Yale

Scaling Up Graph Neural Networks to Large Graphs

CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Readings

- Readings are updated on the website (syllabus page)
- **Lecture 5 readings**:
  - Design Space of Graph Neural Networks
  - OGB Datasets
- **Lecture 6 readings:**
  - GraphSAINT
  - GNN AutoScale

Recap of Lecture 5

- **We introduce the pipeline of GNN training**
  - **Prediction Heads**:
    - Node-level / Edge-level / Graph-level
  - **Predictions & Labels**
    - Supervised / unsupervised
  - **Loss Functions:**
    - Regression / Classification
  - **Evaluation Metrics:**
    - Regression / Classification
  - **Dataset Split**
    - Transductive / inductive
    - Node / edge / graph

Graphs in Modern Applications (1/4)

Scenario 1. Recommender Systems

Examples

- Amazon
- YouTube
- Pinterest

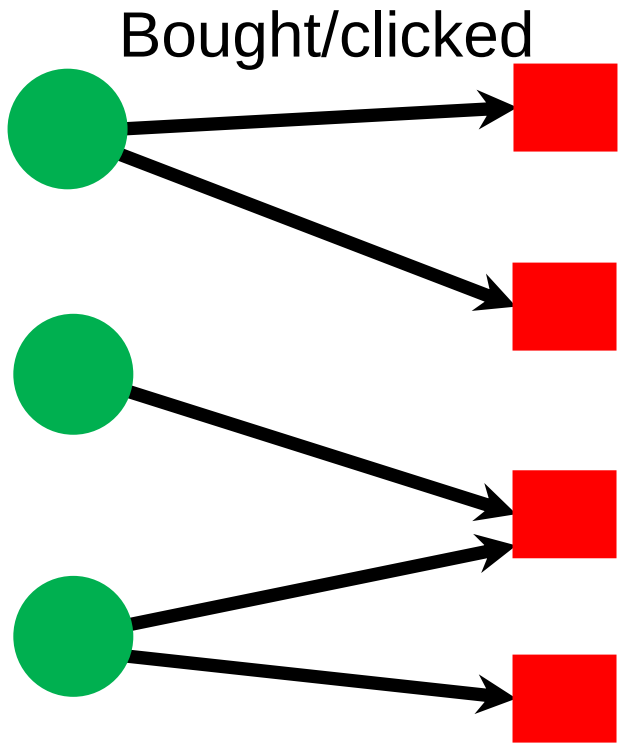
…

Machine Learning Tasks

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

Users
100M~1B

Products/Videos
10M ~ 1B



Graphs in Modern Applications (2/4)

Scenario 2. Social networks

Examples

- Facebook
- Twitter
- Instagram

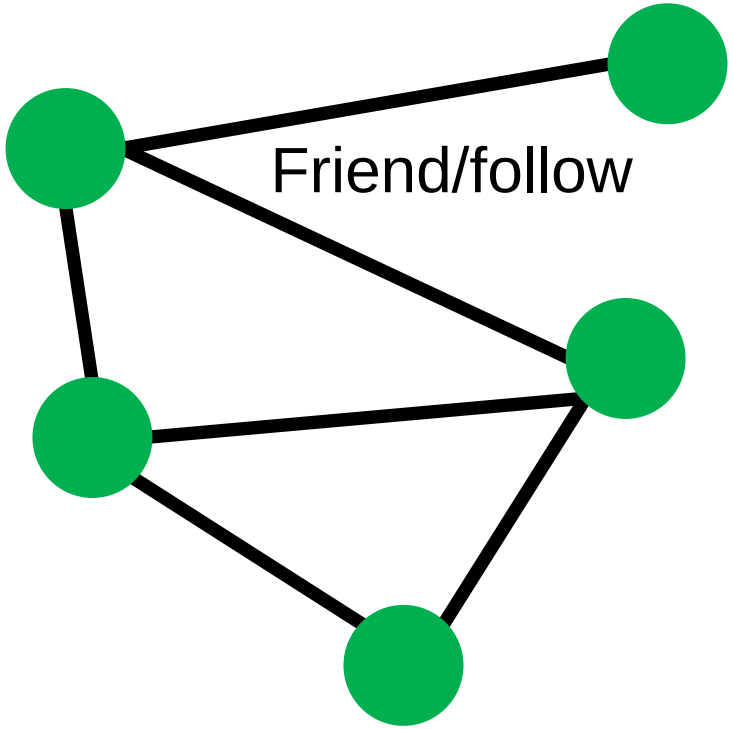
…

Machine Learning Tasks

- **Friend recommendation**
  (link prediction)
- **User property prediction**
  (node classification)

Users

300M~3B



Graphs in Modern Applications (3/4)

Scenario 3. Academic graph

Examples

- Microsoft Academic Graph
- Arxiv Graph

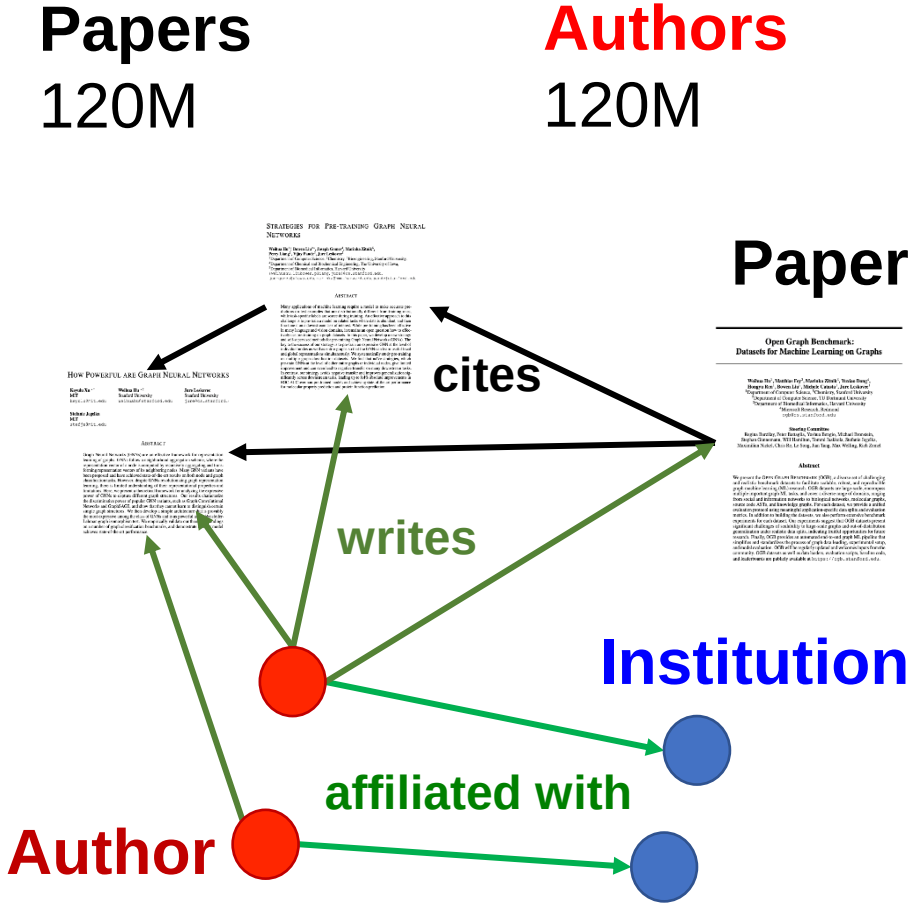
  …

Machine Learning Tasks

- **Paper categorization**

  (node classification)
- **Author collaboration prediction**
- **Paper citation recommendation**

  (link prediction)



Graphs in Modern Applications (4/4)

Scenario 2. Knowledge Graphs (KGs)

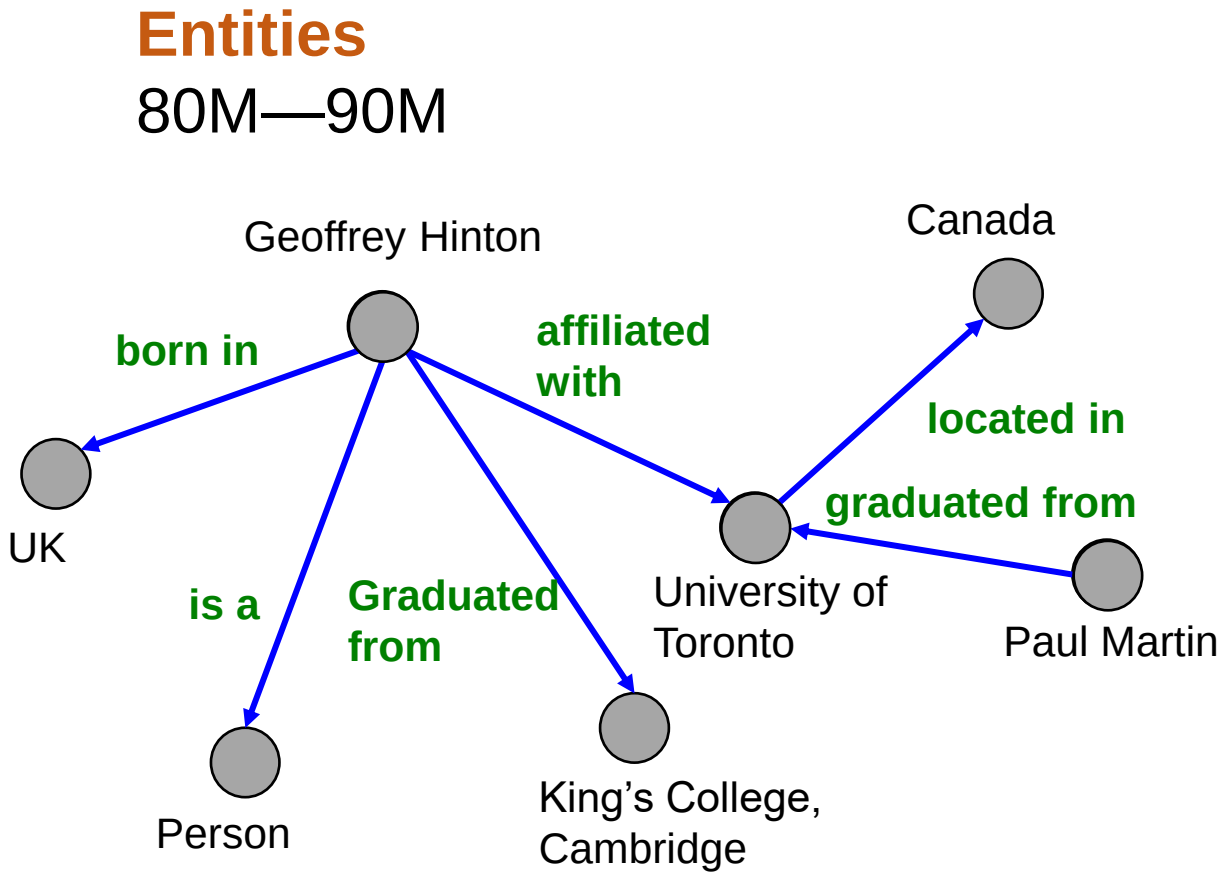
Examples

- Wikidata
- Freebase
- ConceptNet

…

Machine Learning Tasks

- **KG completion** (link prediction)
- **Logical reasoning** (set prediction)



What is in common?

1. Large-Scale:

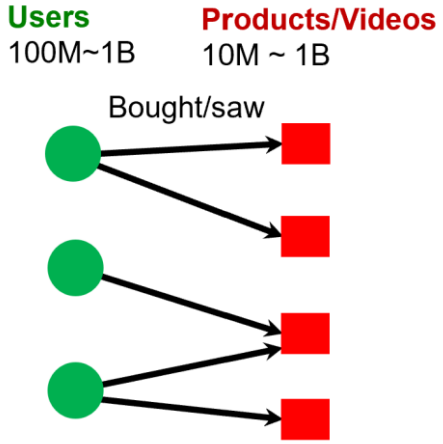
- #nodes ranges from 10M to 10B.
- #edges ranges from 100M to 100B.

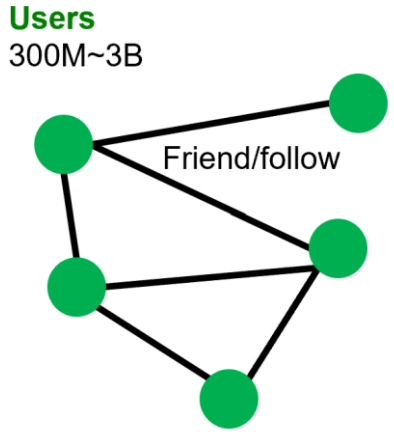
2. Tasks

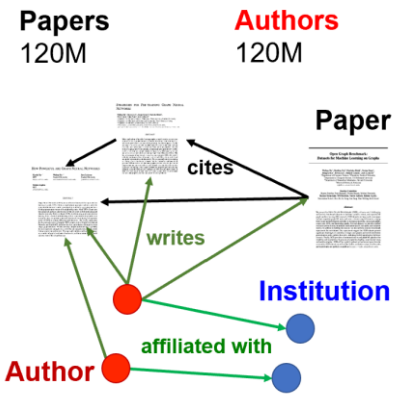
- **Node-level**: User/item/paper classification.
- **Link-level**: Recommendation, completion.

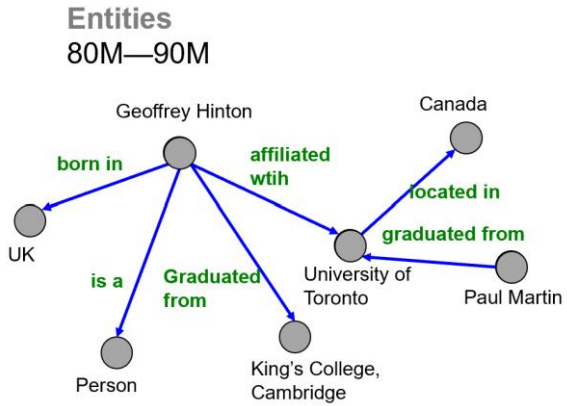
Todays' lecture

- **Scale up GNNs to large graphs!**









Why is it hard? (1/4)

How do we usually train an ML model on large data (N=#data is large)?

Objective: Minimize the averaged loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=0}^{N-1} \mathcal{L}_i(\boldsymbol{\theta})$$

$\boldsymbol{\theta}$: model parameters, $\mathcal{L}_i(\boldsymbol{\theta})$: loss for i-th data point.

Method: Stochastic Gradient Descent (SGD).

- Randomly sample M ($<< N$) data points (**mini-batches** $\mathcal{B}$).
- Compute the $\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$ over the M data points.
- Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$

Why is it hard? (2/4)

Method: Stochastic Gradient Descent (SGD).

- Randomly sample M ($<< N$) data points (**mini-batches**).
- Compute the $\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$ over the M data points.
- Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$

What if we use the standard SGD for GNN on large graphs?

In mini-batch, we sample $\boldsymbol{M}$ ($<<\boldsymbol{N}$) nodes independently

(i) Sampled nodes tend to be **isolated** from each other**.**

(ii) **Recall:** GNNs generate node embeddings by **aggregating neighboring node embeddings**.

(i)&(ii)→ GNN has no access to neighboring nodes within the mini-batch

We need to construct special minibatches to train GNNs.

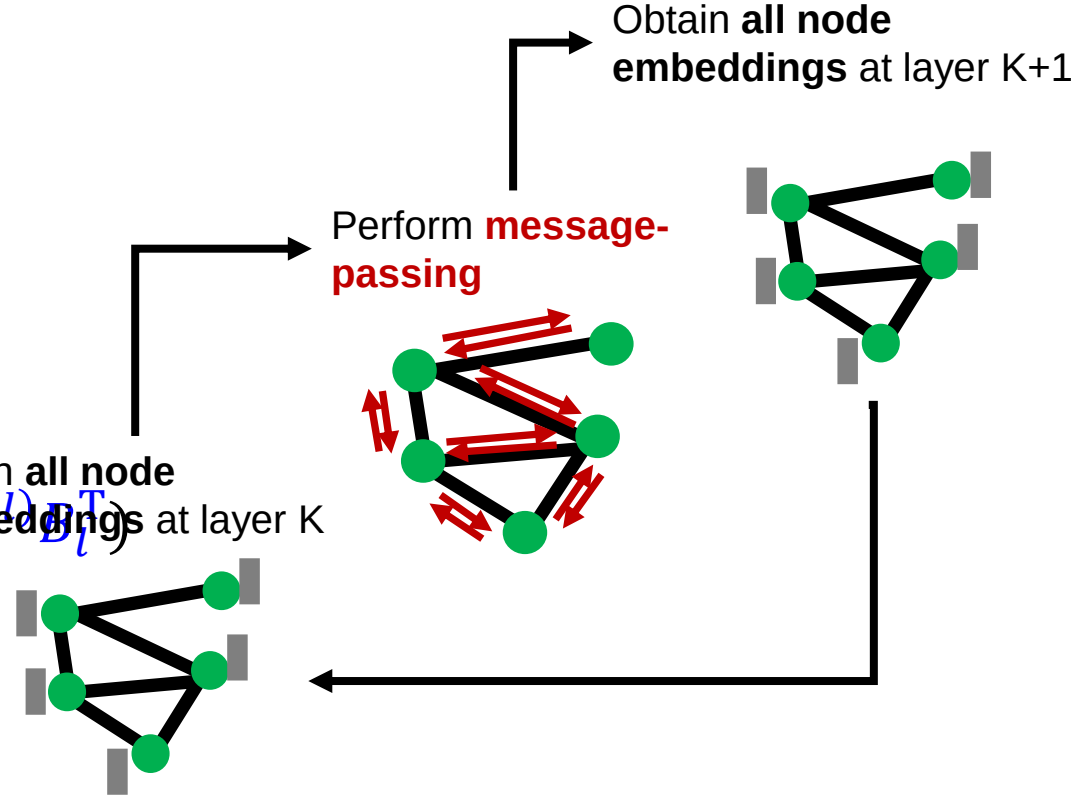
Why is it hard? (3/4)

- **Potential Solution: Naïve full-batch implementation:**
  - Generate embeddings of **all the nodes at the same time**.

1. Load **the entire graph and features.**
2. **At each GNN layer:**
   - compute embeddings of all nodes using all the node embeddings from the previous layer.
   - Use **sparse matrix operation** (recall Lecture 3, 4) For example (GraphSAGE),

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W_l^{\mathrm{T}} + H^{(l)}B_l^{\mathrm{T}})$$

3. Compute the loss
4. Perform gradient descent



Why is it hard? (4/4)

Problem with full-batch implementation

Full-batch implementation is often **not feasible** for large graphs.

Why?

Because we want to use GPU for accelerating training, but GPU memory is **limited** (only 10~40 GB).

The entire graph, features and gradients exceed GPU memory limit.

Outline of Today's Lecture

Three Methods for Scaling up GNNs

1. Neighbor Sampling [Hamilton et al. NeurIPS 2017]

2. Cluster-GCN [Chiang et al. KDD 2019]

→ **Small subgraphs in each mini-batch;** only load subgraphs on GPU at a time.

3. Simplified GCN [Wu et al. ICML 2019] → Simplify a GNN into a **feature-processing operation**.

Outline of Today's Lecture

Three Methods for Scaling up GNNs

1. Neighbor Sampling [Hamilton et al. NeurIPS 2017]

Small subgraphs in each mini-batch; only load subgraphs on GPU at a time.

2. Cluster-GCN [Chiang et al. KDD 2019]

3. Simplified GCN [Wu et al. ICML 2019]

Simplify a GNN into a **feature-processing operation**.

GraphSAGE Neighbor Sampling

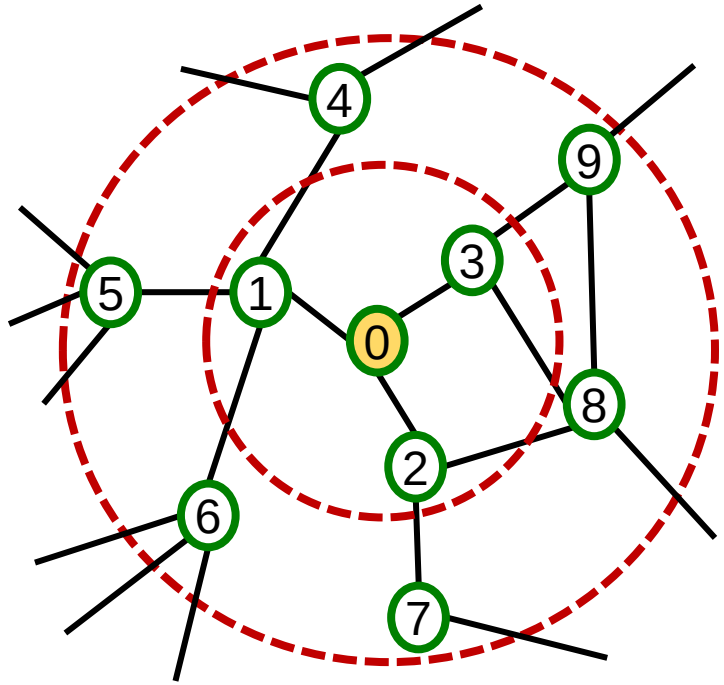
W.L. Hamilton*, R. Ying*, and J. Leskovec. **Inductive representation learning on large graphs.** *Advances in neural information processing systems (NeurIPS)*, *2017*.

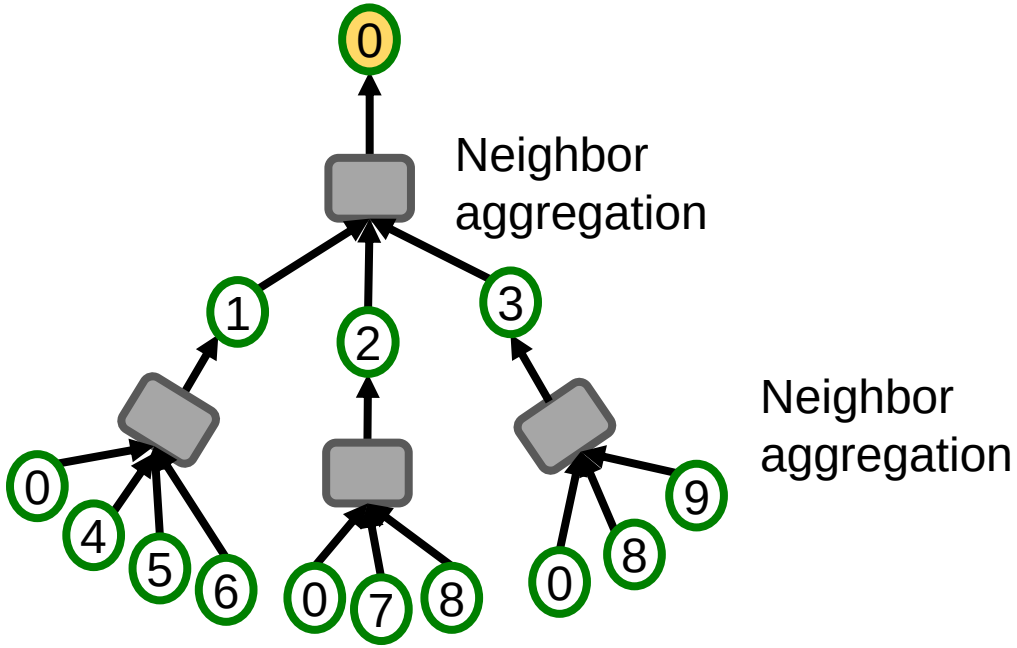
Recall: Computational Graph (1/2)

Recall

GNNs generate node embeddings via neighbor aggregation.

- Represented as a computational graph (right).



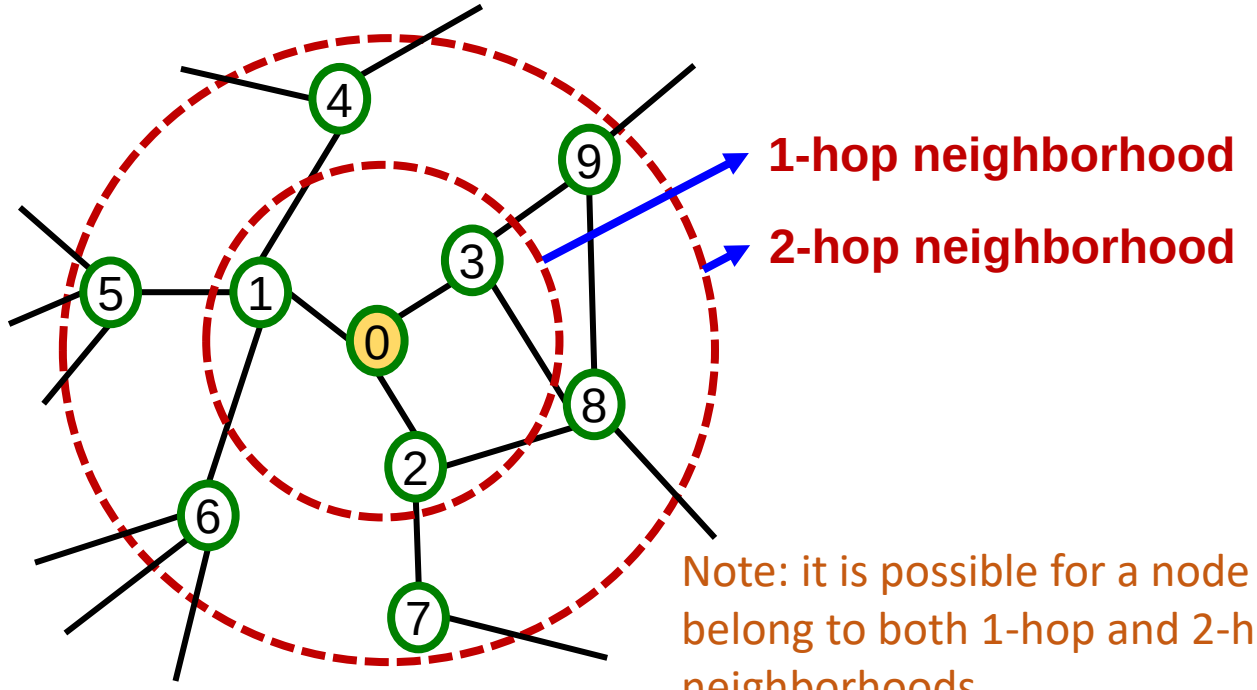


Recall: Computational Graph (2/2)

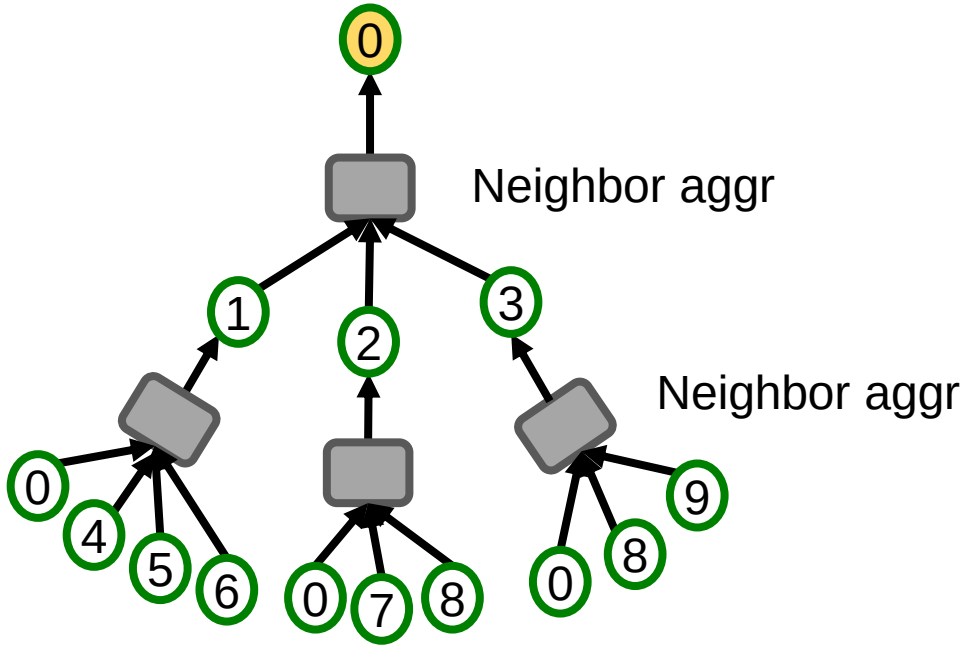
Observation

A **2-layer** GNN generates embedding of node "0" using **2-hop neighborhood** structure and features.

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



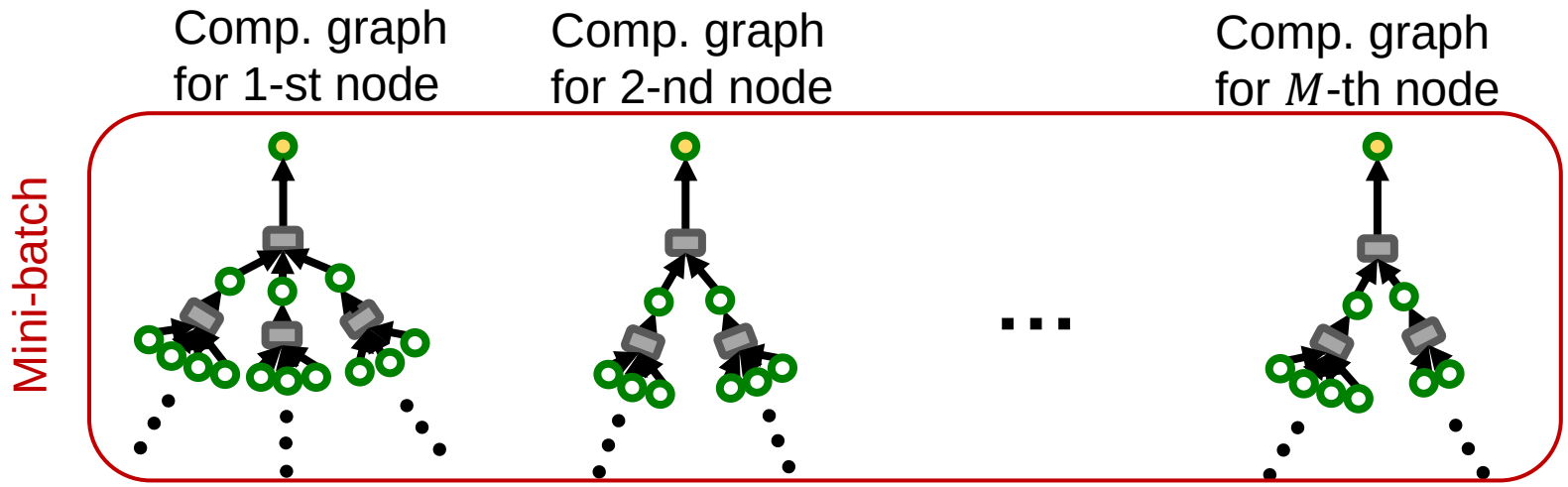
Note: it is possible for a node to belong to both 1-hop and 2-hop neighborhoods



Computing Node Embeddings

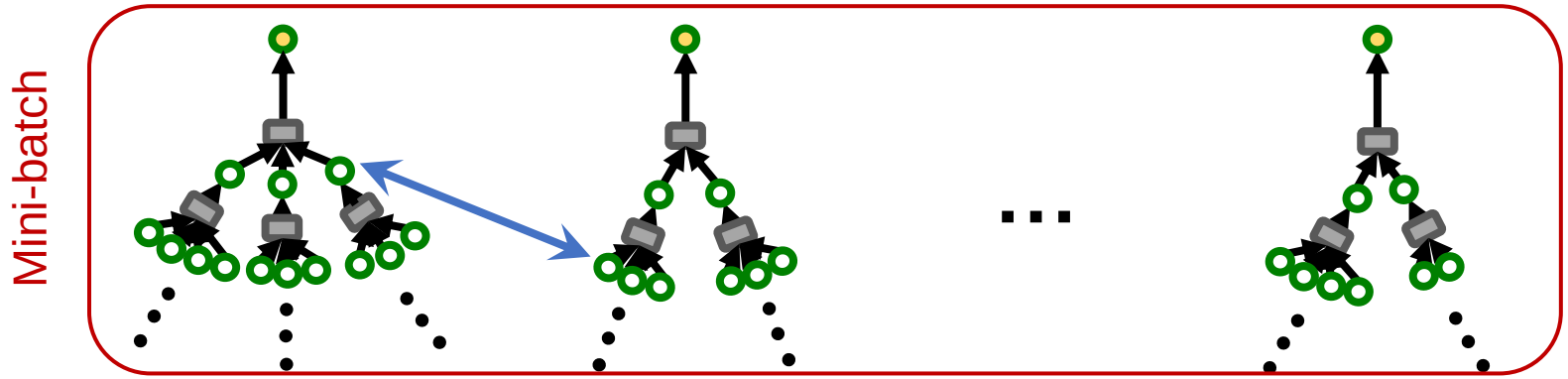
Key Insight

- 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 computation graphs.
- **Can be computed on GPU!**



Node Sharing in Minibatches

- To further reduce memory usage, we could **share the same nodes (features) among different computation trees**, if they correspond to the same node in the original graph
  - The default behavior in Pytorch Geometric Neighbor Sampler
- This is not always possible in special circumstances of **temporal sampling**
  - We sample nodes with **timestamps less than the root node** of the computation tree
  - The same node can have different temporal constraints in different computation trees



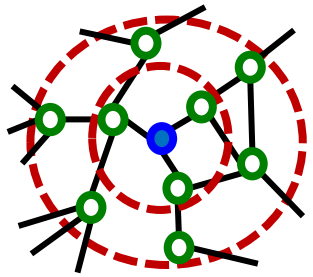
Stochastic Training of GNNs

Potential Solution

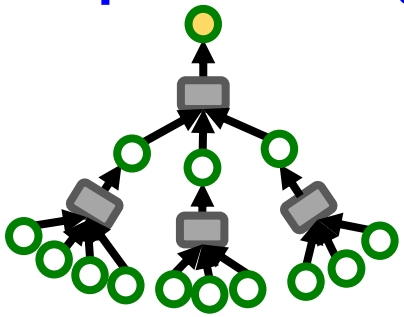
We can now consider the following SGD strategy for training K-layer GNNs

- Randomly sample M $(<< N)$ 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$\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$ averaged over the M nodes.
- Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$

K-hop neighborhood



Computational graph



Issues with Stochastic Training of GNNs (1/2)

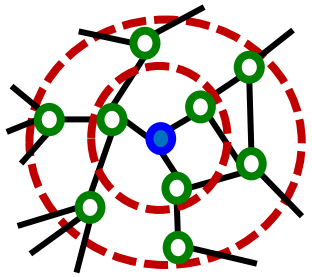
Problems with the Solution

For each node, we need to get the entire K-hop neighborhood and pass it through the computation graph.

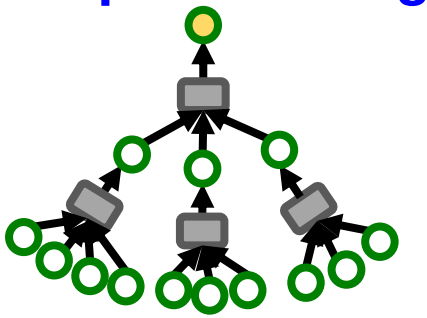
We need to aggregate lot of information just to compute one node embedding.

→ **Computationally expensive**.

K-hop neighborhood



Computational graph



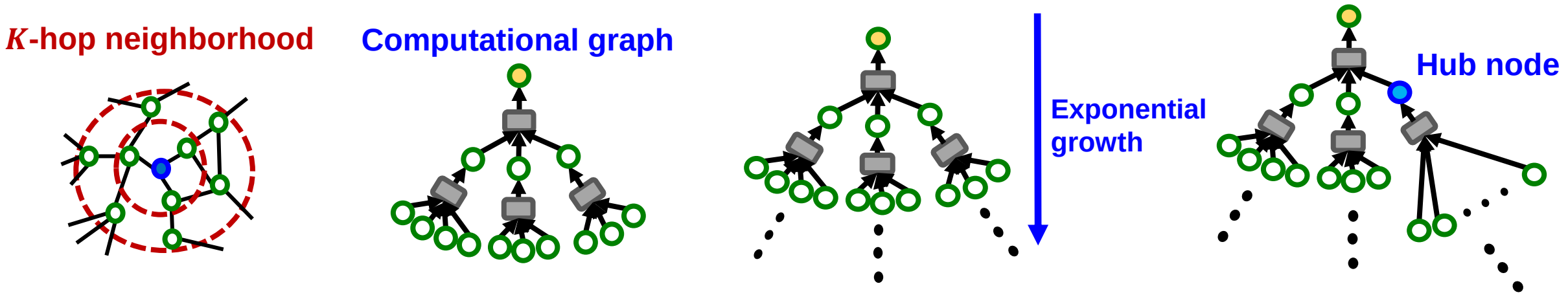
Issues with Stochastic Training of GNNs (2/2)

Problems with the Solution

Specifically, two major issues may occur.

1. **Exponential growth**: Computation graph becomes exponentially large with respect to the layer size K.
2. **Hub node**: Computation graph explodes when it hits a hub node (high-degree node).

How to fix this? Make the computational graph more compact!

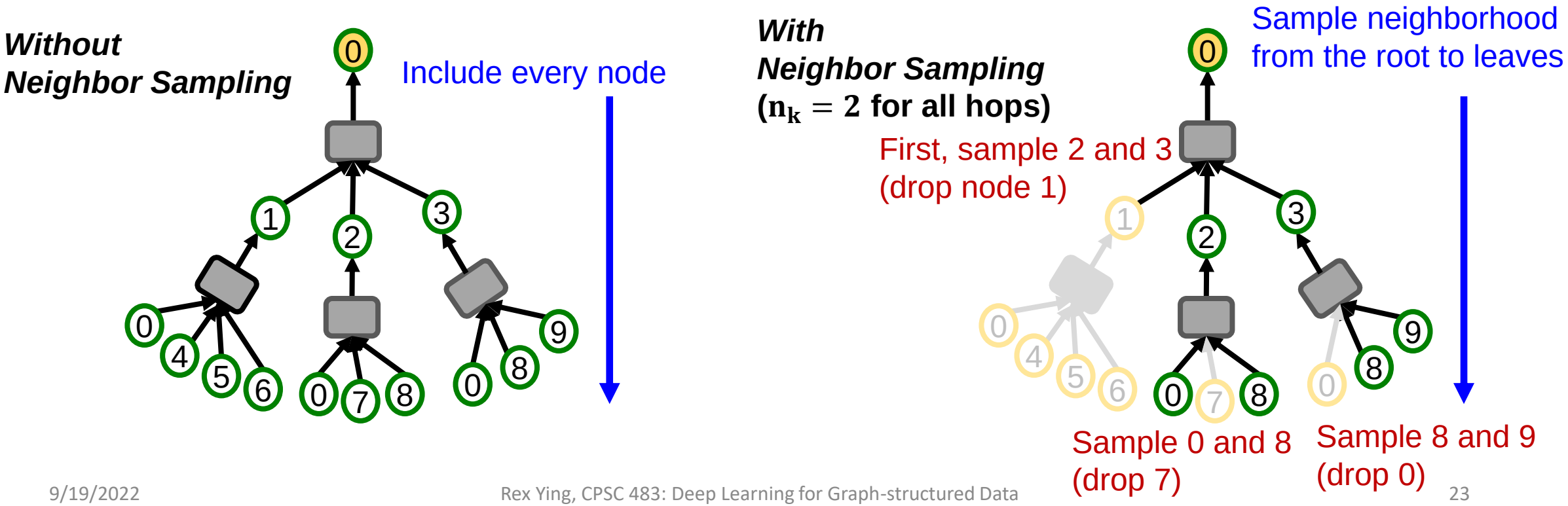


Neighbor Sampling

Key idea

Construct the computational graph by (randomly) **sampling <u>at most</u>** n_k neighbors at each hop.

Pruned computational graph → compute node embeddings **more efficiently**.



Neighbor Sampling Algorithm

Neighbor sampling for K-layer GNN

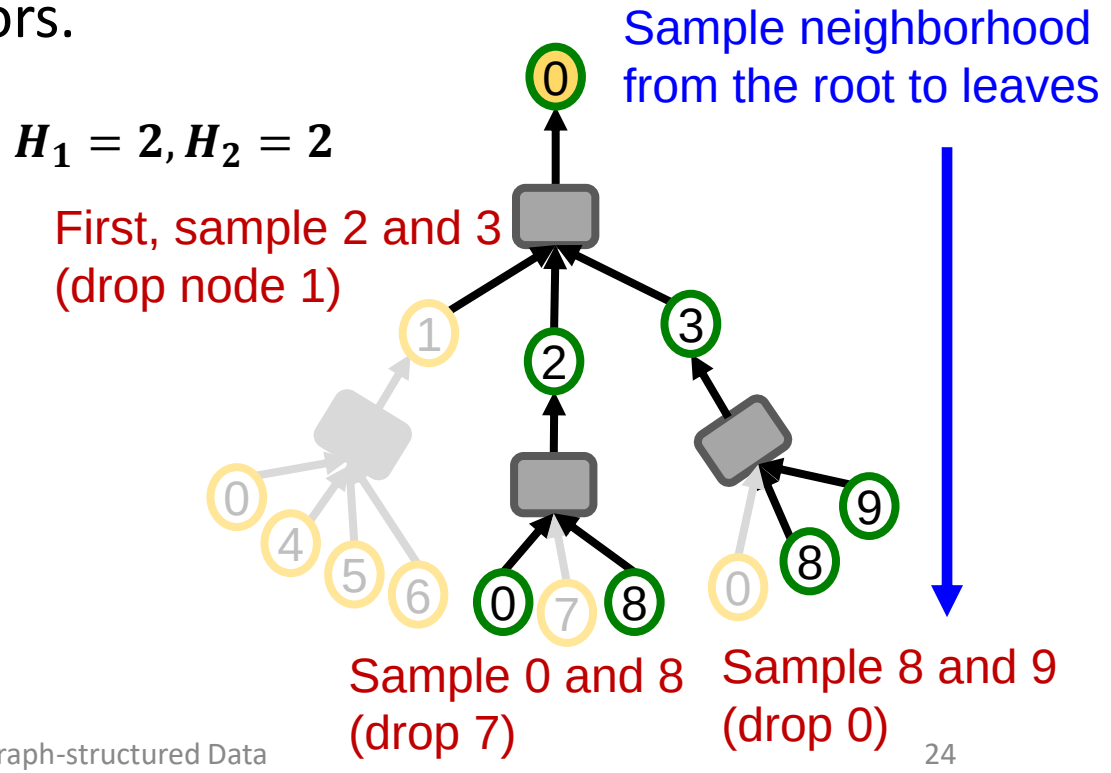
For k = 1, 2, …, K:

- For each node in k-hop neighborhood:
  - (Randomly) sample at most n_k neighbors.

K-layer GNN will at most involve

$\prod_{k=1}^{K} n_k$ leaf nodes

in the computational graph.

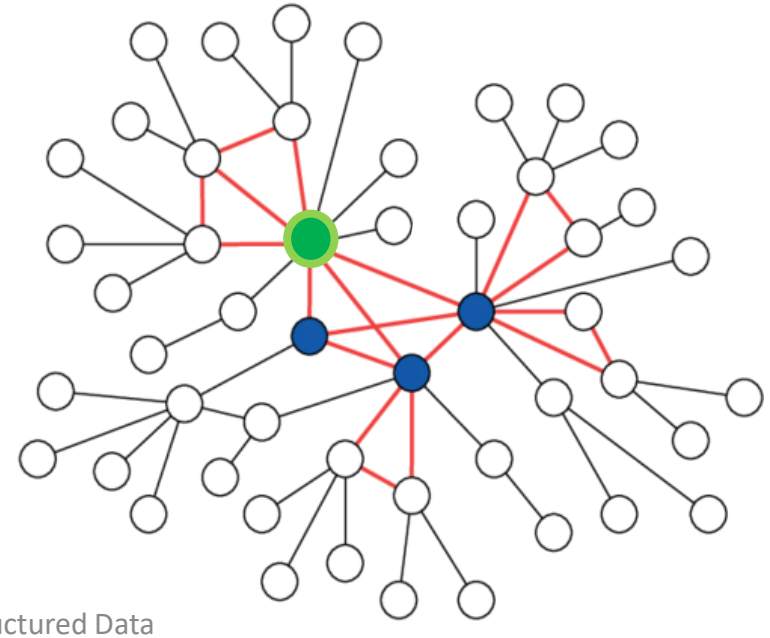


Remarks on Neighbor Sampling

- **Remark 1: Trade-off in sampling number n_k**
  - Smaller n_k 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.
- **Remark 3: How to sample the nodes**
  - (See next slide)

Remark 3: How to sample the nodes

- **Random sampling:** fast but many times not optimal (may sample "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 n_k neighbors i with the highest R_i scores
  - Often more effective



Summary: Neighbor Sampling

- A computational graph is constructed for each node in a mini-batch.
- In neighbor sampling, the computational graph is **pruned/sub-sampled to increase computational efficiency.**
- The pruned computational graph is used to generate a node embedding.

Problem with Neighbor Sampling

Computational graphs can still become large, especially for GNNs with many message-passing layers.

Outline of Today's Lecture

Three Methods for Scaling up GNNs

1. Neighbor Sampling [Hamilton et al. NeurIPS 2017]

2. Cluster-GCN [Chiang et al. KDD 2019]

3. Simplified GCN [Wu et al. ICML 2019]

Small subgraphs in each mini-batch; only load subgraphs on GPU at a time.

Simplify a GNN into a **feature-processing operation**.

Cluster GCN

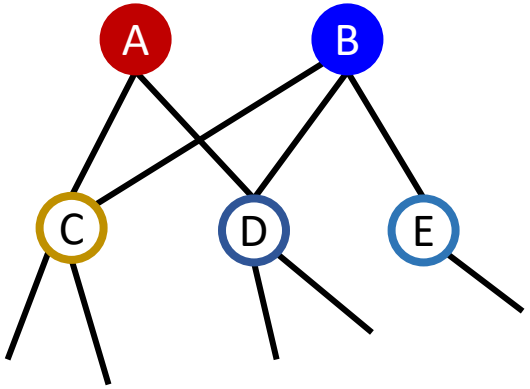
WL Chiang, X Liu, S Si, Y Li, S Bengio and CJ Hsieh. **Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks**. In Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining 2019 Jul 25 (pp. 257-266).

Issues with Neighbor Sampling

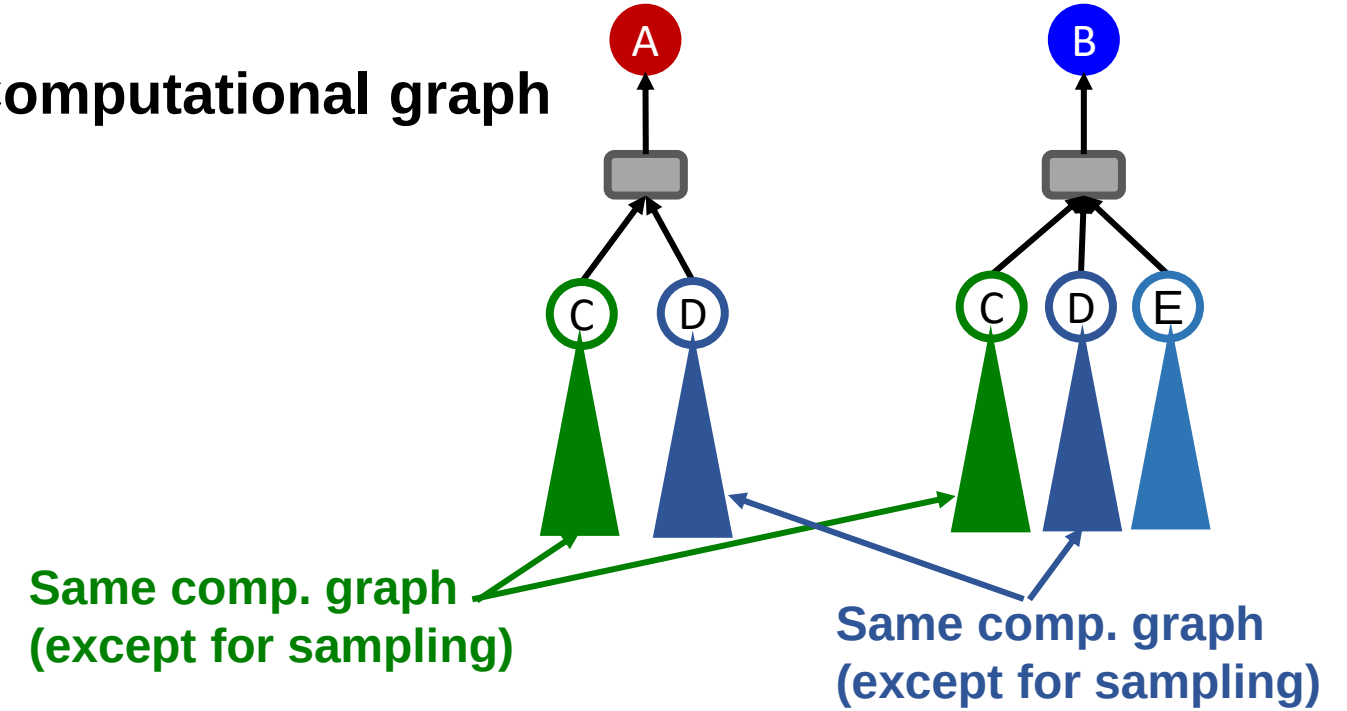
Problems with Neighbor Sampling

- 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.

Input graph



Computational graph



Recall: Full-Batch GNN

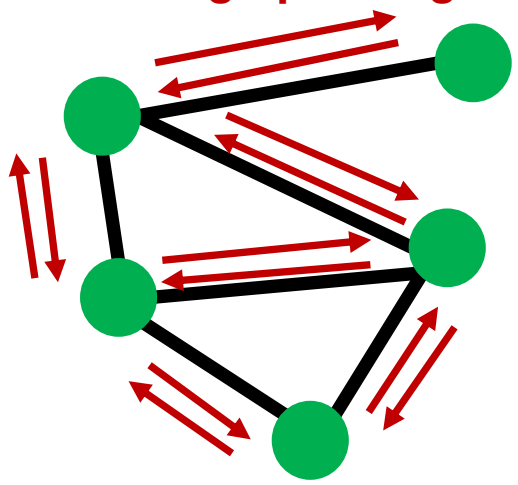
- In full-batch GNN implementation, **all the node embeddings are updated together using embeddings of the previous layer.**

Update for all $v \in V$

Message

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

Message passing



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

Subgraph Sampling (1/3)

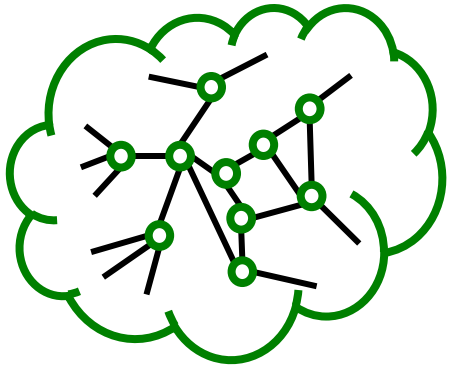
Insight from full-batch GNN

- **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.**

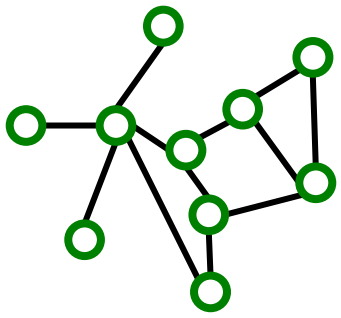
Key idea

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

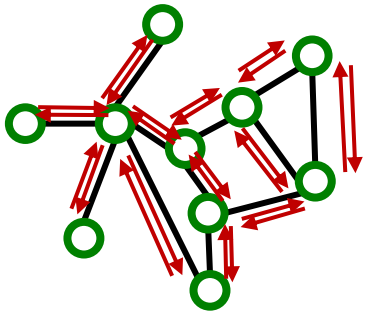
Large graph



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



Layer-wise Node embeddings update on the GPU



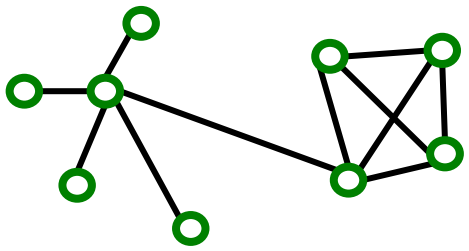
Subgraph Sampling (2/3)

Key question: **What subgraphs are good for training GNNs?**

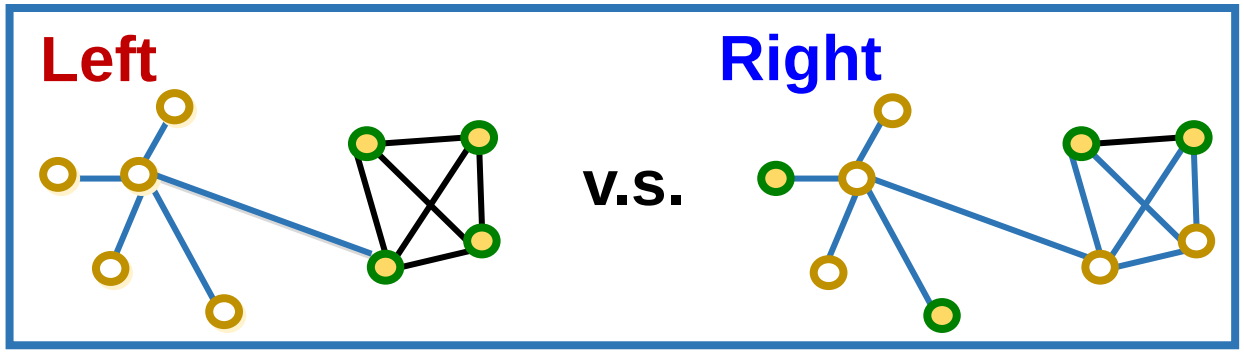
- **Recall: GNN performs node embedding by passing messages via the edges.**
  - Subgraphs **should retain edge connectivity structure of the original graph as much as possible.**
  - This way, the GNN over the subgraph generates embeddings closer to the GNN over the original graph.

Case Study: Which subgraph is good for training GNN?

Original graph



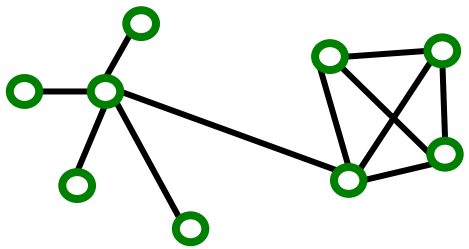
Subgraphs (both 4-node induced subgraph)



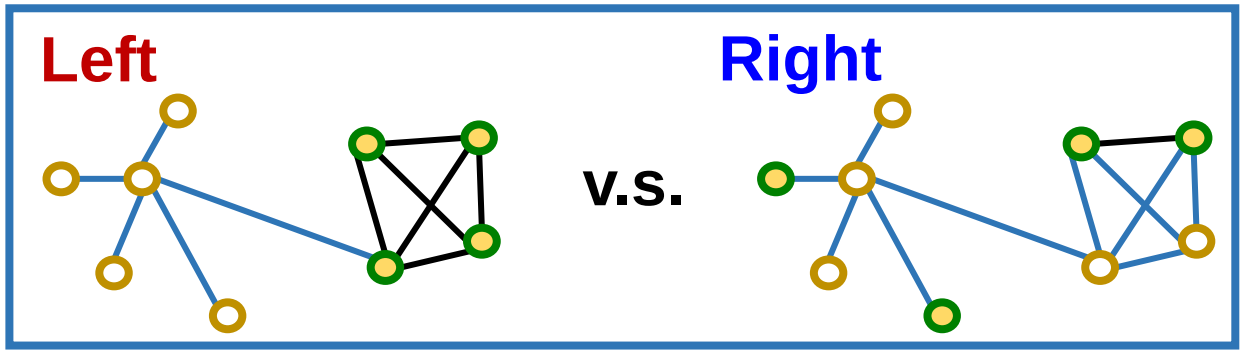
Subgraph Sampling (3/3)

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

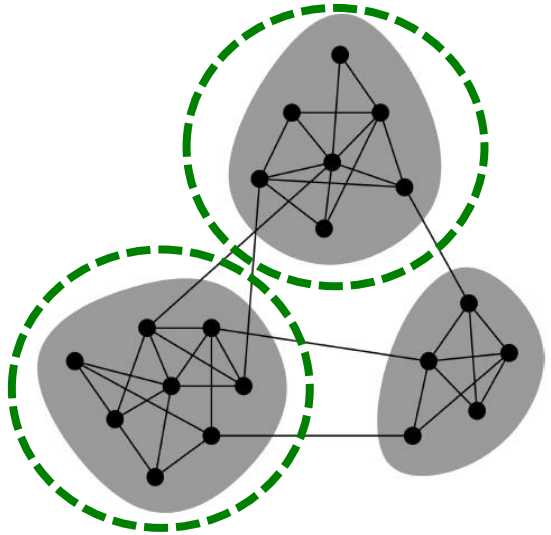
Real-world graph exhibits community structure

- A large graph can be decomposed into many small communities.

Key Insight [Cluster GCN, KDD 2019]

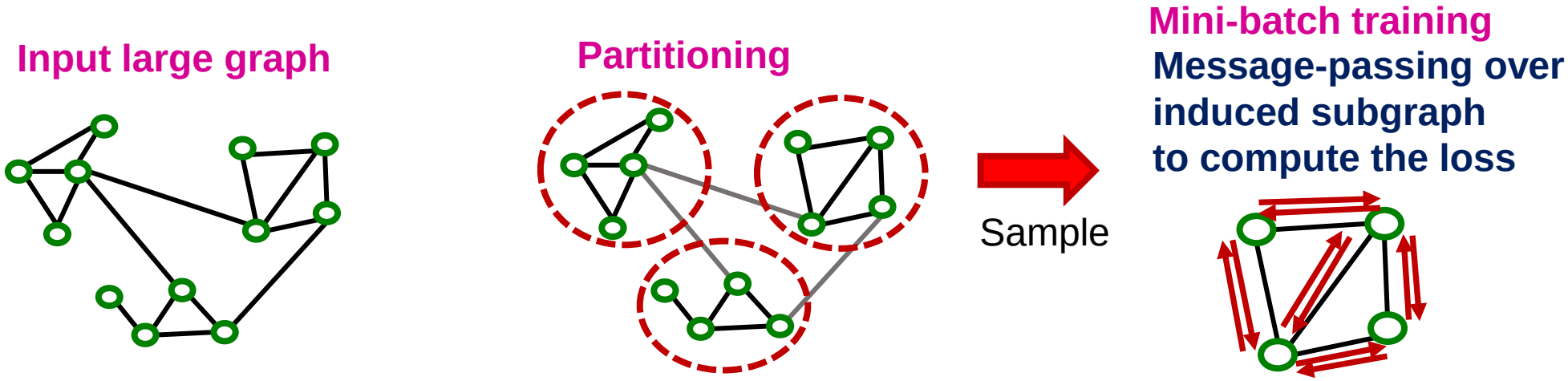
Sample a community as a subgraph.

Each subgraph retains essential local connectivity pattern of the original graph.



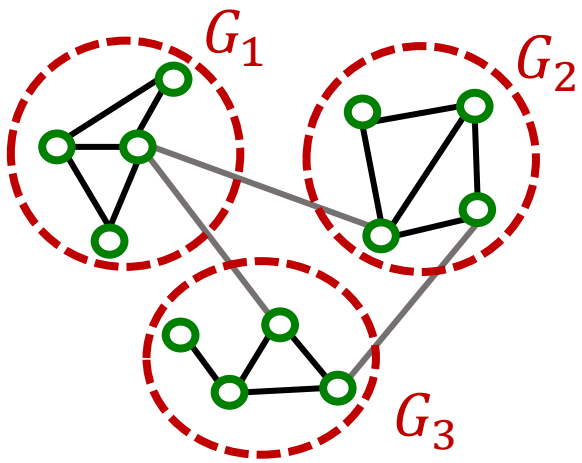
Cluster-GCN: Overview

- **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: Preprocessing

- 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 $E_c = \{(u, v) | u, v \in V_c\}$



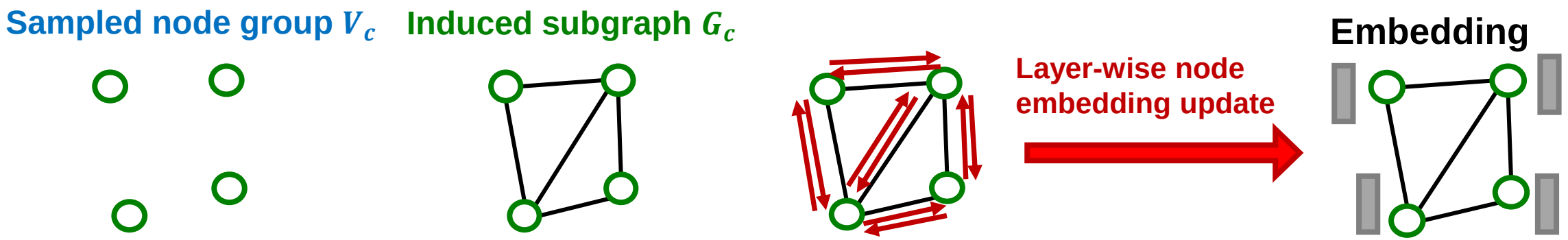
Notice: Between-group edges are *not* included in $G_1, \dots, G_C$.

Cluster-GCN: Mini-batch Training

- For each mini-batch, **randomly sample a node group V_c.**
- Construct **induced subgraph $G_c = (V_c, E_c)$**.
- Apply GNN's **layer-wise node update** over G_c to obtain embedding h_v for each node $v \in V_c$.
- Compute the loss for each node $v \in V_c$ and take average:

$$\ell_{sub}(\boldsymbol{\theta}) = (1/|V_c|) \cdot \sum_{v \in V_c} \mathcal{L}_v(\boldsymbol{\theta})$$

- Update params: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$

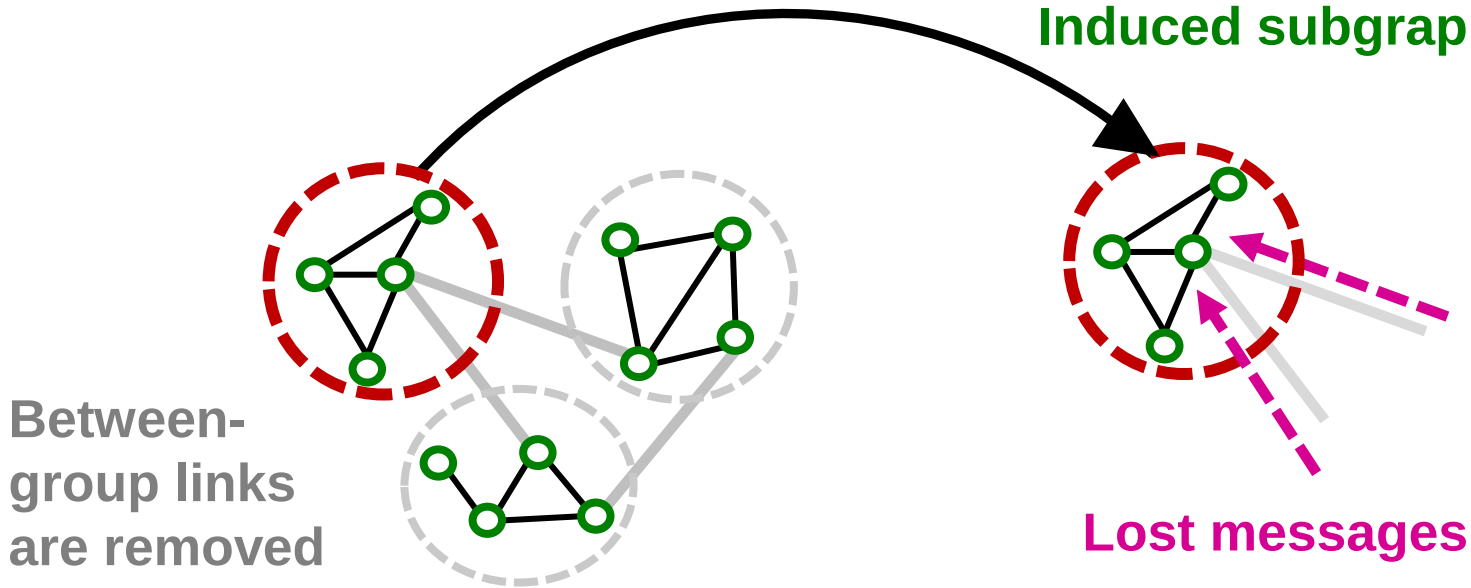


Issues with Cluster-GCN (1/2)

Problems with Cluster-GCN

1. Loss of between-group links

- **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.**



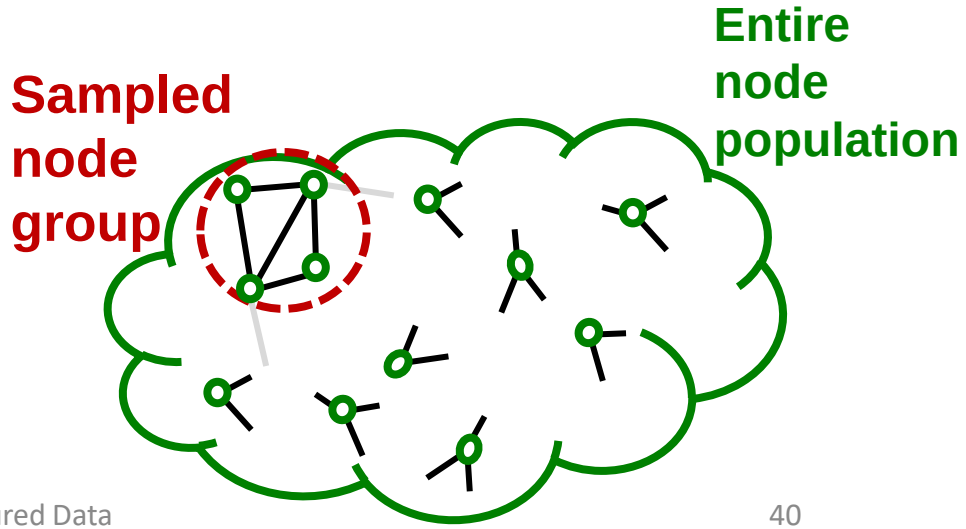
Issues with Cluster-GCN (2/2)

Problems with Cluster-GCN

2. Sampling bias

- 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 node population**.
- **Sampled nodes are not diverse enough to represent the entire graph structure.**
  - The gradient averaged over the sampled nodes, $\frac{1}{|V_c|}\sum_{v\in V_c}\nabla\mathcal{L}_v(\boldsymbol{\theta})$, 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 (1)

Key Idea

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)

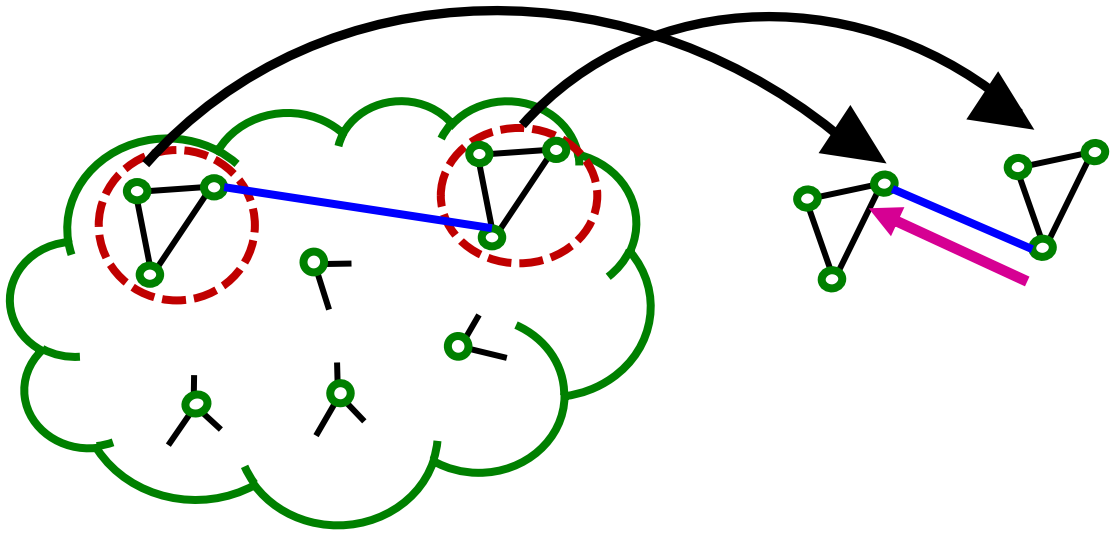
Advanced Cluster-GCN: Overview (2)

Why does this solution work?

Sampling **multiple node groups**

→Makes the sampled nodes more representative of the entire node population.

→Leads to less variance in gradient estimation.



The induced subgraph over aggregated node groups

→ Includes between-group edges

→ Message can flow across groups.

Improved Cluster-GCN

Similar to vanilla Cluster-GCN, improved Cluster-GCN also follows a 2-step approach.

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.

Mini-batch training

- For each mini-batch, **randomly sample a set of q node groups:** $\{V_{t_1}, \dots, V_{t_q}\} \subset \{V_1, \dots, V_C\}$.
- **Obtain all nodes in the sampled node groups**: $V_{aggr} = V_{t_1} \cup \cdots \cup V_{t_q}$
- Extract the **induced subgraph** $\boldsymbol{G_{aggr} = (V_{aggr}, E_{aggr})}$,

  where $\boldsymbol{E_{aggr} = \{(u, v) \mid u, v \in V_{aggr}\}}$
  - $\boldsymbol{E_{aggr}}$ **also includes between-group edges!**

Comparison of Time Complexity

Generate M $(<< N)$ node embeddings using K-layer GNN (N : #all nodes).

Neighbor-sampling (sampling n nodes per layer):

- For each node, the size of K-layer computational graph is n^K.
- For M nodes, the cost is $\boldsymbol{M \cdot n^K}$

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 $\boldsymbol{K \cdot M \cdot D_{avg}}$.

Comparison: Assume $H = D_{avg}/2$. In other words, 50% of neighbors are sampled.

- **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)
  - It also imposes the assumption that the graph has clustering structure

Outline of Today's Lecture

Three Methods for Scaling up GNNs

1. Neighbor Sampling [Hamilton et al. NeurIPS 2017]

2. Cluster-GCN [Chiang et al. KDD 2019]

Small subgraphs in each mini-batch; only load subgraphs on GPU at a time.

3. Simplified GCN [Wu et al. ICML 2019] → Simplify a GNN into a **feature-processing operation**.

Simplified GCN

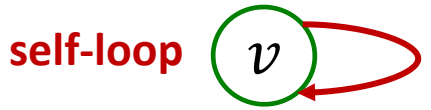
F Wu, A Souza, T Zhang, C Fifty, T Yu and K Weinberger. **Simplifying graph convolutional networks**. In International conference on machine learning 2019 May 24 (pp. 6861-6871). PMLR.

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 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 $\boldsymbol{E}$ **includes the self-loop**:
  - $(\boldsymbol{v}, \boldsymbol{v}) \in \boldsymbol{E}$ **for all** $\boldsymbol{v} \in \boldsymbol{V}$.



- Set input node embeddings: $h_v^{(0)} = X_v$ for $v \in V$.
- For $k \in \{0, \ldots, K-1\}$:
  - For all $v \in V$, aggregate neighboring information as

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

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

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

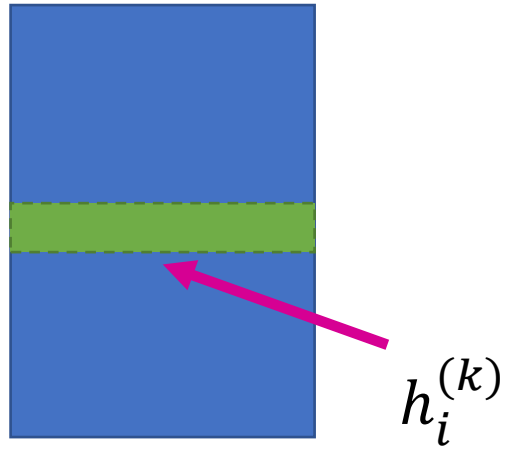
Recall: Matrix formulation of GCN (1)

GCN aggregations can be formulated as matrix vector product:

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

$$\frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)} \rightarrow \boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{H}^{(k)}$$

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



Recall: Matrix formulation of GCN (2)

- **GCN's neighbor aggregation:**

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

- **In matrix form**:

$$\boldsymbol{H}^{(k+1)} = \mathrm{ReLU}\left(\widetilde{\boldsymbol{A}} \boldsymbol{H}^{(k)} \boldsymbol{W}_{\boldsymbol{k}}^{\mathrm{T}}\right)$$

where $\widetilde{\boldsymbol{A}} = \boldsymbol{D}^{-1}\boldsymbol{A}$.

- **Note:** The original GCN uses re-normalized version: $\widetilde{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$.
  - Empirically, this version of $\widetilde{\boldsymbol{A}}$ often gives better performance than $\boldsymbol{D}^{-1}\boldsymbol{A}$.

Simplifying GCN (1/3)

- **Simplify GCN by removing ReLU non-linearity:**
  (recall that $\tilde{A}$ is the normalized adjacency in Lecture 4)

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

- The final node embedding matrix is given as

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

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

$$= \tilde{A}\left(\tilde{A}\left(\cdots\left(\tilde{A}\, H^{(0)}\, W_0^{\mathrm{T}}\right)\cdots\right) W_{K-2}^{\mathrm{T}}\right) W_{K-1}^{\mathrm{T}}$$

$$= \tilde{A}^K\, X\, \left(W_0^{\mathrm{T}} \cdots W_{K-1}^{\mathrm{T}}\right)$$

Composition of linear transformation is still linear!

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

Simplifying GCN (2/3)

- Removing ReLU significantly simplifies GCN!

$$\boldsymbol{H}^{(K)} = \widetilde{\boldsymbol{A}}^{K}\, \boldsymbol{X}\, \boldsymbol{W}^{\mathrm{T}}$$

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

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

$$\boldsymbol{H}^{(K)} = \widetilde{\boldsymbol{X}}\, \boldsymbol{W}^{\mathrm{T}}$$

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

$$h_v^{(K)} = \boldsymbol{W}\, \widetilde{\boldsymbol{X}}_v$$

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

Simplifying GCN (3/3)

- Node embedding form:

$$h_v^{(K)} = \boldsymbol{W}\,\widetilde{\boldsymbol{X}}_v$$

- Once $\widetilde{\boldsymbol{X}}$ is pre-computed, embeddings of M nodes can be generated in time linear in M:
  - Given M nodes $\{v_1, v_2, \ldots, v_M\}$, their embeddings are
    - $h_{v_1}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_1}$,
    - $h_{v_2}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_2}$,
    - …
    - $h_{v_M}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_M}$.

Simplifying GCN: Summary

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

- **Pre-processing step**
  - Pre-compute $\widetilde{\boldsymbol{X}} = \widetilde{\boldsymbol{A}}^K \boldsymbol{X}$. 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
    - $h_{v_1}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_1}, h_{v_2}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_2}, \dots, h_{v_M}^{(K)} = \boldsymbol{W}\widetilde{\boldsymbol{X}}_{v_M}$
  - 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 **GraphSAGE 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.

Problems with Simplified GCN

- Its 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

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

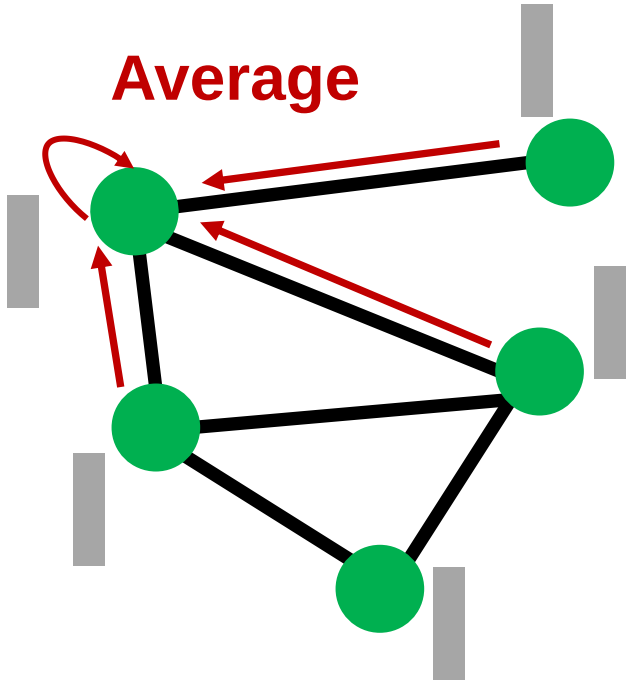
- **Why?**
- Many node classification tasks exhibit **homophily structure**, i.e., **nodes connected by edges tend to share the same target labels.**
- **Examples**:
  - Paper category classification in paper-citation network
    - Two papers tend to share the same category if one cites another.
  - 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? (1)

- 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? (2)

- **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 GCN'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.