

Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://cs224w.Stanford.edu>

Scaling Up GNNs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



Announcements

- **Homework 3** due today
 - Late submissions accepted until end of day Monday, 11/20
- **Colab 5** released on course website
 - Due Tuesday, 12/05
- **Regrade request deadlines**
 - **Homework 2:** Saturday, 11/18
 - Solutions and statistics released on Ed
 - Regrade request policy update on Ed

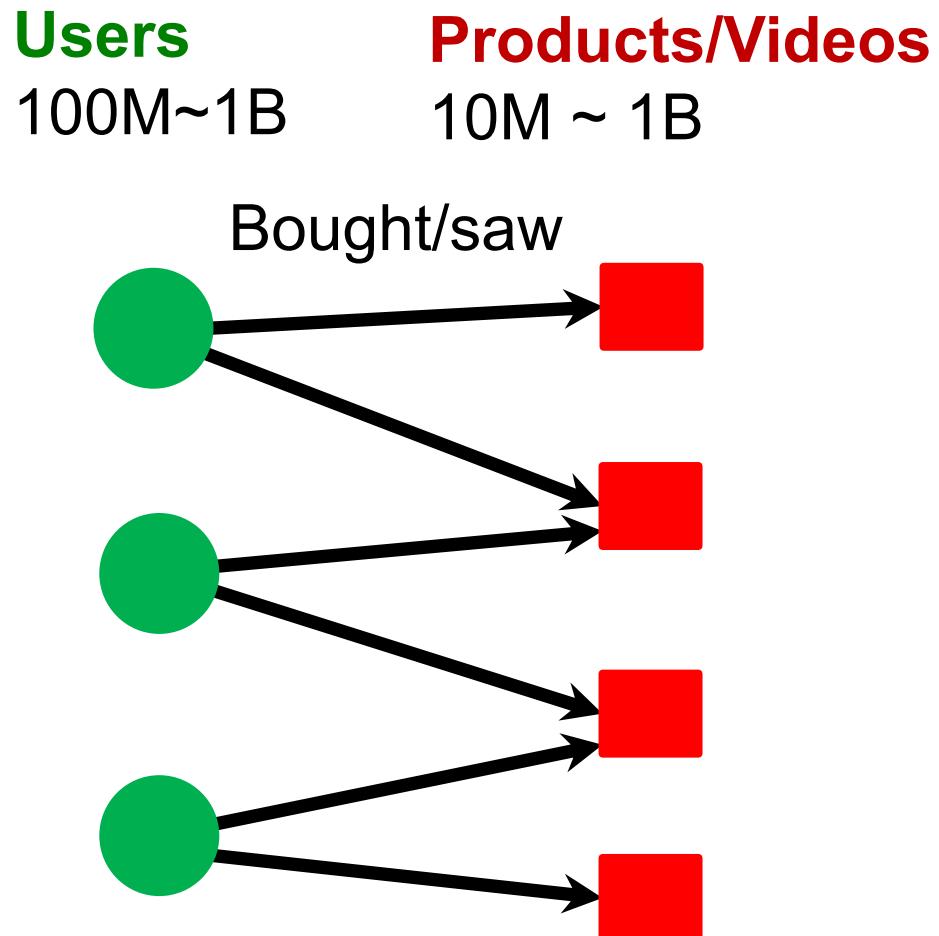
Graphs in Modern Applications

■ Recommender systems:

- Amazon
- YouTube
- Pinterest
- Etc.

■ ML tasks:

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



Graphs in Modern Applications

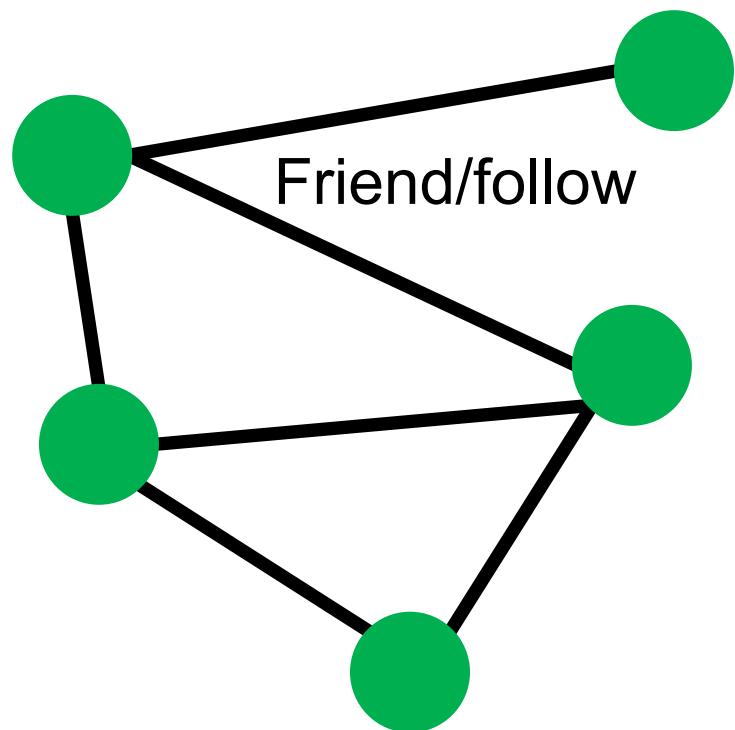
- **Social networks:**

- 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

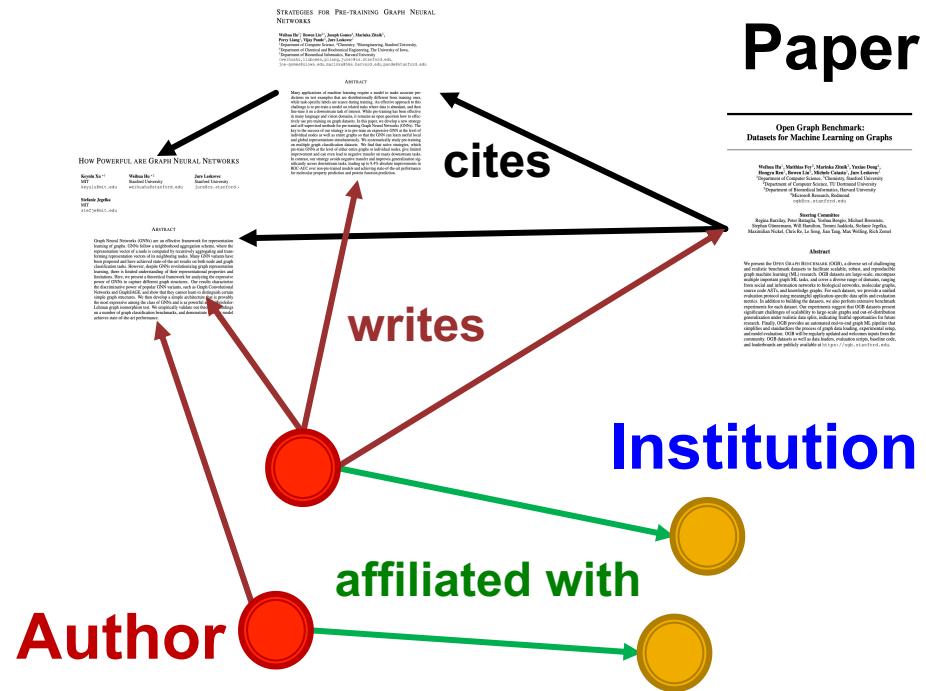
■ ML tasks:

- Paper categorization
(node classification)
- Author collaboration recommendation
- Paper citation recommendation
(link prediction)

Papers
120M

Authors
120M

Paper



Graphs in Modern Applications

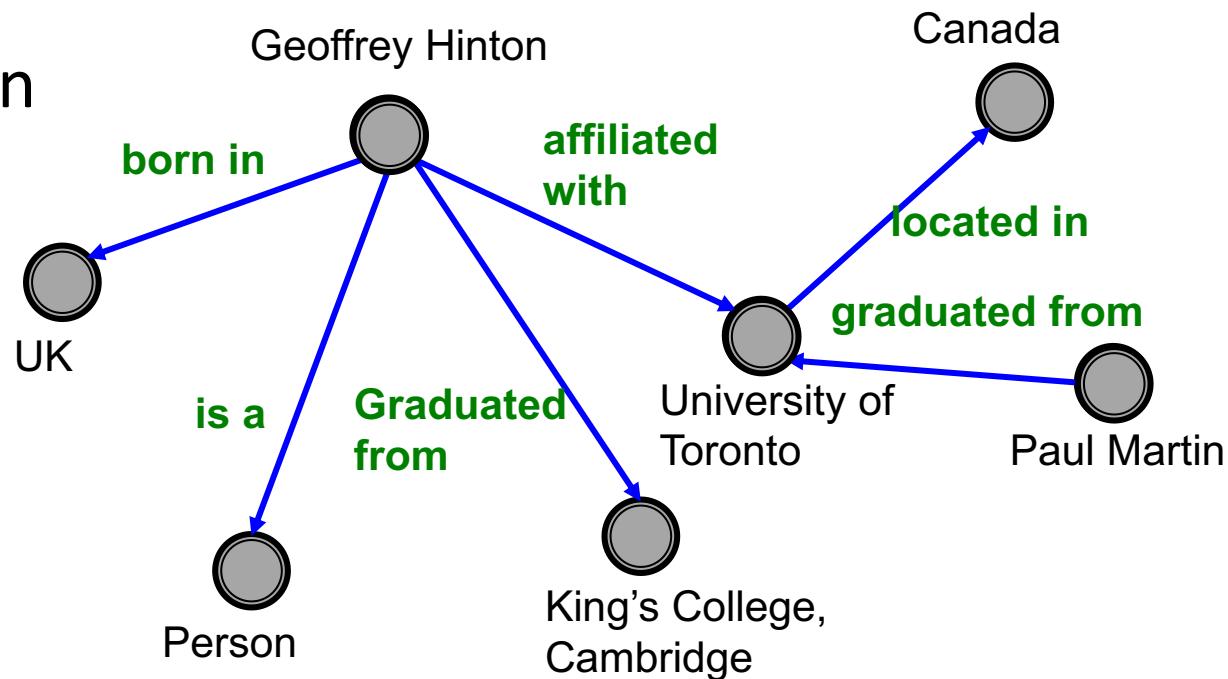
■ Knowledge Graphs (KGs):

- 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

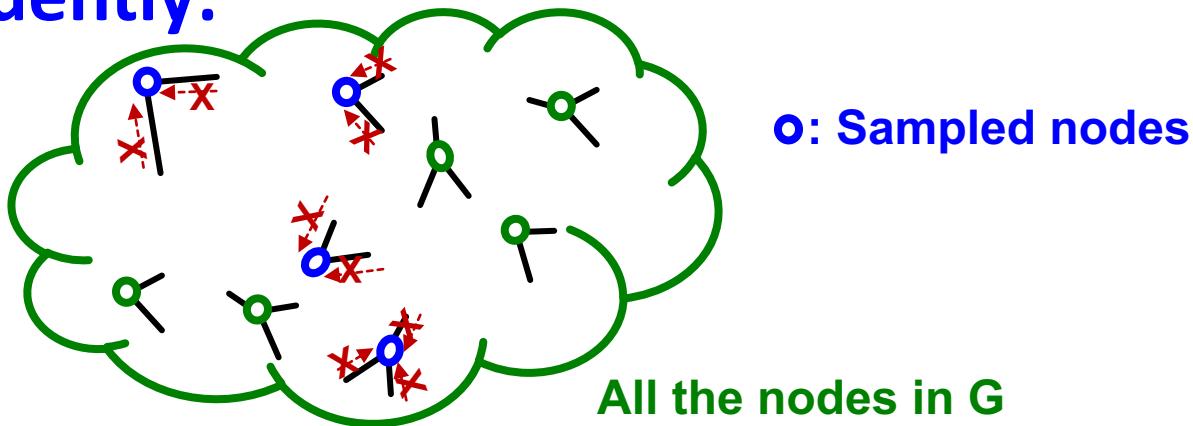
$$\ell(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=0}^{N-1} \ell_i(\boldsymbol{\theta})$$

- $\boldsymbol{\theta}$: model parameters, $\ell_i(\boldsymbol{\theta})$: loss for i -th data point.
- We perform **Stochastic Gradient Descent (SGD)**.
 - Sample M ($\ll N$) data points (**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 were to use the standard SGD for GNN?

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



- Sampled nodes will be isolated from each other!
- GNN generates 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?

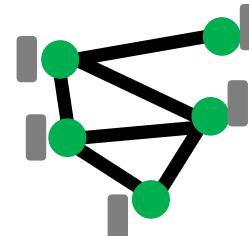
■ Naïve full-batch implementation:

Generate embeddings of all the nodes **at the same time**:

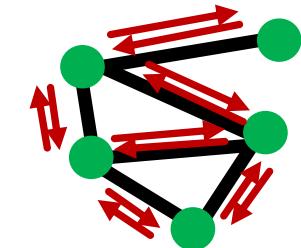
$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)$$

- Load **the entire graph A and features**
- X. Set $H^{(0)} = X$.
- **At each GNN layer:** Compute embeddings of all nodes using all the node embeddings from the previous layer.
- Compute the loss
- Perform gradient descent

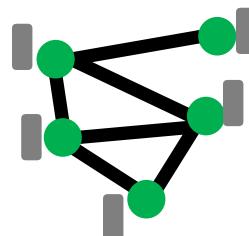
Given all node embeddings at layer K



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 (10GB-80GB).
 - **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

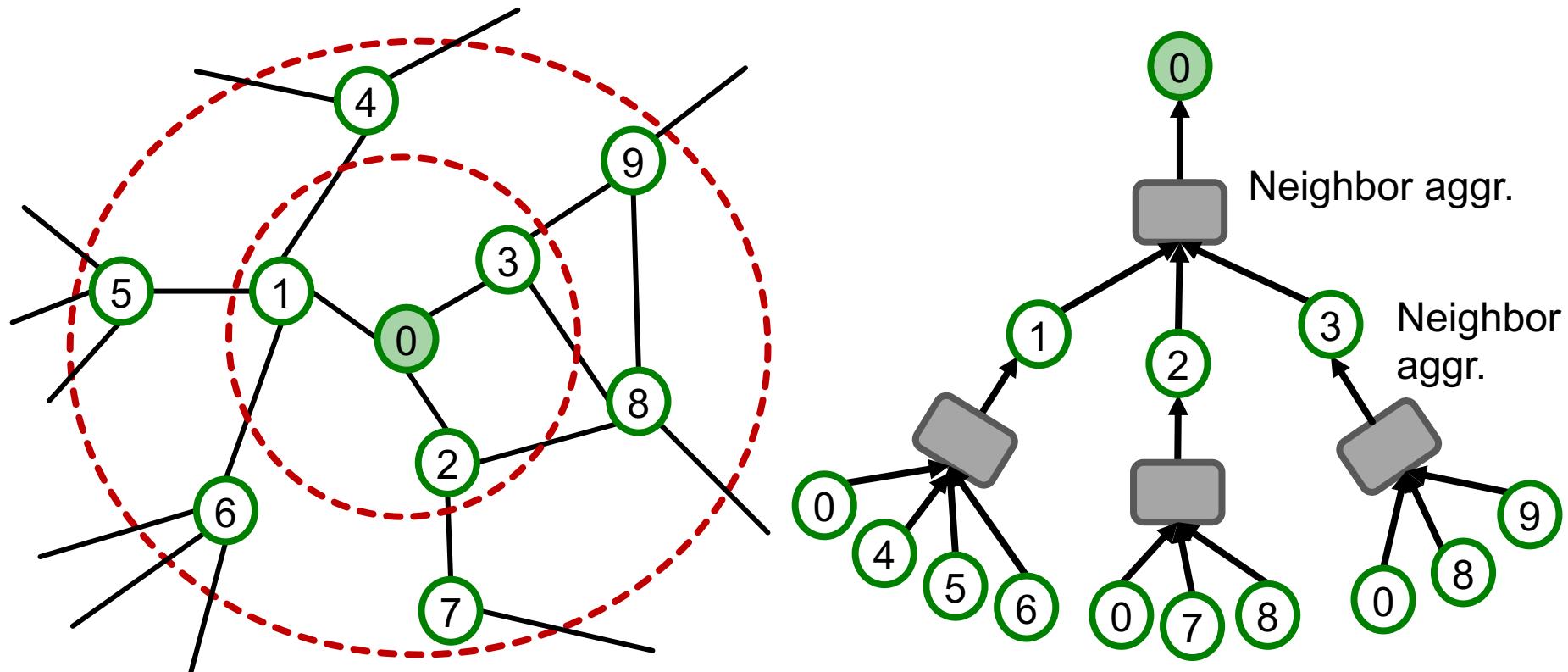
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



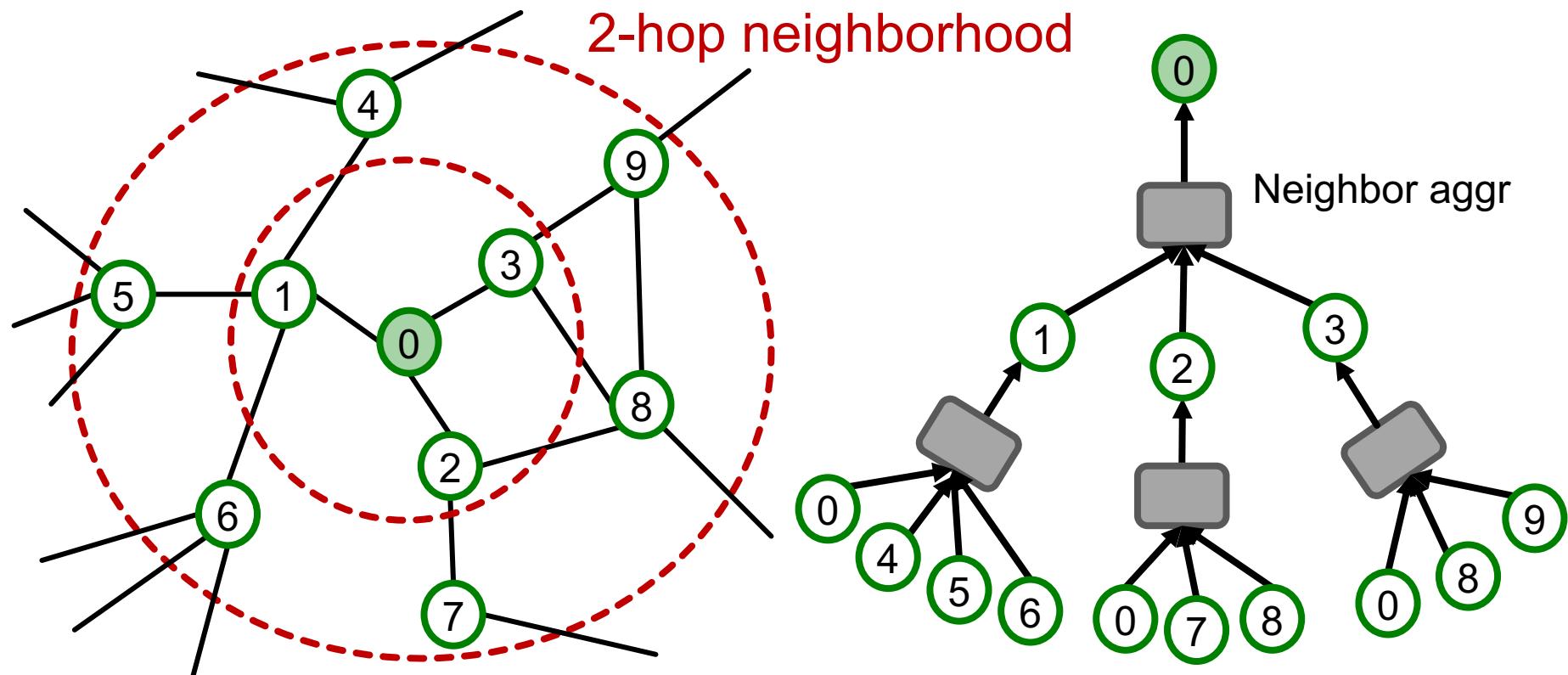
Recall: Computational Graph

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



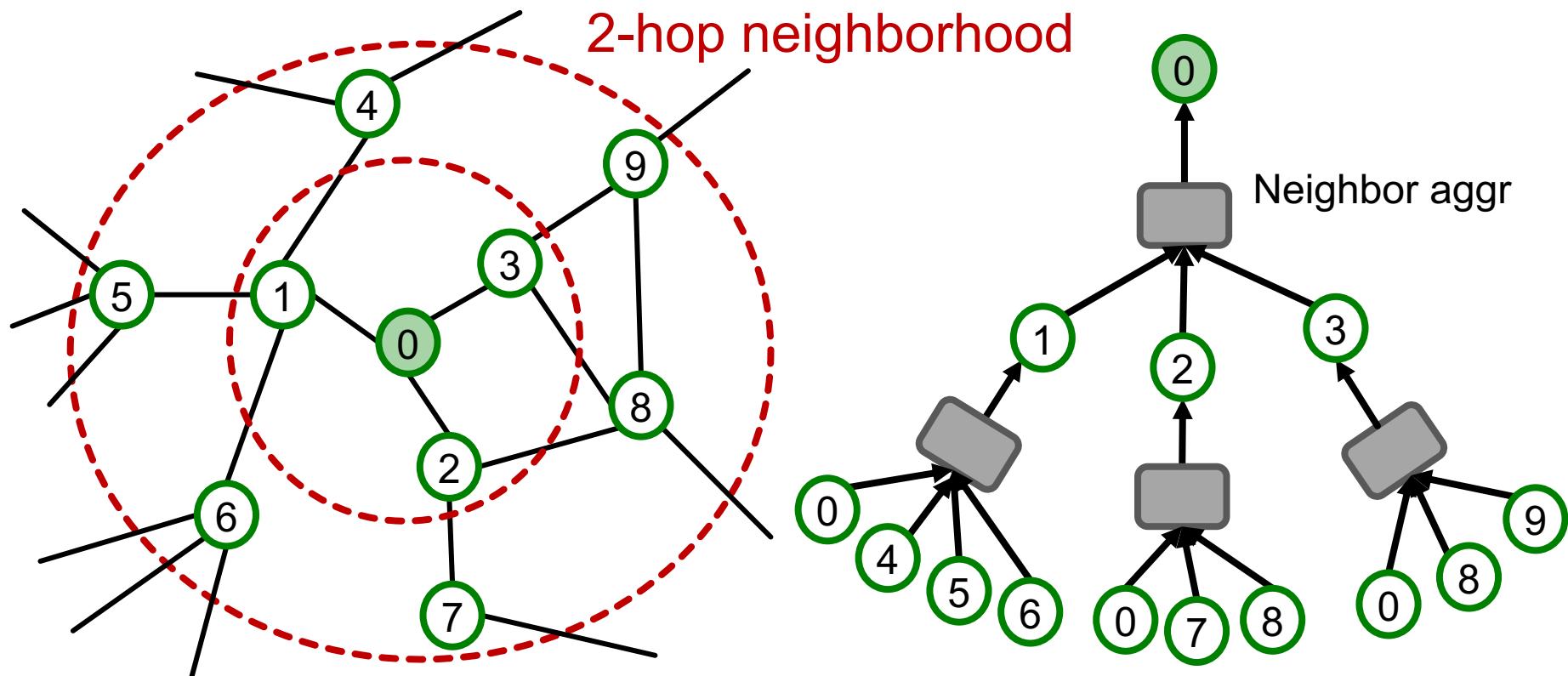
Recall: Computational Graph

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



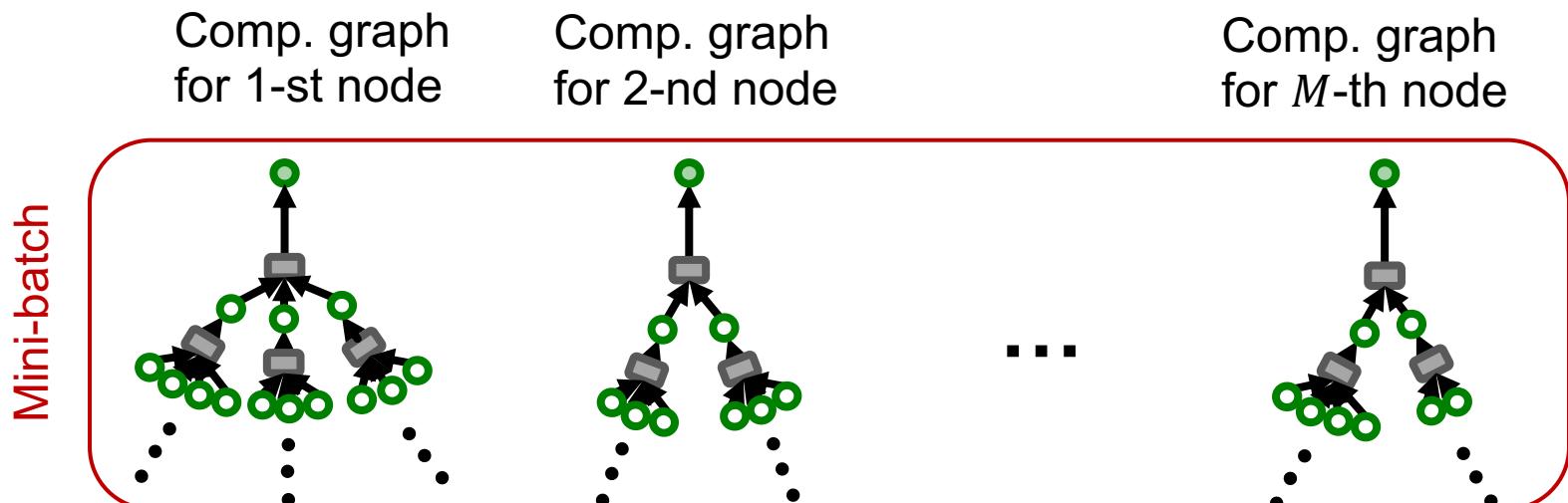
Recall: Computational Graph

- **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!**

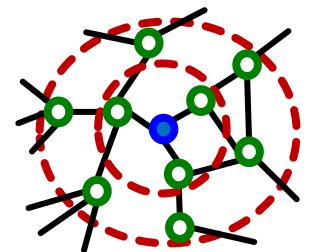


Stochastic Training of GNNs

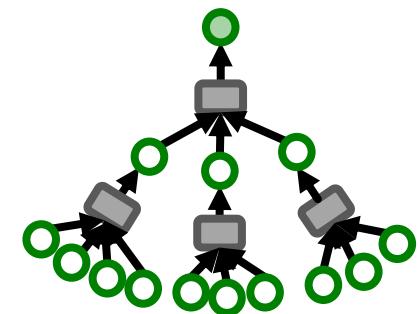
- We can now consider the following SGD strategy for training K -layer GNNs:

- Randomly sample M ($\ll N$) root nodes.
- For each sampled root 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

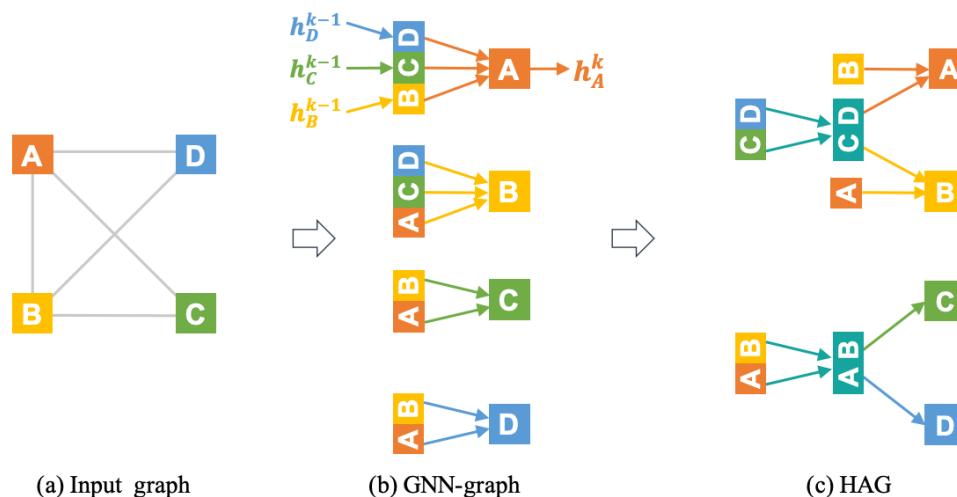


Computational graph



Issue with Stochastic Training (1)

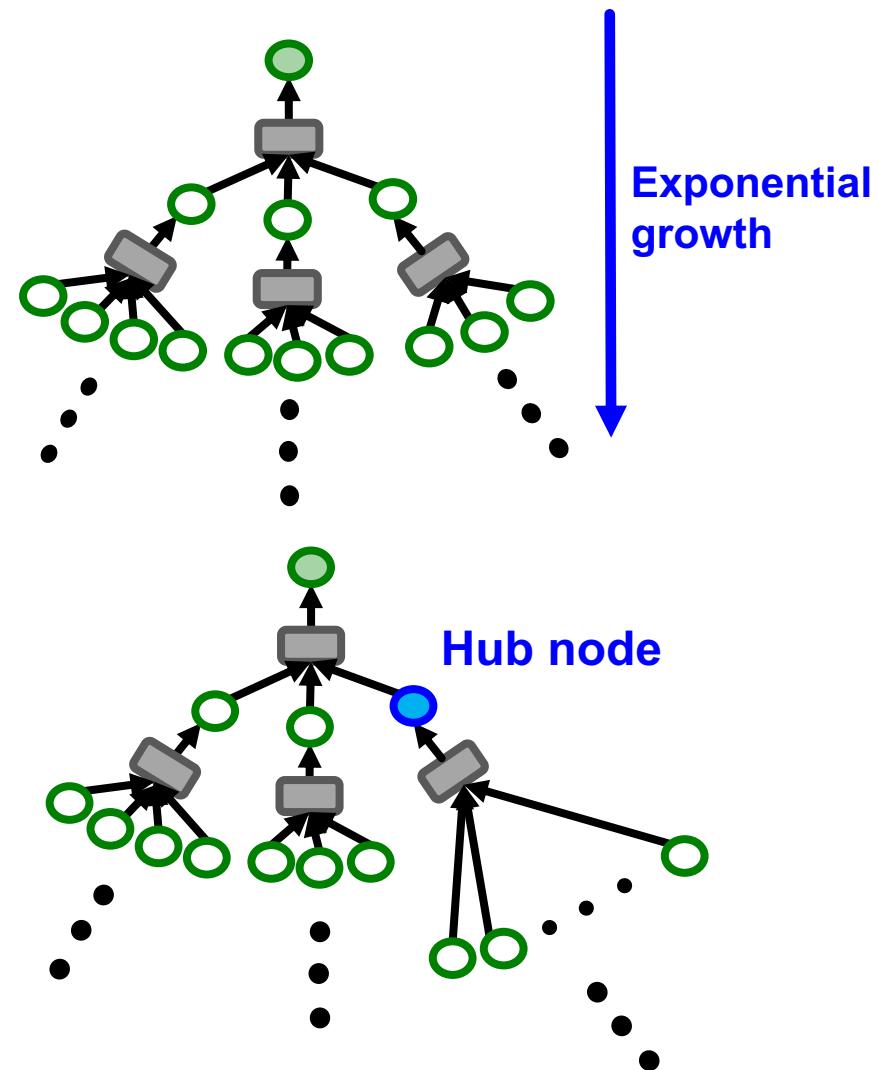
- For each node, we need to get the entire **K-hop neighborhood** and pass it through the computation graph.
- We need to aggregate lot of information just to compute one node embedding.
- **Some computational redundancy:**



Issue with Stochastic Training (2)

■ 2nd issue:

- 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!



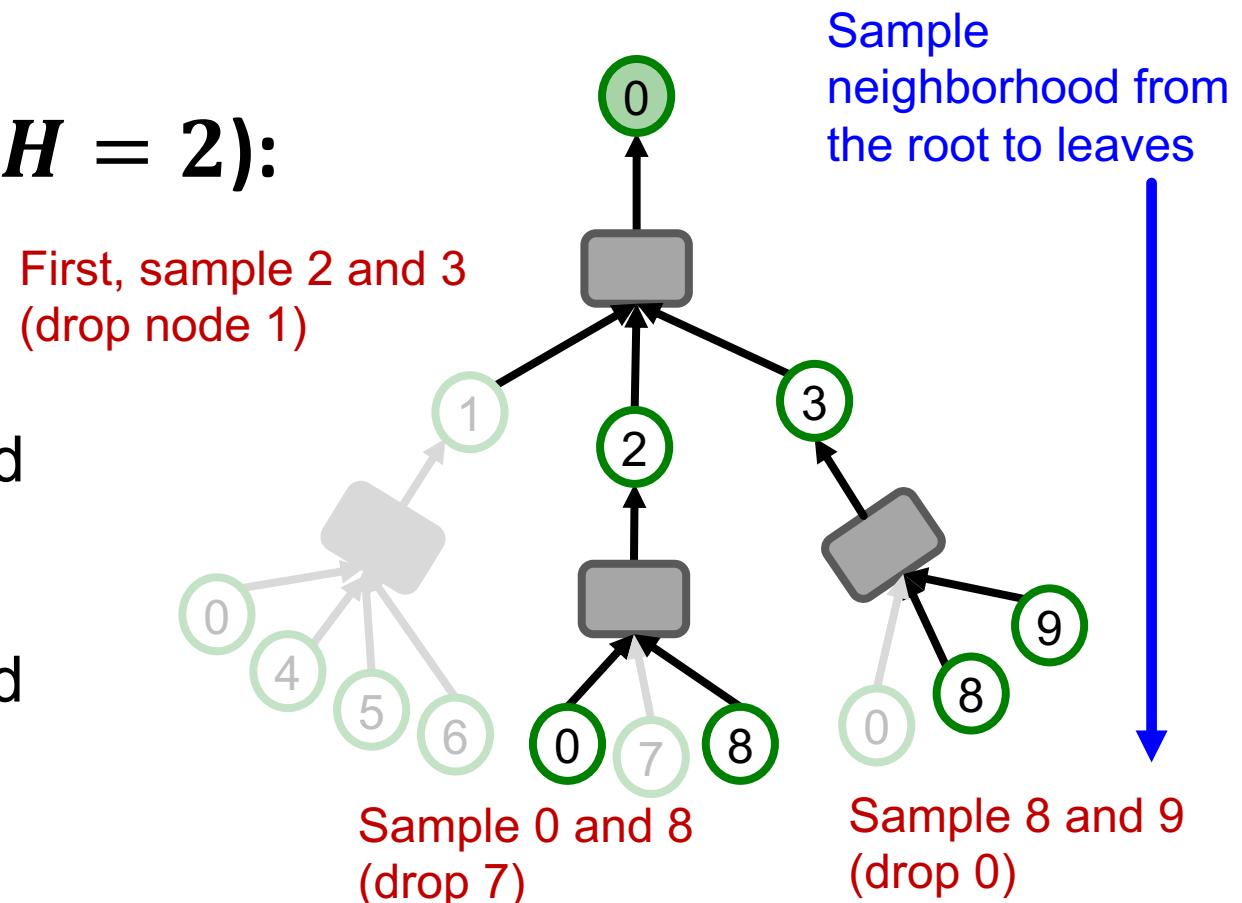
Neighborhood Sampling

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

- Example ($H = 2$):

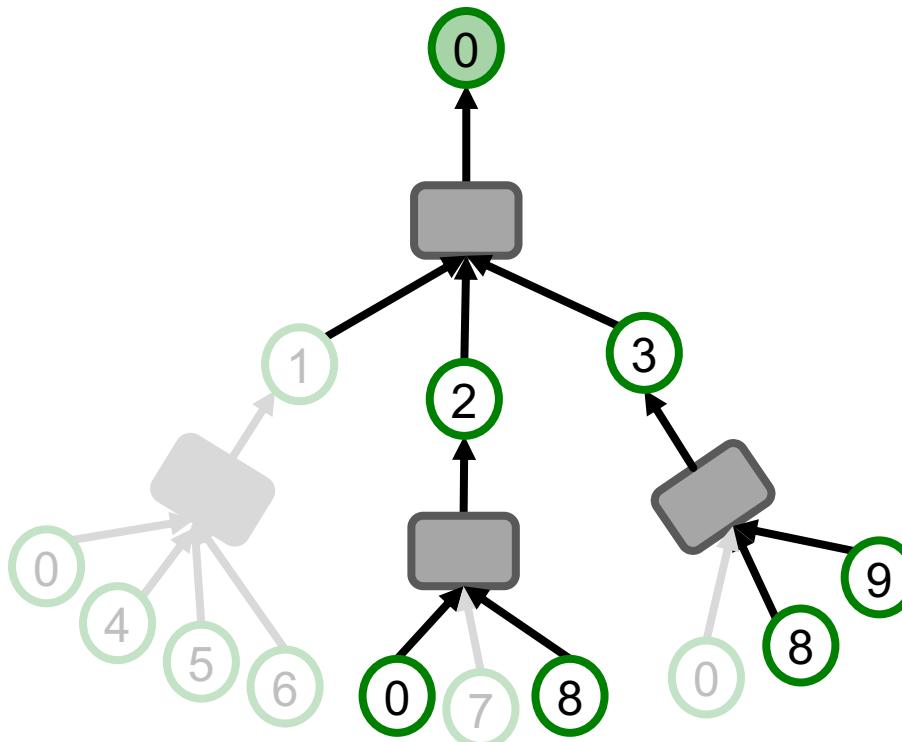
1st-hop neighborhood

2nd-hop neighborhood



Neighborhood Sampling

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



Neighborhood Sampling Algorithm

Neighbor sampling for K -layer GNN

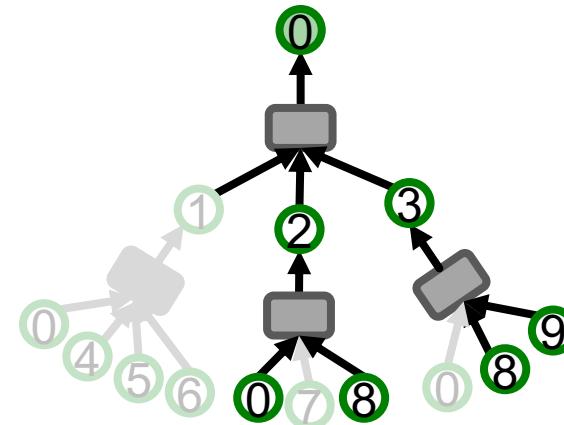
- For $k = 1, 2, \dots, K$:
 - For each node in k -hop neighborhood:
 - (Randomly) sample at most H_k neighbors:

1st-hop neighborhood

Sample $H_1 = 2$ neighbors

2nd-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.

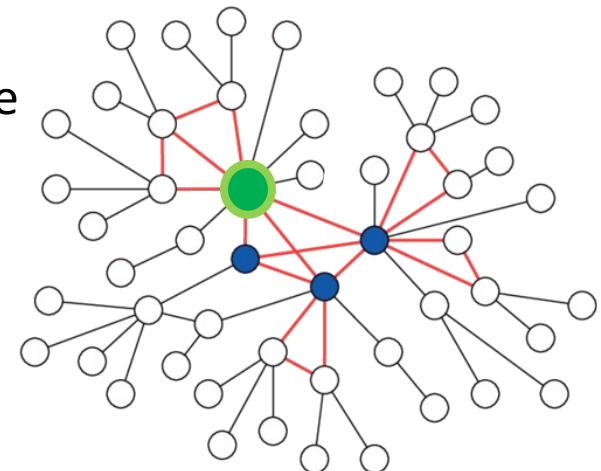
Remarks on Neighbor Sampling (1)

- **Remark 1: Trade-off in sampling number H**
 - Smaller H leads to more efficient neighbor aggregation, but results are less stable 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 .**
 - Adding 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

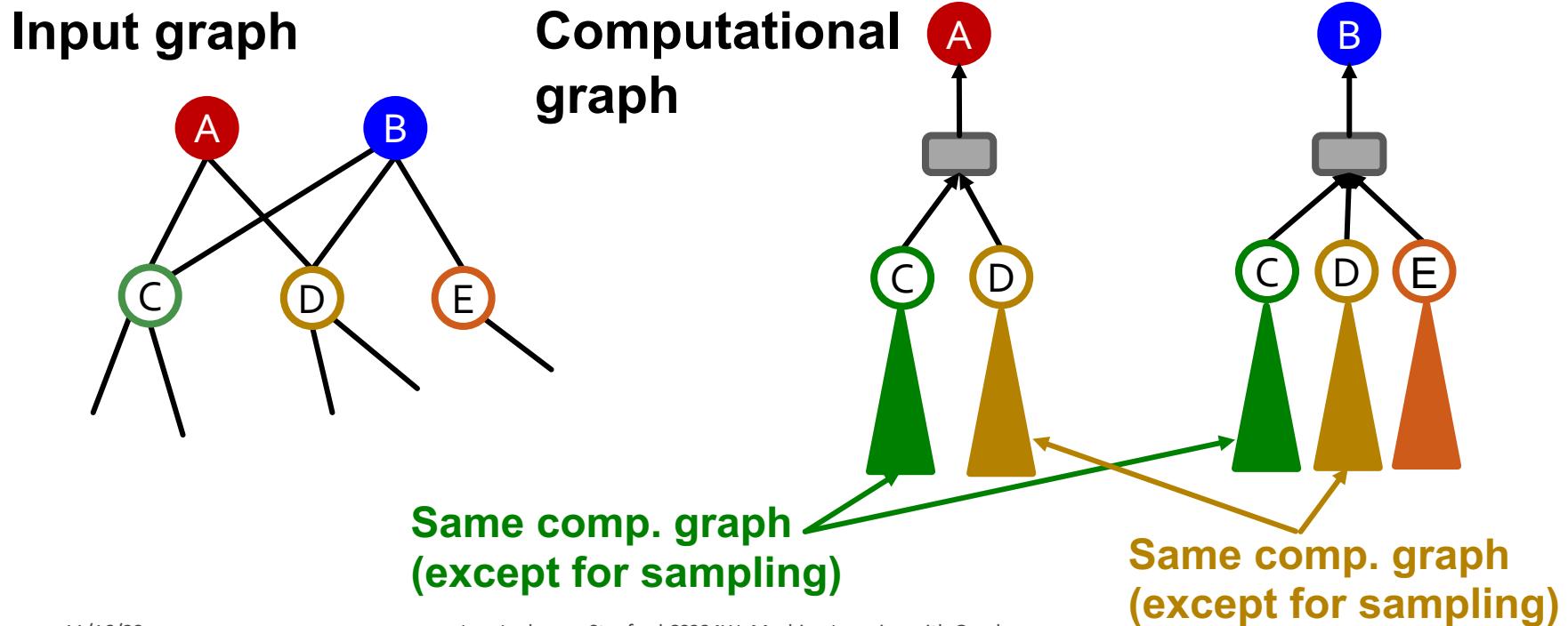
Jure Leskovec, Stanford University

<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.



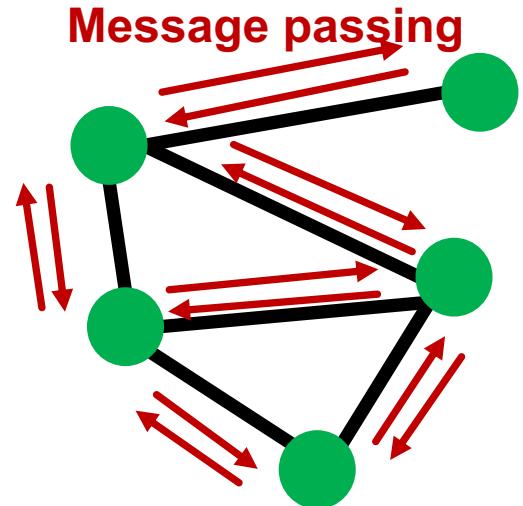
Recall: Full Batch GNN

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

Update for all $v \in V$

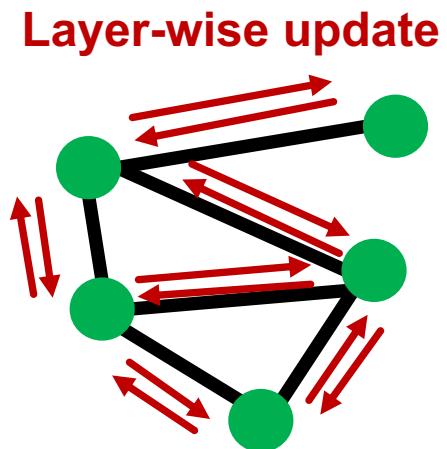
$$h_v^{(\ell)} = COMBINE \left(h_v^{(\ell-1)}, AGGR \left(\left\{ \mathbf{h}_u^{(\ell-1)} \right\}_{u \in N(v)} \right) \right)$$

- 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

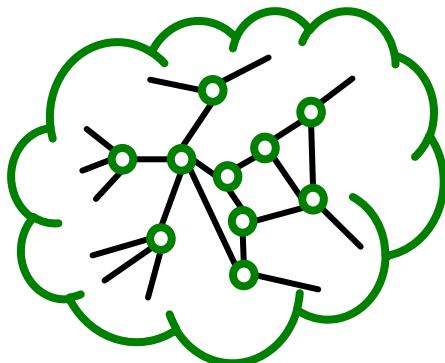
- The **layer-wise** node embedding update allows the re-use of embeddings from the previous layer.
- This significantly **reduces the computational redundancy of neighbor sampling.**
 - Of course, the **layer-wise** update is **not feasible** for a large graph due to **limited GPU memory.**
 - Requires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



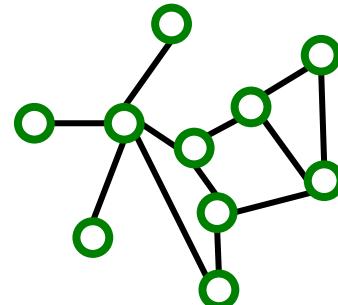
Subgraph Sampling

- **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.

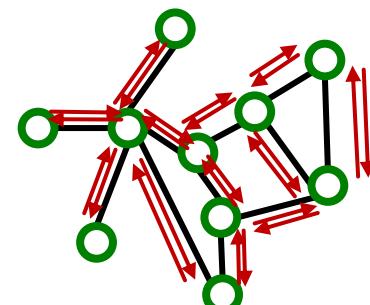
Large graph



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



Layer-wise
node embeddings
update on the GPU



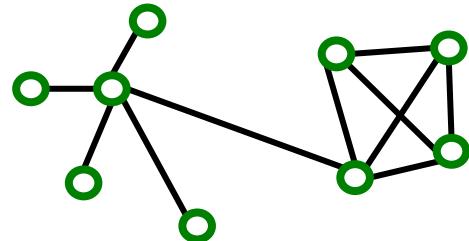
Subgraph Sampling

- **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.

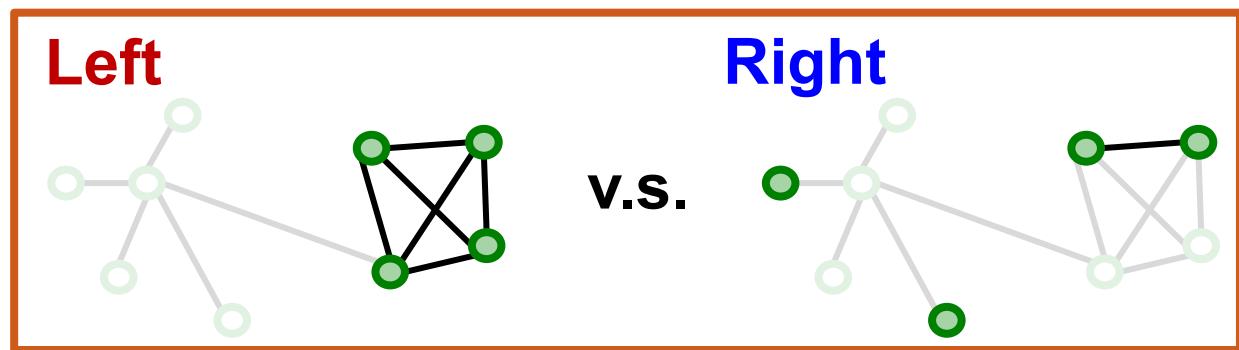
Subgraph Sampling: Case Study

- Which subgraph is good for training GNN?

Original graph



Subgraphs (both 4-node induced subgraph)

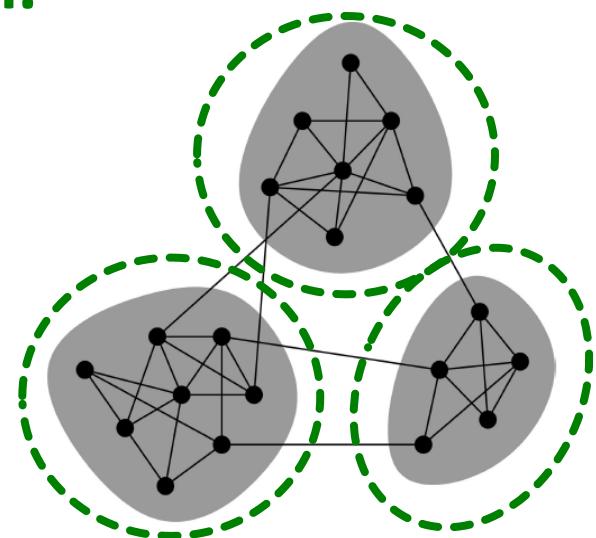


- **Left subgraph** retains the essential community structure among the 4 nodes → **Good**
- **Right subgraph** drops many connectivity patterns, even leading to isolated nodes → **Bad**

Exploiting Community Structure

Real-world graph exhibits community structure

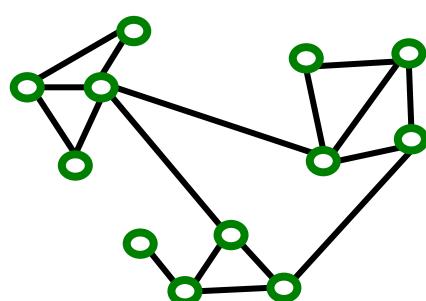
- A large graph can be decomposed into many small communities.
- **Key insight** [Chiang et al. KDD 2019]:
**Sample a community as a subgraph.
Each subgraph retains essential
local connectivity pattern of the
original graph.**



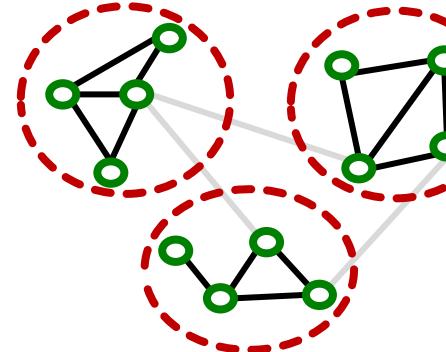
Cluster-GCN: Overview

- 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**.

Input large graph

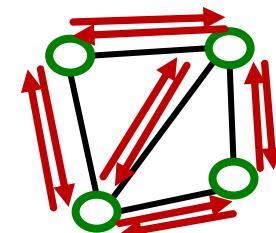


Partitioning



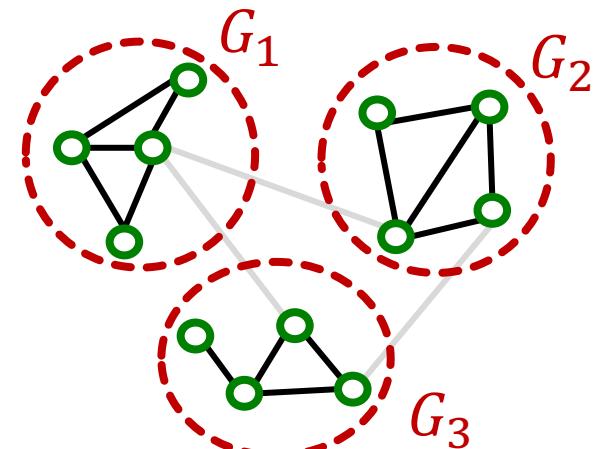
Sample

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



Cluster-GCN: Pre-processing

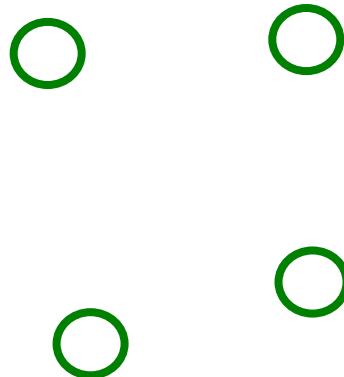
- Given a large graph $G = (V, E)$, **partition its nodes V into C groups:** V_1, \dots, V_C .
 - We can use any scalable community detection methods, e.g., Louvain, METIS [Karypis et al. SIAM 1998].
 - V_1, \dots, V_C induces C subgraphs, G_1, \dots, G_C ,
 - Recall: $G_c \equiv (V_c, E_c)$, where $E_c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



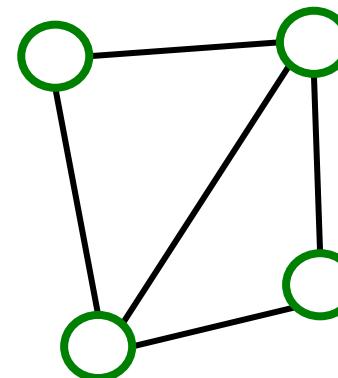
Cluster-GCN: Mini-batch Training

- 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



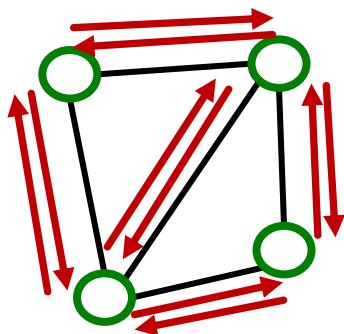
Induced subgraph G_c



Cluster-GCN: Mini-batch Training

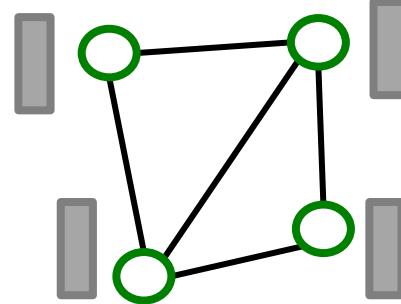
- Apply GNN's **layer-wise node update** over G_c to obtain embedding \mathbf{h}_v for each node $v \in V_c$.
- Compute the loss for each node $v \in V_c$ and take average: $\ell_{sub}(\theta) = (1/|V_c|) \cdot \sum_{v \in V_c} \ell_v(\theta)$
- Update params: $\theta \leftarrow \theta - \nabla \ell_{sub}(\theta)$

Induced subgraph G_c



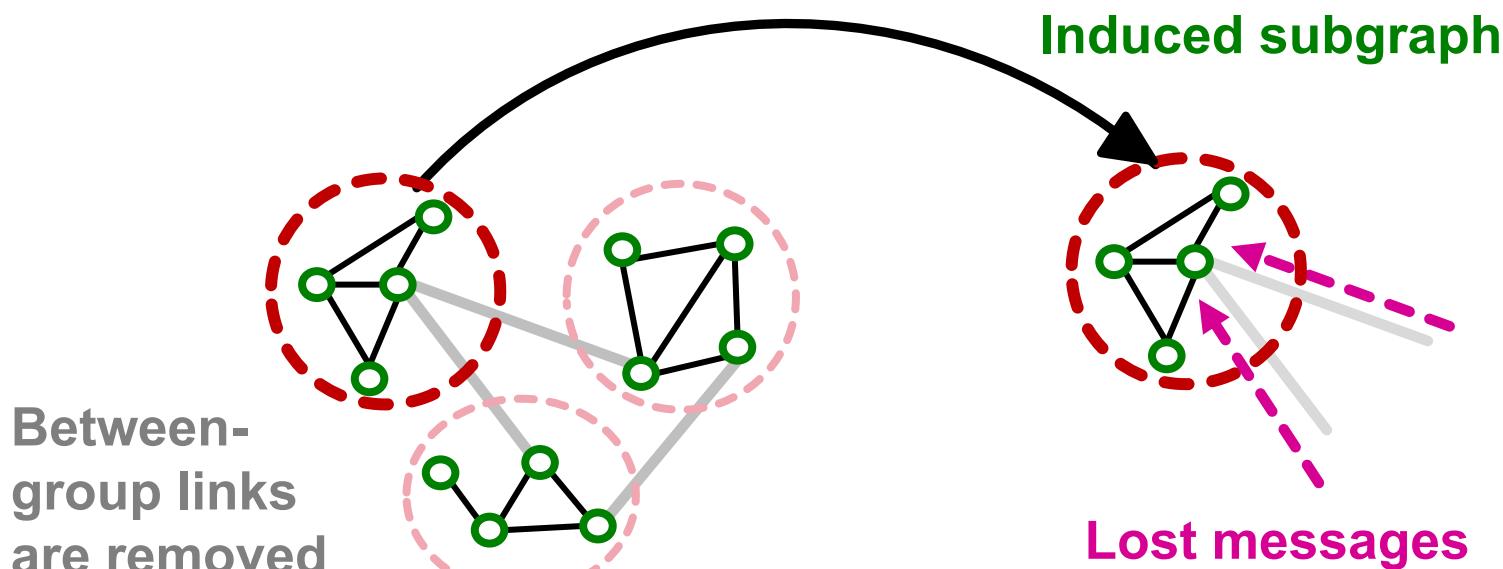
Layer-wise node
embedding update

Embedding



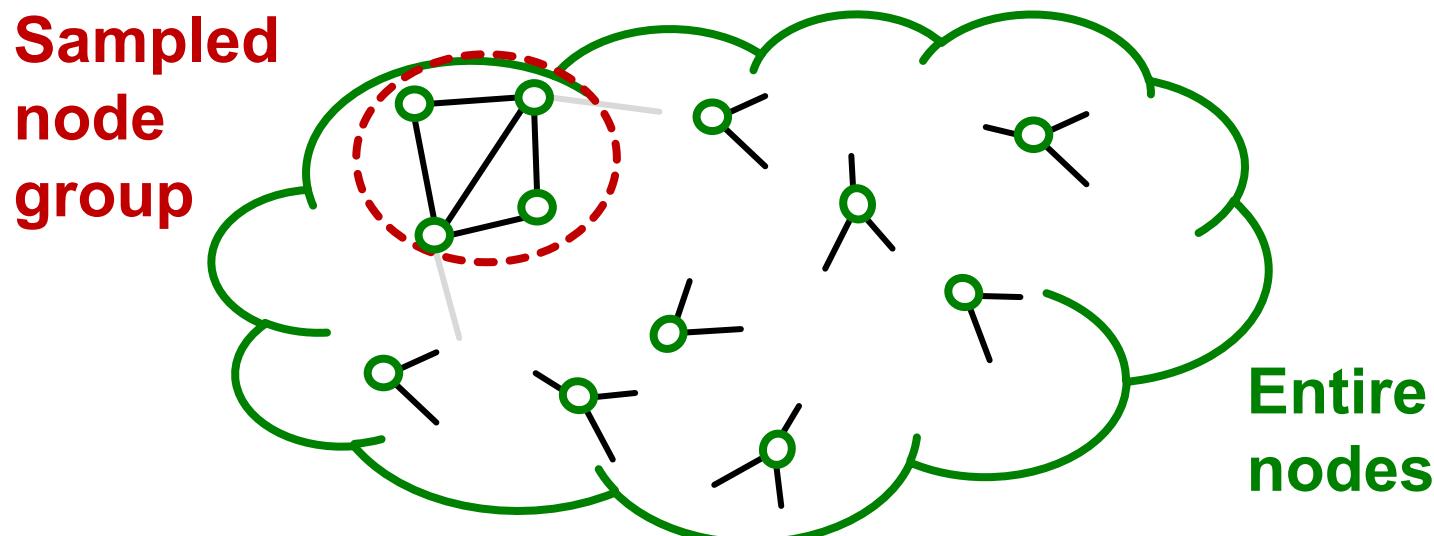
Issues with Cluster-GCN (1)

- 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, $\frac{1}{|V_c|} \sum_{v \in V_c} \ell_v(\theta)$, 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

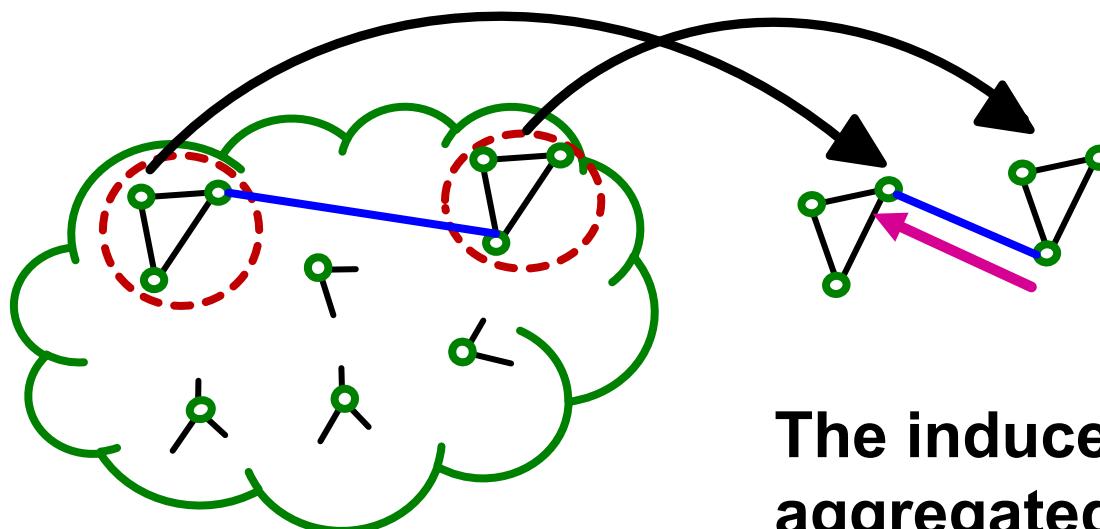
- **Solution: Aggregate multiple node groups per mini-batch.**
- Partition the graph into **relatively-small groups of nodes**.
- **For each mini-batch:**
 - Sample and aggregate **multiple node groups**.
 - **Construct the induced subgraph of the *aggregated node group*.**
 - The rest is the same as 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 edges between groups
→ Message can flow across groups.

Advanced Cluster-GCN

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

1) 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.

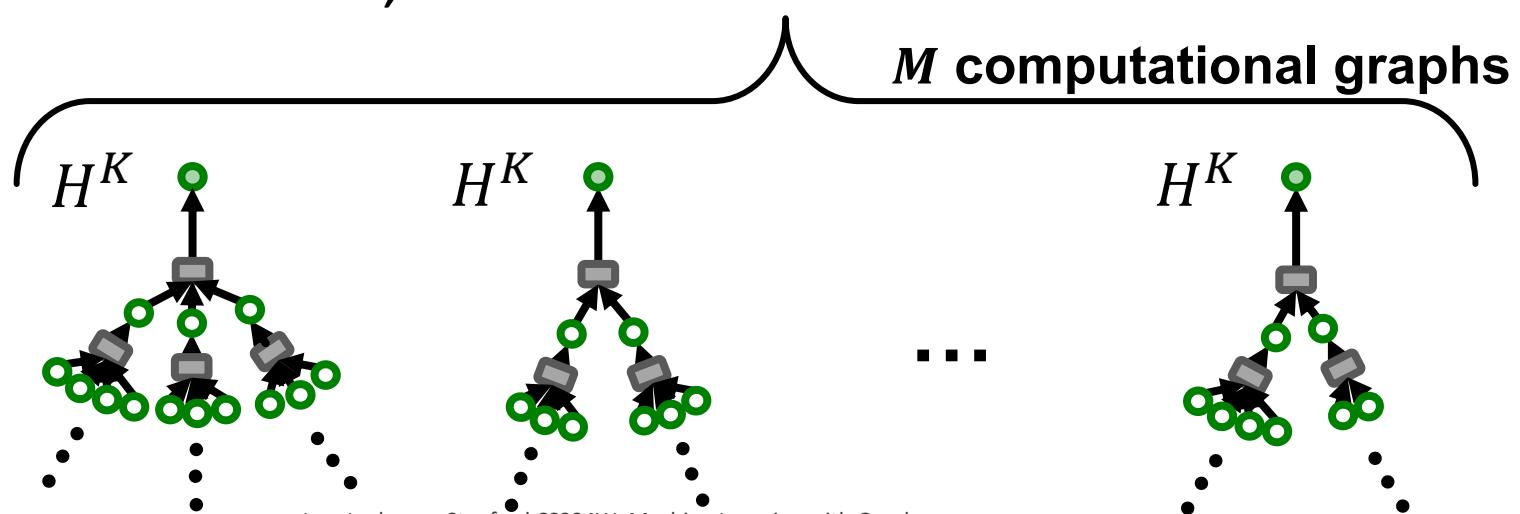
Advanced Cluster-GCN

2) 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 ($\ll N$) 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 ($\ll N$) 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 GNN Architecture

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



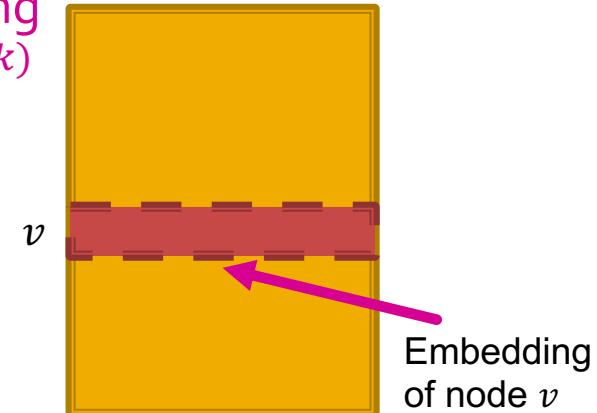
Roadmap of Simplifying GCN

- We start from Graph Convolutional Network (GCN) [Kipf & Welling ICLR 2017].
- We simplify GCN (“*SimplGCN*”) by **removing the non-linear activation** from the GCN [Wu et al. ICML 2019].
 - SimplGCN 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.
 - **The simplification strategy is very similar to the one used by LightGCN for recommender systems.**

Quick Overview of LightGCN (1)

- Adjacency matrix: A
- Degree matrix: D
- Normalized adjacency matrix:
 $\tilde{A} \equiv D^{-1/2} A D^{-1/2}$
- Let $E^{(k)}$ be the embedding matrix at k -th layer.
- Let E be the input embedding matrix.
 - We backprop into E .
- GCN's aggregation in the matrix form
 - $E^{(k+1)} = \text{ReLU}(\tilde{A}E^{(k)}W^{(k)})$

Embedding
matrix $E^{(k)}$



Quick Overview of LightGCN (2)

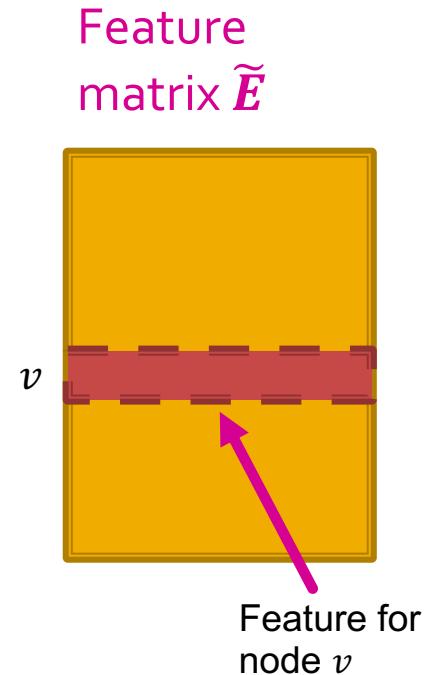
- Removing ReLU non-linearity gives us
 - $E^{(K)} = \tilde{A}^K E W$, where $W \equiv W^{(0)} \dots W^{(K-1)}$
**Diffusing node embeddings
along the graph**
- Efficient algorithm to obtain $\tilde{A}^K E$
 - Start from input embedding matrix E .
 - Apply $E \leftarrow \tilde{A} E$ for K times.
- Weight matrix W can be ignored for now.
 - W acts as a linear classifier over the diffused node embeddings $\tilde{A}^K E$.

Differences to LightGCN

- SimplGCN adds **self-loops** to adjacency matrix A :
 - $A \leftarrow A + I$
 - Follows the original GCN by Kipf & Welling.
- SimplGCN assumes input node embeddings E to be **given as features**:
 - Input embedding matrix E is **fixed** rather than learned.
 - **Important consequence:** $\tilde{A}^K E$ needs to be calculated **only once**.
 - Can be treated as a **pre-processing step**.

Simplified GCN: “SimplGCN”

- Let $\tilde{E} = \tilde{A}^K E$ be pre-processed feature matrix.
 - Each row stores the pre-processed feature for each node.
 - \tilde{E} can be used as input to any scalable ML models (e.g., linear model, MLP).
- SimplGCN empirically shows learning a linear model over \tilde{E} often gives performance comparable to GCN!



Comparison with Other Methods

- Compared to neighbor sampling and cluster-GCN, **SimplGCN is much more efficient.**
 - **SimplGCN computes \tilde{E} only once at the beginning.**
 - The pre-processing (sparse matrix vector product, $(E \leftarrow \tilde{A} E)$) can be performed efficiently on CPU.
 - Once \tilde{E} is obtained, getting an embedding for node v only takes **constant time!**
 - Just look up a row for node v in \tilde{E} .
 - No need to build a computational graph or sample a subgraph.
- But the model is **less expressive** (next).

Potential Issue of Simplified GCN

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

Performance of Simplified GCN

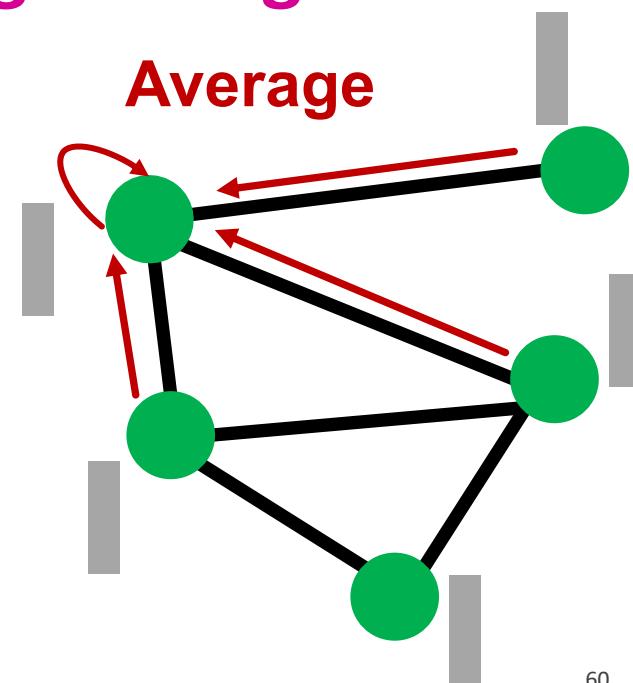
- Surprisingly, in semi-supervised node classification benchmark, **SimplGCN works comparably to the original GNNs despite being less expressive.**
- **Why?**

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 $E \leftarrow \tilde{A} E$ for K times.**
 - E is node feature matrix $E = X$
- 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.