

# Stanford CS224W: Limitations of Graph Neural Networks

CS224W: Machine Learning with Graphs

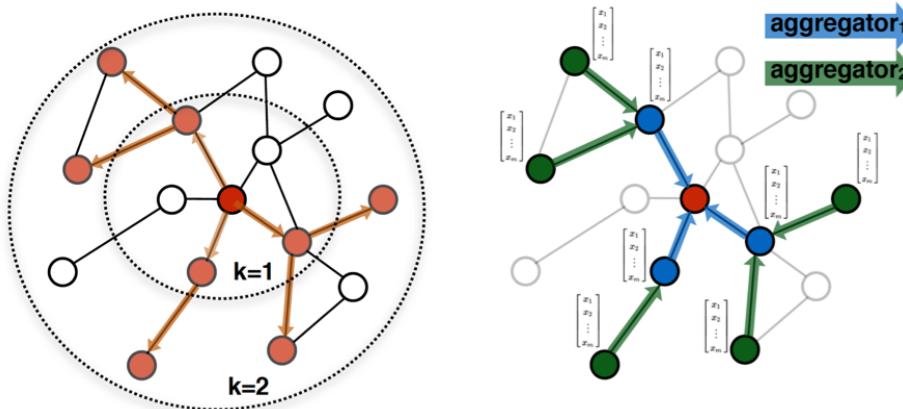
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



# A “Perfect” GNN Model

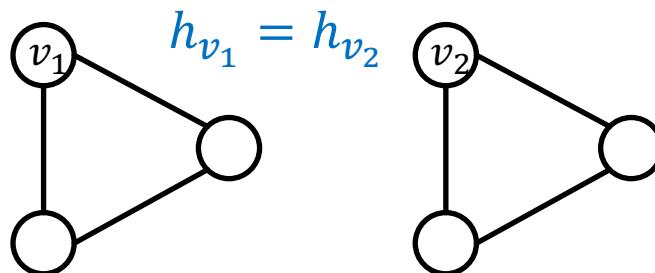
- A thought experiment: What should a perfect GNN do?
  - A  $k$ -layer GNN embeds a node based on the  $K$ -hop neighborhood structure



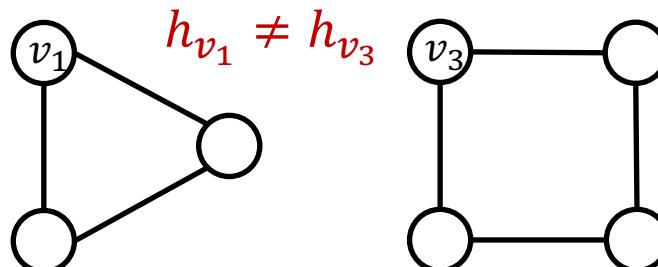
- A perfect GNN should build an **injective function** between **neighborhood structure** (regardless of hops) and **node embeddings**

# 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

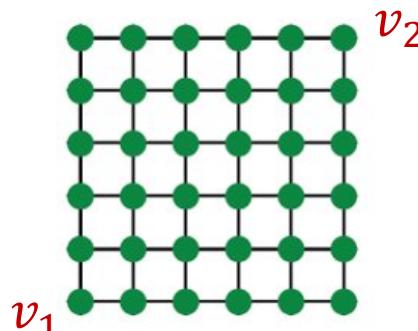


- **Observation 2:** If two nodes have different neighborhood structure, they must have different embeddings



# 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

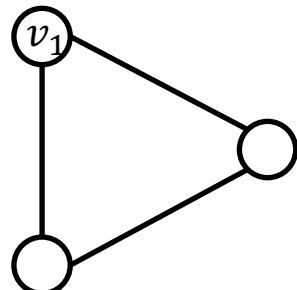


NYC road network

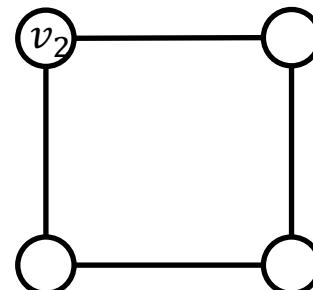
# Imperfections of Existing GNNs

- **Observation 2 often cannot be satisfied:**
  - The GNNs we have introduced so far are not perfect
  - In Lecture 9, we discussed that their expressive power is **upper bounded by the WL test**
  - For example, message passing GNNs **cannot count cycle length**:

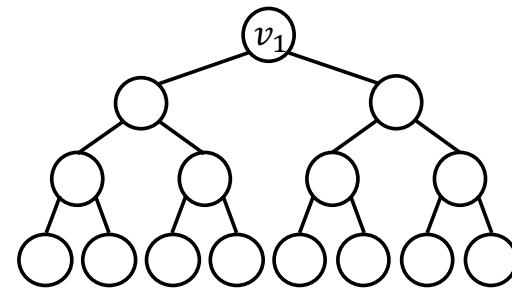
$v_1$  resides in a cycle with length 3



$v_2$  resides in a cycle with length 4



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



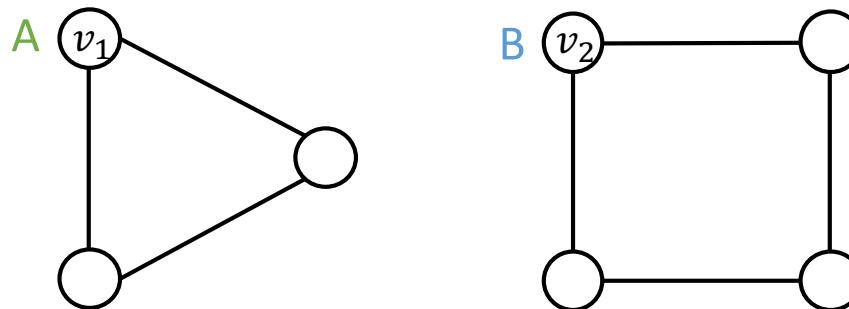
# Plan for the Lecture

- 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:



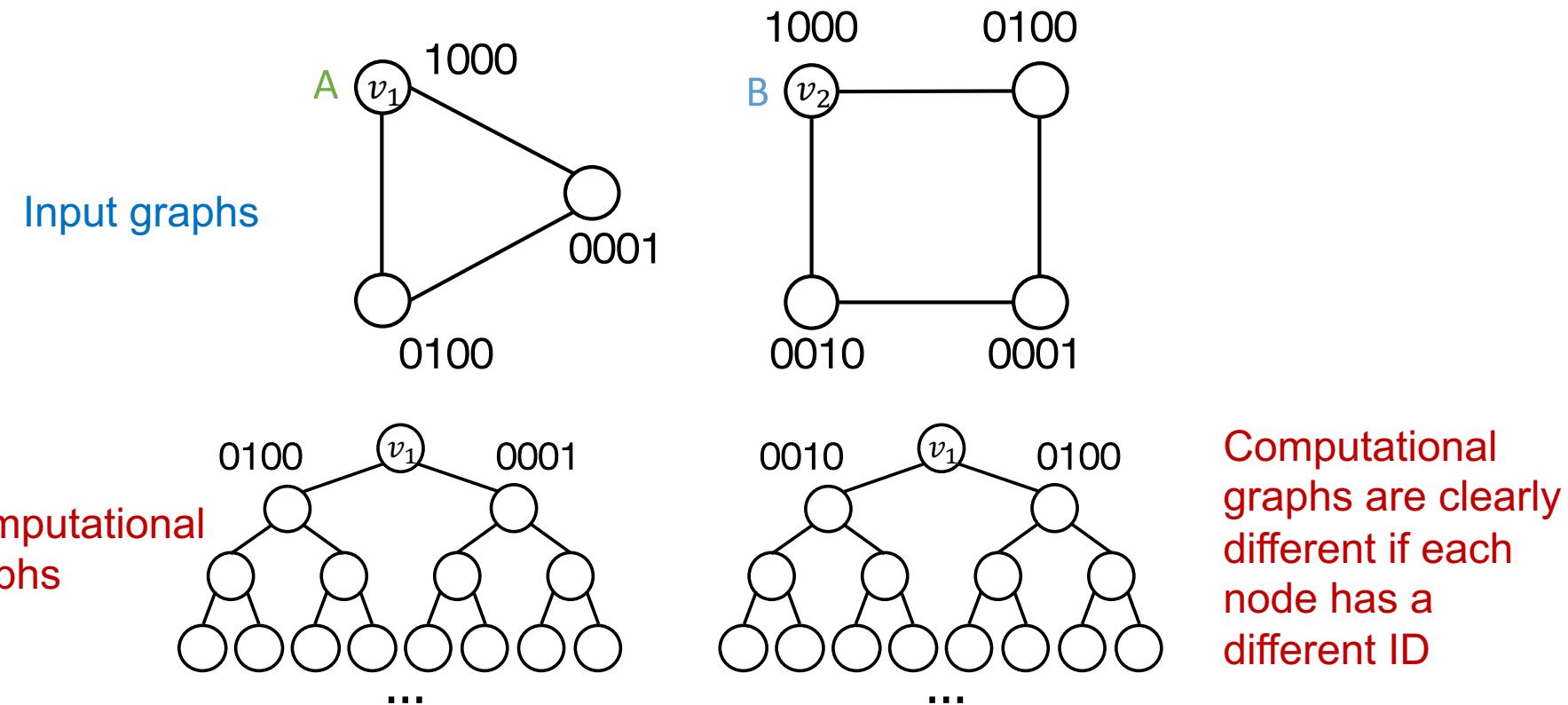
**Two inputs:** nodes  $v_1$  and  $v_2$

**Different labels:** A and B

**Goal:** assign different embeddings to  $v_1$  and  $v_2$

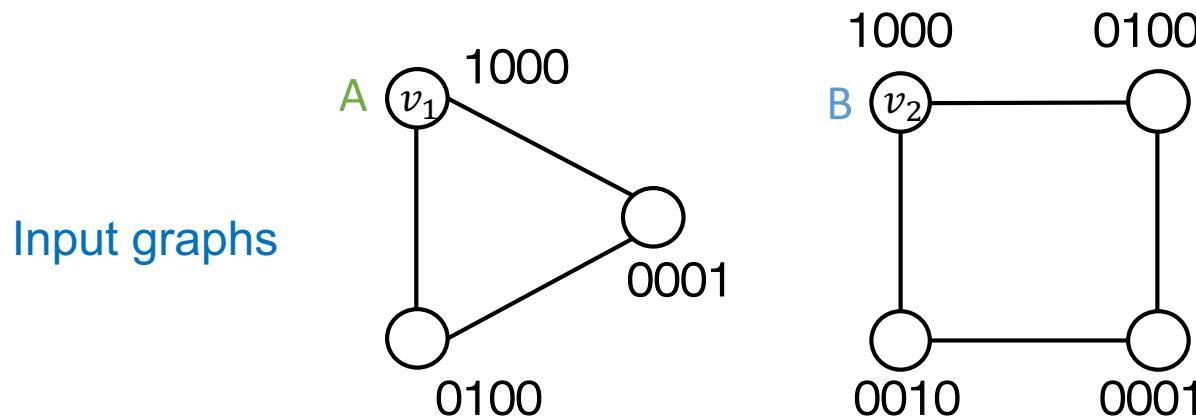
# Naïve Solution is not Desirable

- 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

# Stanford CS224W: Position-aware Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

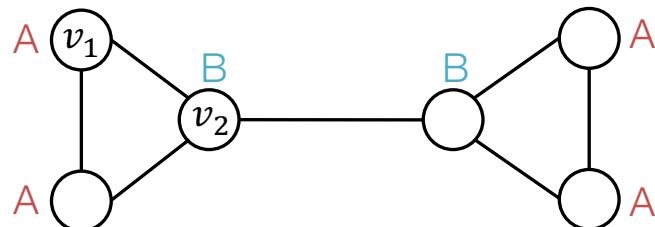
<http://cs224w.stanford.edu>



# Two Types of Tasks on Graphs

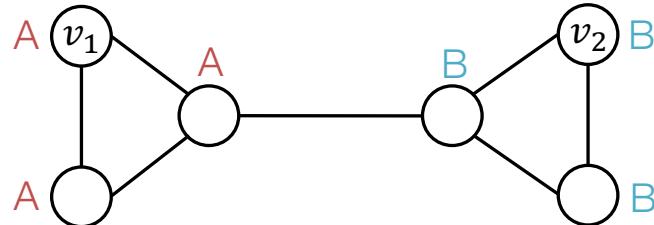
- There are two types of tasks on graphs

## Structure-aware task



- Nodes are labeled by their **structural roles** in the graph

## Position-aware task

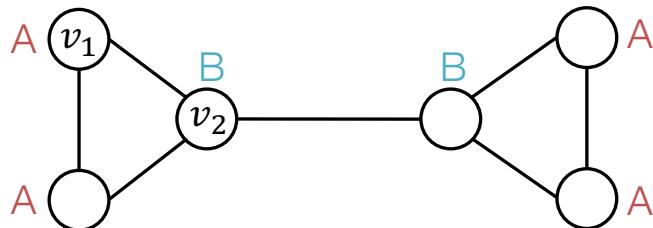


- Nodes are labeled by their **positions** in the graph

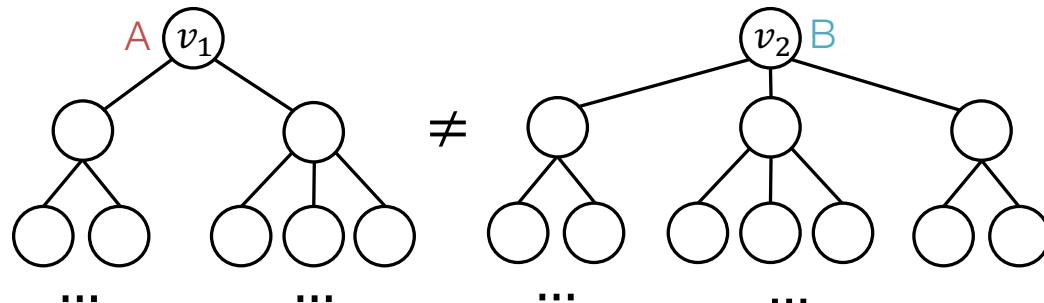
# Structure-aware Tasks

- GNNs often work well for structure-aware tasks

## Structure-aware task



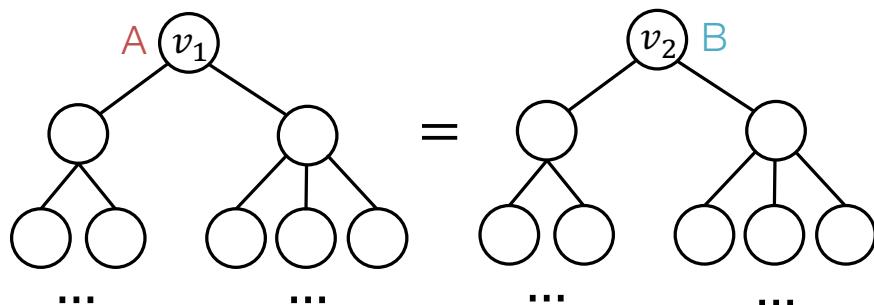
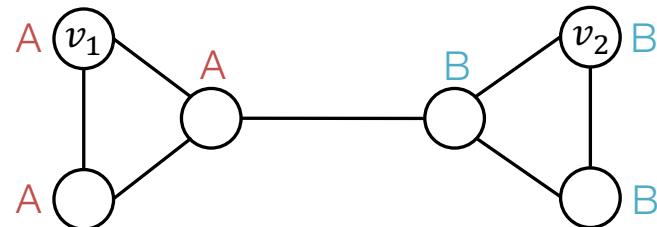
- GNNs work 😊
- Can differentiate  $v_1$  and  $v_2$  by using different computational graphs



# Position-aware Tasks

- GNNs will always fail for position-aware tasks

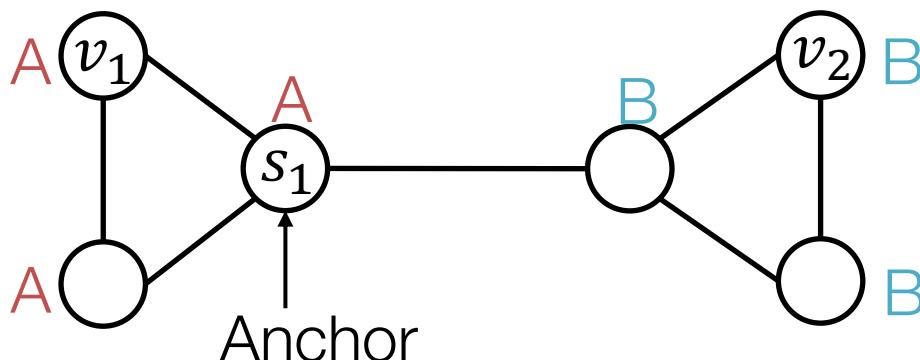
## Position-aware task



- 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**

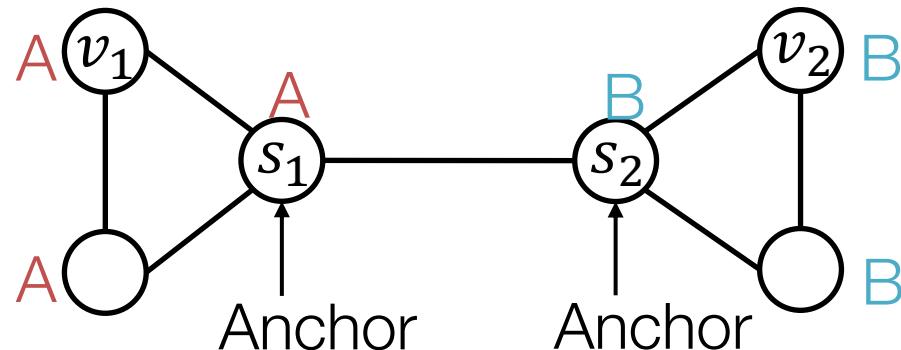


Relative  
Distances

	$s_1$
$v_1$	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

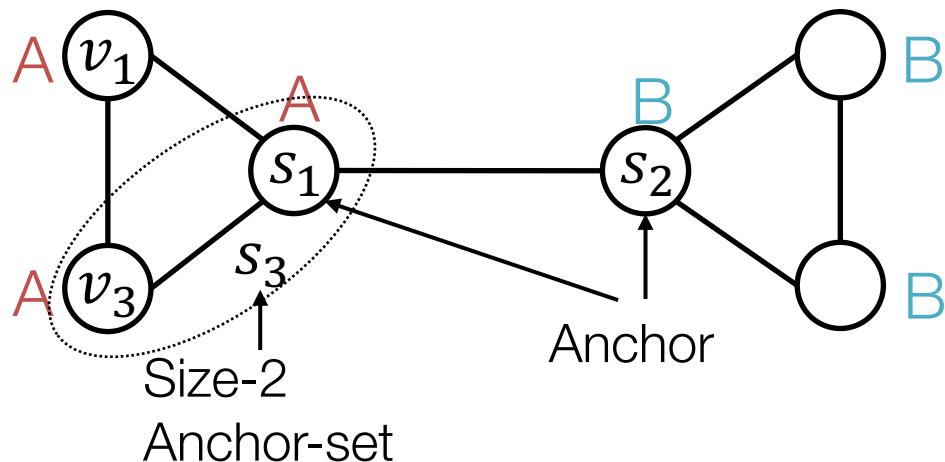


Relative  
Distances

	$s_1$	$s_2$
$v_1$	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 anchor-set
- **Observation:** Large anchor-sets can sometimes provide more precise position estimate
  - We can save the total number of anchors



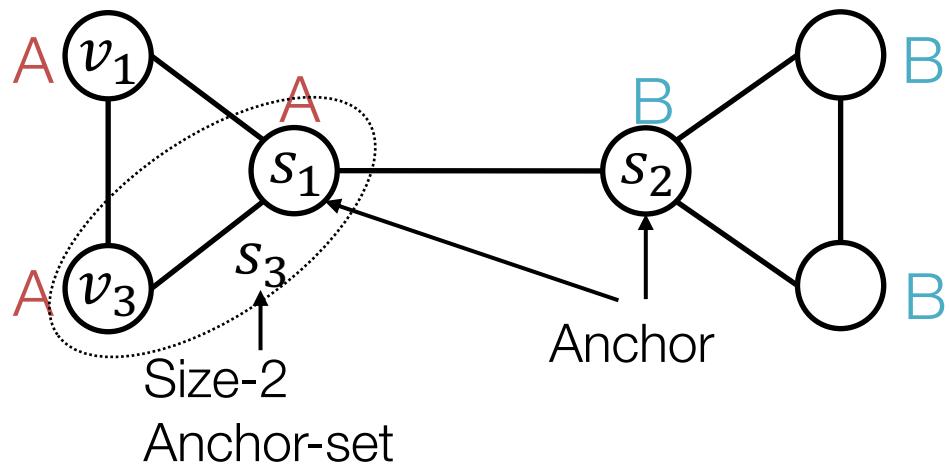
Relative Distances

	$s_1$	$s_2$	$s_3$
$v_1$	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

# 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, **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
  - Refer to the Position-aware GNN paper for more details

# Stanford CS224W: Identity-Aware Graph Neural Networks

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



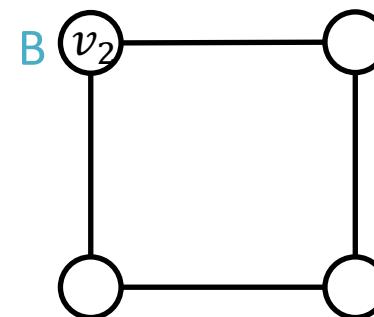
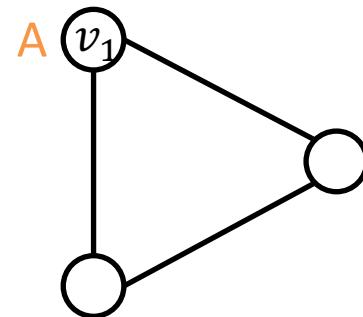
# More Failure Cases for GNNs

- 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

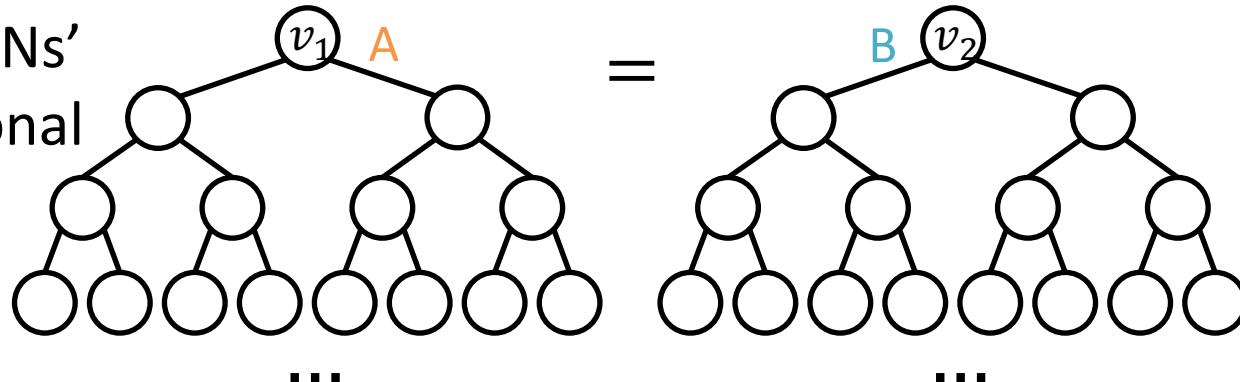
# GNN Failure 1: Node-level Tasks

Different Inputs but the same computational graph → GNN fails

Example input  
graphs



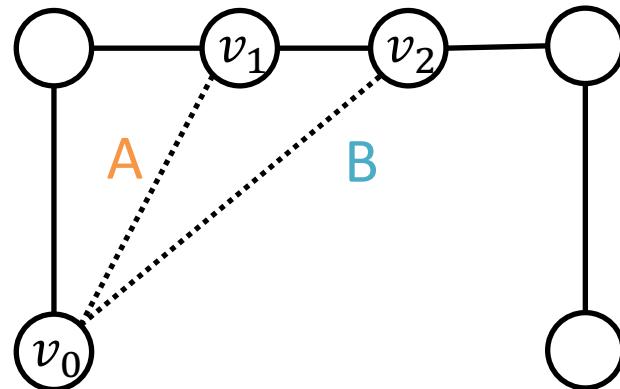
Existing GNNs'  
computational  
graphs



# GNN Failure 2: Edge-level Tasks

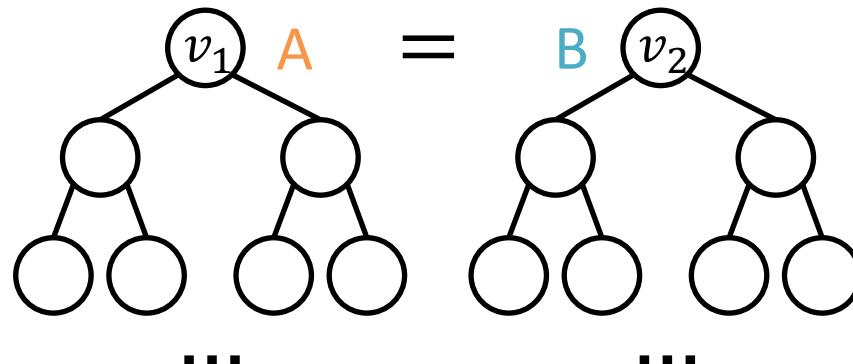
Different Inputs but the same computational graph → GNN fails

Example input  
graphs



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

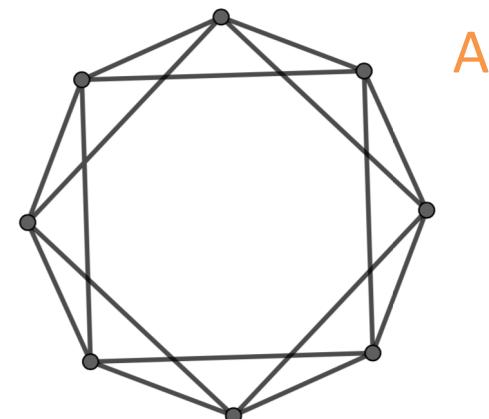
Existing GNNs'  
computational  
graphs



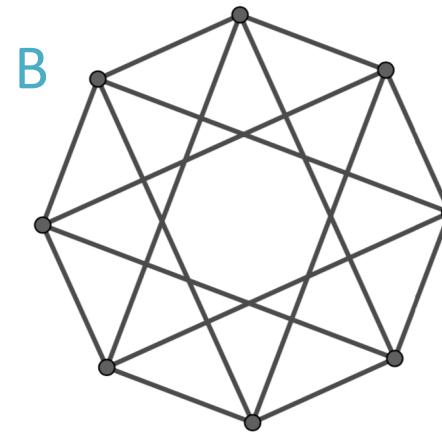
# GNN Failure 3: Graph-level Tasks

Different Inputs but the same computational graph → GNN fails

Example input graphs



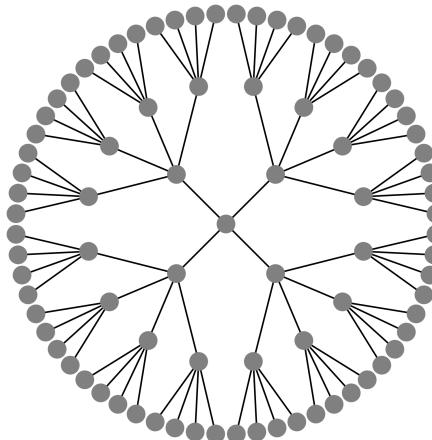
A



B

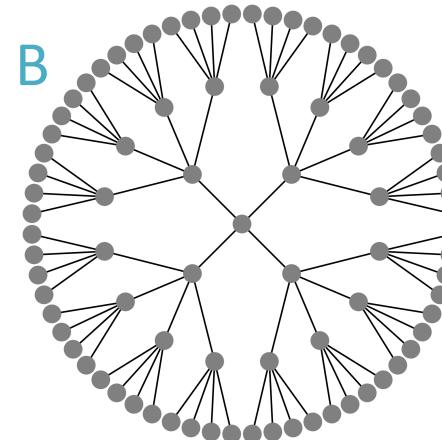
We look at embeddings for each node

For each node:



A

=



For each node:

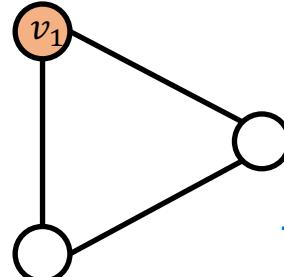
Existing GNNs' computational graphs

# Idea: Inductive Node Coloring

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

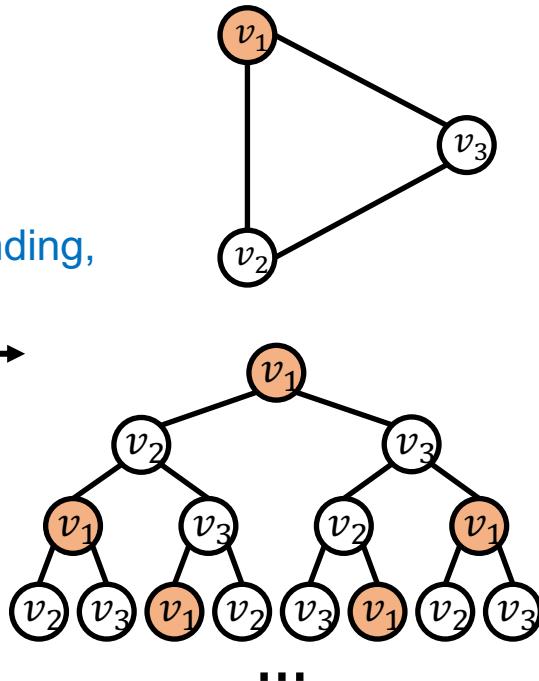
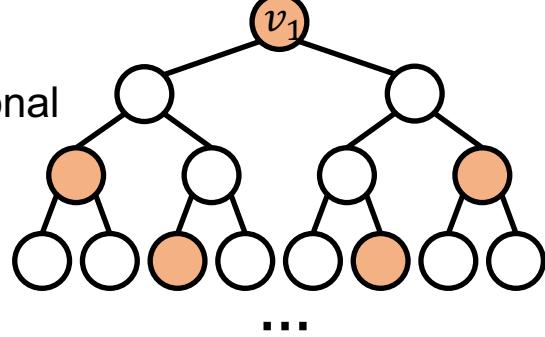
- The node we want to embed
- The rest of nodes

Input graph



To assist understanding,  
we label the nodes

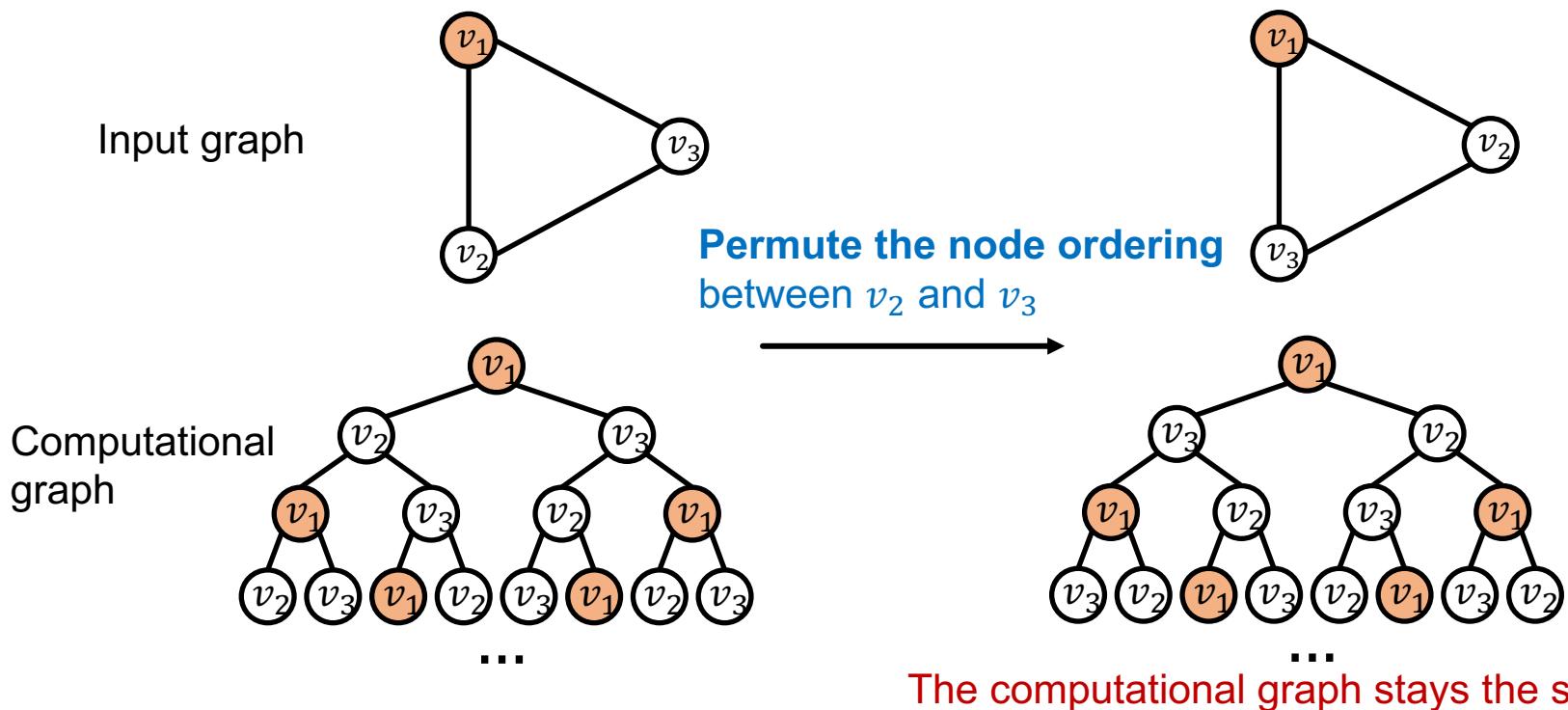
Computational  
graph



# Idea: Inductive Node Coloring

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

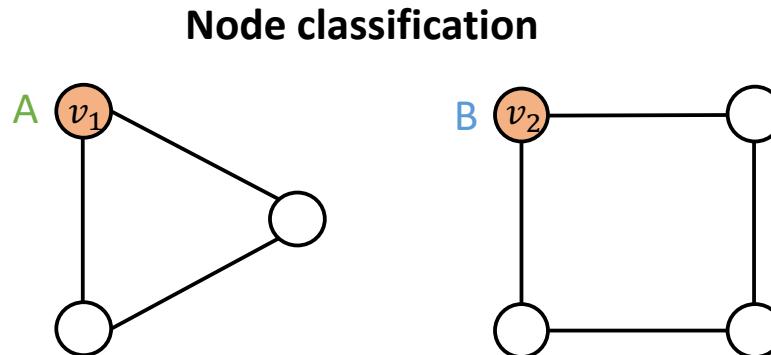
- The node we want to embed
- The rest of nodes



# Inductive Node Coloring – Node level

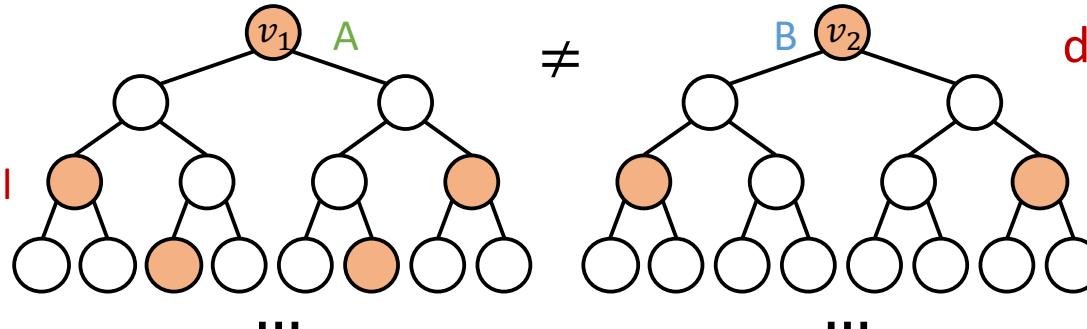
- Inductive node coloring can help **node classification**

Example input graphs



We color root nodes with identity

ID-GNNs' computational graphs



Different computational graphs  
→ Successfully differentiate nodes

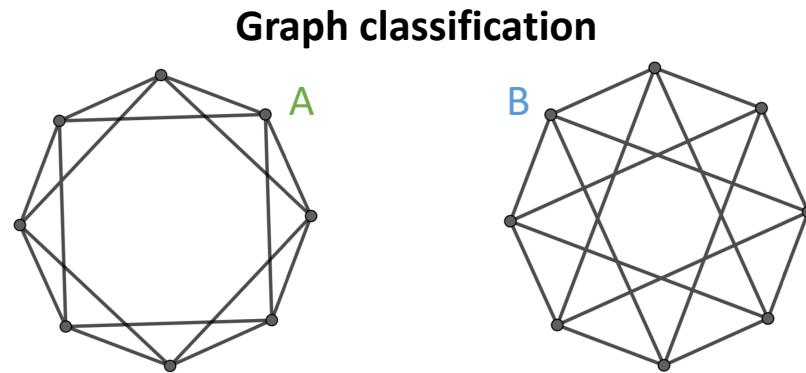
Two types of nodes:

- node with augmented identity (orange circle)
- node without augmented identity (white circle)

# Inductive Node Coloring – Graph Level

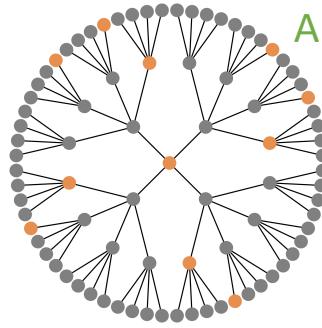
- Inductive node coloring can help **graph classification**

Example input graphs



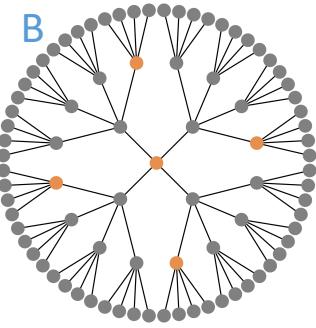
We color root nodes with identity

For each node:



≠

For each node:



ID-GNNs' computational graphs

Two types of nodes:

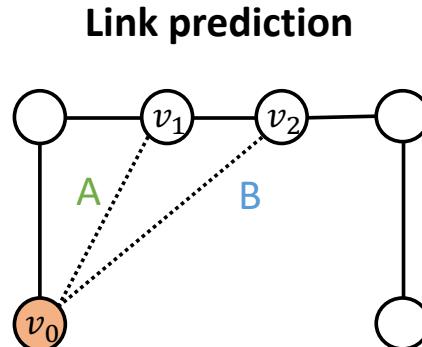
- node with augmented identity
- node without augmented identity

Different computational graphs  
→ Successful differentiate graphs

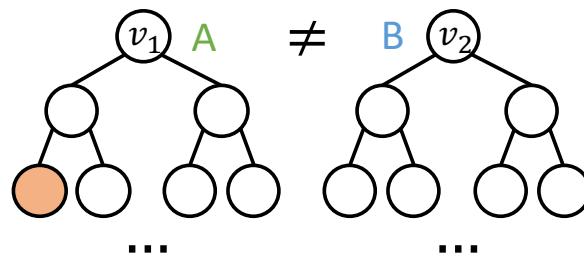
# Inductive Node Coloring – Edge Level

- Inductive node coloring can help **link prediction**

Example input graphs



ID-GNNs' computational graphs



Two types of nodes:

- node with augmented identity (orange)
- node without augmented identity (white)

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

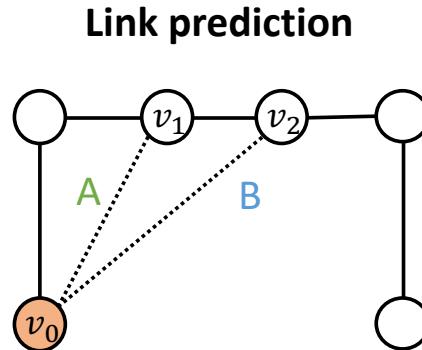
- We color one of the nodes ( $v_0$ )
- We then embed the other node in the node pair ( $v_1$  or  $v_2$ )
- We use the node embedding for  $v_1$  or  $v_2$  conditioned on  $v_0$  being colored or not to make edge-level prediction

Different computational graphs  
→ Successfully differentiate edges

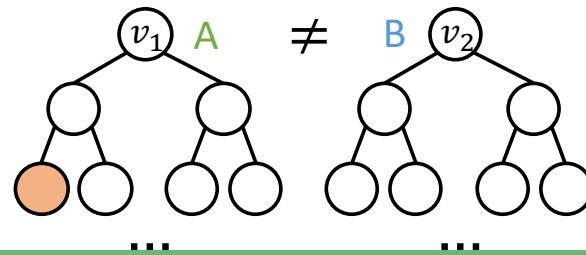
# Inductive Node Coloring – Edge Level

- Inductive node coloring can help **link prediction**

Example input graphs



ID-GNNs'  
computational  
graphs



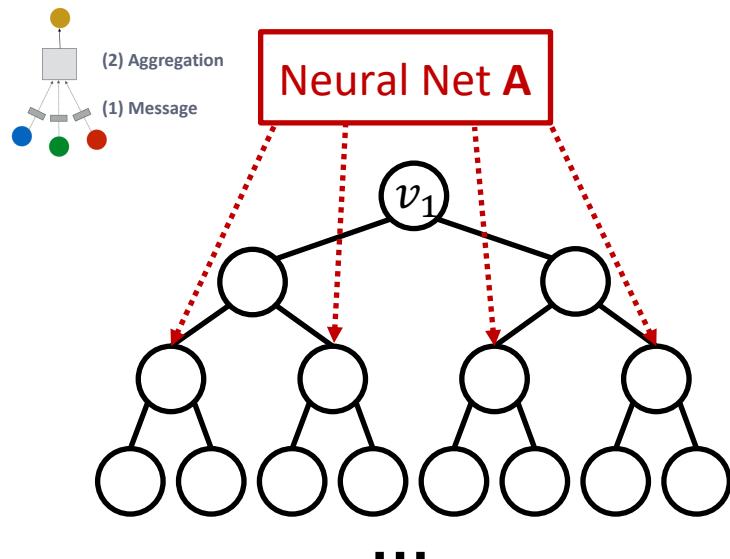
An edge-level task involves classifying a pair of nodes:

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

Two How to build a GNN using node coloring?

# Identity-aware GNN

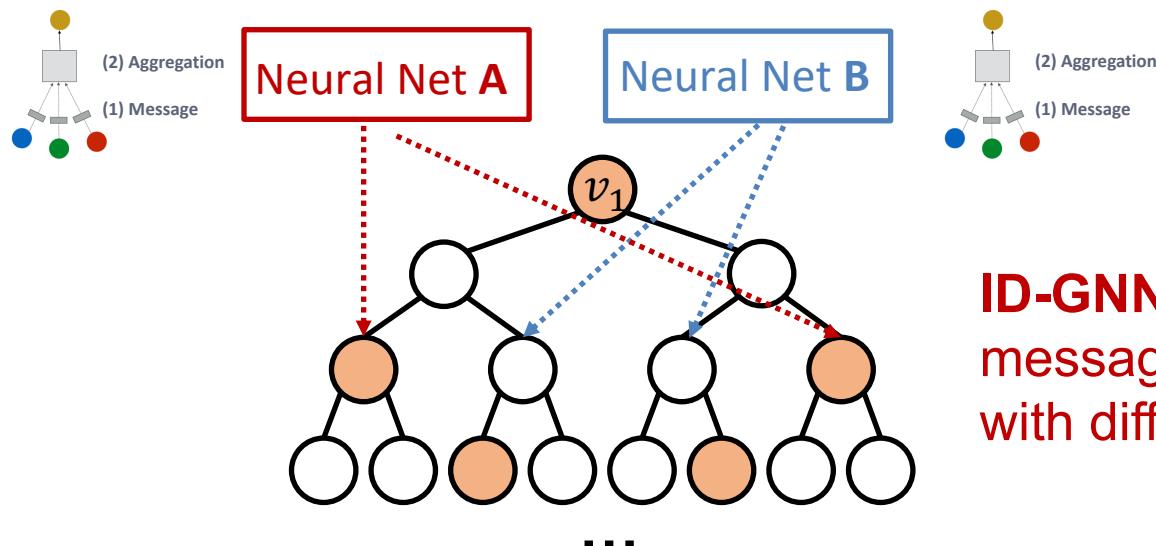
- Utilize **inductive node coloring** in embedding computation
  - Idea: **Heterogenous message passing**
    - Normally, a GNN applies **the same message/aggregation computation to all the nodes**



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

# Identity-aware GNN

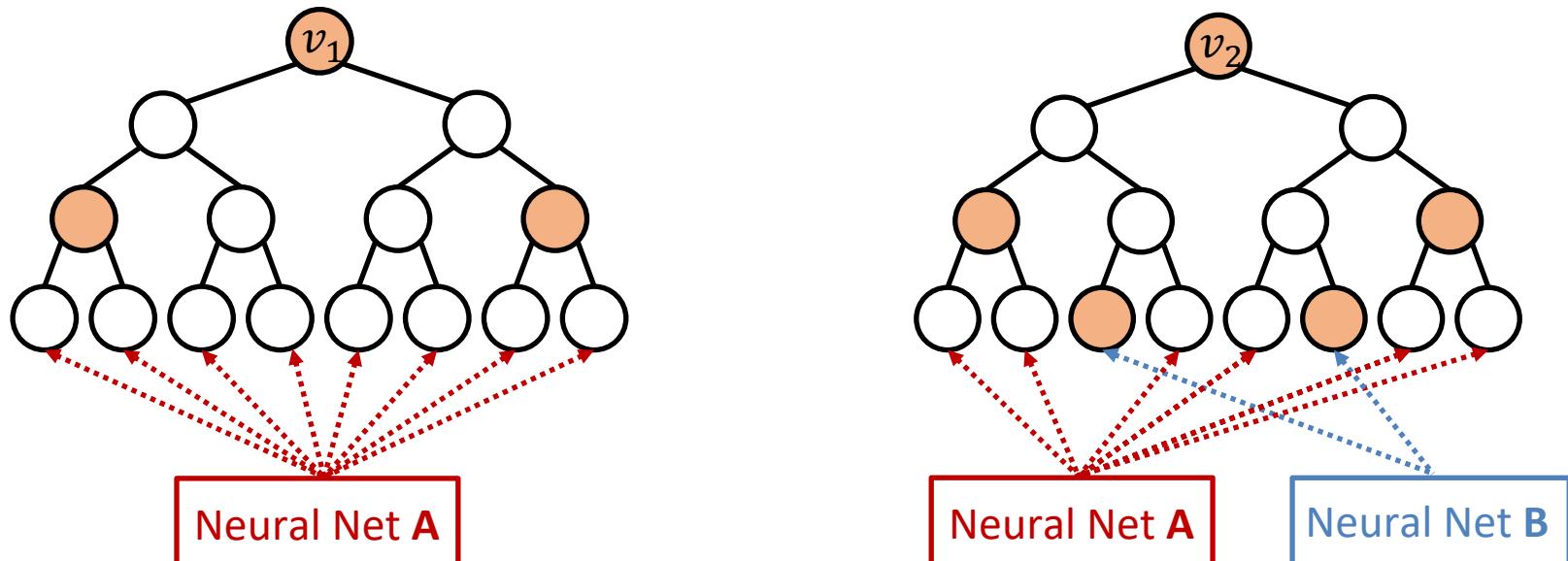
- Idea: Heterogenous message passing
  - Heterogenous: different types of message passing is applied to different nodes
  - An ID-GNN applies different message/aggregation to nodes with different colorings



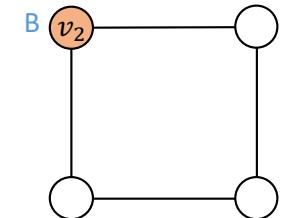
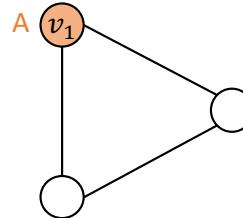
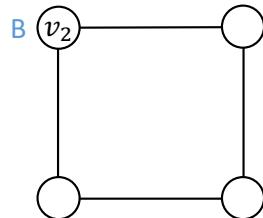
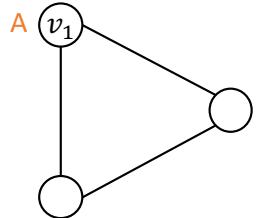
**ID-GNN:** At a given layer, different message/aggregation to nodes with different colorings

# Identity-aware GNN

- 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

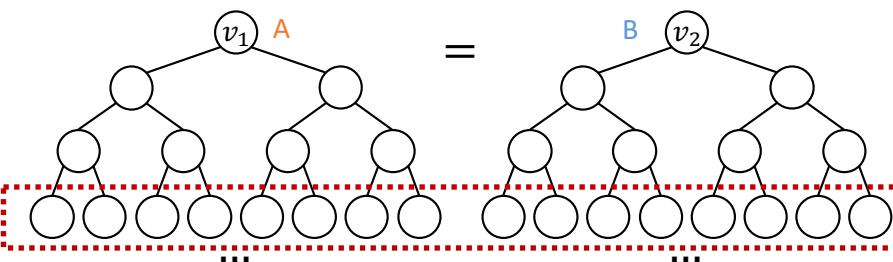


# GNN vs ID-GNN

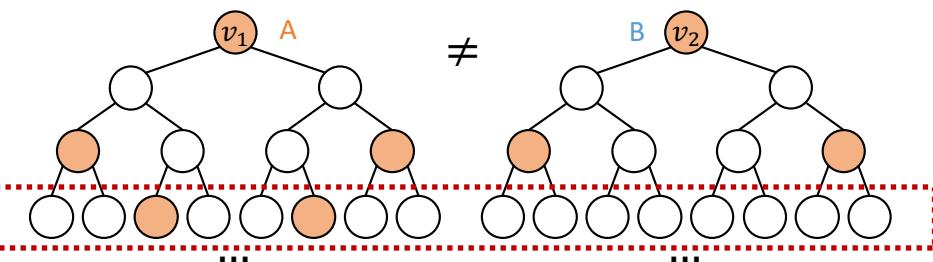


Goal: classify  $v_1$  and  $v_2$

GNN computational graph



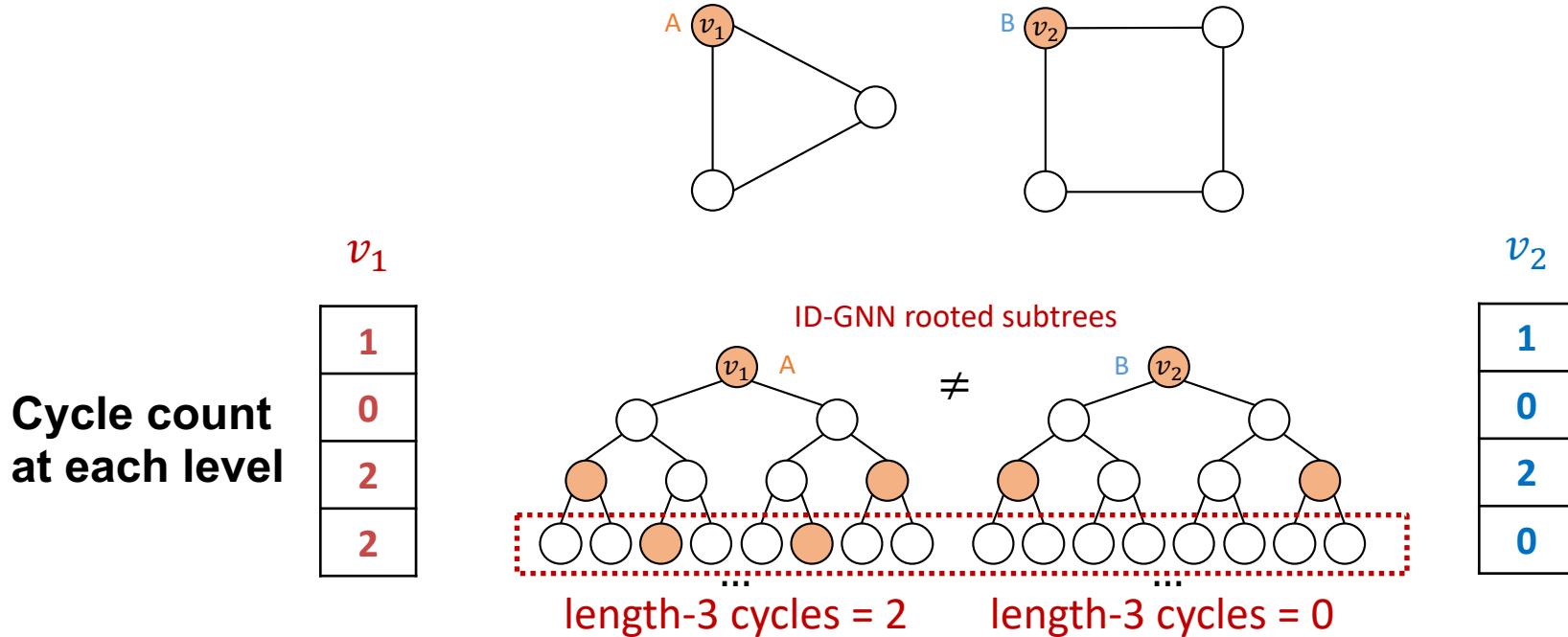
ID-GNN rooted subtrees



From the node coloring, we can tell that:  
 $v_1$ : length-3 cycles = 2       $v_2$ : length-3 cycles = 0

- 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 propose 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, ...)

# **Stanford CS224W:** **Robustness of** **Graph Neural Networks**

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>

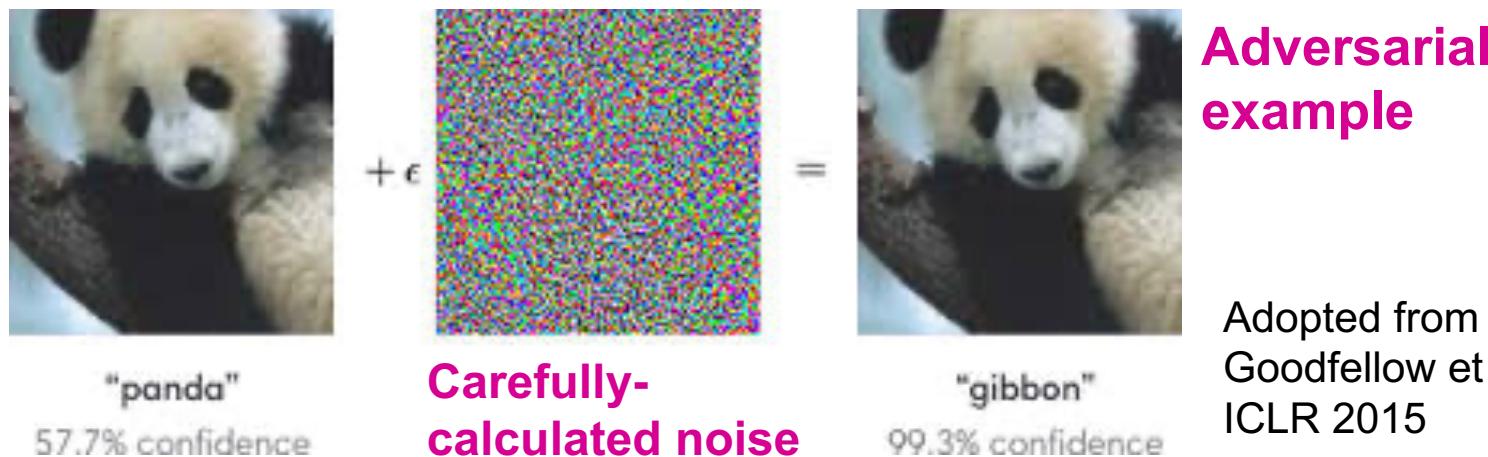


# Deep Learning Performance

- Recent years have seen **impressive performance of deep learning models in a variety of applications.**
  - Ex) 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.

# Implication 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

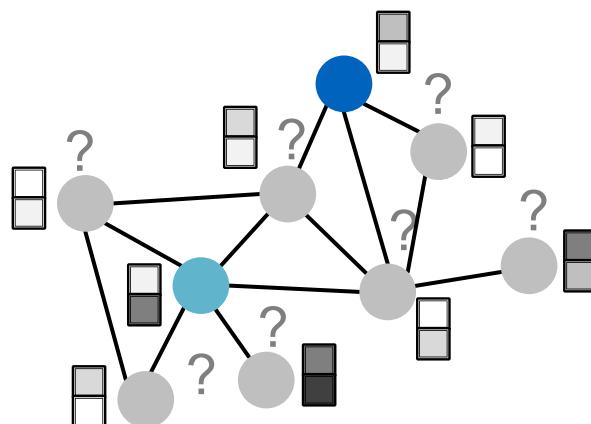
# Robustness of GNNs

- **This lecture: 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.

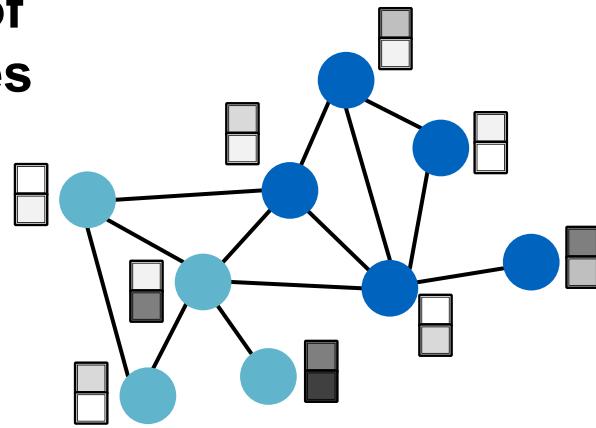
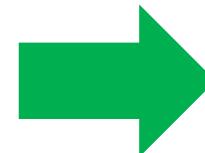
# Setting to Study GNNs' Robustness

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

?: Unlabeled



Predict labels of  
unlabeled nodes

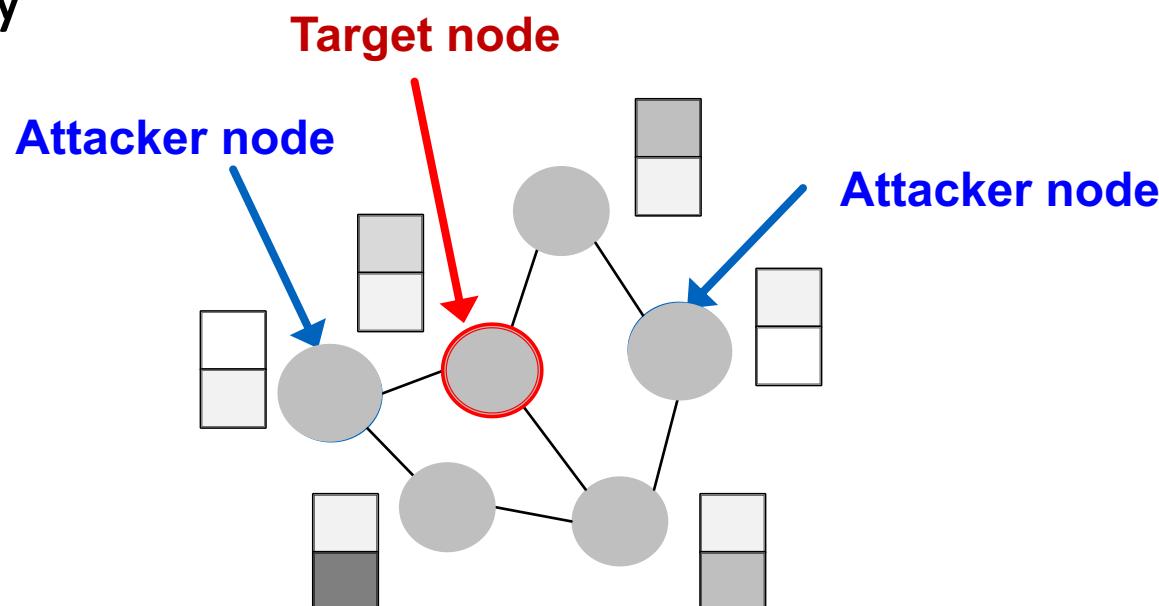


# Roadmap

- 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 attack problem as an optimization problem.**
- **We empirically see how vulnerable GCN's prediction is to the adversarial attack.**

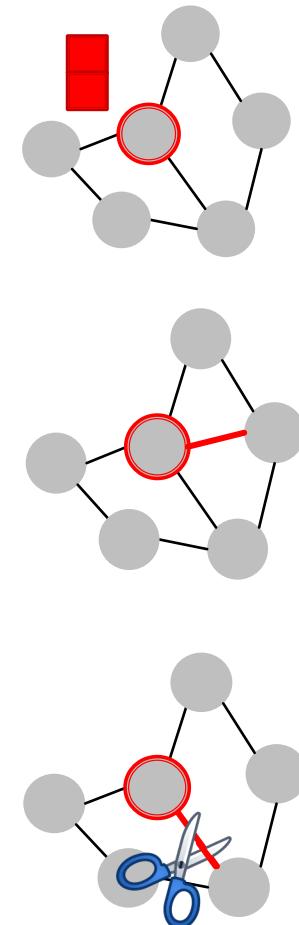
# Attack Possibilities

- 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



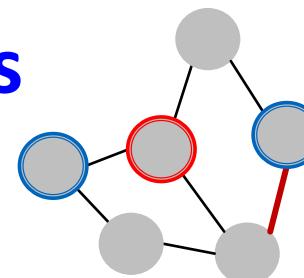
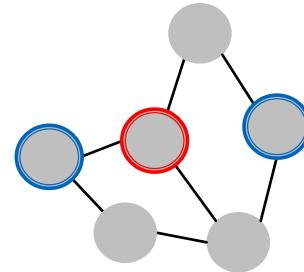
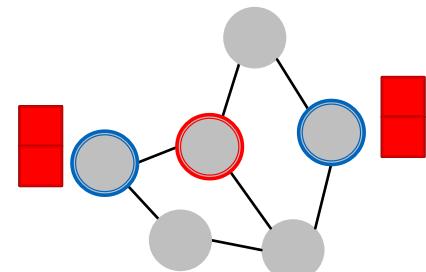
# Attack Possibilities: Direct Attack

- **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

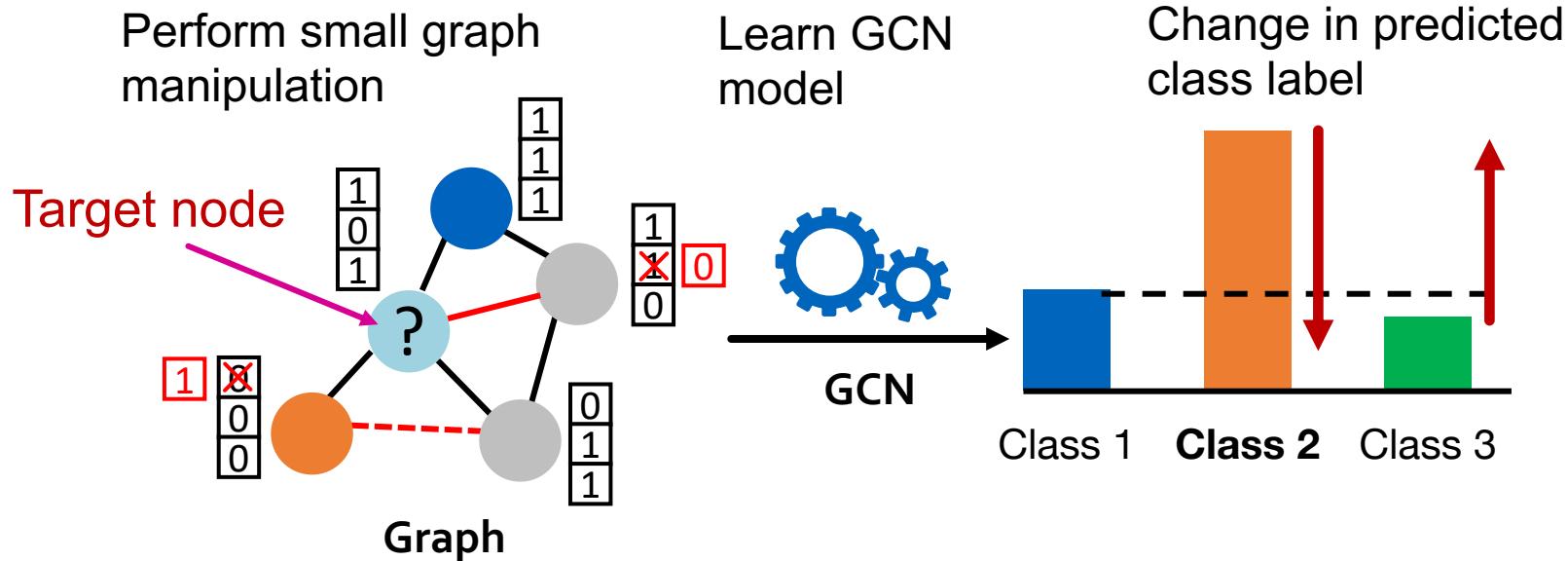


# Formalizing Adversarial Attacks

## ■ 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:**
  - $A$ : adjacency matrix,  $X$ : feature matrix
- **Manipulated graph (after adding noise):**
  - $A'$ : adjacency matrix,  $X'$ : feature matrix
- **Assumption:**  $(A', X') \approx (A, X)$ 
  - 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)

- **Target node:**  $v \in V$
- GCN learned over the **original graph**  
$$\theta^* = \operatorname{argmin}_{\theta} \mathcal{L}_{train}(\theta; A, X)$$
- GCN's original prediction on the **target node**:

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

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

# Mathematical Formulation (3)

- GCN learned over the **manipulated graph**

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

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

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

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

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

# Mathematical Formulation (4)

- Change of prediction on target node  $v$ :

$$\Delta(v; A', X') =$$

$$\log f_{\theta^{*'}}(A', X')_{v, c_v^{*'}} - \log f_{\theta^{*'}}(A', X')_{v, c_v^*}$$

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

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



Want to increase  
this term



Want to decrease  
this term

# Mathematical Formulation (5)

- **Final optimization objective:**

$$\operatorname{argmax}_{A', X'} \Delta(v; A', X')$$

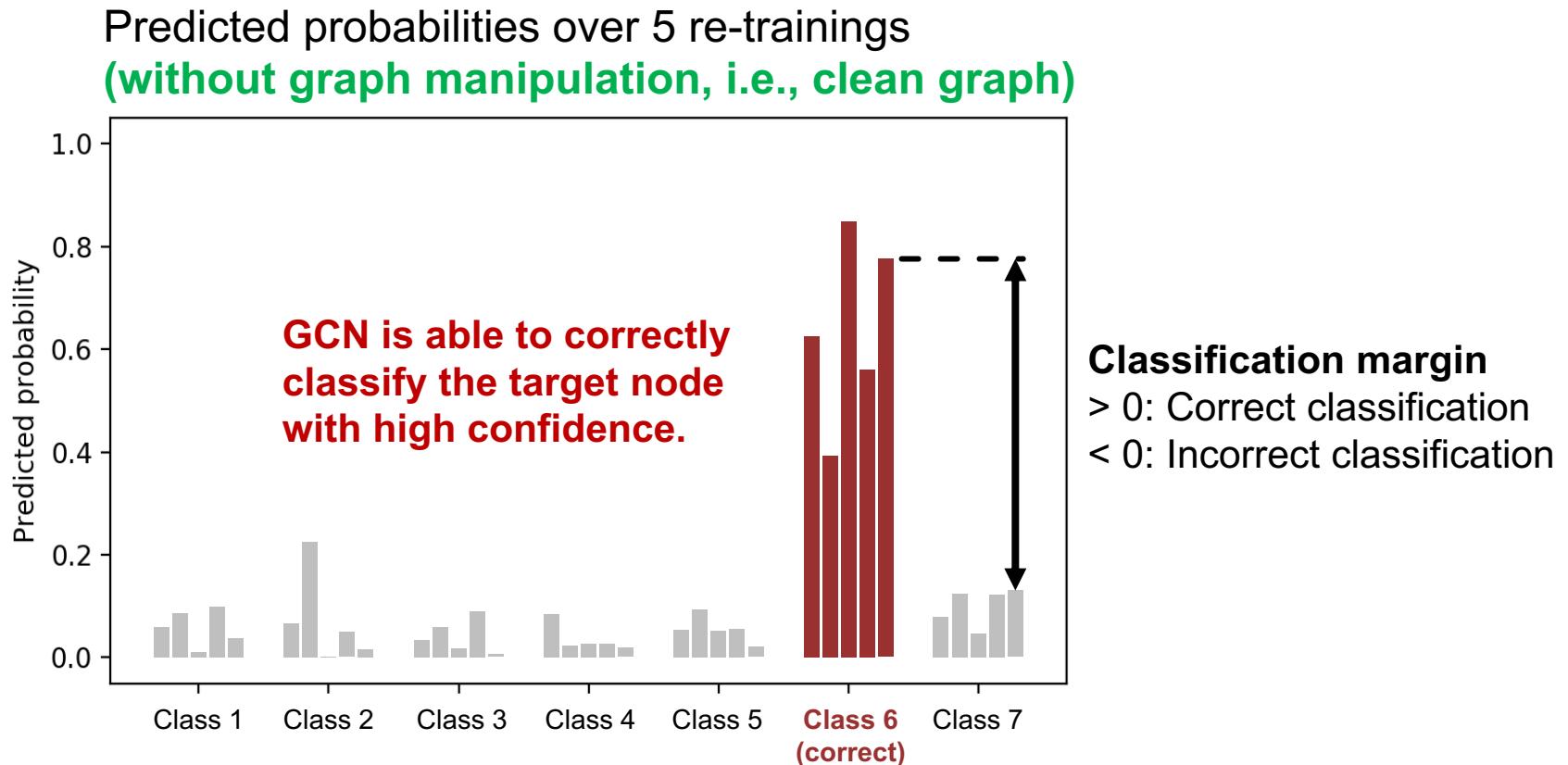
subject to  $(A', X') \approx (A, X)$

- **Challenges in optimizing the objective**

- Adjacency matrix  $A'$  is a discrete object: gradient-based optimization cannot be used.
- For every modified graph  $A'$  and  $X'$ , GCN needs to be re-trained (this is computationally expensive):
  - $\theta^{*'} = \operatorname{argmin}_{\theta} \mathcal{L}_{train}(\theta; A', X')$
- Several approximations are proposed to make the optimization tractable [Zügner et al. KDD2018].

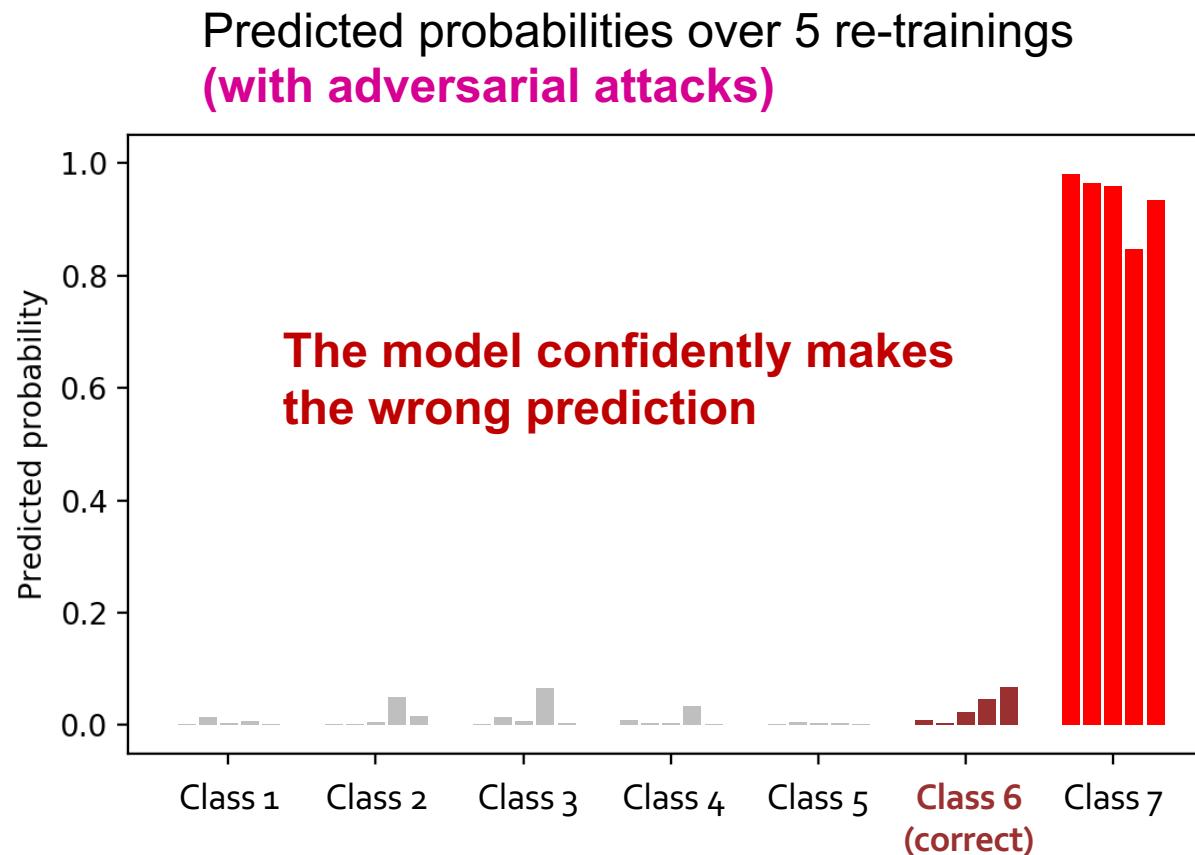
# Experiments: Adversarial Attack

Semi-supervised node classification with GCN on a paper citation network (2,800 nodes, 8,000 edges).



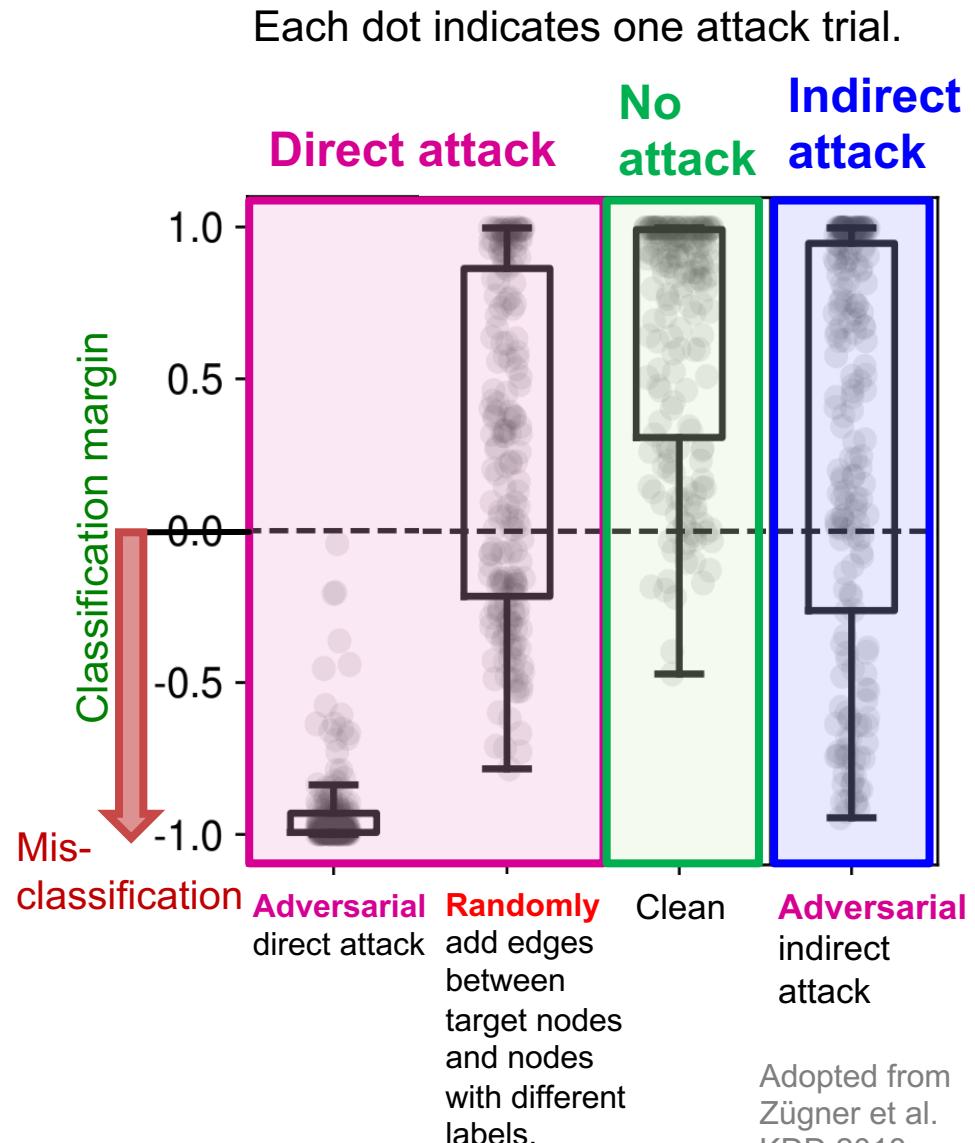
# Experiments: Adversarial Attack

GCN's prediction after carefully modifying just 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.**

# Scaling Up Graph Neural Networks to Large Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



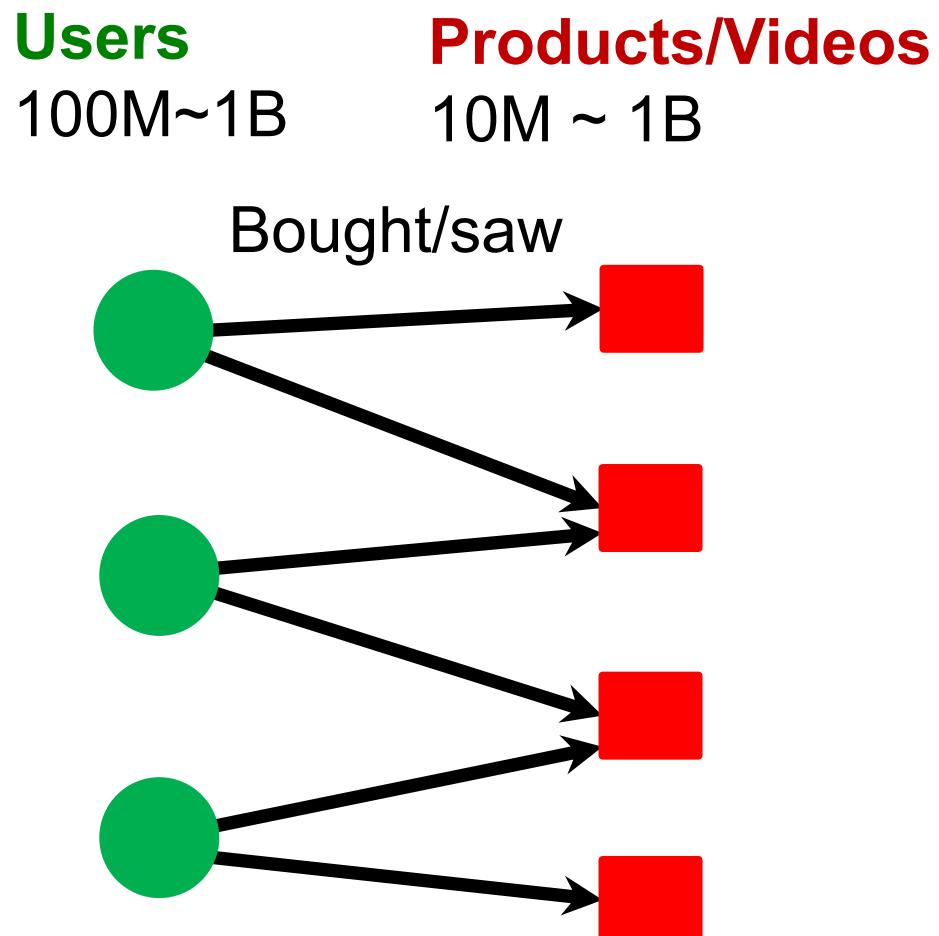
# Graphs in Modern Applications

## ■ Recommender systems:

- Amazon
- YouTube
- Pinterest
- Etc.

## ■ ML tasks:

- Recommend items  
(link prediction)
- Classify users/items  
(node classification)



# Graphs in Modern Applications

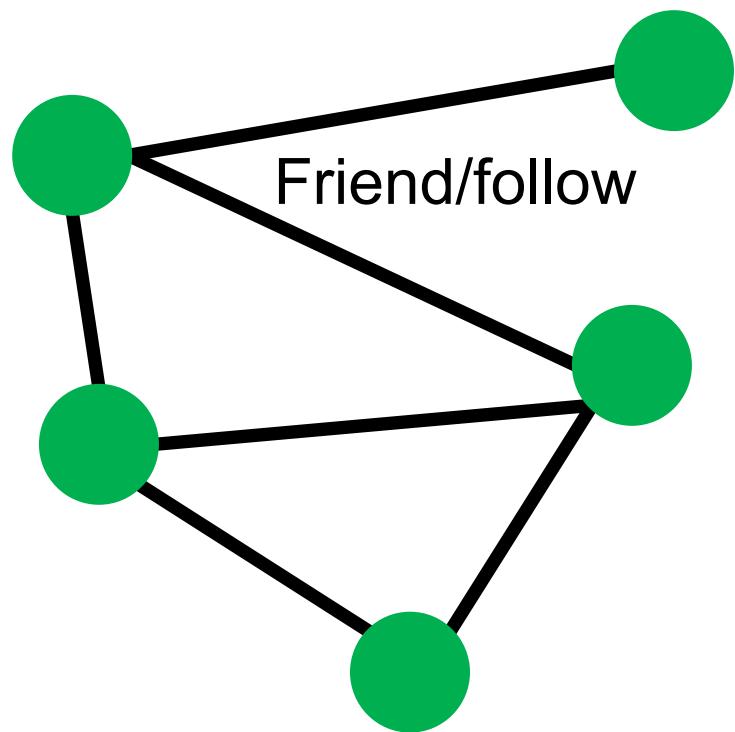
- **Social networks:**

- Facebook
- Twitter
- Instagram
- Etc.

- **ML tasks:**

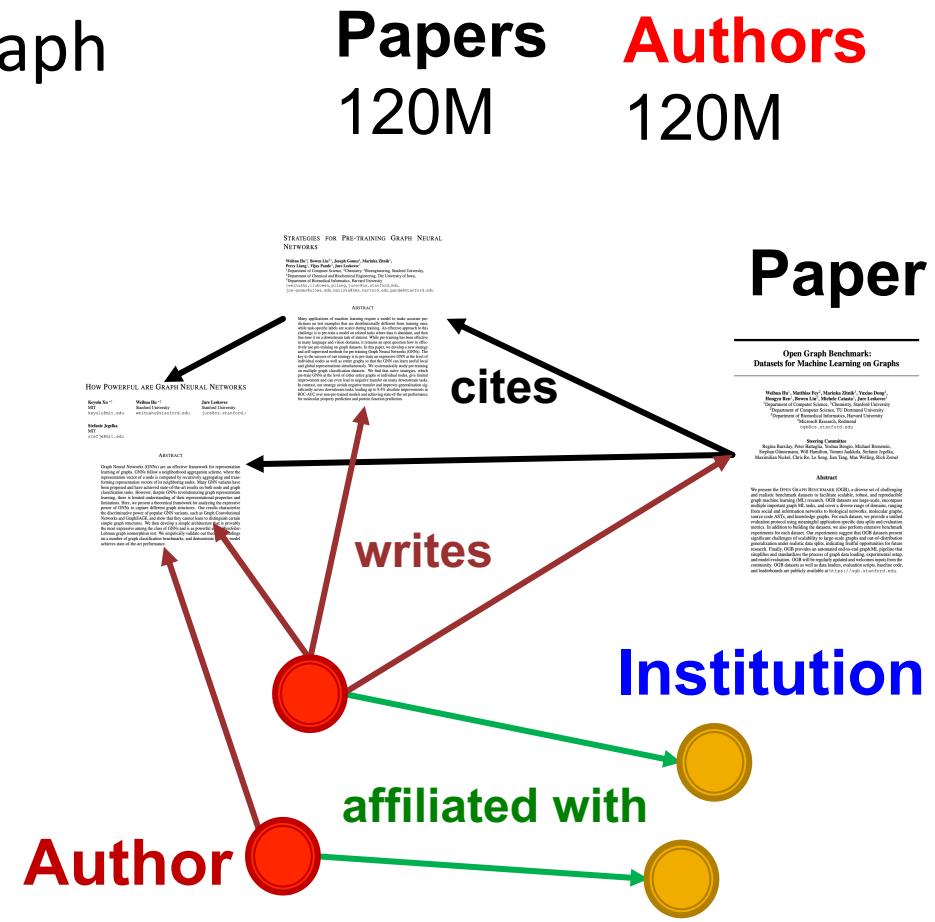
- Friend recommendation  
(link-level)
- User property prediction  
(node-level)

**Users**  
300M~3B



# Graphs in Modern Applications

- **Academic graph:**
    - Microsoft Academic Graph
  - **ML tasks:**
    - Paper categorization  
(node classification)
    - Author collaboration  
recommendation
    - Paper citation  
recommendation
    - (link prediction)



# Graphs in Modern Applications

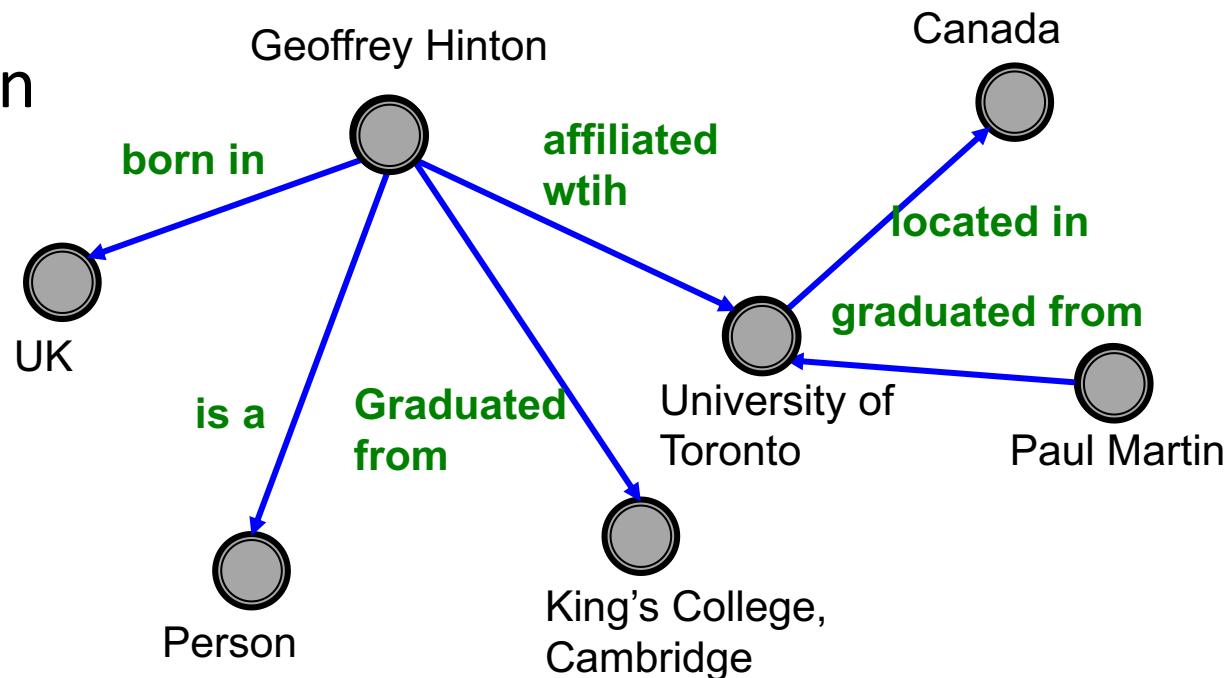
## ■ Knowledge Graphs (KGs):

- Wikidata
- Freebase

## ■ ML tasks:

- KG completion
- Reasoning

**Entities**  
80M—90M



# What is in Common?

- **Large-scale:**
  - #nodes ranges from 10M to 10B.
  - #edges ranges from 100M to 100B.
- **Tasks**
  - **Node-level:** User/item/paper classification.
  - **Link-level:** Recommendation, completion.
- **Todays' lecture**
  - **Scale up GNNs to large graphs!**

# Why is it Hard?

- **Recall:** How we usually train an ML model on large data ( $N=\#\text{data}$  is large)
- **Objective:** Minimize the averaged loss

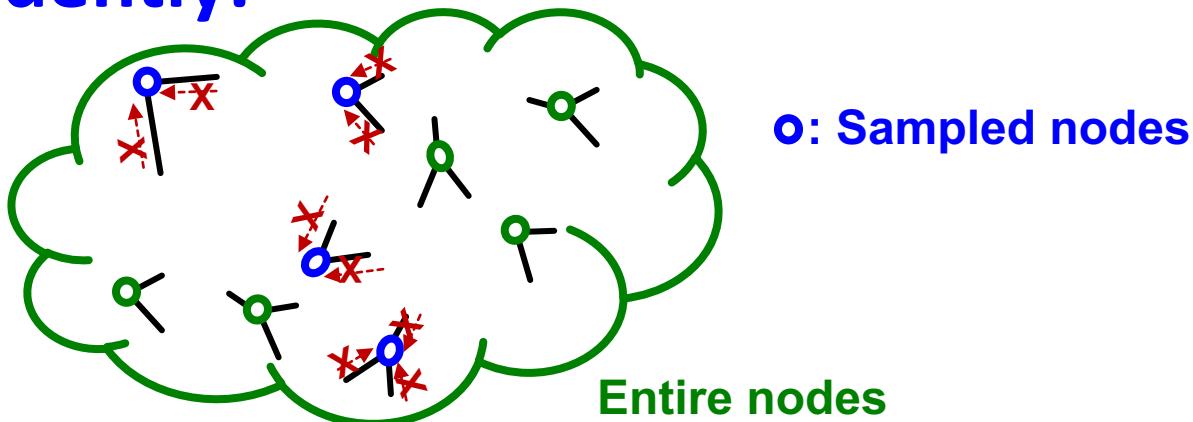
$$\ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=0}^{N-1} \ell_i(\boldsymbol{\theta})$$

- $\boldsymbol{\theta}$ : model parameters,  $\ell_i(\boldsymbol{\theta})$ : loss for  $i$ -th data point.
- We perform **Stochastic Gradient Descent (SGD)**.
  - Randomly sample  $M$  ( $<< N$ ) data (**mini-batches**).
  - Compute the  $\ell_{sub}(\boldsymbol{\theta})$  over the  $M$  data points.
  - Perform SGD:  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\boldsymbol{\theta})$

# Why is it Hard?

What if we use the standard SGD for GNN?

- In mini-batch, we sample  $M$  ( $<< N$ ) nodes independently:

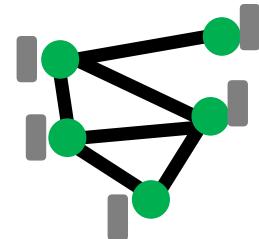


- Sampled nodes tend to be isolated from each other.
- Recall: GNN generate node embeddings by aggregating neighboring node features.
  - GNN does not access to neighboring nodes within the mini-batch!
- Standard SGD cannot effectively train GNNs.

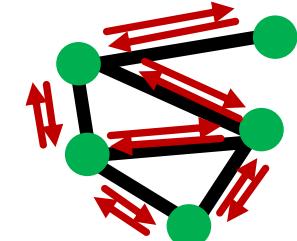
# Why is it Hard?

- **Naïve full-batch implementation:** Generate embeddings of all the nodes **at the same time.**
  - Load **the entire graph and features**
  - **At each GNN layer:**
    - Compute embeddings of all nodes using all the node embeddings from the previous layer.
  - Compute the loss
  - Perform gradient descent

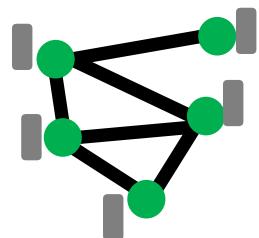
Given all node embeddings at layer K



Perform message-passing



Obtain all node embeddings at layer K+1



# Why is it Hard?

- However, **Full-batch** implementation is **not feasible** for a large graphs. **Why?**
- Because we want to use GPU for fast training, but GPU memory is extremely limited (only 10GB--20GB).
  - **The entire graph and the features cannot be loaded on GPU.**

Slow computation,  
large memory

CPU  
1TB—10TB

Fast computation,  
limited memory

GPU  
10GB—20GB

# Today's Lecture

We introduce **three methods for scaling up GNNs**:

- Two methods perform message-passing over **small subgraphs in each mini-batch**; only the subgraphs need to be loaded on a GPU at a time.
  - **Neighbor Sampling** [Hamilton et al. NeurIPS 2017]
  - **Cluster-GCN** [Chiang et al. KDD 2019]
- One method **simplifies a GNN into feature-preprocessing operation** (can be efficiently performed even on a CPU)
  - **Simplified GCN** [Wu et al. ICML 2019]

# GraphSAGE Neighbor Sampling: Scaling up GNNs

CS224W: Machine Learning with Graphs

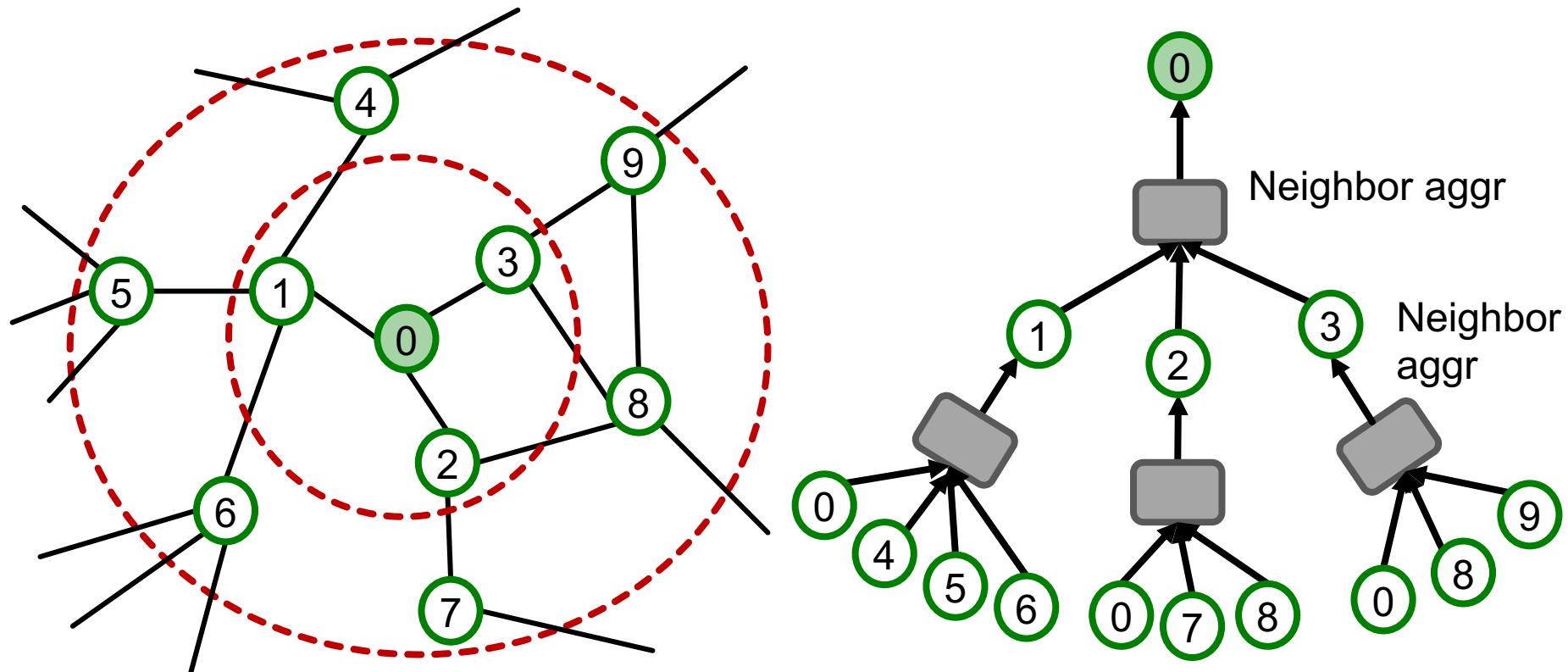
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



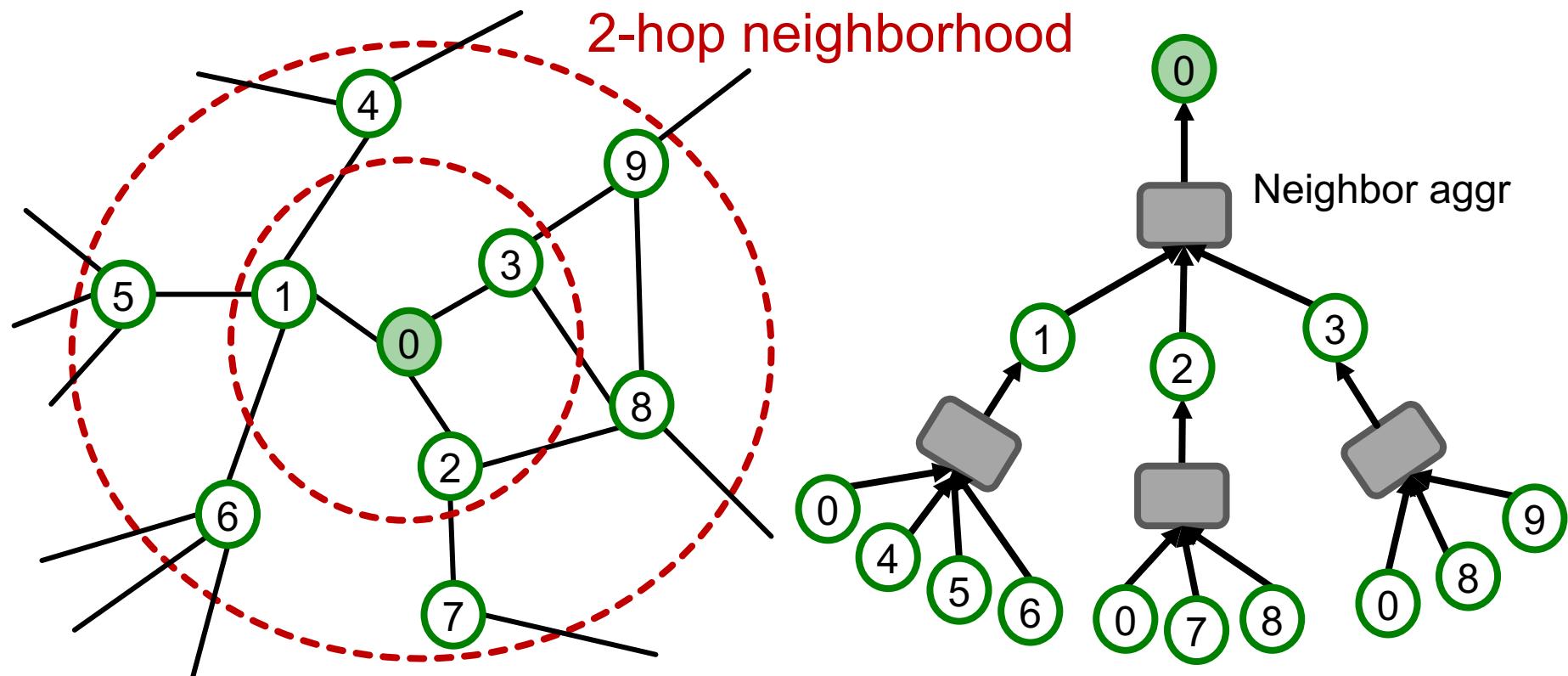
# Recall: Computational Graph

- **Recall:** GNNs generate node embeddings via neighbor aggregation.
  - Represented as a computational graph (right).



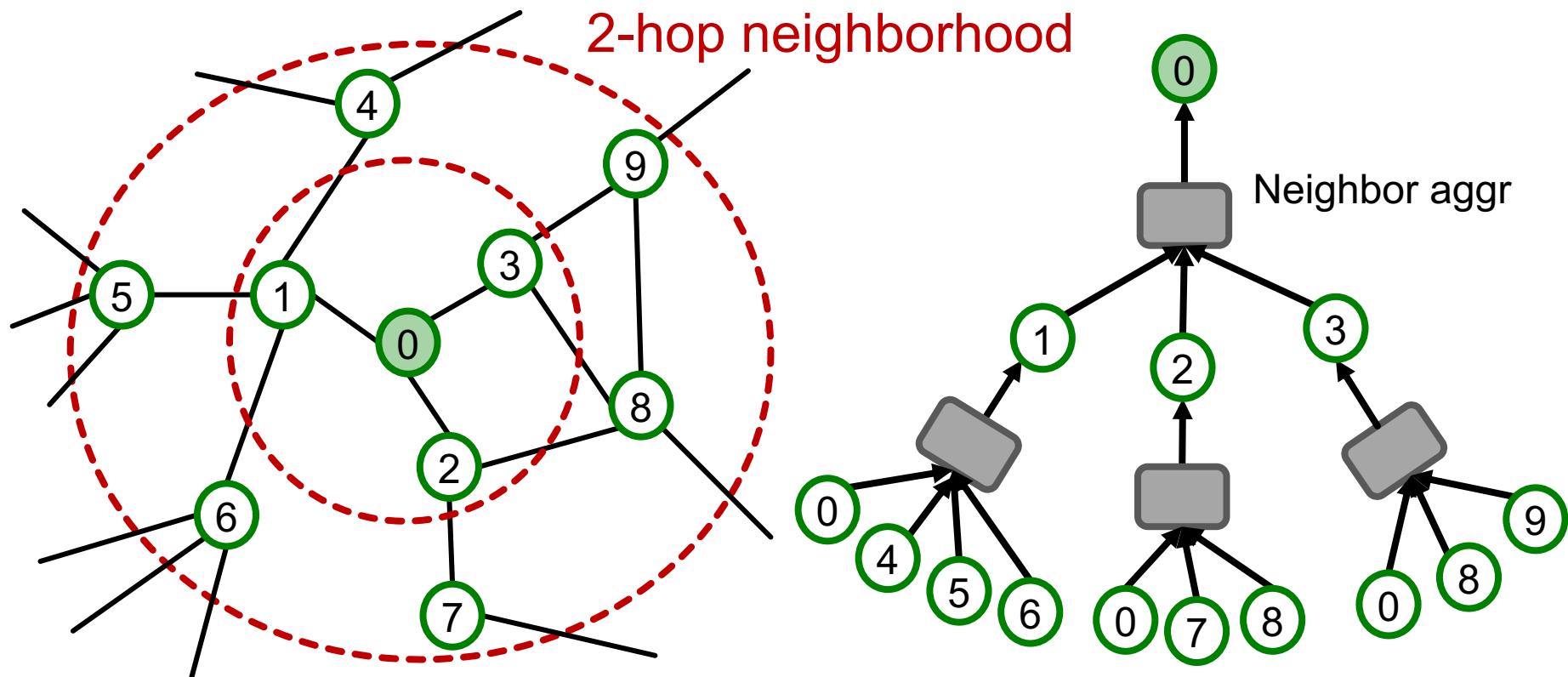
# Recall: Computational Graph

- **Observation:** A 2-layer GNN generates embedding of node “0” using 2-hop neighborhood structure and features.



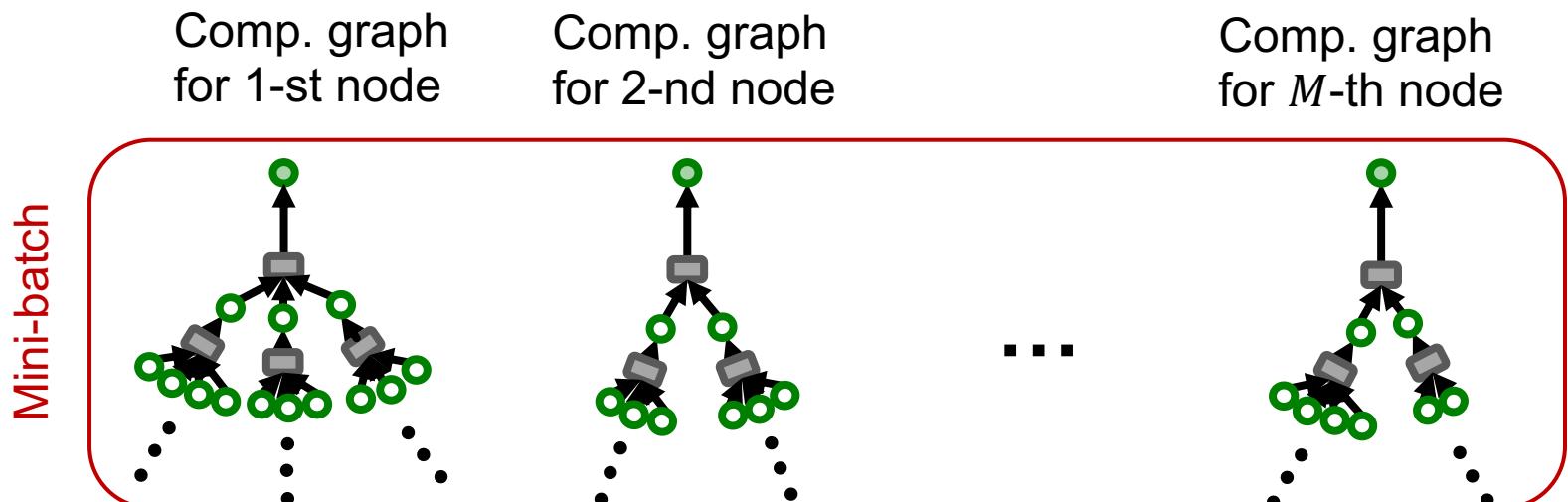
# Recall: Computational Graph

- **Observation:** More generally,  $K$ -layer GNNs generate embedding of a node using  $K$ -hop neighborhood structure and features.



# Computing Node Embeddings

- **Key insight:** To compute embedding of a single node, all we need is **the  $K$ -hop neighborhood** (which defines the computation graph).
- Given a set of  $M$  different nodes in a **mini-batch**, we can generate their embeddings using  $M$  computational graphs. **Can be computed on GPU!**

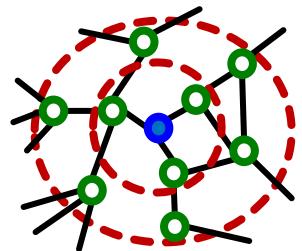


# Stochastic Training of GNNs

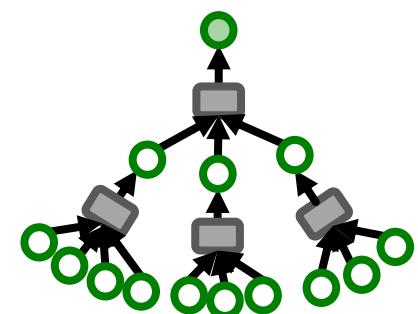
- We can now consider the following SGD strategy for training  $K$ -layer GNNs:

- Randomly sample  $M$  ( $\ll N$ ) nodes.
- For each sampled node  $v$ :
  - Get  **$K$ -hop neighborhood**, and construct the **computation graph**.
  - Use the above to generate  $v$ 's embedding.
- Compute the loss  $\ell_{sub}(\theta)$  averaged over the  $M$  nodes.
- Perform SGD:  $\theta \leftarrow \theta - \nabla \ell_{sub}(\theta)$

**$K$ -hop neighborhood**

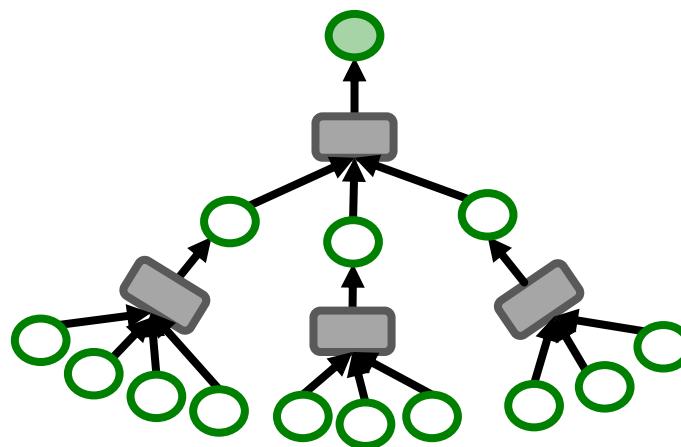


**Computational graph**



# Issue with Stochastic Training

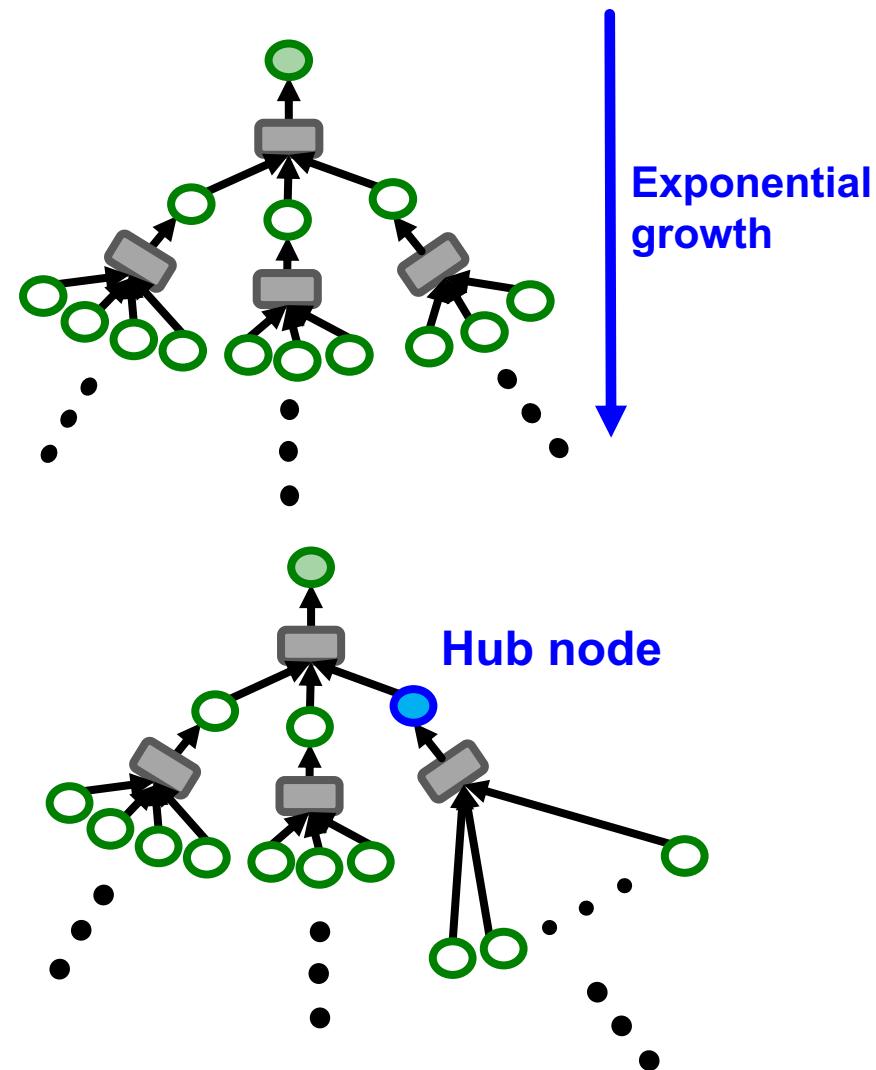
- For each node, we need to get the entire  **$K$ -hop neighborhood** and pass it through the computation graph.
- We need to aggregate lot of information just to compute one node embedding.
- **Computationally expensive.**



# Issue with Stochastic Training

## More details:

- Computation graph becomes **exponentially large** with respect to the layer size  $K$ .
- Computation graph explodes when it hits a **hub node** (high-degree node).
- Next:** Make the comp. graph more compact!



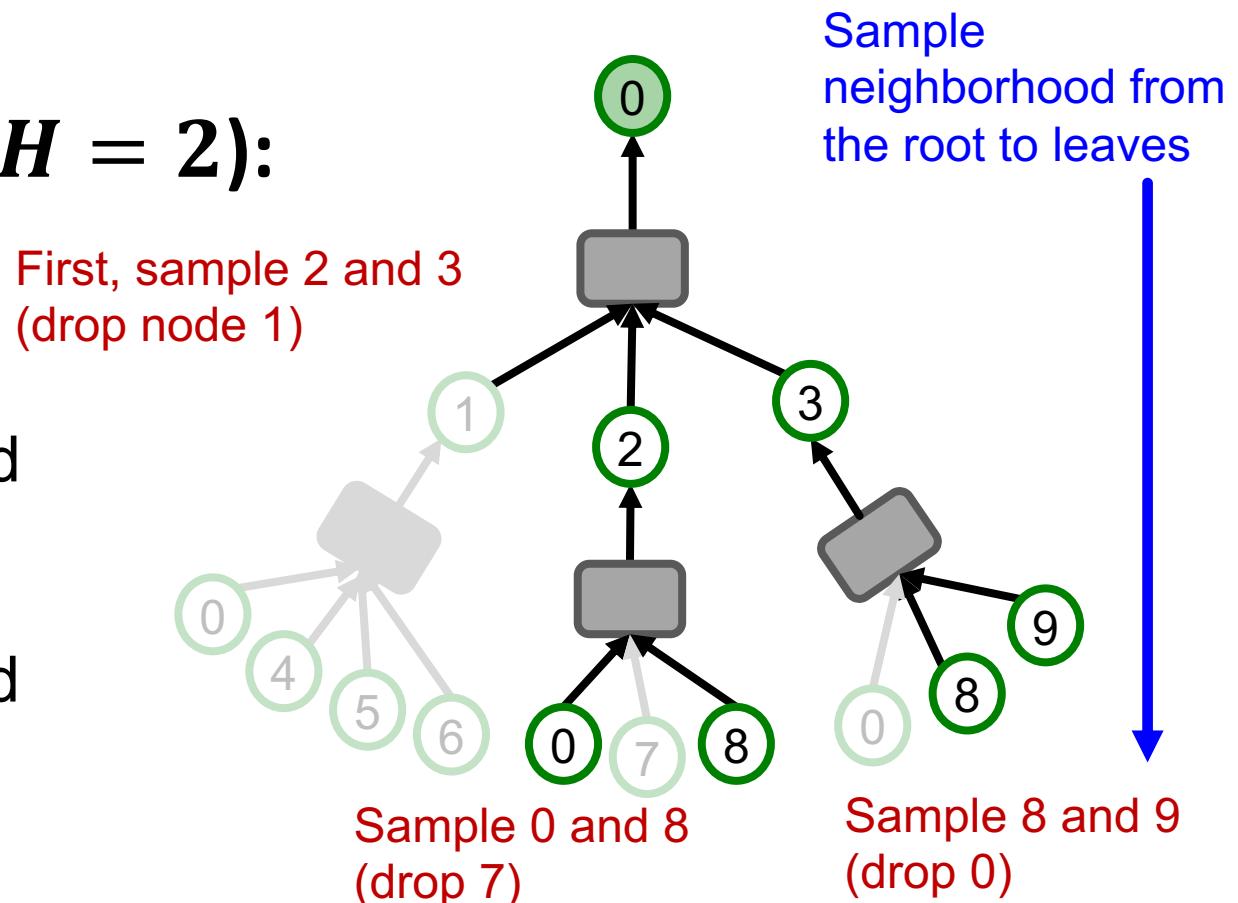
# Neighbor Sampling

**Key idea:** Construct the computational graph by (randomly) sampling at most  $H$  neighbors at each hop.

- Example ( $H = 2$ ):

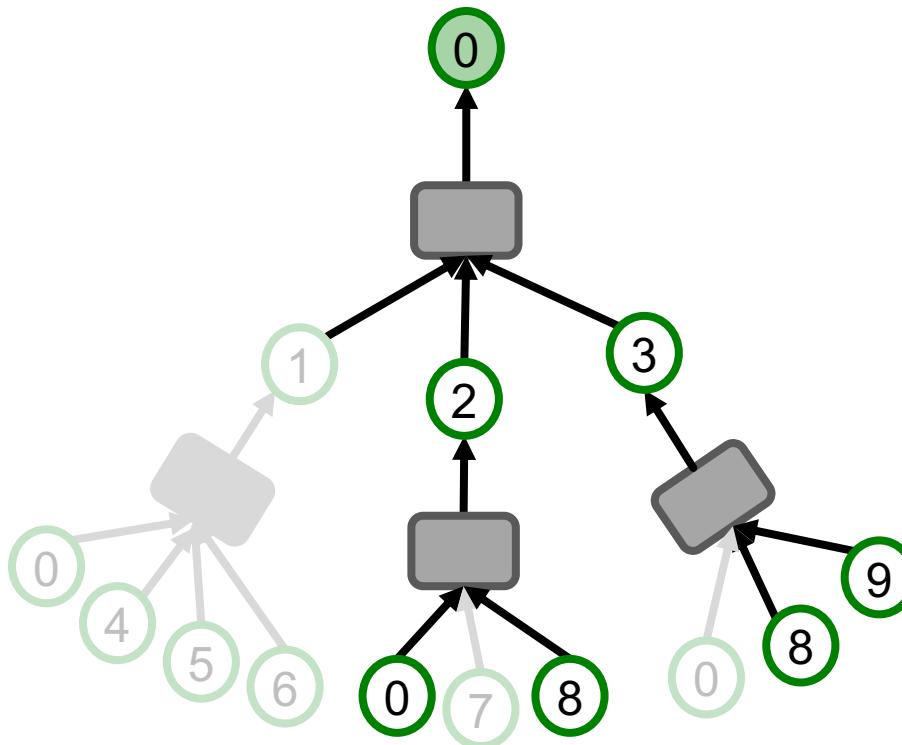
1<sup>st</sup>-hop neighborhood

2<sup>nd</sup>-hop neighborhood



# Neighbor Sampling

We can use the pruned computational graph to more efficiently compute node embeddings.



# Neighbor Sampling Algorithm

## Neighbor sampling for $K$ -layer GNN

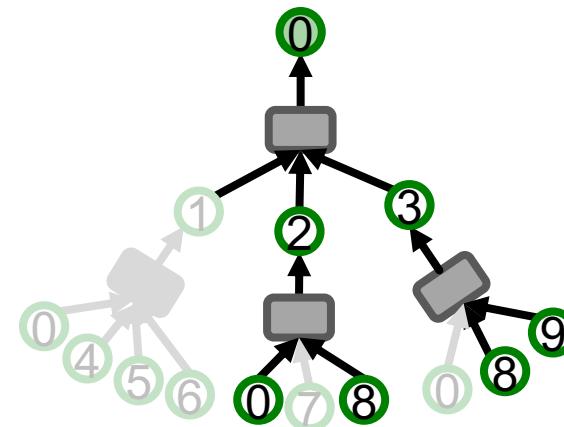
- For  $k = 1, 2, \dots, K$ :
  - For each node in  $k$ -hop neighborhood:
  - (Randomly) sample at most  $H_k$  neighbors:

**1<sup>st</sup>-hop neighborhood**

Sample  $H_1 = 2$  neighbors

**2<sup>nd</sup>-hop neighborhood**

Sample  $H_2 = 2$  neighbors



- $K$ -layer GNN will at most involve  $\prod_{k=1}^K H_k$  leaf nodes in comp. graph.

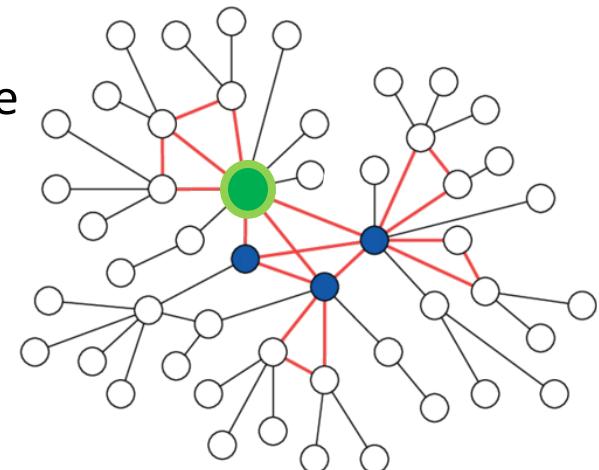
# Remarks on Neighbor Sampling (1)

- **Remark 1: Trade-off in sampling number  $H$** 
  - Smaller  $H$  leads to more efficient neighbor aggregation, but results in more unstable training **due to the larger variance** in neighbor aggregation.
- **Remark 2: Computational time**
  - Even with neighbor sampling, **the size of the computational graph is still exponential with respect to number of GNN layers  $K$ .**
  - Increasing one GNN layer would make computation  $H$  times more expensive.

# Remarks on Neighbor Sampling (2)

## ■ Remark 3: How to sample the nodes

- **Random sampling:** fast but many times not optimal (may sample many “unimportant” nodes)
- **Random Walk with Restarts:**
  - Natural graphs are “scale free”, sampling random neighbors, samples many low degree “leaf” nodes.
  - Strategy to sample important nodes:
    - Compute Random Walk with Restarts score  $R_i$  starting at the **green** node
    - At each level sample  $H$  neighbors  $i$  with the highest  $R_i$
  - This strategy works much better in practice.



# Summary: Neighbor Sampling

- A computational graph is constructed for each node in a mini-batch.
- In neighbor sampling, the comp. graph is pruned/sub-sampled to increase computational efficiency.
- The pruned comp. graph is used to generate a node embedding.
- However, computational graphs can still become large, especially for GNNs with many message-passing layers.

# Cluster-GCN: Scaling up GNNs

CS224W: Machine Learning with Graphs

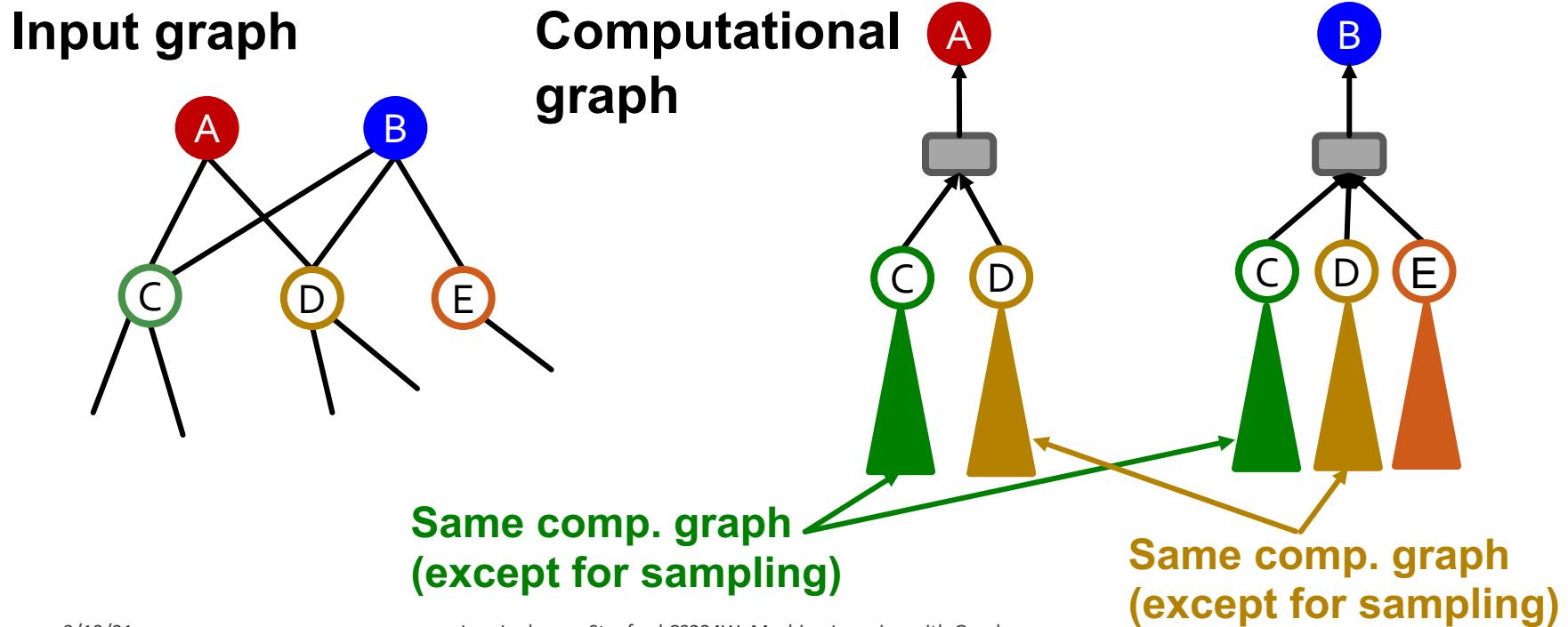
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



# Issues with Neighbor Sampling

- The size of computational graph becomes exponentially large w.r.t. the #GNN layers.
- Computation is redundant, especially when nodes in a mini-batch share many neighbors.



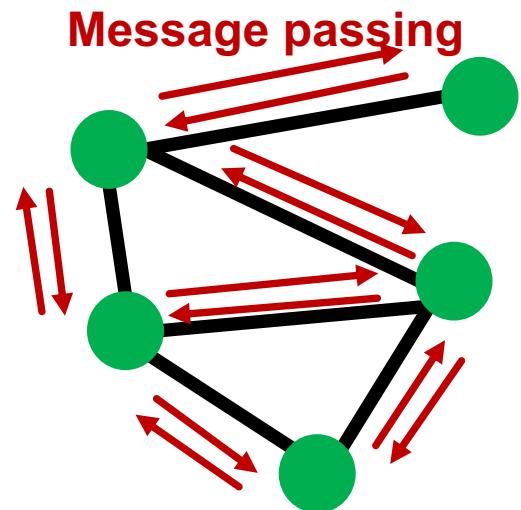
# Recall: Full Batch GNN

- In full-batch GNN implementation, **all the node embeddings are updated together using embeddings of the previous layer.**

Update for all  $v \in V$

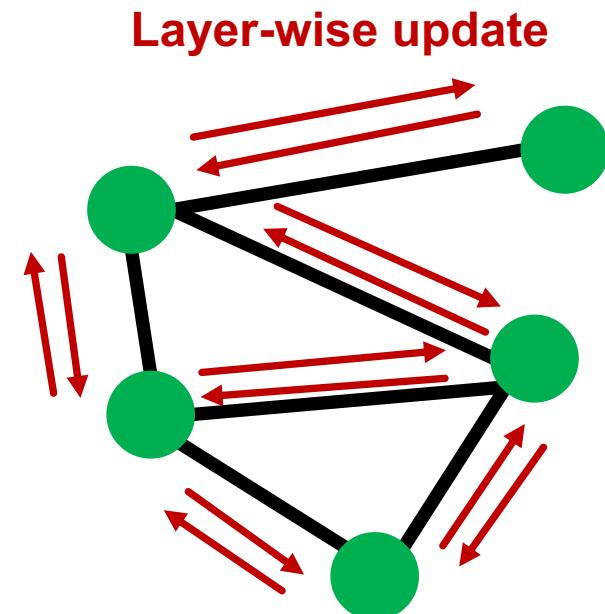
$$h_v^{(\ell)} = COMBINE \left( h_v^{(\ell-1)}, AGGR \left( \left\{ \mathbf{h}_u^{(\ell-1)} \right\}_{u \in N(v)} \right) \right)$$

- In each layer, only **2\*(#edges) messages** need to be computed.
- For  $K$ -layer GNN, only  $2K * \#(\text{edges})$  messages need to be computed.
- GNN's entire computation is only **linear** in  $\#(\text{edges})$  and  $\#(\text{GNN layers})$ . **Fast!**



# Insight from Full-batch GNN

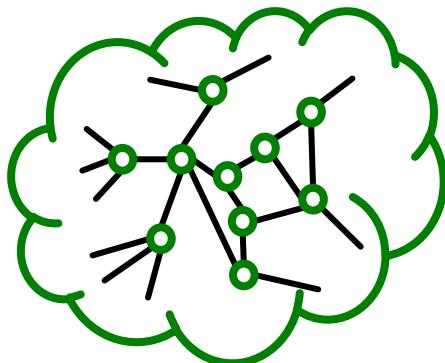
- The **layer-wise** node embedding update allows the re-use of embeddings from the previous layer.
- This significantly **reduces the computational redundancy of neighbor sampling**.
  - Of course, the **layer-wise** update is **not feasible** for a large graph due to **limited GPU memory**.



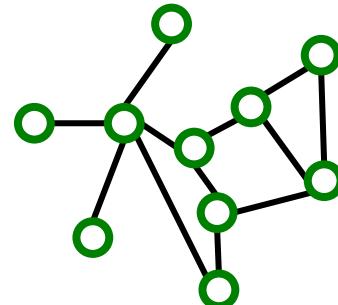
# Subgraph Sampling

- **Key idea:** We can **sample a small subgraph of the large graph** and then perform the efficient **layer-wise** node embeddings update over the subgraph.

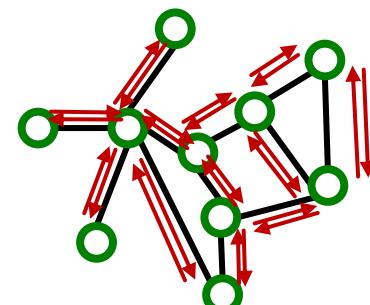
Large graph



Sampled subgraph  
(small enough to  
be put on a GPU)



Layer-wise  
node embeddings  
update on the GPU



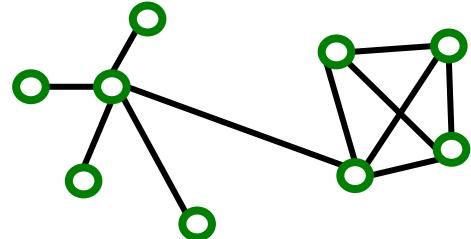
# Subgraph Sampling

- **Key question:** What subgraphs are good for training GNNs?
  - Recall: GNN performs node embedding by passing messages **via the edges**.
    - Subgraphs **should retain edge connectivity structure of the original graph as much as possible.**
    - This way, the GNN over the subgraph generates embeddings closer to the GNN over the original graph.

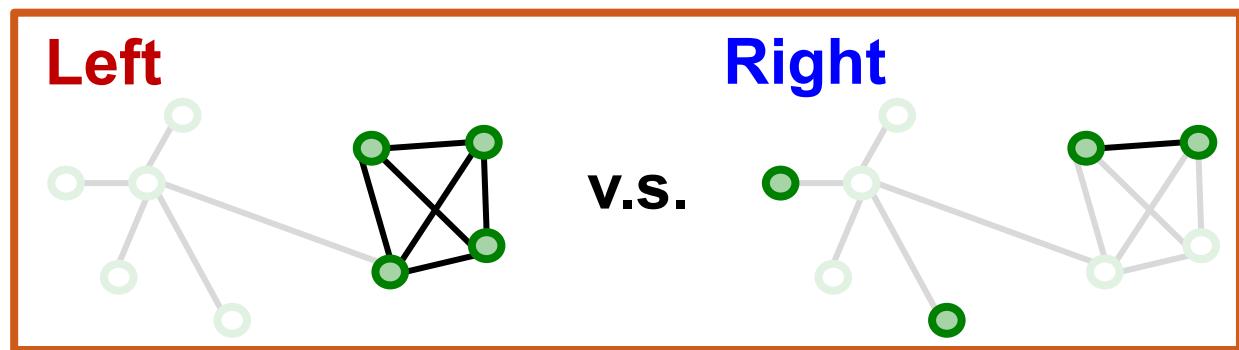
# Subgraph Sampling: Case Study

- Which subgraph is good for training GNN?

Original graph



Subgraphs (both 4-node induced subgraph)

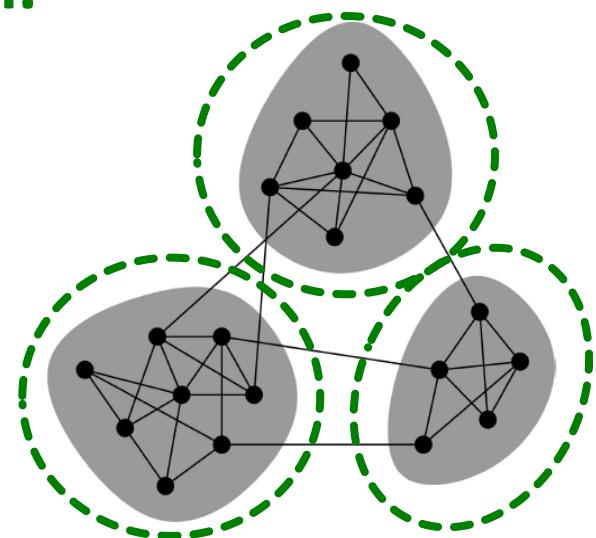


- **Left subgraph** retains the essential community structure among the 4 nodes → **Good**
- **Right subgraph** drops many connectivity patterns, even leading to isolated nodes → **Bad**

# Exploiting Community Structure

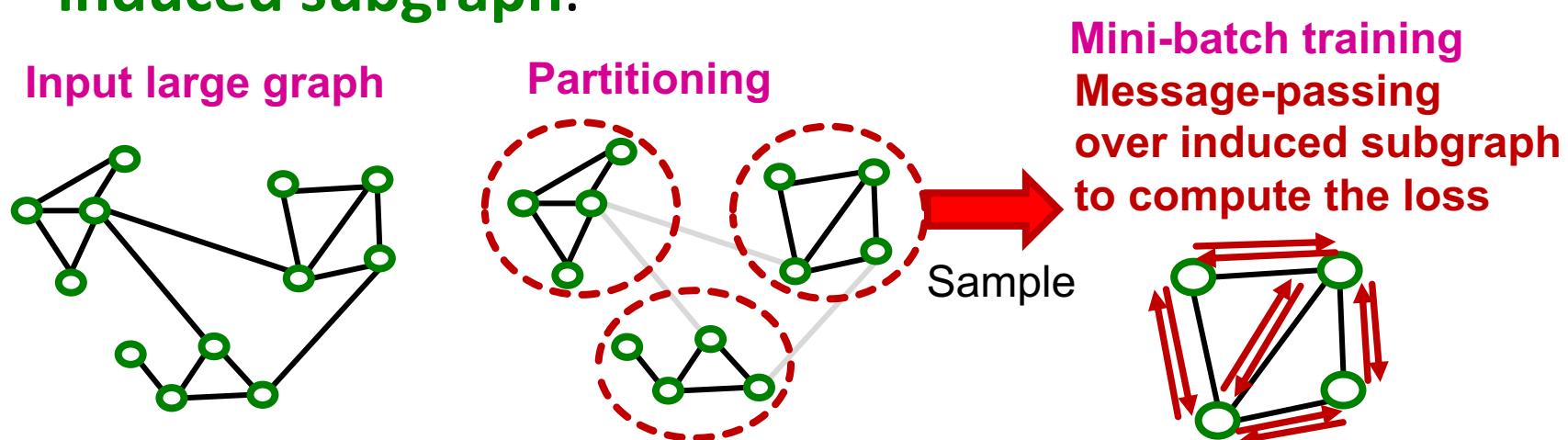
Real-world graph exhibits community structure

- A large graph can be decomposed into many small communities.
- **Key insight** [Chiang et al. KDD 2019]:  
**Sample a community as a subgraph.**  
**Each subgraph retains essential local connectivity pattern of the original graph.**



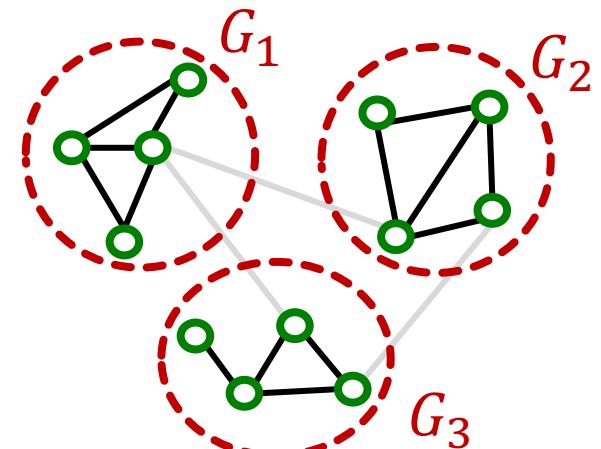
# Cluster-GCN: Overview

- We first introduce “vanilla” Cluster-GCN.
- Cluster-GCN consists of two steps:
  - **Pre-processing**: Given a large graph, partition it into groups of nodes (i.e., subgraphs).
  - **Mini-batch training**: Sample one node group at a time. Apply GNN’s message passing over the **induced subgraph**.



# Cluster-GCN: Pre-processing

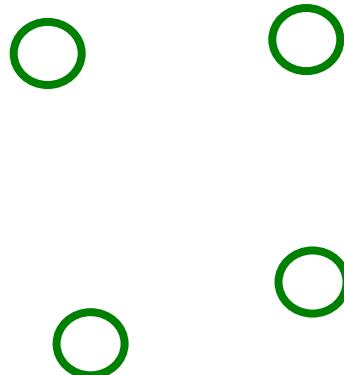
- Given a large graph  $G = (V, E)$ , **partition its nodes  $V$  into  $C$  groups:**  $V_1, \dots, V_C$ .
    - We can use any scalable community detection methods, e.g., Louvain, METIS [Karypis et al. SIAM 1998].
  - $V_1, \dots, V_C$  induces  $C$  subgraphs,  $G_1, \dots, G_C$ ,
    - Recall:  $G_c \equiv (V_c, E_c)$ ,
    - where  $E_c = \{(u, v) | u, v \in V_c\}$
- Notice: Between-group edges are *not* included in  $G_1, \dots, G_C$ .**



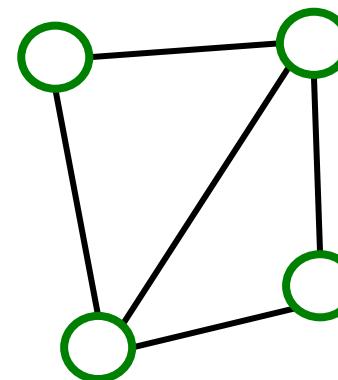
# Cluster-GCN: Mini-batch Training

- For each mini-batch, **randomly sample a node group  $V_c$ .**
- Construct **induced subgraph  $G_c = (V_c, E_c)$**

Sampled node group  $V_c$



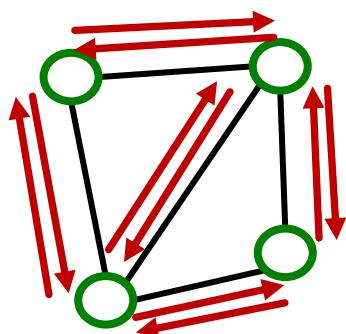
Induced subgraph  $G_c$



# Cluster-GCN: Mini-batch Training

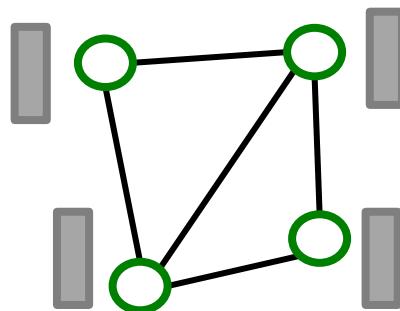
- Apply GNN's **layer-wise node update** over  $G_c$  to obtain embedding  $\mathbf{h}_v$  for each node  $v \in V_c$ .
- Compute the loss for each node  $v \in V_c$  and take average:  $\ell_{sub}(\theta) = (1/|V_c|) \cdot \sum_{v \in V_c} \ell_v(\theta)$
- Update params:  $\theta \leftarrow \theta - \nabla \ell_{sub}(\theta)$

Induced subgraph  $G_c$



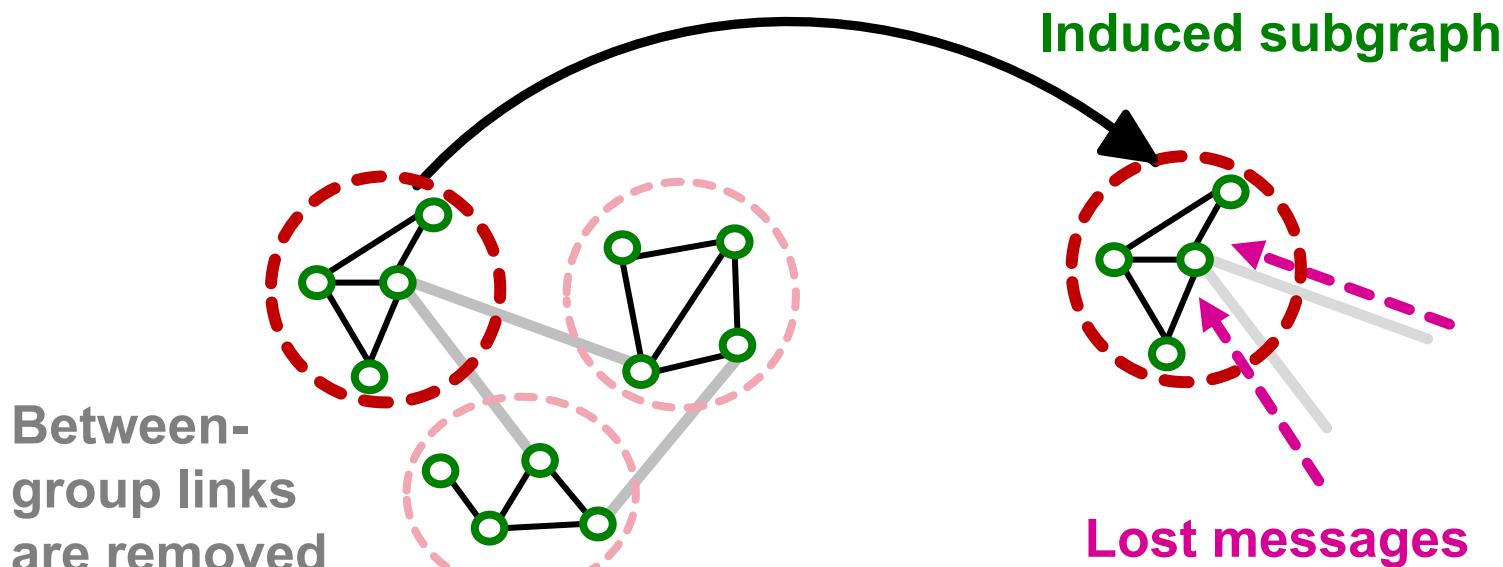
Layer-wise node  
embedding update

Embedding



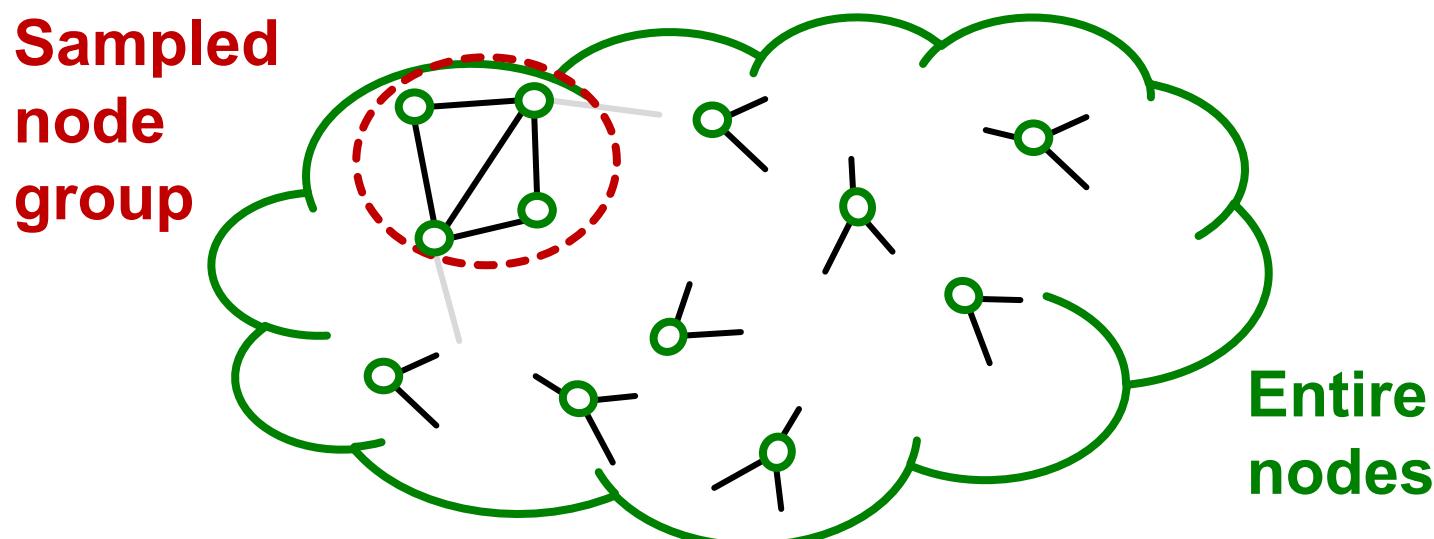
# Issues with Cluster-GCN (1)

- The induced subgraph removes between-group links.
- As a result, messages from other groups will *be lost during message passing*, which could hurt the GNN's performance.



# Issues with Cluster-GCN (2)

- Graph community detection algorithm **puts similar nodes together in the same group.**
- **Sampled node group** tends to only cover the small-concentrated portion of the **entire data**.



# Issues with Cluster-GCN (3)

Sampled nodes are not diverse enough to be represent the entire graph structure:

- As a result, the gradient averaged over the sampled nodes,  $\frac{1}{|V_c|} \sum_{v \in V_c} \nabla \ell_v(\theta)$ , becomes unreliable.
  - Fluctuates a lot from a node group to another.
  - In other words, the gradient has high variance.
- Leads to slow convergence of SGD

# Advanced Cluster-GCN: Overview

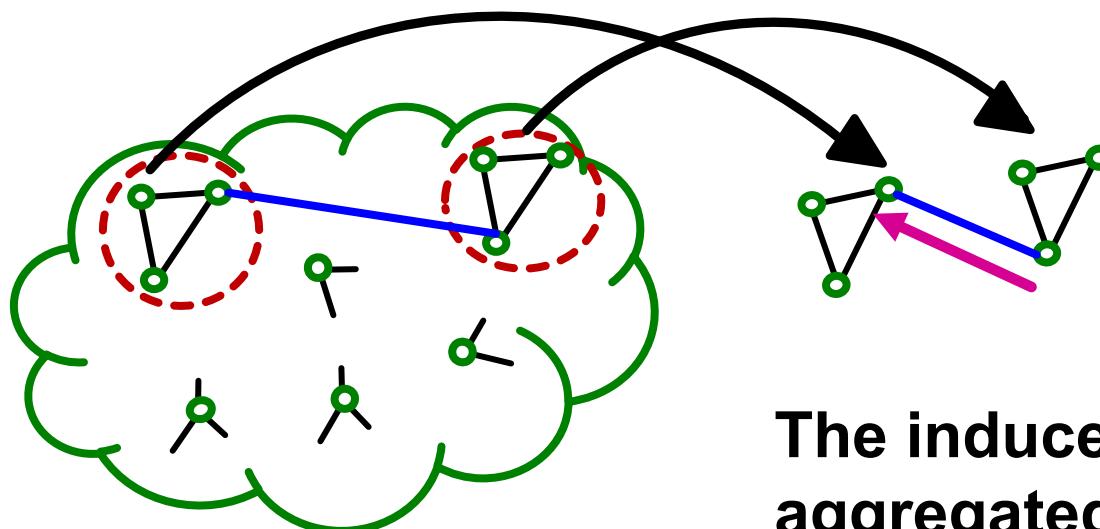
- **Solution: Aggregate multiple node groups per mini-batch.**
- Partition the graph into **relatively-small groups of nodes**.
- **For each mini-batch:**
  - Sample and aggregate **multiple node groups**.
  - **Construct the induced subgraph of the *aggregated node group*.**
  - The rest is the same as vanilla Cluster-GCN (compute node embeddings and the loss, update parameters)

# Advanced Cluster-GCN: Overview

## ■ Why does the solution work?

### Sampling multiple node groups

→ Makes the sampled nodes more representative of the entire nodes. Leads to less variance in gradient estimation.



**The induced subgraph over aggregated node groups**

- Includes between-group edges
- Message can flow across groups.

# Advanced Cluster-GCN

Similar to vanilla Cluster-GCN, advanced Cluster-GCN also follows 2-step approaches.

## Pre-processing step:

- Given a large graph  $G = (V, E)$ , partition its nodes  $V$  into  $C$  **relatively-small** groups:  
 $V_1, \dots, V_C$ .
  - $V_1, \dots, V_C$  needs to be small so that even if multiple of them are aggregated, the resulting group would not be too large.

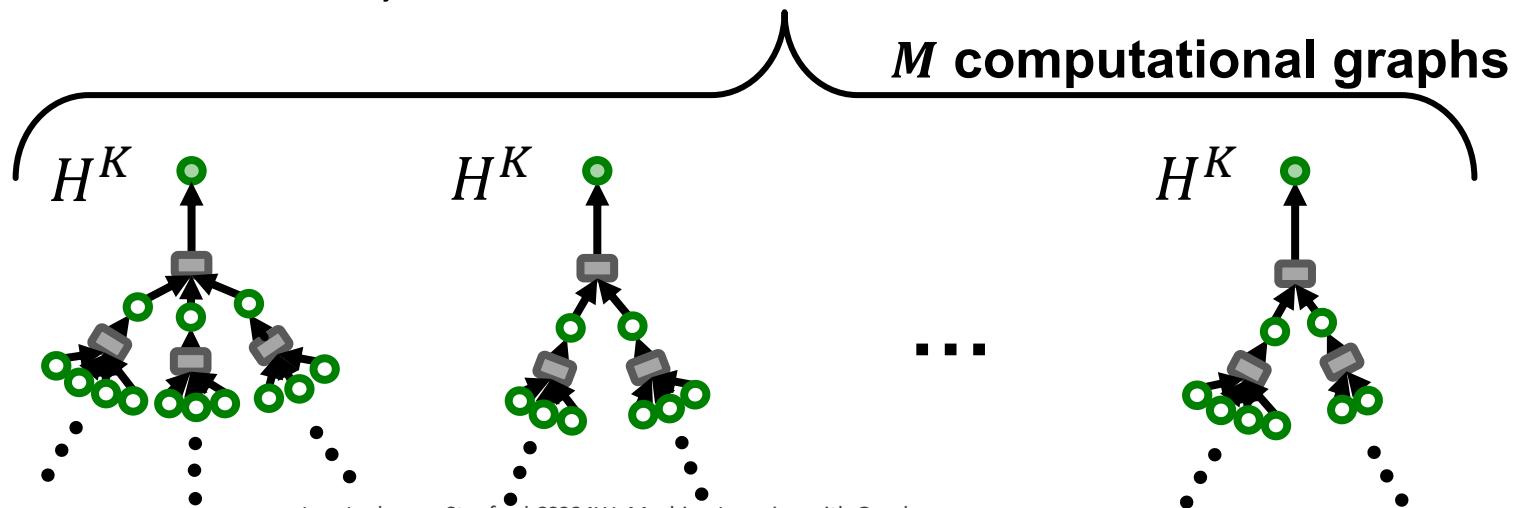
# Advanced Cluster-GCN

## Mini-batch training:

- For each mini-batch, **randomly sample a set of  $q$  node groups:**  $\{V_{t_1}, \dots, V_{t_q}\} \subset \{V_1, \dots, V_C\}$ .
- **Aggregate all nodes across the sampled node groups:**  $V_{aggr} = V_{t_1} \cup \dots \cup V_{t_q}$
- Extract the **induced subgraph**  
 $G_{aggr} = (V_{aggr}, E_{aggr}),$   
where  $E_{aggr} = \{(u, v) \mid u, v \in V_{aggr}\}$ 
  - $E_{aggr}$  also includes between-group edges!

# Comparison of Time Complexity

- Generate  $M$  ( $<< N$ ) node embeddings using  $K$ -layer GNN ( $N$  : #all nodes).
- Neighbor-sampling (sampling  $H$  nodes per layer):
  - For each node, the size of  $K$ -layer computational graph is  $H^K$ .
  - For  $M$  nodes, the cost is  $M \cdot H^K$



# Comparison of Time Complexity

- Generate  $M$  ( $<< N$ ) node embeddings using  $K$ -layer GNN ( $N$  : #all nodes).
- Cluster-GCN:
  - Perform message passing over a subgraph induced by the  $M$  nodes.
  - The subgraph contains  $M \cdot D_{avg}$  edges, where  $D_{avg}$  is the average node degree.
  - $K$ -layer message passing over the subgraph costs at most  $K \cdot M \cdot D_{avg}$ .

# Comparison of Time Complexity

- In summary, the cost to generate embeddings for  $M$  nodes using  $K$ -layer GNN is:
  - **Neighbor-sampling (sample  $H$  nodes per layer):**  
 $M \cdot H^K$
  - **Cluster-GCN:**  $K \cdot M \cdot D_{avg}$
- Assume  $H = D_{avg}/2$ . In other words, 50% of neighbors are sampled.
  - Then, Cluster-GCN (cost:  $2MHK$ ) is much more efficient than neighbor sampling (cost:  $MH^K$ ).
  - Linear (instead of exponential) dependency w.r.t.  $K$ .

# Cluster-GCN: Summary

- Cluster-GCN first **partitions the entire nodes into a set of small node groups.**
- At each mini-batch, multiple node groups are sampled, and their nodes are aggregated.
- **GNN performs layer-wise node embeddings update over the induced subgraph.**
- Cluster-GCN is more computationally efficient than neighbor sampling, especially when #(GNN layers) is large.
- But Cluster-GCN leads to systematically biased gradient estimates (due to missing cross-community edges)

# Scaling up by Simplifying GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



# Roadmap of Simplifying GCN

- We start from Graph Convolutional Network (GCN) [Kipf & Welling ICLR 2017].
- We simplify GCN by **removing the non-linear activation** from the GCN [Wu et al. ICML 2019].
  - *Wu et al.* demonstrated that the performance on benchmark is not much lower by the simplification.
- Simplified GCN turns out to be extremely scalable by the model design.

# Recall: GCN (mean-pool)

- **Given:** Graph  $G = (V, E)$  with input node features  $X_v$  for  $v \in V$ , where  **$E$  includes the self-loop**:
    - $(v, v) \in E$  for all  $v \in V$ .
  - Set input node embeddings:  $h_v^{(0)} = X_v$  for  $v \in V$ .
  - For  $k \in \{0, \dots, K - 1\}$ :
    - For all  $v \in V$  update  $h_v^{(k)}$  via information from neighbors

$$h_v^{(k+1)} = \text{ReLU} \left( W_k \left[ \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)} \right] \right)$$

Trainable weight matrices  
(i.e., what we learn)      Mean-pooling

- **Final node embedding:**  $z_v = h_v^{(K)}$

# Recall: Matrix Formulation of GCN

GCN aggregations can be formulated as matrix vector product:

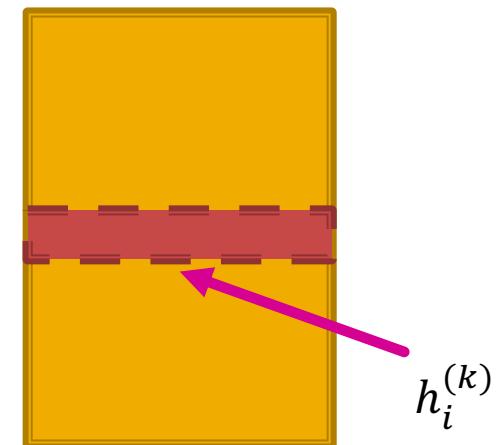
- Let  $\mathbf{H}^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$
- Let  $A$  be the adjacency matrix (w/ self-loop)
- Then:  $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:} \mathbf{H}^{(k)}$
- Let  $D$  be diagonal matrix where  $D_{v,v} = \text{Deg}(v) = |N(v)|$ 
  - The inverse of  $D$ :  $D^{-1}$  is also diagonal:  
$$D_{v,v}^{-1} = 1/|N(v)|$$
- Therefore,

$$\frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)}$$



$$D^{-1} A \mathbf{H}^{(k)}$$

Matrix of hidden embeddings  $\mathbf{H}^{(k)}$



# Recall: Matrix Formulation of GCN

- GCN's neighbor aggregation:

$$h_v^{(k+1)} = \text{ReLU} \left( W_k \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)} \right)$$

- In matrix form:

$$\mathbf{H}^{(k+1)} = \text{ReLU}(\tilde{\mathbf{A}} \mathbf{H}^{(k)} \mathbf{W}_k^T)$$

where  $\tilde{\mathbf{A}} = \mathbf{D}^{-1} \mathbf{A}$ .

- Note: The original GCN uses re-normalized version:  $\tilde{\mathbf{A}} = \mathbf{D}^{-1/2} \mathbf{A} \mathbf{D}^{-1/2}$ .

- Empirically, this version of  $\tilde{\mathbf{A}}$  often gives better performance than  $\mathbf{D}^{-1} \mathbf{A}$ .

# Simplifying GCN

- Simplify GCN by removing ReLU non-linearity:

$$H^{(k+1)} = \tilde{A} H^{(k)} W_k^T$$

- The final node embedding matrix is given as

$$H^{(K)} = \tilde{A} H^{(K-1)} W_{K-1}^T$$

$$= \tilde{A} (\tilde{A} H^{(K-2)} W_{K-2}^T) W_{K-1}^T$$

$$\dots = \tilde{A} (\tilde{A} (\dots (\tilde{A} H^{(0)} W_0^T) \dots) W_{K-2}^T) W_{K-1}^T$$

$$= \tilde{A}^K X (W_0^T \dots W_{K-1}^T)$$

Composition of linear transformation is still linear!

$$= \tilde{A}^K X W^T \quad \text{where } W \equiv W_{K-1} \dots W_0$$

# Simplifying GCN (1)

- Removing ReLU significantly simplifies GCN!

$$\mathbf{H}^{(K)} = \tilde{\mathbf{A}}^K \mathbf{X} \mathbf{W}^T$$

- Notice  $\tilde{\mathbf{A}}^K \mathbf{X}$  does not contain any learnable parameters; hence, it **can be pre-computed**.
  - Efficiently computable as a sequence of sparse-matrix vector products:
  - Do  $\mathbf{X} \leftarrow \tilde{\mathbf{A}} \mathbf{X}$  for  $K$  times.

# Simplifying GCN (2)

- Let  $\tilde{X} = \tilde{A}^K X$  be pre-computed matrix.
- Simplified GCN's final embedding is

$$H^{(K)} = \tilde{X} W^T$$

- It's just a **linear transformation of pre-computed matrix!**
- Back to the node embedding form:

$$h_v^{(K)} = W \boxed{\tilde{X}_v}$$

Pre-computed feature vector for node  $v$

- Embedding of node  $v$  only depends on its own (pre-processed) feature!

# Simplifying GCN (3)

- Once  $\tilde{X}$  is pre-computed, embeddings of  $M$  nodes can be generated in time linear in  $M$ :
  - Given  $M$  nodes  $\{\nu_1, \nu_2, \dots, \nu_M\}$ , their embeddings are
    - $h_{\nu_1}^{(K)} = \mathbf{W}\tilde{X}_{\nu_1}$ ,
    - $h_{\nu_2}^{(K)} = \mathbf{W}\tilde{X}_{\nu_2}$ ,
    - ...
    - $h_{\nu_M}^{(K)} = \mathbf{W}\tilde{X}_{\nu_M}$ .

# Simplified GCN: Summary

In summary, simplified GCN consists of **two steps**:

- **Pre-processing step:**

- Pre-compute  $\tilde{X} = \tilde{A}^K X$ . Can be done on CPU.

- **Mini-batch training step:**

- For each mini-batch, randomly-sample  $M$  nodes  $\{v_1, v_2, \dots, v_M\}$ .
  - Compute their embeddings by
    - $h_{v_1}^{(K)} = W\tilde{X}_{v_1}, h_{v_2}^{(K)} = W\tilde{X}_{v_2}, \dots, h_{v_M}^{(K)} = W\tilde{X}_{v_M}$
  - Use the embeddings to make prediction and compute the loss averaged over the  $M$  data points.
  - Perform SGD parameter update.

# Comparison with Other Methods

- **Compared to neighbor sampling:**
  - Simplified GCN generates node embeddings much more efficiently (no need to construct the giant computational graph for each node).
- **Compared to Cluster-GCN:**
  - Mini-batch nodes of simplified GCN can be sampled completely randomly from the entire nodes (no need to sample from multiple groups as Cluster-GCN does)
  - Leads to lower SGD variance during training.
- But the model is much **less expressive** (next).

# Potential Issue of Simplified GCN

- Compared to the original GNN models,  
**simplified GCN's expressive power is limited  
due to the lack of non-linearity in generating  
node embeddings.**

# Performance of Simplified GCN

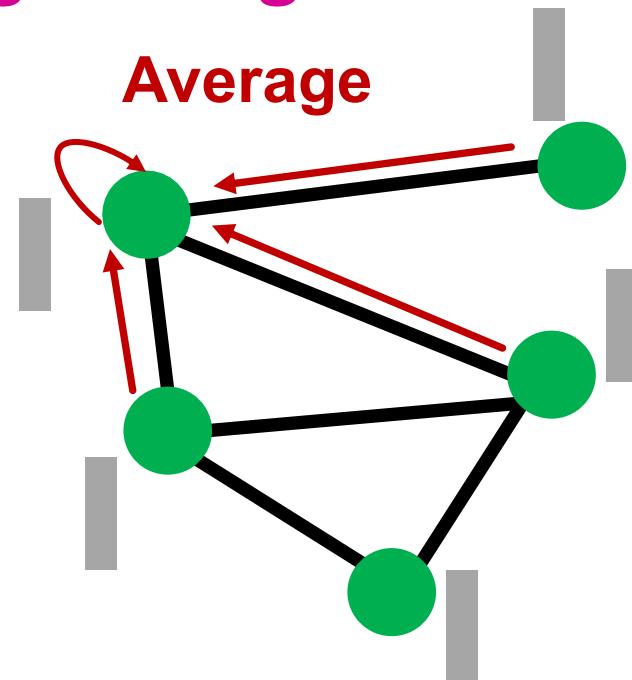
- Surprisingly, in semi-supervised node classification benchmark, **simplified GCN works comparably to the original GNNs despite being less expressive.**
- **Why?**

# Graph Homophily

- Many node classification tasks exhibit homophily structure, i.e., **nodes connected by edges tend to share the same target labels.**
- **Examples:**
  - Paper category classification in paper-citation network
    - Two papers tend to share the same category if one cites another.
  - Movie recommendation for users in social networks
    - Two users tend to like the same movie if they are friends in a social network.

# When does Simplified GCN Work?

- Recall the preprocessing step of the simplified GCN: **Do  $X \leftarrow \tilde{A} X$  for  $K$  times.**
- Pre-processed features are obtained **by iteratively averaging their neighboring node features.**
- As a result, nodes connected by edges tend to have similar pre-processed features.



# When does Simplified GCN Work?

- **Premise:** Model uses the pre-processed node features to make prediction.
- Nodes connected by edges tend to get similar pre-processed features.  
→ **Nodes connected by edges tend to be predicted the same labels by the model**
- **Simplified SGC's prediction aligns well with the graph homophily in many node classification benchmark datasets.**

# Simplified GCN: Summary

- **Simplified GCN removes non-linearity in GCN and reduces to the simple pre-processing of node features.**
- Once the pre-processed features are obtained, scalable mini-batch SGD can be directly applied to optimize the parameters.
- **Simplified GCN works surprisingly well in node classification benchmark.**
  - The feature pre-processing aligns well with graph homophily in real-world prediction tasks.