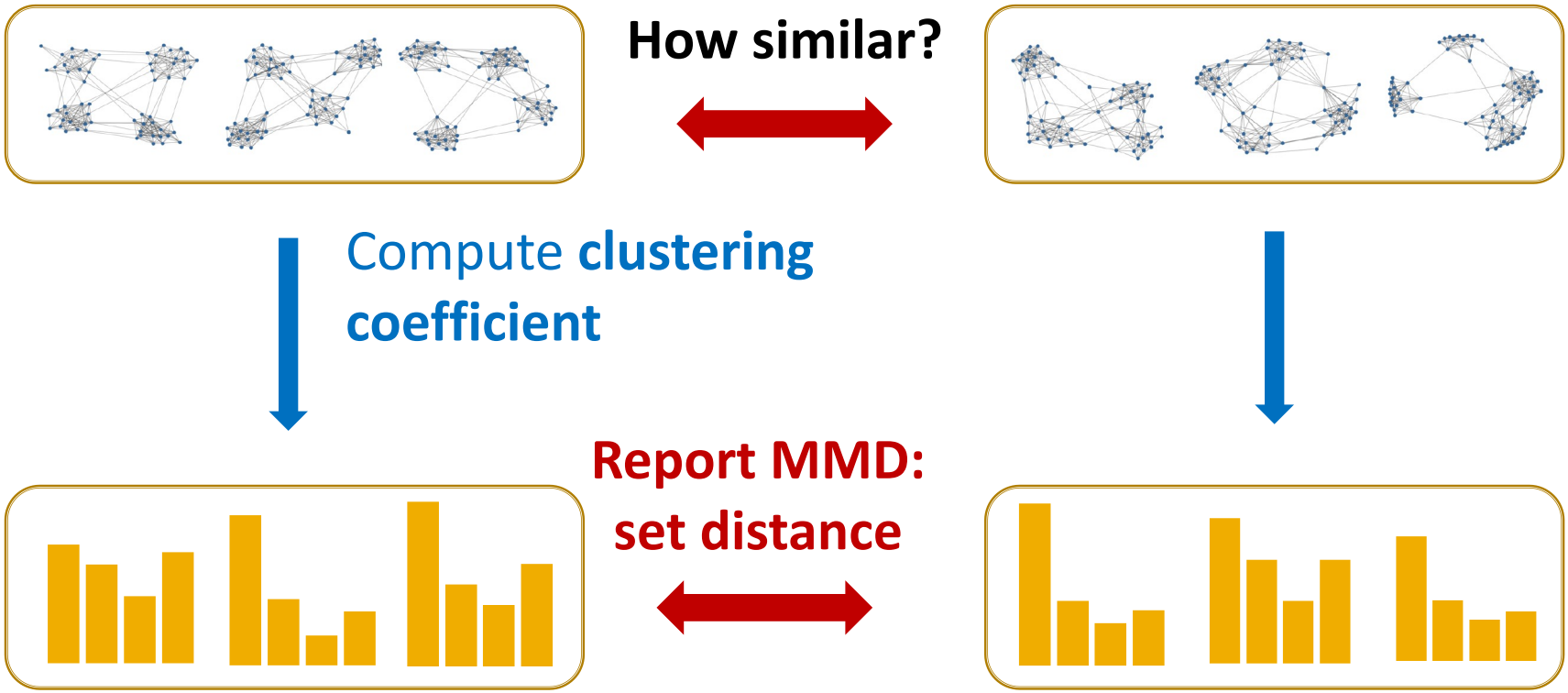
(2) Graph statistics similarity

- Putting things together



(2) Graph statistics similarity

- **Example**



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

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

CS224W: Machine Learning with Graphs

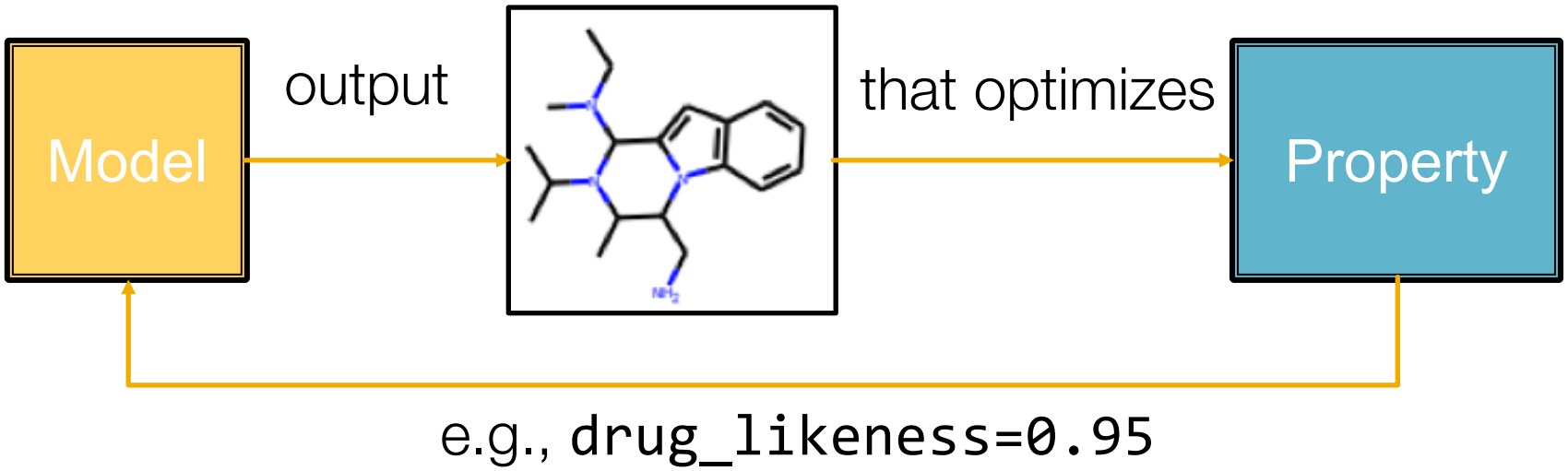
Jure Leskovec, Stanford University

http://cs224w.stanford.edu

[You et al., NeurIPS 2018]

Application: Drug Discovery

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



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

Goal-Directed Graph Generation

Generating graphs that:

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

The Hard Part:

Generating graphs that:

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

Idea: Reinforcement Learning

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

Solution: GCPN

Graph Convolutional Policy Network (GCPN) combines graph representation + RL

Key component of GCPN:

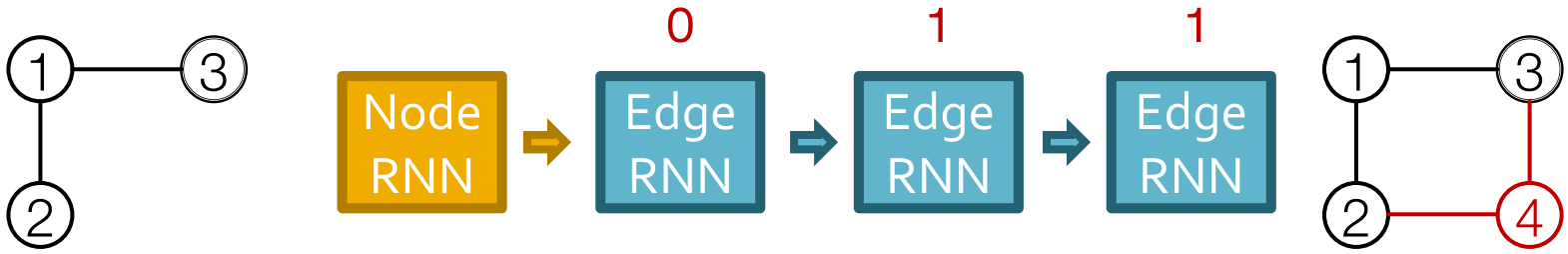
- Graph Neural Network captures graph structural information
- Reinforcement learning guides the generation towards the desired objectives
- Supervised training imitates examples in given datasets

GCPN vs GraphRNN

- **Commonality of GCPN & GraphRNN:**
  - Generate graphs sequentially
  - Imitate a given graph dataset
- **Main Differences:**
  - GCPN uses **GNN** to predict the generation action
    - **Pros:** GNN is more expressive than RNN
    - **Cons:** GNN takes longer time to compute than RNN
  - GCPN further uses **RL** to direct graph generation to our goals
    - RL enables goal-directed graph generation

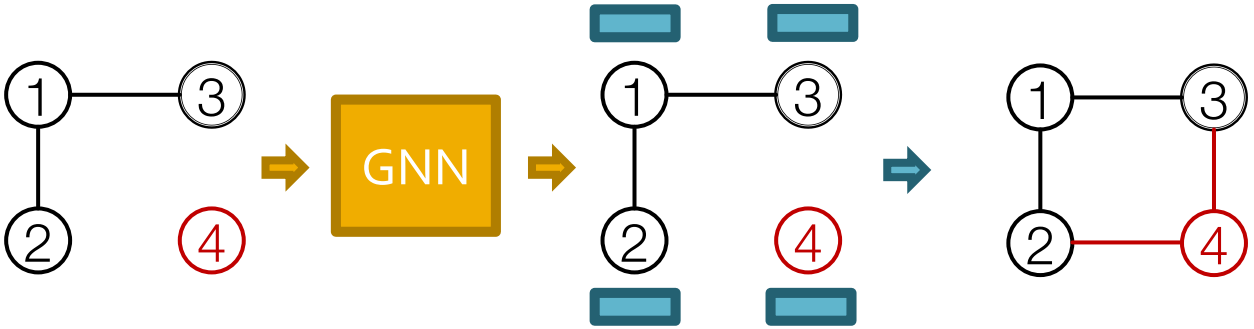
GCPN vs GraphRNN

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



RNN hidden state captures the generated graph so far

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



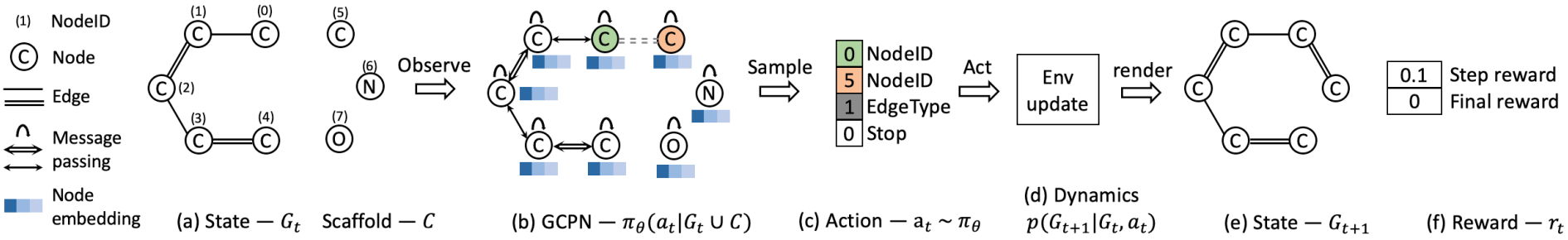
Node embeddings

Predict potential links using node embeddings

Recall the link prediction head:

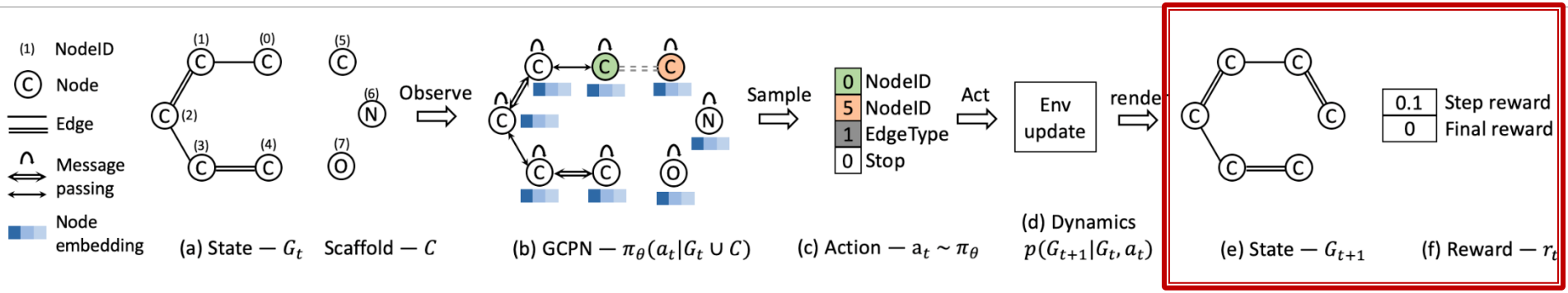
$$\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}) := \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$$

Overview of GCPN



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

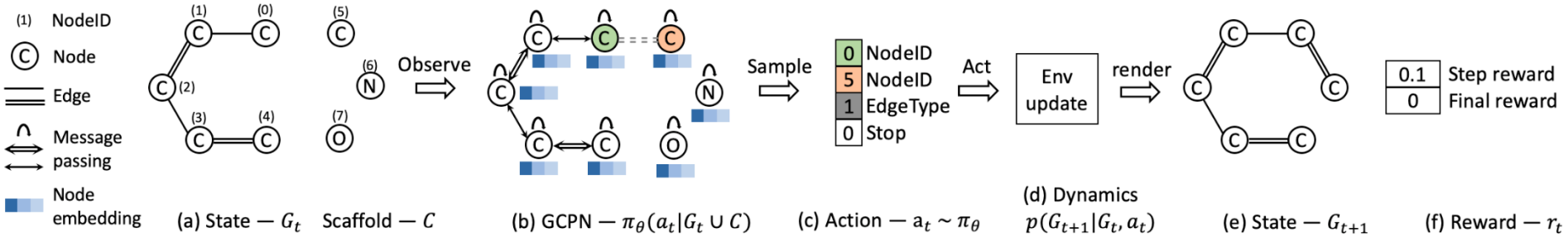
How Do We Set the Reward?



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

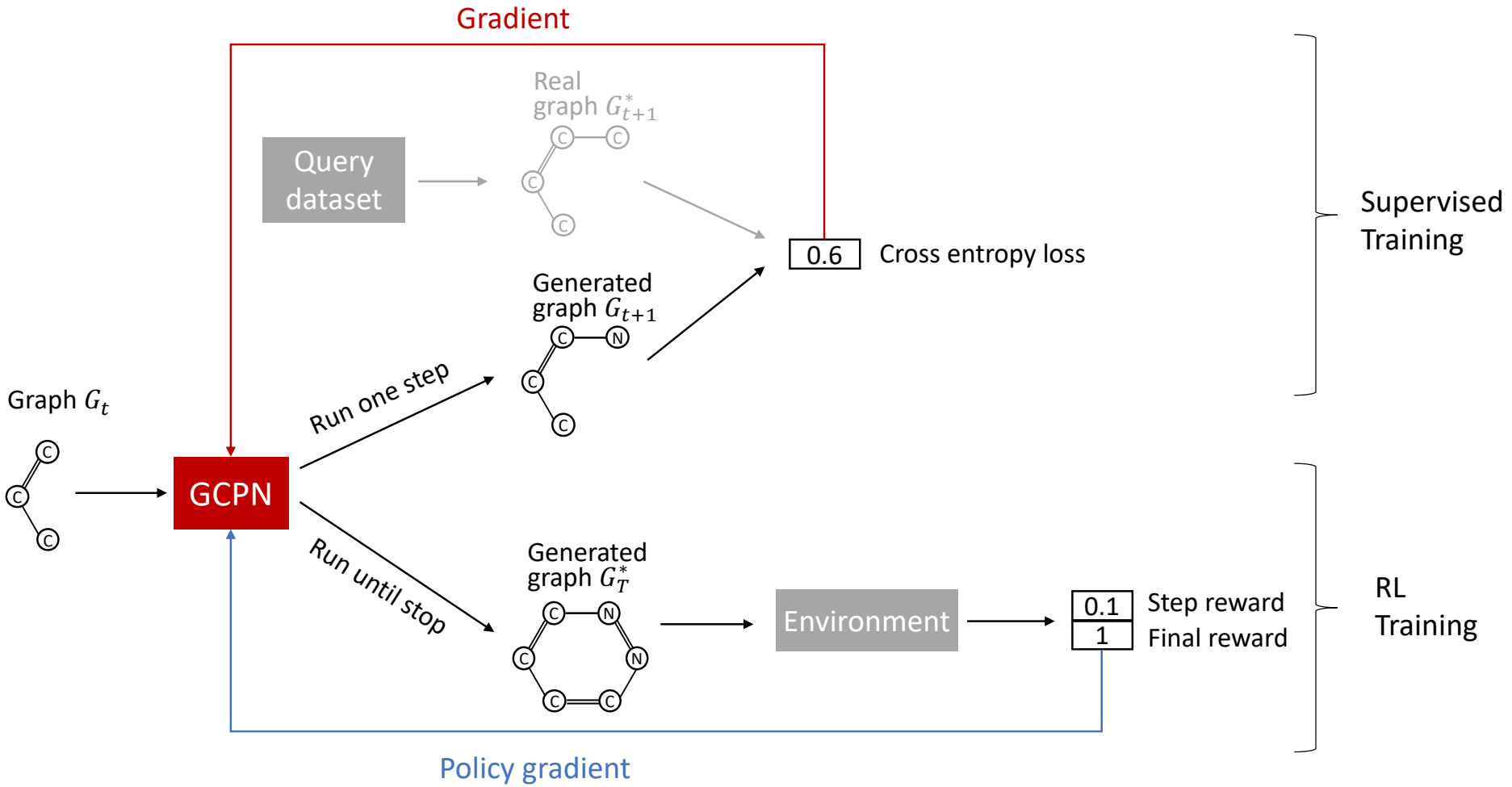
Reward = Final reward + Step reward

How Do We Train?



- **Two parts:**
- **(1) Supervised training:** Train policy by imitating the action given by real observed graphs. Use gradient.
  - We have covered this idea in GraphRNN
- **(2) RL training:** Train policy to optimize rewards. Use standard policy gradient algorithm
  - Refer to any RL course, e.g., CS234

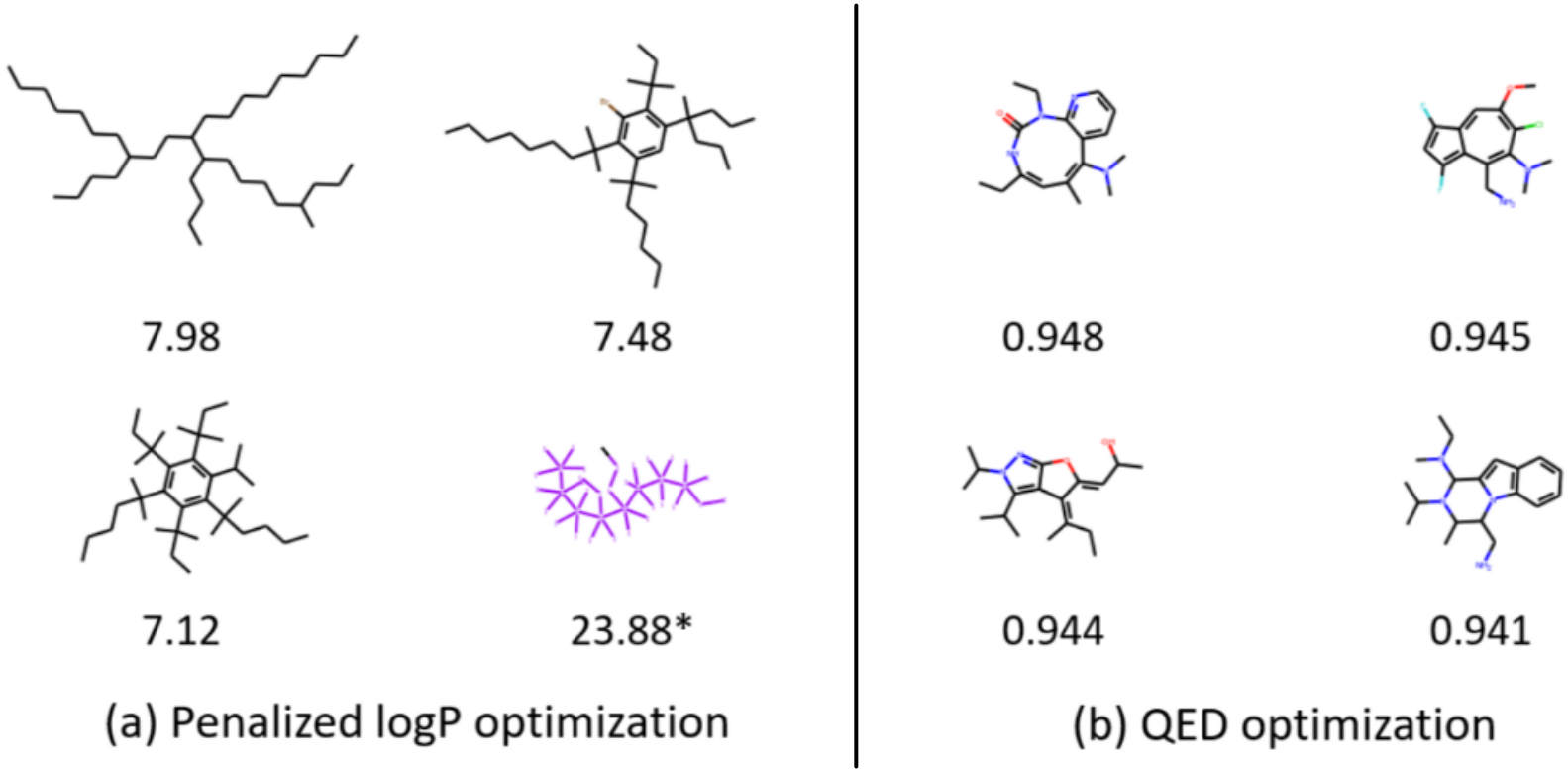
Training GCPN



Qualitative Results

Visualization of GCPN graphs:

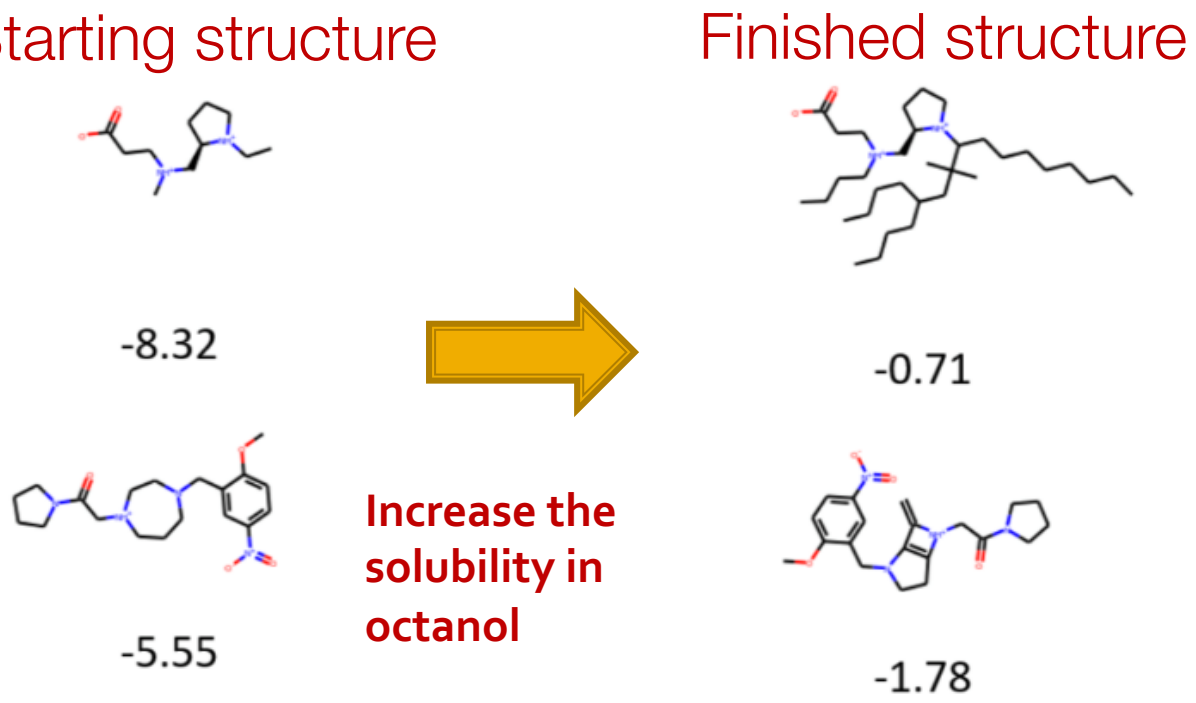
- Property optimization Generate molecules with high specified property score



Qualitative Results

Visualization of GCPN graphs:

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



(c) Constrained optimization of penalized logP

Summary of Graph Generation

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