

# Scaling Up Graph Neural Networks to Large Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



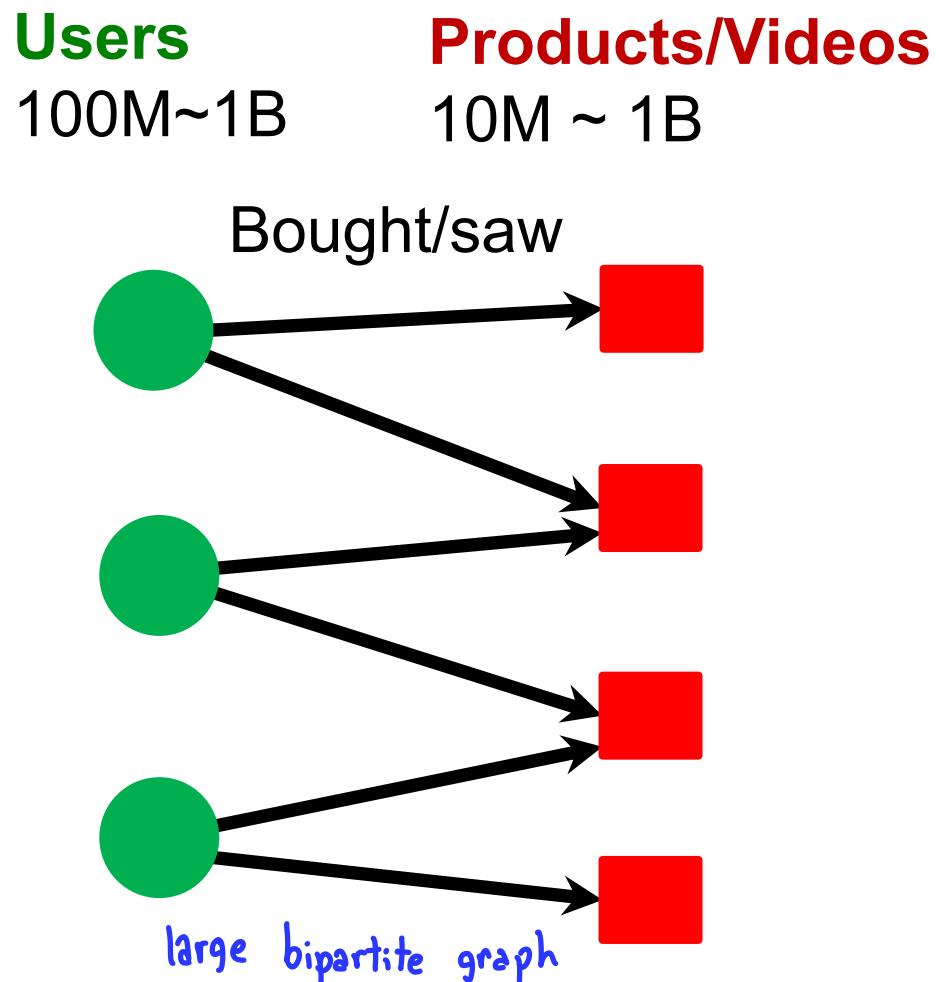
# Graphs in Modern Applications

## ■ Recommender systems:

- Amazon
- YouTube
- Pinterest
- Etc.

## ■ ML tasks:

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



# Graphs in Modern Applications

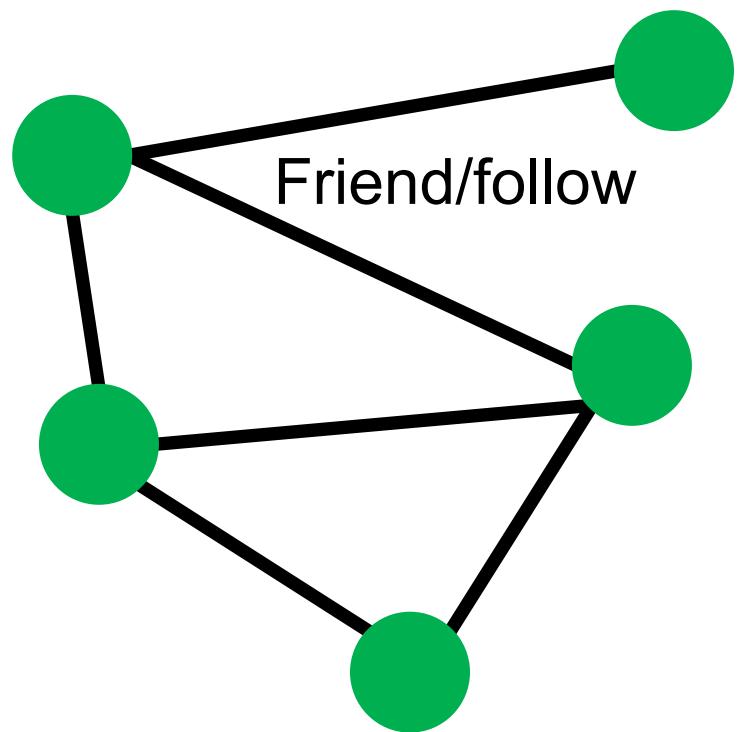
- **Social networks:**

- 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)

heterogeneous graph

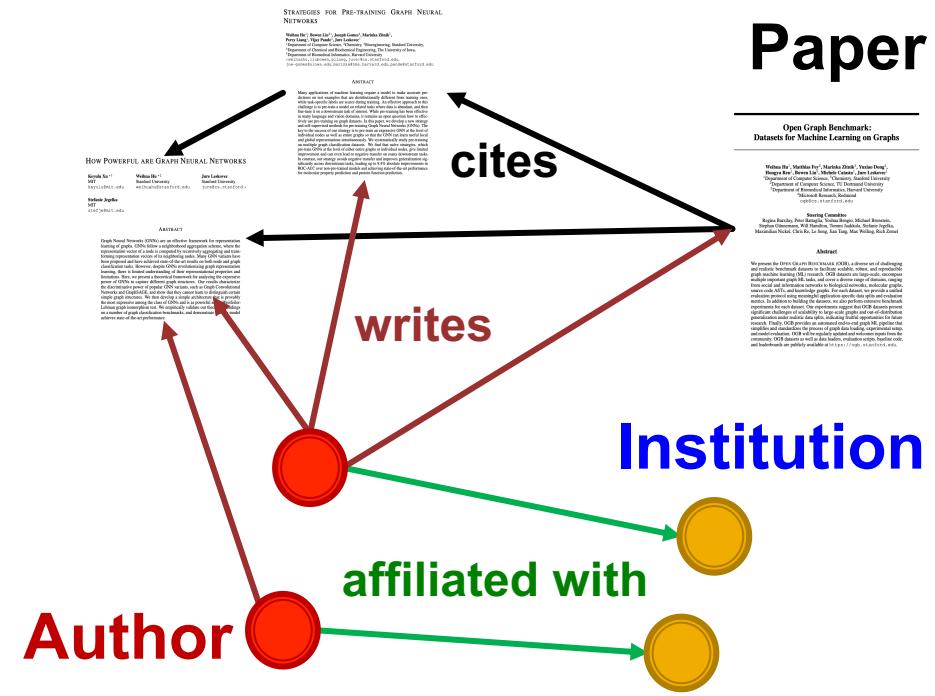
Papers

120M

Authors

120M

Paper



# Graphs in Modern Applications

## ■ Knowledge Graphs (KGs):

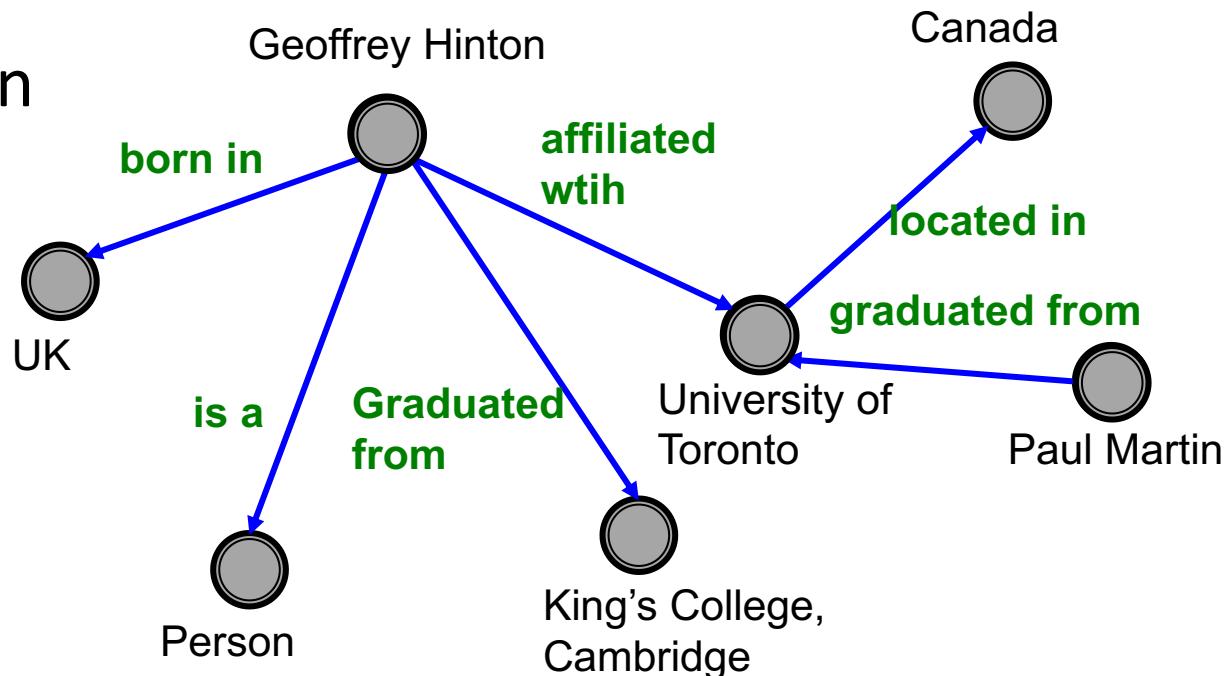
- Wikidata
- Freebase

## ■ ML tasks:

- KG completion
- Reasoning

Entities  
80M—90M

kG



# 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

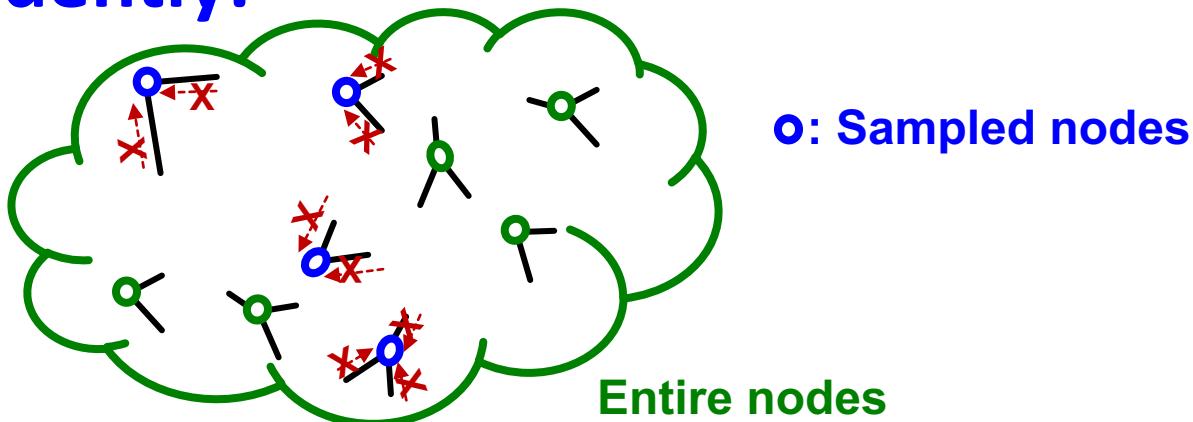
$$\ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=0}^{N-1} \ell_i(\boldsymbol{\theta})$$

- $\boldsymbol{\theta}$ : 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.  
*won't work in GNN*
    - 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:

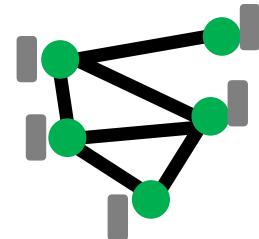


- Sampled nodes tend to be isolated from each other.
- Recall: GNN generate node embeddings by aggregating neighboring node features. no message passing
  - GNN does not access to neighboring nodes within the mini-batch!
- Standard SGD cannot effectively train GNNs.

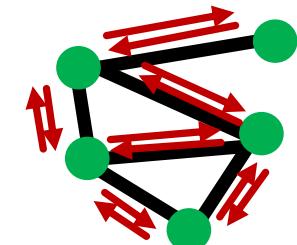
# Why is it Hard?

- **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

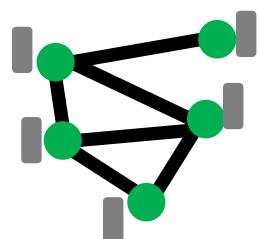
Given all node embeddings at layer K



Perform message-passing



Obtain all node embeddings at layer K+1



# Why is it Hard?

time-wise / data-wise inefficient

- 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-processing 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

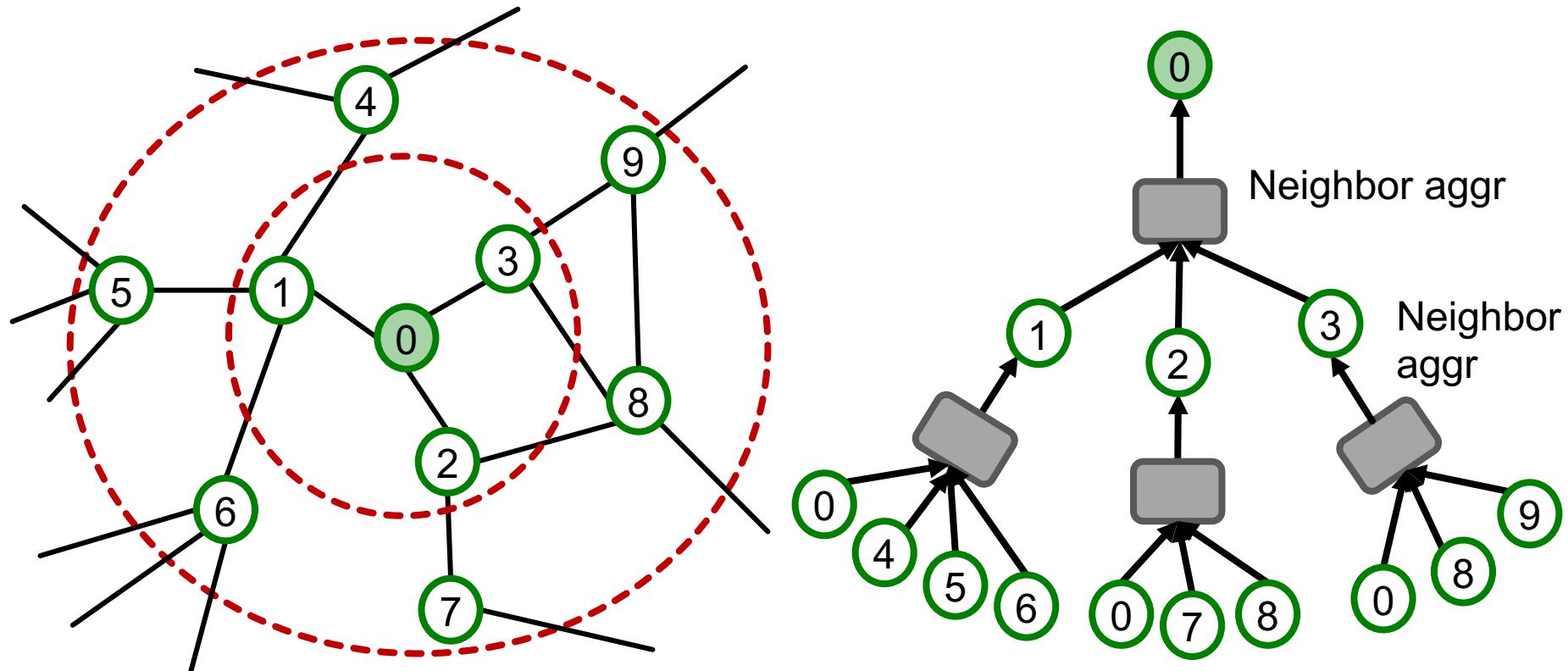
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



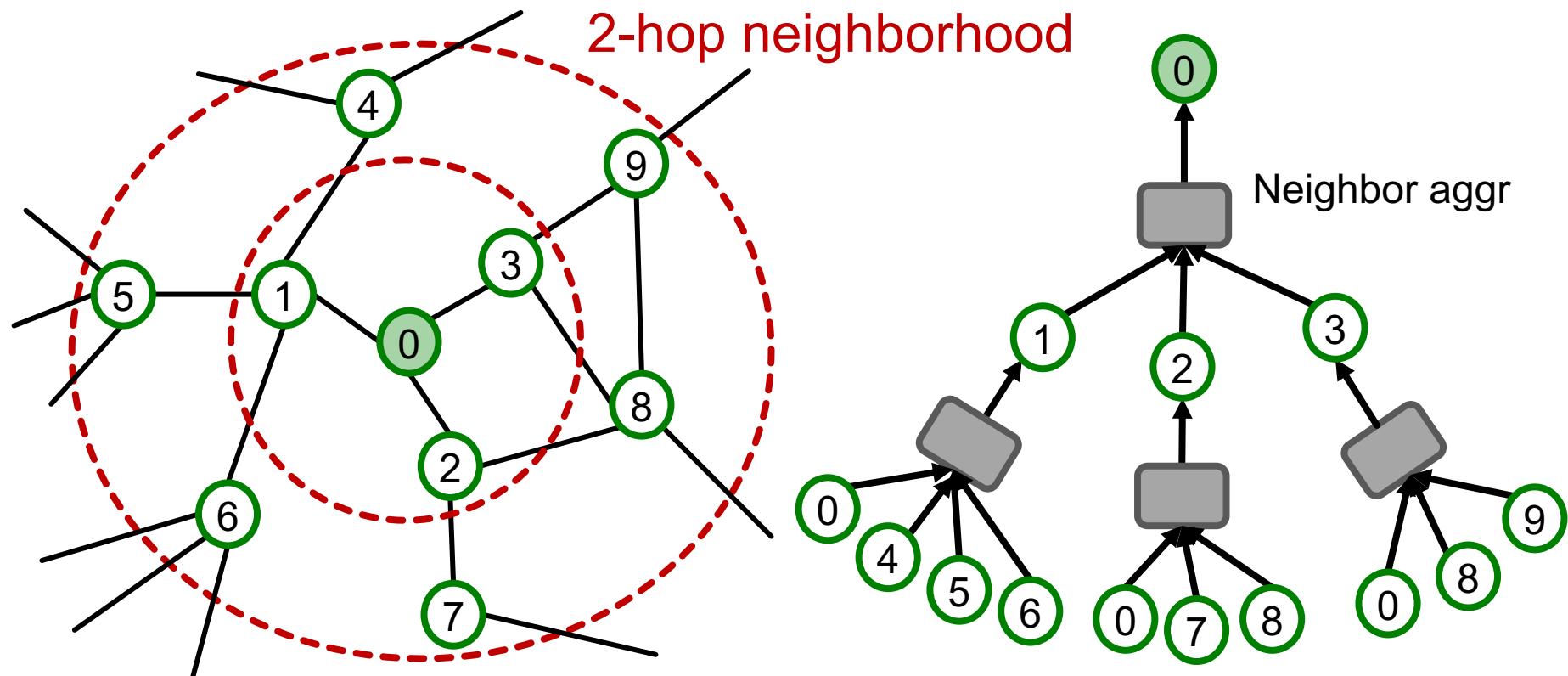
# Recall: Computational Graph

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



# Recall: Computational Graph

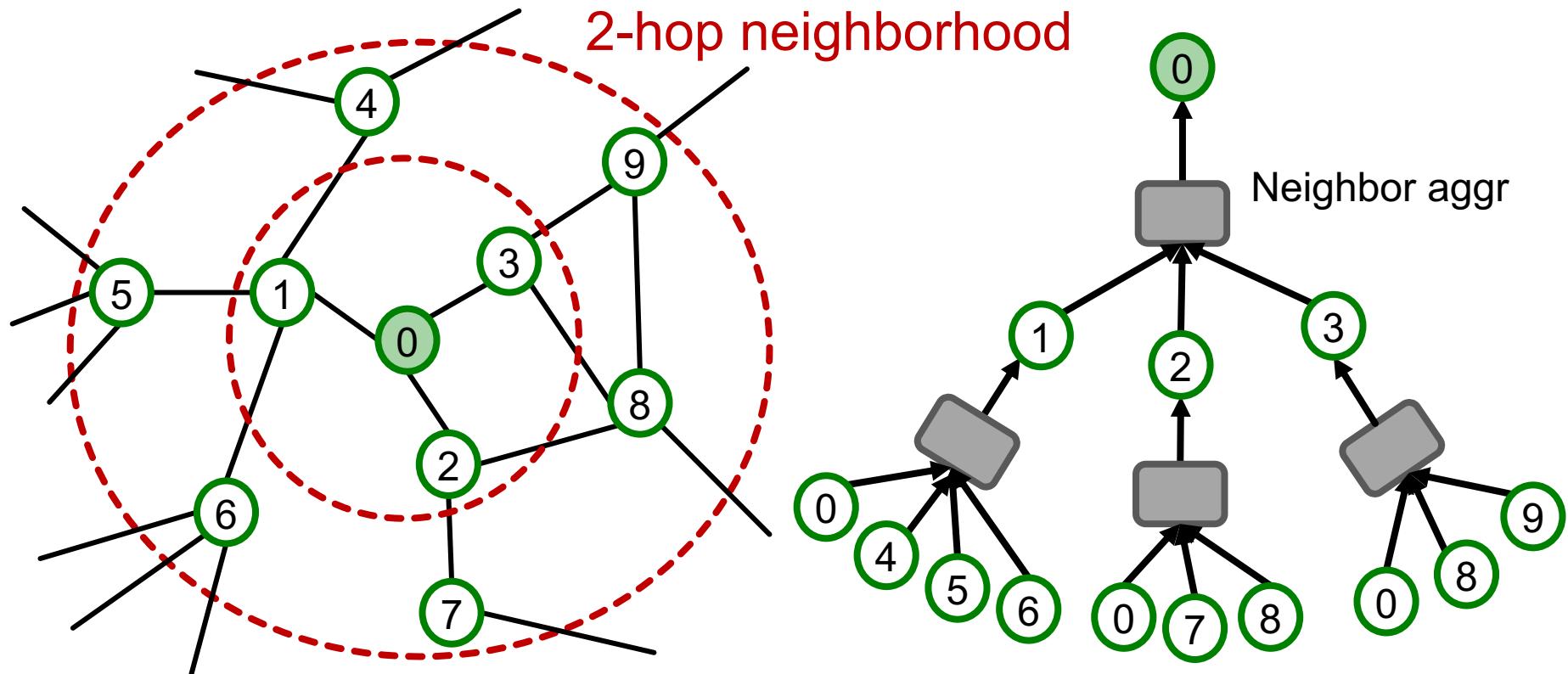
- **Observation:** A 2-layer GNN generates embedding of node “0” using 2-hop neighborhood structure and features.



# Recall: Computational Graph

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

can ignore the rest  
of the network



# Computing Node Embeddings

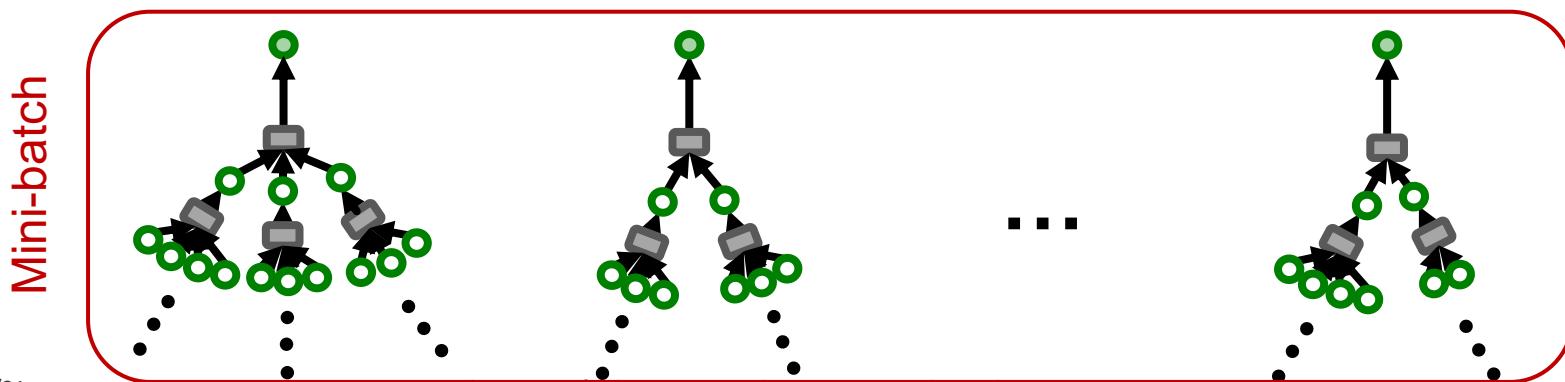
- **Key insight:** To compute embedding of a single node, all we need is **the  $K$ -hop neighborhood** (which defines the computation graph).  
the rest → not affect  
emb of node of interest
- Given a set of  $M$  different nodes in a **mini-batch**, we can generate their embeddings using  $M$  computational graphs. **Can be computed on GPU!**

rather than putting nodes into mini-batches, putting comp. graph ( $k$ -hop neigh.)

Comp. graph  
for 1-st node

Comp. graph  
for 2-nd node

Comp. graph  
for  $M$ -th node

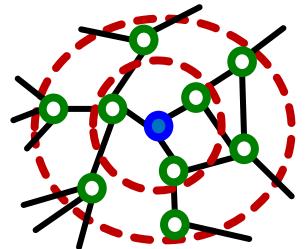


# Stochastic Training of GNNs

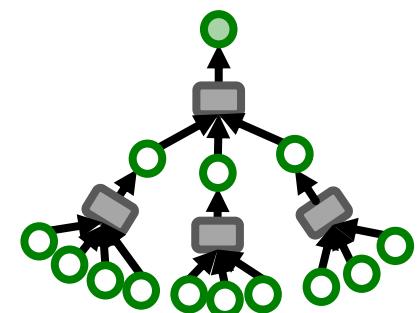
- We can now consider the following SGD strategy for training  $K$ -layer GNNs:

- 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:  $\theta \leftarrow \theta - \nabla \ell_{sub}(\theta)$   
update model parameter w.r.t. to the gradient

**$K$ -hop neighborhood**

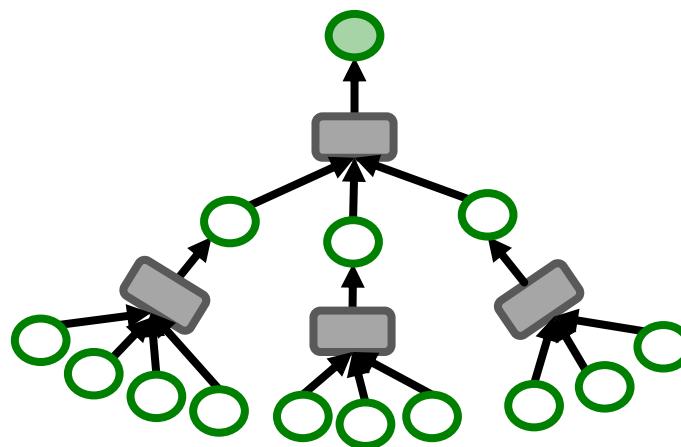


**Computational graph**



# Issue with Stochastic Training

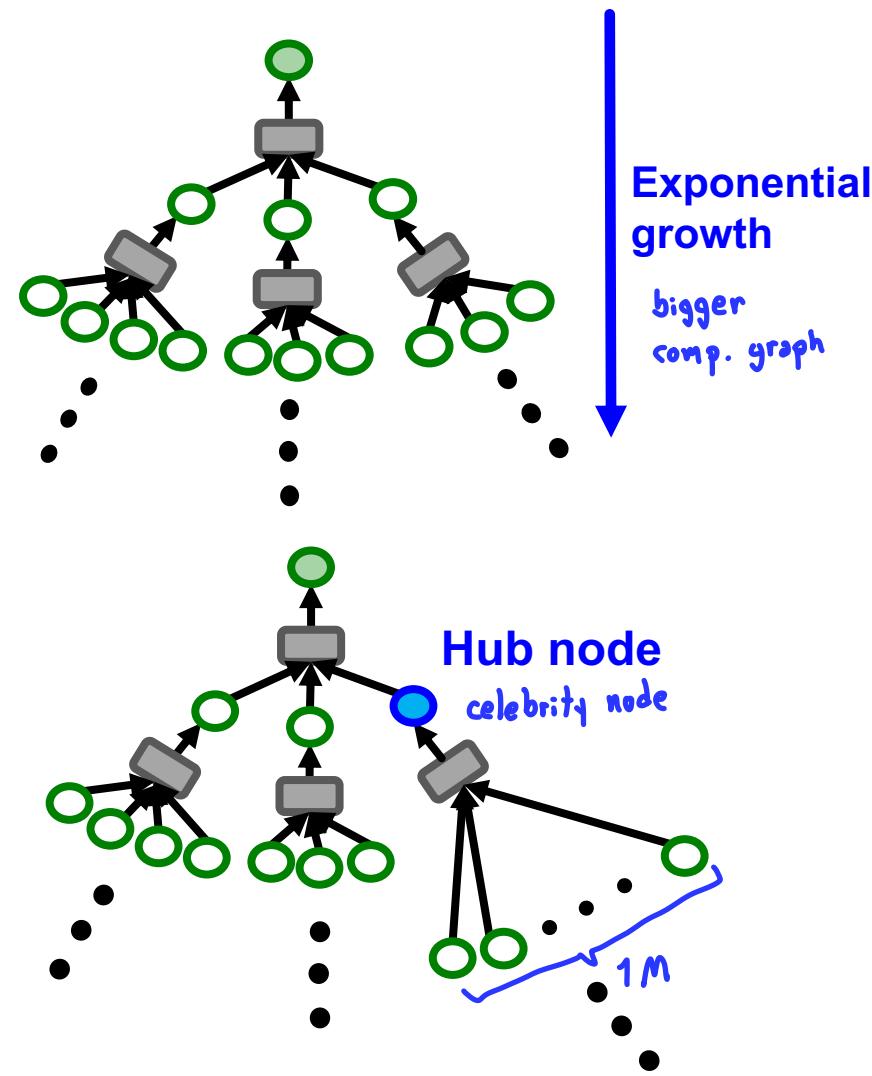
- For each node, we need to get the entire **K-hop neighborhood** and pass it through the computation graph. *then load to GPU memory*
- We need to aggregate lot of information just to compute one node embedding.
- Computationally expensive.



# 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

Sampling at most  $H$  neighbors  
of every node

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

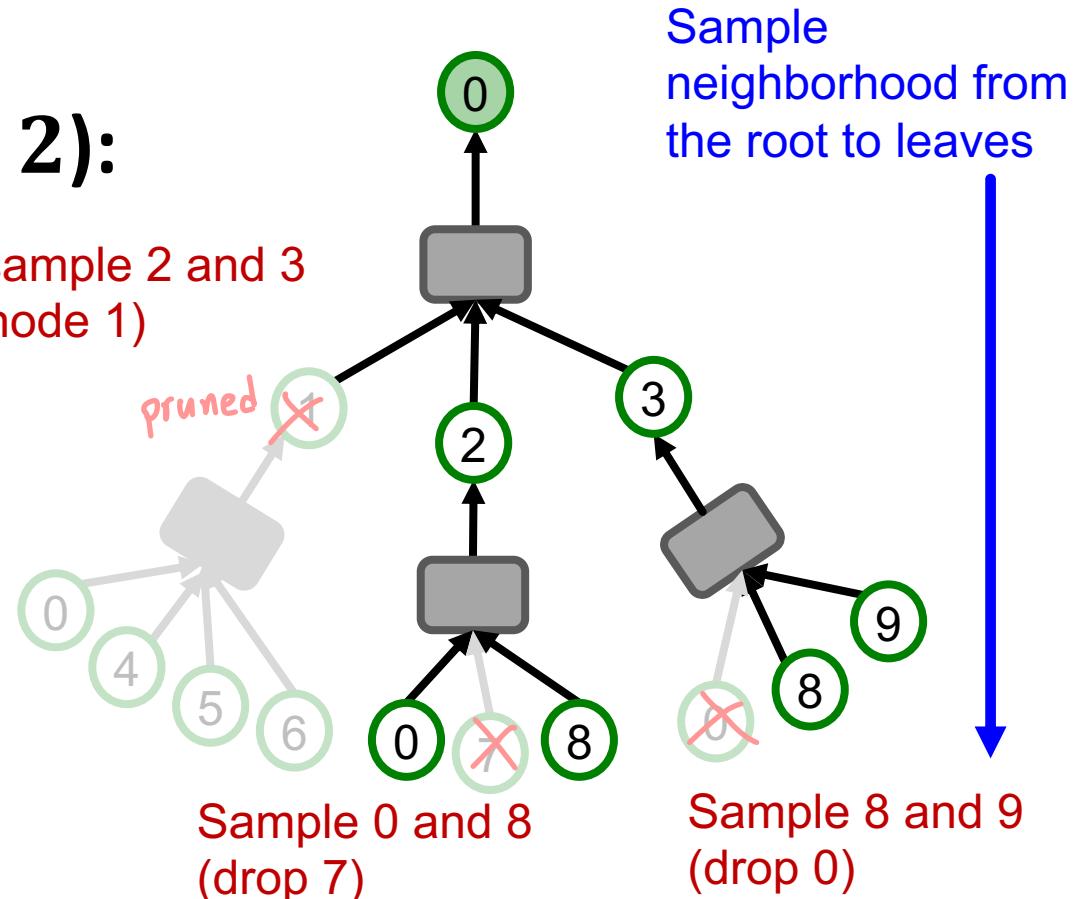
- Example ( $H = 2$ ):

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

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

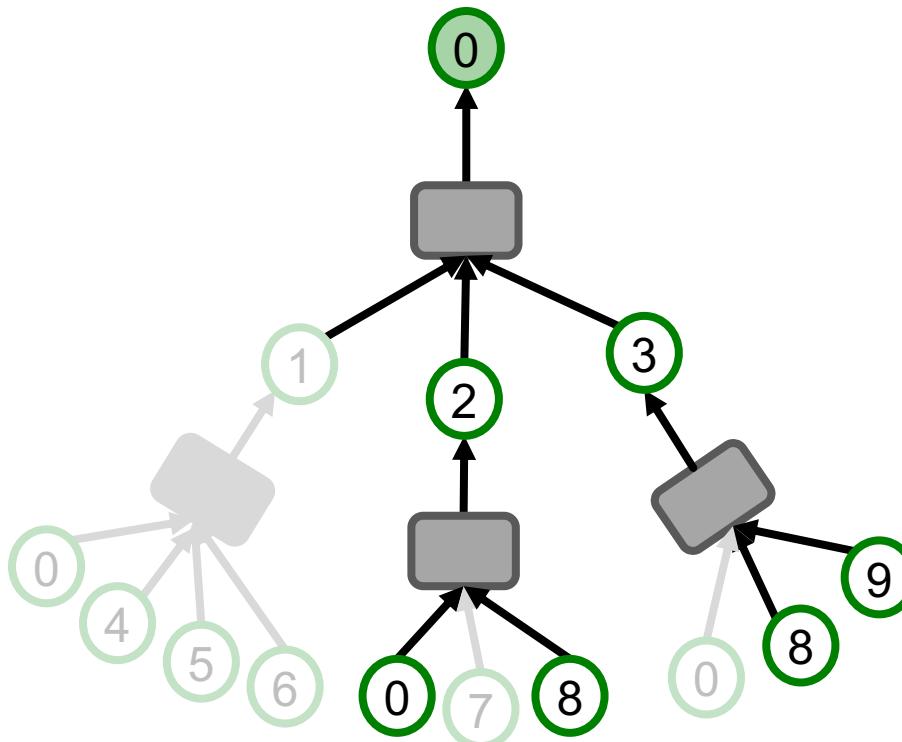
more manageable

First, sample 2 and 3  
(drop node 1)



# Neighbor Sampling

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



# Neighbor Sampling Algorithm

## Neighbor sampling for $K$ -layer GNN

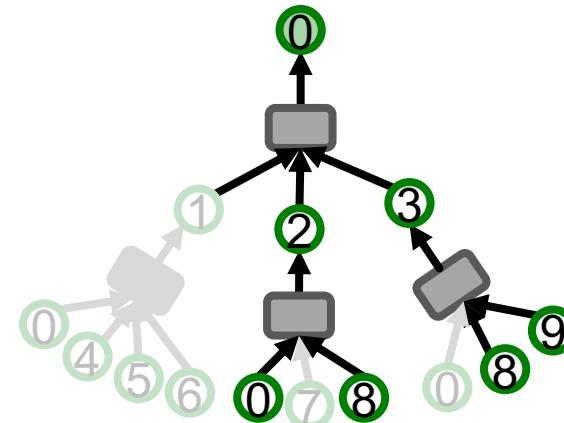
- For  $k = 1, 2, \dots, K$ :
  - For each node in  $k$ -hop neighborhood:
  - (Randomly) sample at most  $H_k$  neighbors:

**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.  
more noise, higher variance
- **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)

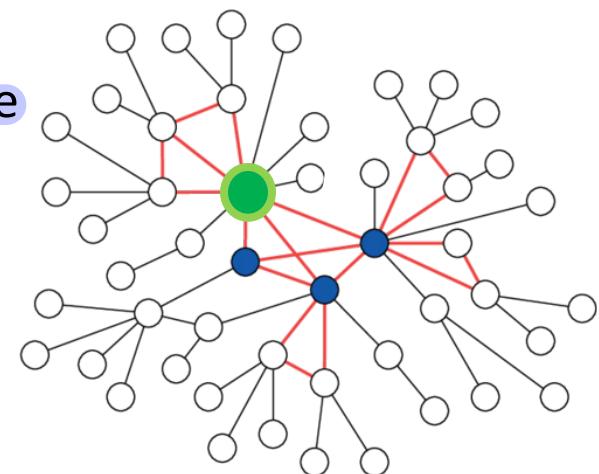
## ■ Remark 3: How to sample the nodes

- **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  $R_i$  *take H most important neighbors*

- 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. *like form of dropout*
  - The pruned comp. graph is used to generate a node embedding. *too much → not reliable*
- caveat
- 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

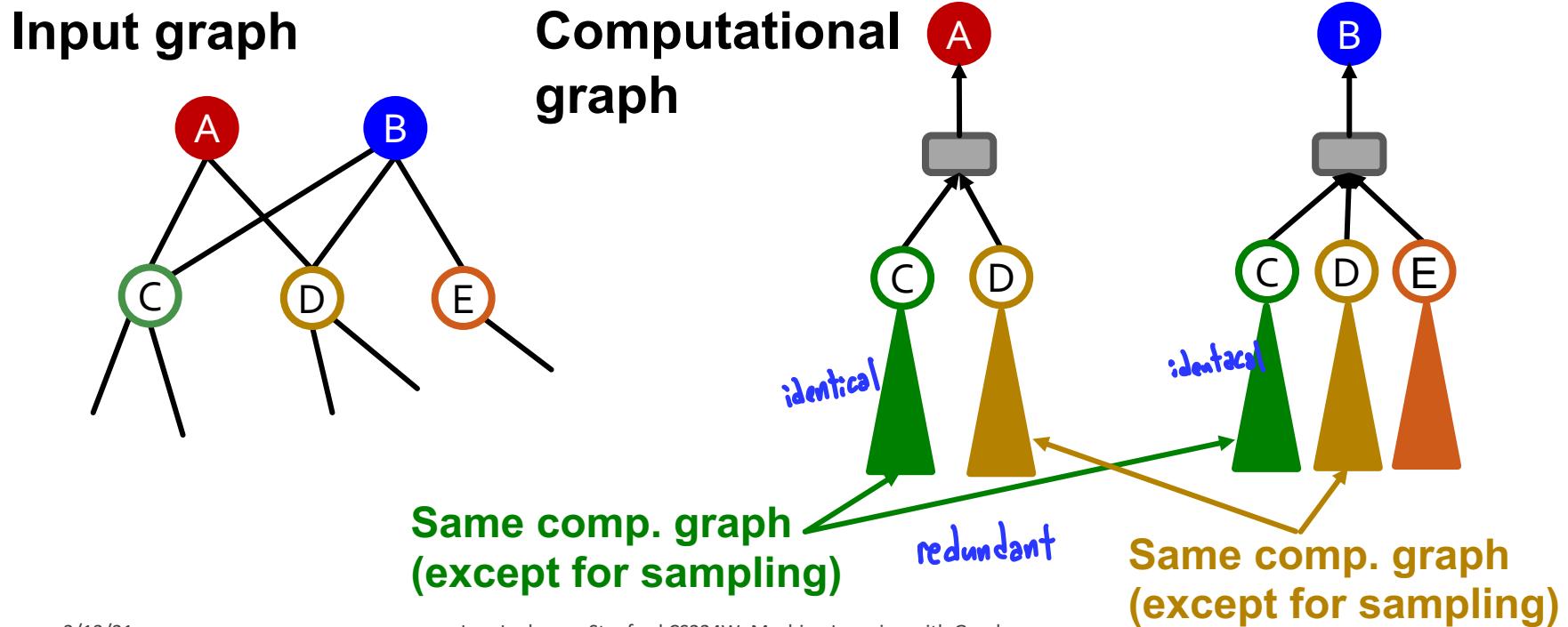
Jure Leskovec, Stanford University

<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.



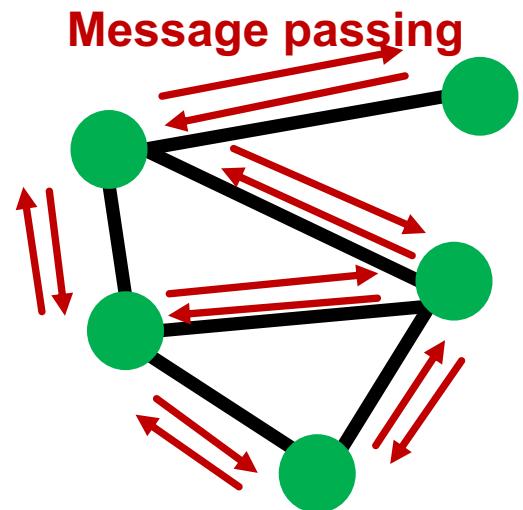
# Recall: Full Batch GNN

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

Update for all  $v \in V$

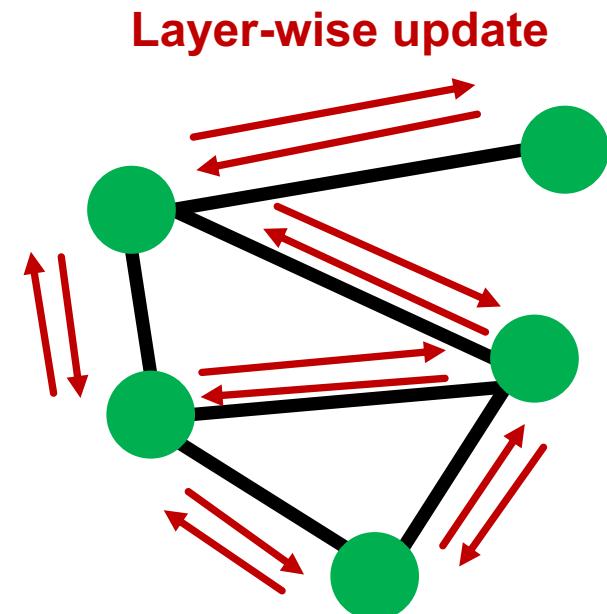
$$h_v^{(\ell)} = COMBINE \left( h_v^{(\ell-1)}, AGGR \left( \left\{ \mathbf{h}_u^{(\ell-1)} \right\}_{u \in N(v)} \right) \right)$$

- 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

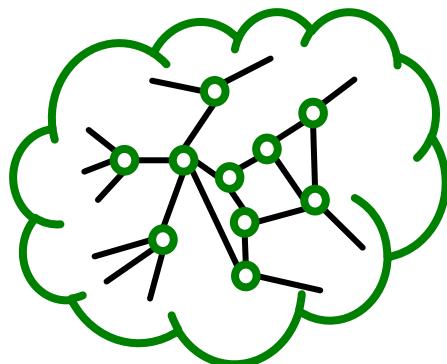
- The **layer-wise** node embedding update allows the re-use of embeddings from the previous layer.
- This significantly **reduces the computational redundancy of neighbor sampling**.
  - Of course, the **layer-wise** update is **not feasible** for a large graph due to **limited GPU memory**.



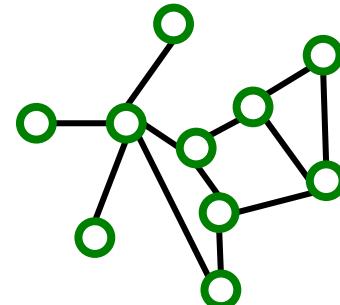
# Subgraph Sampling

- **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. *Full-batch implementation*

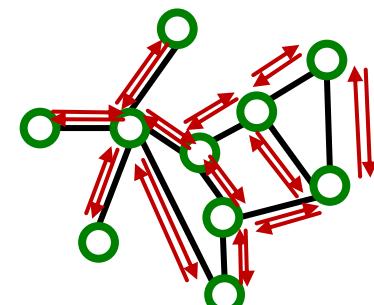
Large graph



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



Layer-wise  
node embeddings  
update on the GPU



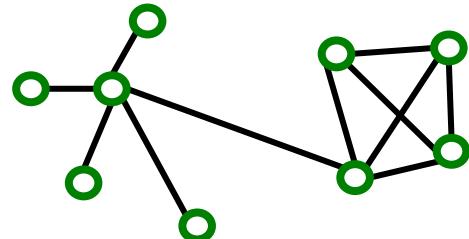
# Subgraph Sampling

- **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.**mimic computation on the big graph
    - This way, the GNN over the subgraph generates embeddings closer to the GNN over the original graph.

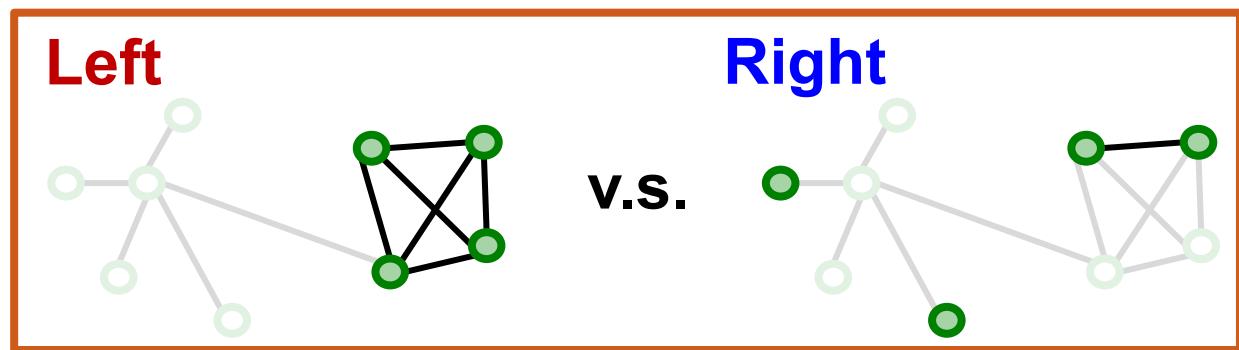
# Subgraph Sampling: Case Study

- Which subgraph is good for training GNN?

Original graph



Subgraphs (both 4-node induced subgraph)

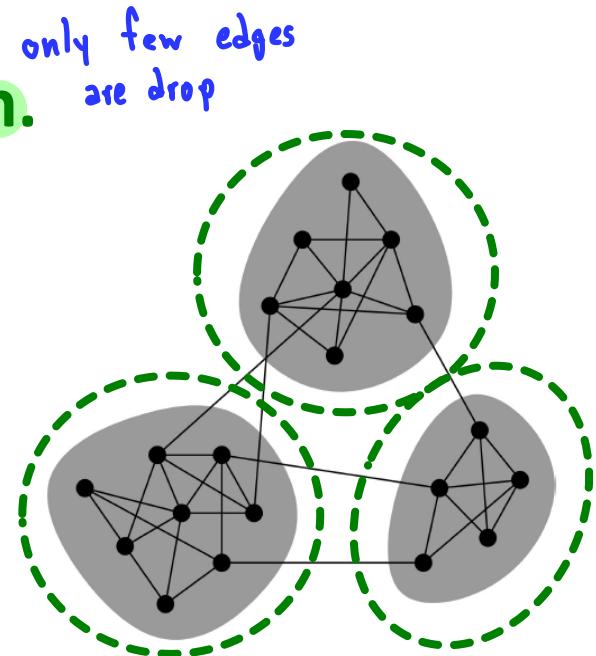


- **Left subgraph** retains the essential community structure among the 4 nodes → **Good ✓**
- **Right subgraph** drops many connectivity patterns, even leading to isolated nodes → **Bad ✗**

# Exploiting Community Structure

Real-world graph exhibits community structure

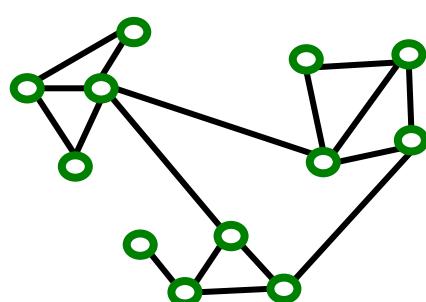
- A large graph can be decomposed into many small communities.
- **Key insight** [Chiang et al. KDD 2019]:  
Sample a **community as a subgraph**.  
Each subgraph retains essential local connectivity pattern of the original graph.



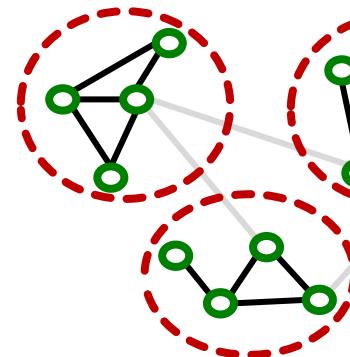
# Cluster-GCN: Overview

- 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**.

Input large graph

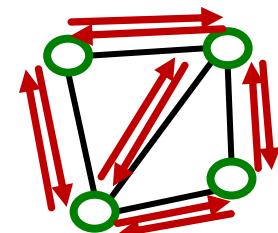


Partitioning



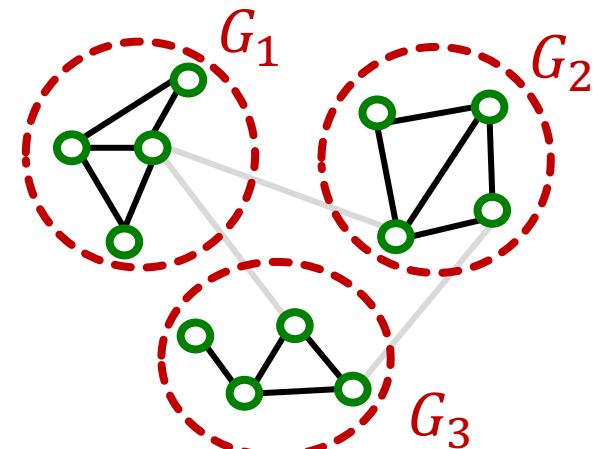
Sample

Mini-batch training  
Message-passing  
over induced subgraph  
to compute the loss



# Cluster-GCN: Pre-processing

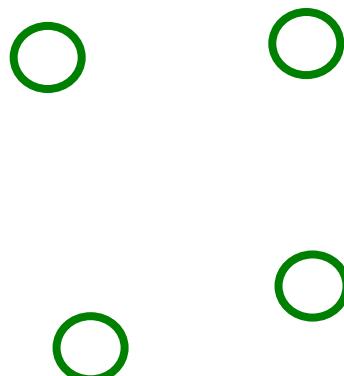
- Given a large graph  $G = (V, E)$ , **partition its nodes  $V$  into  $C$  groups:**  $V_1, \dots, V_C$ .
    - We can use any scalable community detection methods, e.g., Louvain, METIS, <sup>BigCLAM</sup> [Karypis et al. SIAM 1998].
  - $V_1, \dots, V_C$  induces  $C$  subgraphs,  $G_1, \dots, G_C$ ,
    - 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  $G_1, \dots, G_C$ .**



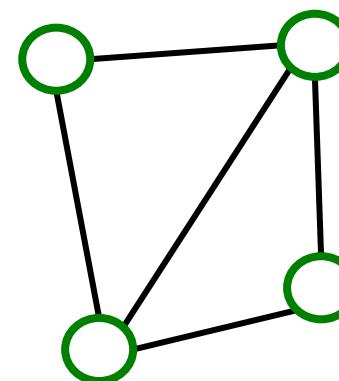
# Cluster-GCN: Mini-batch Training

- 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  $G_c$

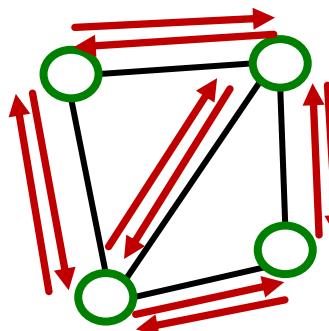


create an induced subgraph over that set of nodes

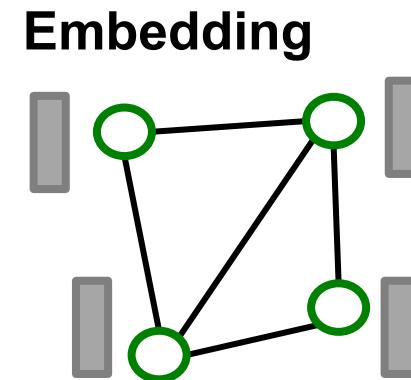
# Cluster-GCN: Mini-batch Training

- 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:  $\theta \leftarrow \theta - \nabla \ell_{sub}(\theta)$  *gradient update*

*fit into GPU memory*  
Induced subgraph  $G_c$

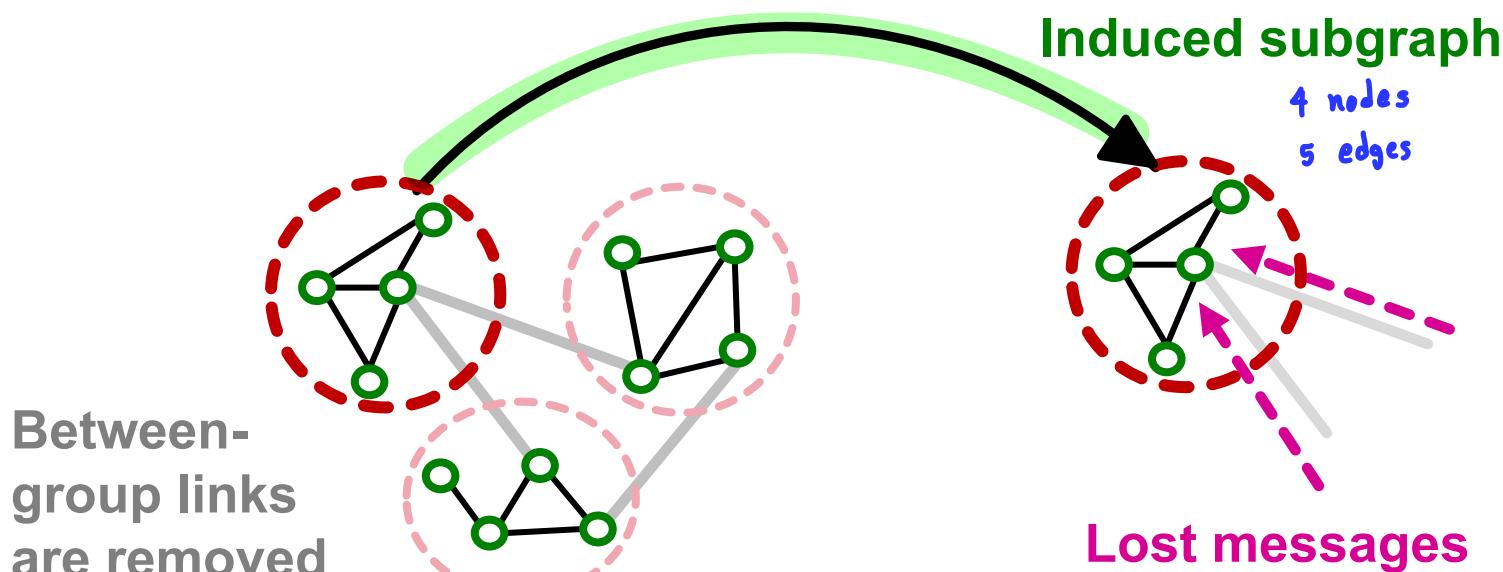


Layer-wise node  
embedding update



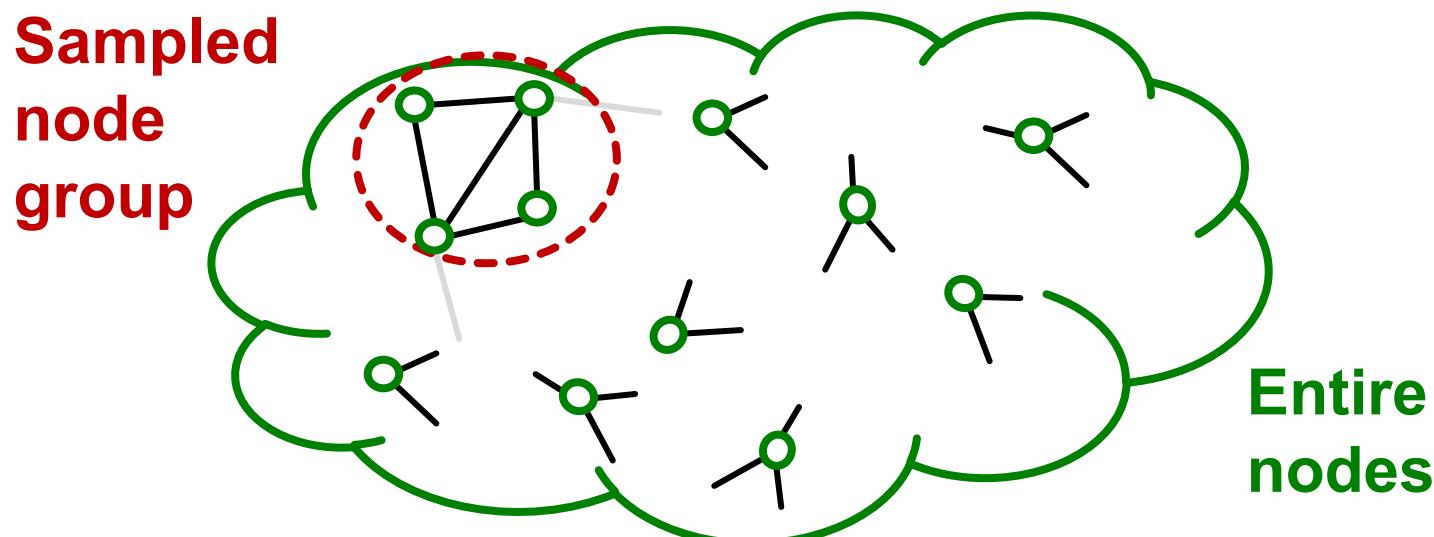
# Issues with Cluster-GCN (1)

- 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.  
fluctuate a lot and training is unstable → slow convergence
- 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)
- subgroup can be more diverse → gradient more stable/  
representative*

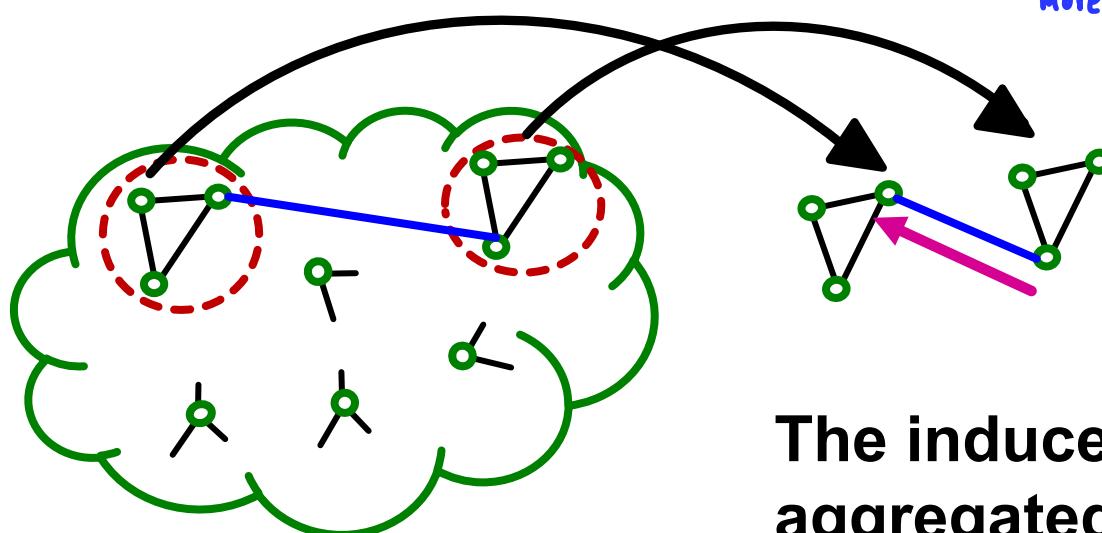
# Advanced Cluster-GCN: Overview

## ■ 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.

*more stable*



**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:  
 $V_1, \dots, V_C$ . much smaller in the original version
- $V_1, \dots, V_C$  needs to be small so that even if multiple of them are aggregated, the resulting group would not be too large.

# Advanced Cluster-GCN

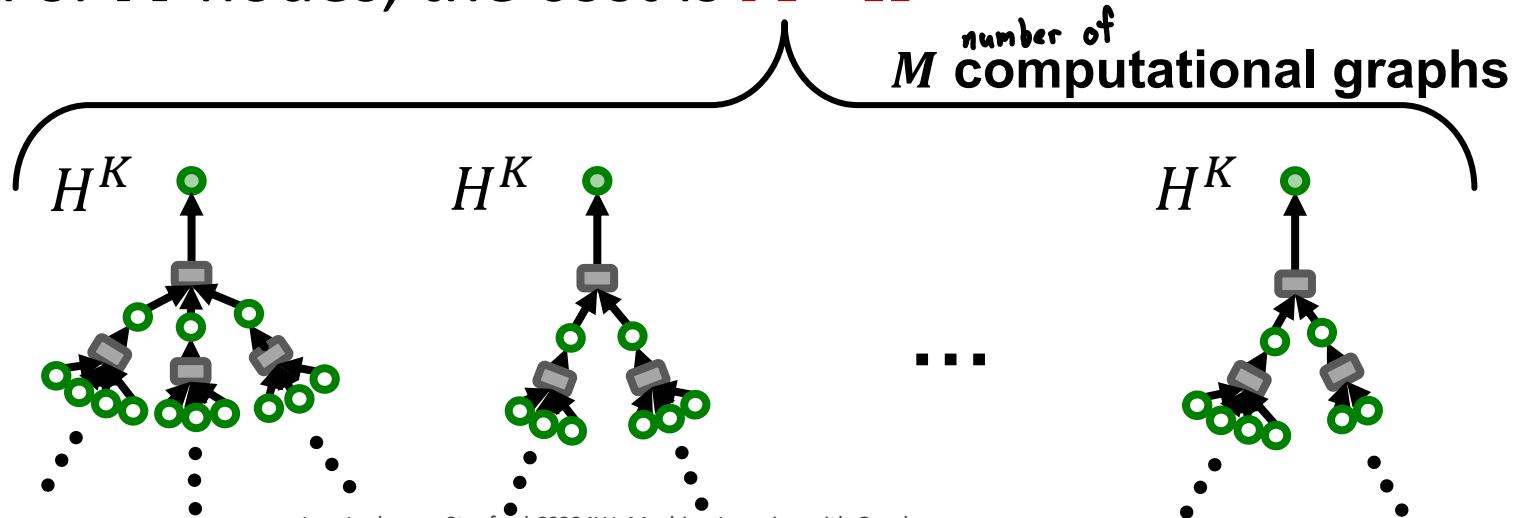
**Mini-batch training:** sample multiple groups of nodes, then aggregate into one super group

- 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

Cluster GCN vs. Neighbor Sampling

- 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  $H^K$ .  
layer  
fan out of comp. graph
  - 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.  $K$ .

But Neighborhood Sampling is use more bc of additional flexibility

# 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. *when number of layers large,  $m \cdot H^k$  grow exponentially*
- But Cluster-GCN leads to systematically biased gradient estimates (due to missing cross-community edges)  
*if # layers is deep, Neighbor sampling can go deep (real depth of ori.), but Cluster-GCN is going in sampled graph*

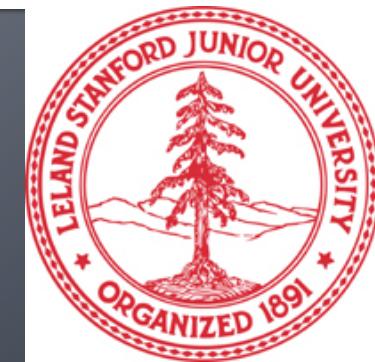
# Scaling up by Simplifying GNNs

kind of an orthogonal approach  
to our first 2 approaches

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)

GCN

- **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  $v \in V$ , aggregate neighboring information as

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

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

The equation shows the computation of the next layer's embeddings. It consists of three main parts: 1) The previous layer's embeddings  $h_u^{(k)}$  are summed over all neighbors  $u$  of node  $v$ . 2) This sum is divided by the degree of node  $v$ ,  $|N(v)|$ . 3) The result is passed through a learned weight matrix  $W_k$  and a ReLU activation function.

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

# Recall: Matrix Formulation of GCN

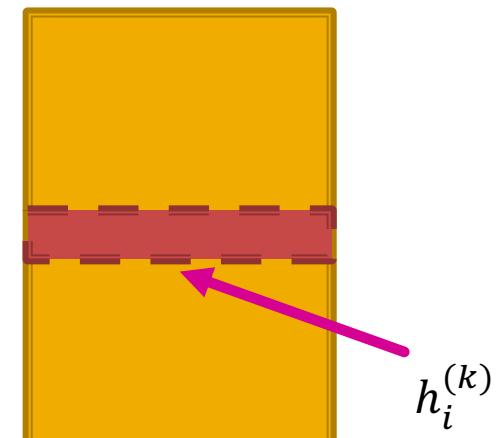
nicely write into the matrix form

GCN aggregations can be formulated as matrix vector product:

- Let  $\mathbf{H}^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$  stack them into an emb matrix
- Let  $\mathbf{A}$  be the adjacency matrix (w/ self-loop)
- Then:  $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:} \mathbf{H}^{(k)}$
- Let  $\mathbf{D}$  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,

$$\frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)} \Rightarrow \mathbf{D}^{-1} \mathbf{A} \mathbf{H}^{(k)} = \mathbf{H}^{(k+1)}$$

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



# Recall: Matrix Formulation of GCN

- GCN's neighbor aggregation: *based on the network*

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

- In matrix form:

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

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

*transformation matrix*

- 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

remove non-linearity

- 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!

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

lect. 2 Power of Adj. matrix → counting path!

connects the target node to its neighbors of neighbor of ...

# 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.** on CPU
- 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:

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

matrix  $\tilde{X}$   
one row for every node  
final emb of  $v$  = row  $v$  times  $W$

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
    - $h_{\nu_1}^{(K)} = \mathbf{W}\tilde{X}_{\nu_1}$ ,      depends on its own row in the matrix  $\tilde{X}$
    - $h_{\nu_2}^{(K)} = \mathbf{W}\tilde{X}_{\nu_2}$ ,
    - ...
    - $h_{\nu_M}^{(K)} = \mathbf{W}\tilde{X}_{\nu_M}$ .

# Simplified GCN: Summary

super fast

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

- **Pre-processing step:**

- Pre-compute  $\tilde{\mathbf{X}} = \tilde{\mathbf{A}}^K \mathbf{X}$ . 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
    - $h_{v_1}^{(K)} = \mathbf{W}\tilde{\mathbf{X}}_{v_1}, h_{v_2}^{(K)} = \mathbf{W}\tilde{\mathbf{X}}_{v_2}, \dots, h_{v_M}^{(K)} = \mathbf{W}\tilde{\mathbf{X}}_{v_M}$
  - 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 **Simplified GCN** 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.**

Theory that we discussed:

- Computation graphs
- WL isomorphism test
- keeping the structure of the underlying subgraphs through that injective mapping

} out of the window  
Bc we don't have non-linearity anymore

# Performance of Simplified GCN

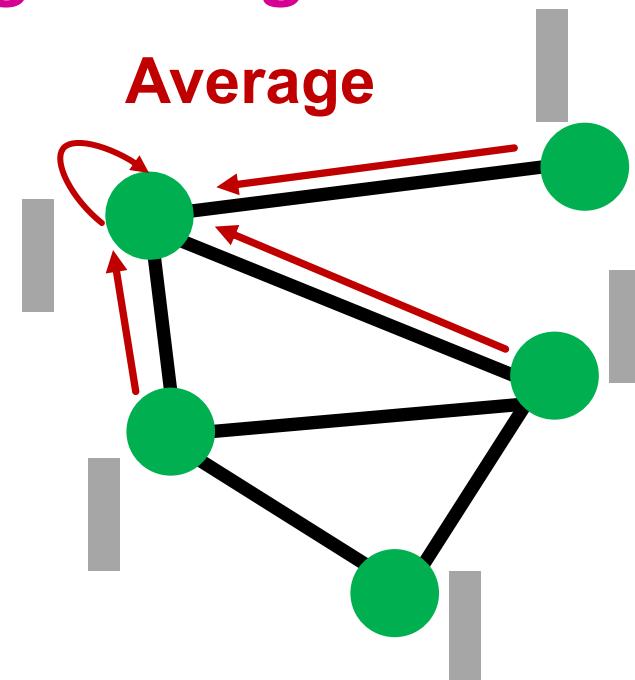
- Surprisingly, in semi-supervised node classification benchmark, **simplified GCN works comparably to the original GNNs despite being less expressive.**
- Why?

# 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.**  $\tilde{A}$  is the  $k$ -hop neighborhood
- 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. *clustered across the homophilic part of the network*



# 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.

easy