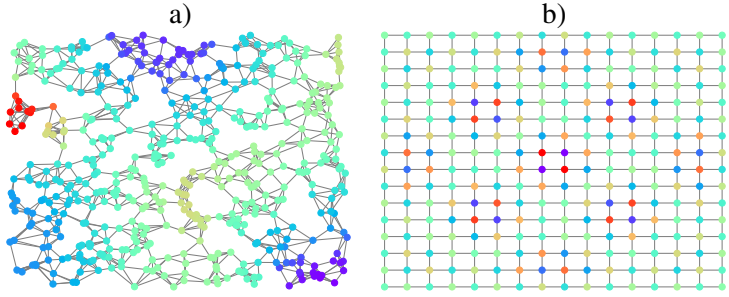
Scaling Up Graph Neural Networks

APM_5DS30_TP, Machine Learning with Graphs

Jhony H. Giraldo (Enseignant-Chercheur).
jhony.giraldo@telecom-paris.fr
October 24, 2025

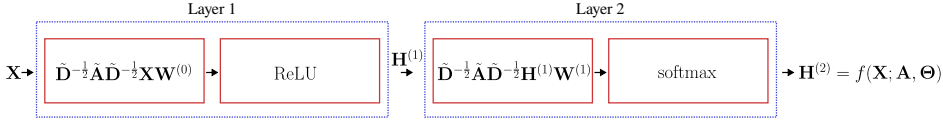
What do you remember from the last lecture?

- GNNs are composed of layers with graph convolutional filters and point-wise non-linearities.

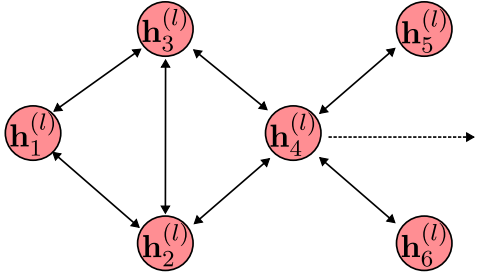


Process a) with graph convolutional NNs. Process b) with convolutional NNs

- We already know that the graph convolution is given by
$\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

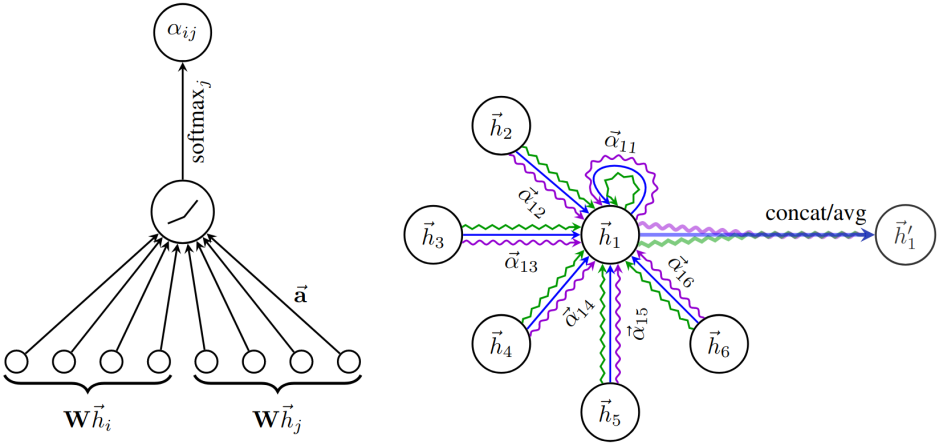


Block diagram of Vanilla GCN for node classification



$$\mathbf{m}_4^{(l+1)} = \sum_{j \in \mathcal{N}_4} \mathbf{S}(4,j)\psi_l(\mathbf{h}_4^{(l)}, \mathbf{h}_j^{(l)})$$

$$\mathbf{h}_4^{(l+1)} = \phi_l\left(\mathbf{h}_4^{(l)}, \mathbf{m}_4^{(l+1)}\right)$$



Graph Attention Network (GAT). **Source:** [Veličković et al.(2018)]

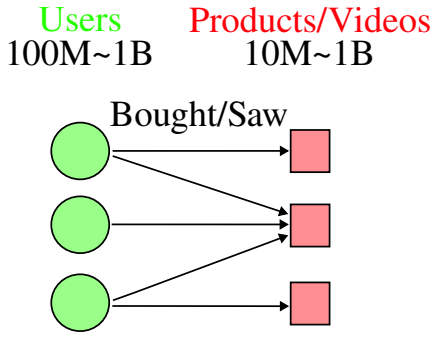
Recap

Limitations: Heterophily and Over-smoothing

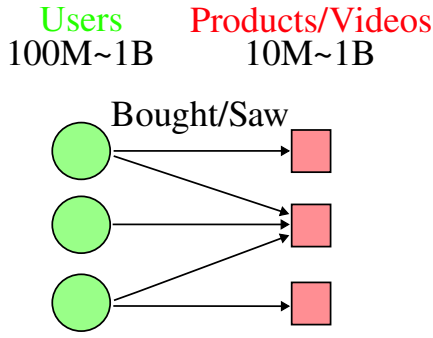
Outline

Graphs in Modern Applications

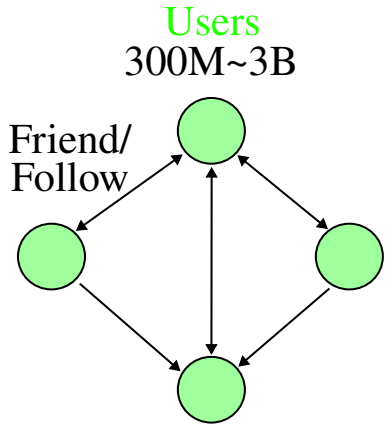
- Recommender Systems:
  - Amazon
  - YouTube
  - Instagram



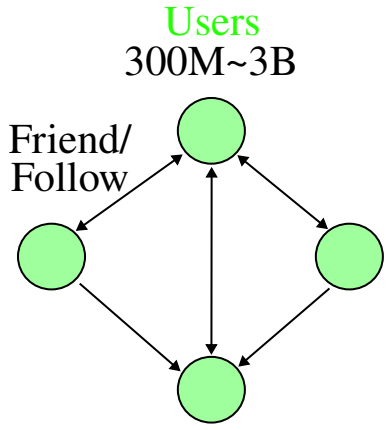
- Recommender Systems:
  - Amazon
  - YouTube
  - Instagram
- Machine Learning Tasks:
  - Recommend items (link prediction)
  - Classify users/items (node classification)



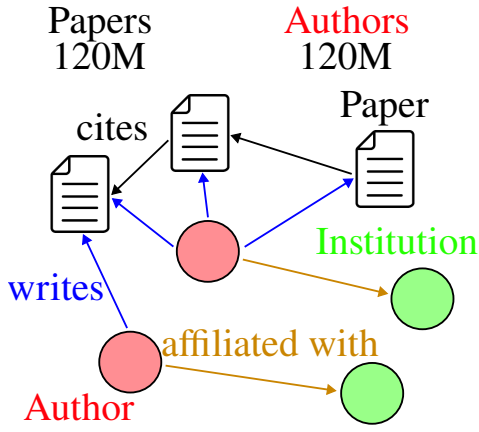
- Social Networks:
  - Facebook
  - Bluesky
  - LinkedIn



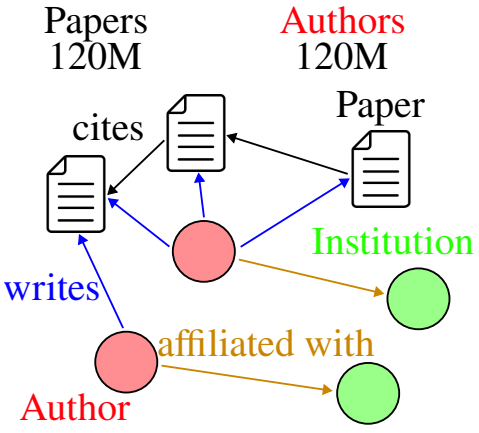
- Social Networks:
  - Facebook
  - Bluesky
  - LinkedIn
- Machine Learning Tasks:
  - Friend Recommendation (link level)
  - User property prediction (node level)



- Academic graph:
  - Microsoft Academic Graph



- Academic graph:
  - Microsoft Academic Graph
- Machine Learning Tasks:
  - Paper categorization (node classification)
  - Author collaboration recommendation
  - Paper citation recommendation (link prediction)



Graphs in Modern Applications

What is in Common?

- Large-scale:
  - Number of nodes ranges from 10M to 10B
  - Number of edges ranges from 100M to 100B

- Large-scale:
  - Number of nodes ranges from 10M to 10B
  - Number of edges ranges from 100M to 100B
- Tasks:
  - Node level: user/item/paper classification
  - Link level: recommendation

Graphs in Modern Applications

What is in Common?

- Large-scale:
  - Number of nodes ranges from 10M to 10B
  - Number of edges ranges from 100M to 100B
- Tasks:
  - Node level: user/item/paper classification
  - Link level: recommendation
- Today's lecture is about scaling up GNNs to very large graphs!

- How we usually train a Machine Learning model on a large dataset ($N =$ number of data is large), *e.g.* training a CNN on ImageNet?

- How we usually train a Machine Learning model on a large dataset ($N =$ number of data is large), *e.g.* training a CNN on ImageNet?
- Our objective is to minimize the averaged loss:

$$\mathcal{L}(\mathbf{\Theta}) = \frac{1}{N} \sum_{i=0}^{N-1} \mathcal{L}_i(\mathbf{\Theta}),$$

where $\mathbf{\Theta}$ is the model parameters and $\mathcal{L}_i(\mathbf{\Theta})$ is the loss for the ith data point

- How we usually train a Machine Learning model on a large dataset ($N =$ number of data is large), *e.g.* training a CNN on ImageNet?
- Our objective is to minimize the averaged loss:

$$\mathcal{L}(\mathbf{\Theta}) = \frac{1}{N} \sum_{i=0}^{N-1} \mathcal{L}_i(\mathbf{\Theta}),$$

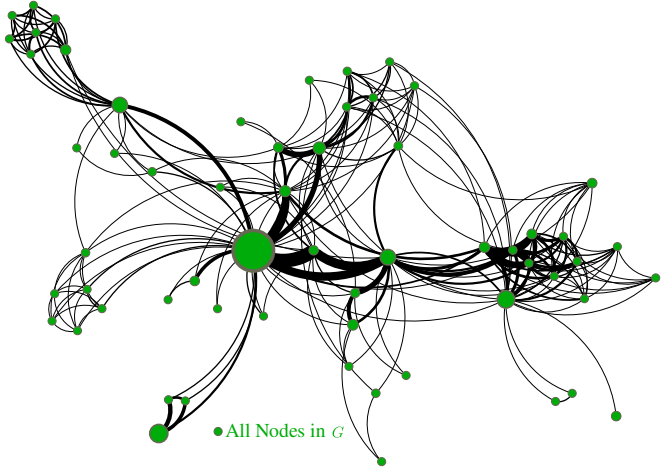
  where $\mathbf{\Theta}$ is the model parameters and $\mathcal{L}_i(\mathbf{\Theta})$ is the loss for the ith data point
- We perform Stochastic Gradient Descent (SGD)
  - Randomly sample M ($<< N$) data (**mini-batches**)
  - Compute the loss function over the M data points
  - Update model parameters according to the gradient of the loss

- Why not just simply using standard SGD for GNNs?

Graphs in Modern Applications

Why is it Hard?

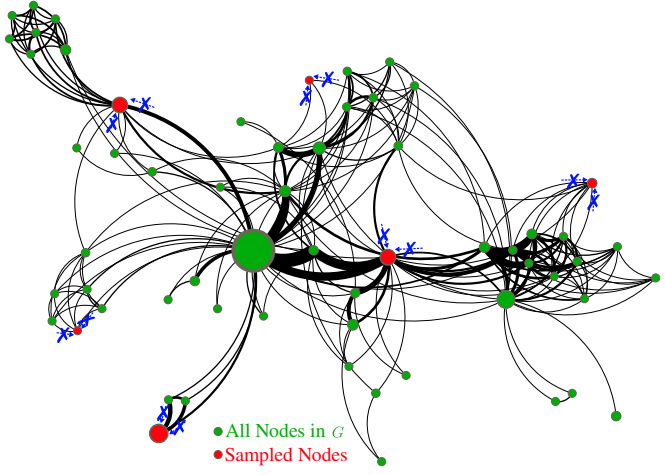
- Why not just simply using standard SGD for GNNs?



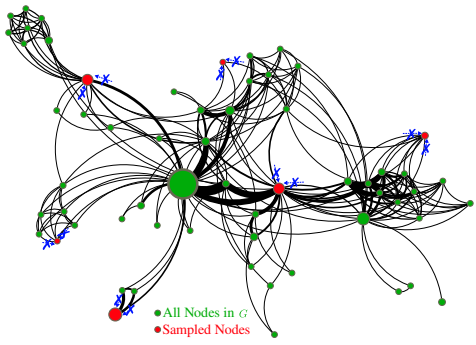
Graphs in Modern Applications

Why is it Hard?

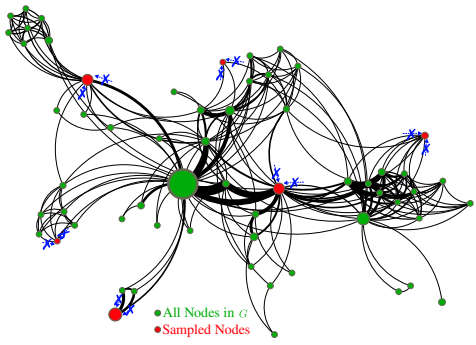
- Why not just simply using standard SGD for GNNs?



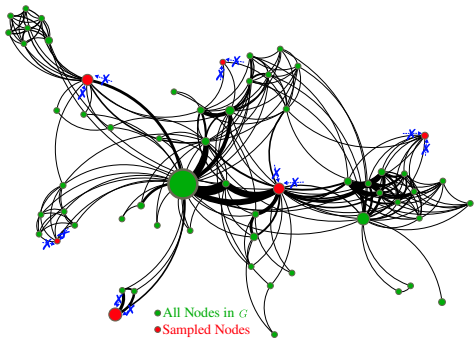
- In mini-batch, we sample M ($<< N$) nodes independently



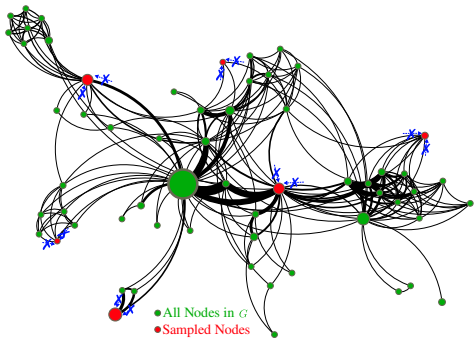
- In mini-batch, we sample M ($<< N$) nodes independently
- Sampled nodes tend to be isolated from each other



- In mini-batch, we sample M ($<< N$) nodes independently
- Sampled nodes tend to be isolated from each other
- GNNs generate node embeddings by aggregating information from the neighborhood



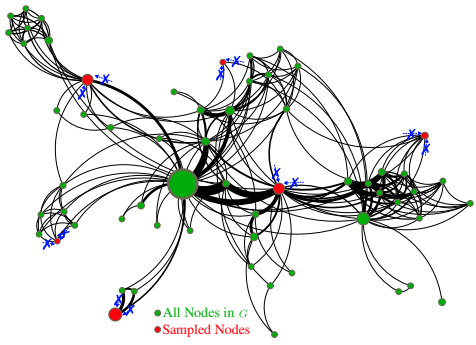
- In mini-batch, we sample M ($<< N$) nodes independently
- Sampled nodes tend to be isolated from each other
- GNNs generate node embeddings by aggregating information from the neighborhood
- The GNN does not have access to neighboring nodes within the mini-batch!



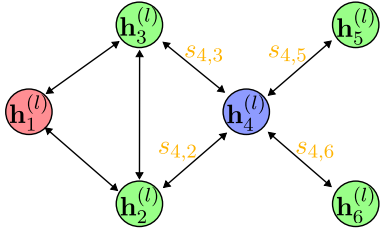
Graphs in Modern Applications

Why is it Hard?

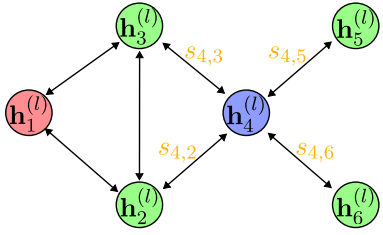
- In mini-batch, we sample M ($<< N$) nodes independently
- Sampled nodes tend to be isolated from each other
- GNNs generate node embeddings by aggregating information from the neighborhood
- The GNN does not have access to neighboring nodes within the mini-batch!
- Standard SGD cannot effectively train GNNs on very large graphs



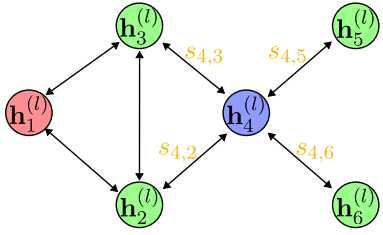
- Why we don't do full-batch, where we generate embeddings of all nodes at the same time? We could use GCN which is a pretty efficient model, isn't it?



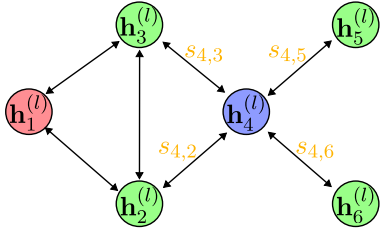
- Why we don't do full-batch, where we generate embeddings of all nodes at the same time? We could use GCN which is a pretty efficient model, isn't it?
- Do you remember how much RAM we need for the graph with $25'000,000$ nodes and 300 neighbors on average? It was 120 GB of memory



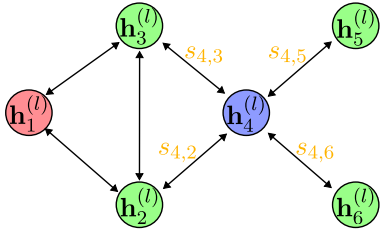
- Why we don't do full-batch, where we generate embeddings of all nodes at the same time? We could use GCN which is a pretty efficient model, isn't it?
- Do you remember how much RAM we need for the graph with $25'000,000$ nodes and 300 neighbors on average? It was 120 GB of memory
- We might save this graph in a powerful CPU in a server for example



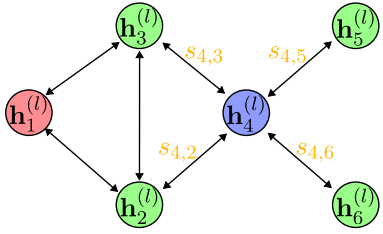
- However, we need GPUs to train our modern Machine Learning models, you already know that



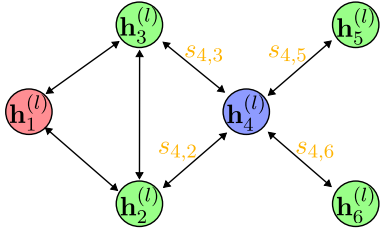
- However, we need GPUs to train our modern Machine Learning models, you already know that
- The most powerful GPUs in the market nowadays have 40 to 80 GB of memory RAM, and we also need some space for our model



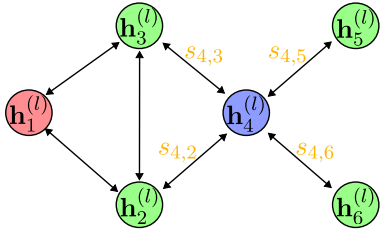
- However, we need GPUs to train our modern Machine Learning models, you already know that
- The most powerful GPUs in the market nowadays have 40 to 80 GB of memory RAM, and we also need some space for our model
- Now imagine that you work at Amazon and you need to train a GNN for the graph representing the customers of Europe



- However, we need GPUs to train our modern Machine Learning models, you already know that
- The most powerful GPUs in the market nowadays have 40 to 80 GB of memory RAM, and we also need some space for our model
- Now imagine that you work at Amazon and you need to train a GNN for the graph representing the customers of Europe
- We'll have more nodes than 25M nodes!



- However, we need GPUs to train our modern Machine Learning models, you already know that
- The most powerful GPUs in the market nowadays have 40 to 80 GB of memory RAM, and we also need some space for our model
- Now imagine that you work at Amazon and you need to train a GNN for the graph representing the customers of Europe
- We'll have more nodes than 25M nodes!
- We need to do something else



GraphSAGE Neighbor Sampling

GraphSAGE Neighbor Sampling

Computational Graph

- GNNs generate node embeddings via neighbor aggregation with MPNN

GraphSAGE Neighbor Sampling

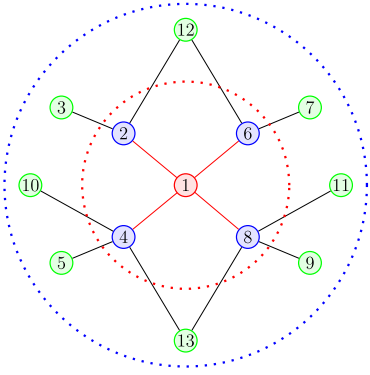
Computational Graph

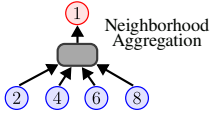
- GNNs generate node embeddings via neighbor aggregation with MPNN
- Let's focus on one single node. How looks like a two-layers GNN?

GraphSAGE Neighbor Sampling

Computational Graph

- GNNs generate node embeddings via neighbor aggregation with MPNN
- Let's focus on one single node. How looks like a two-layers GNN?

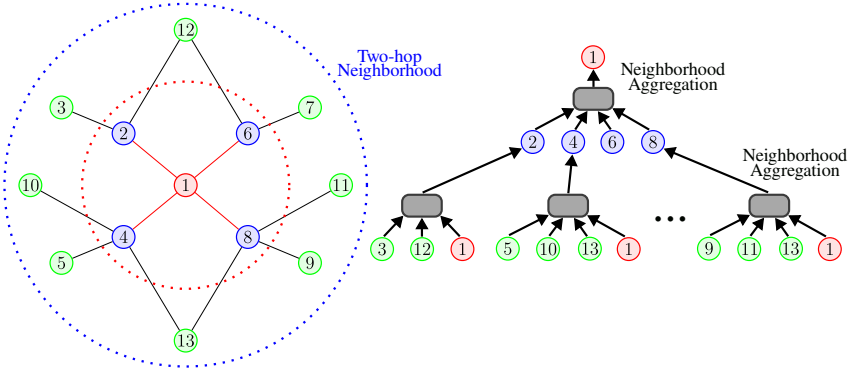




GraphSAGE Neighbor Sampling

Computational Graph

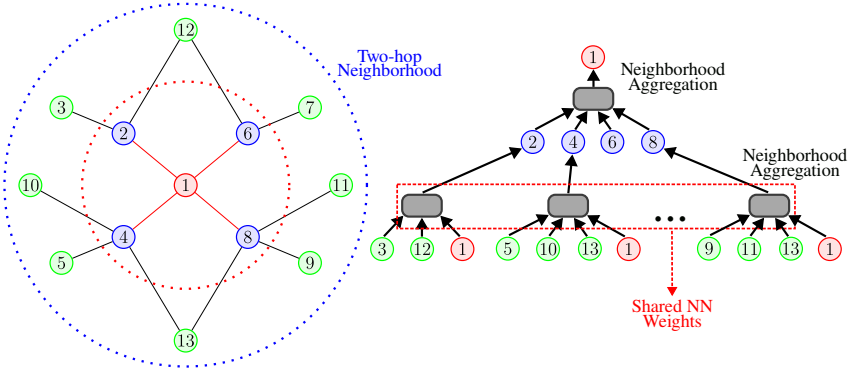
- GNNs generate node embeddings via neighbor aggregation with MPNN
- Let's focus on one single node. How looks like a two-layers GNN?



GraphSAGE Neighbor Sampling

Computational Graph

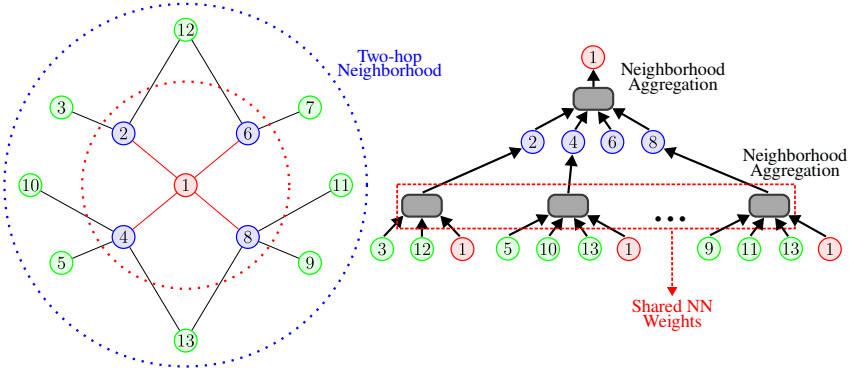
- GNNs generate node embeddings via neighbor aggregation with MPNN
- Let's focus on one single node. How looks like a two-layers GNN?



GraphSAGE Neighbor Sampling

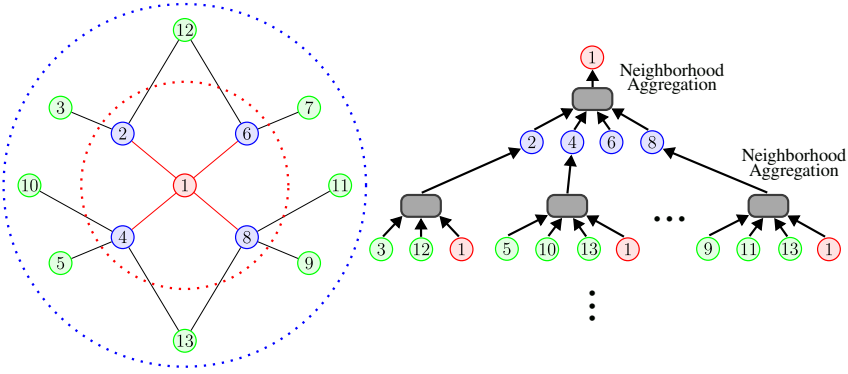
Computational Graph

- GNNs generate node embeddings via neighbor aggregation with MPNN
- Let's focus on one single node. How looks like a two-layers GNN?



- A two-layer GNN generates embedding of each node using two-hop neighborhood structure and features.

- **Observation**: A K-layers GNN generates embedding of a node using K-hop neighborhood structure and features



GraphSAGE Neighbor Sampling

Computing Node Embeddings

- **Key insight**: We need the K-hop neighborhood to compute the embedding of a single node

GraphSAGE Neighbor Sampling

Computing Node Embeddings

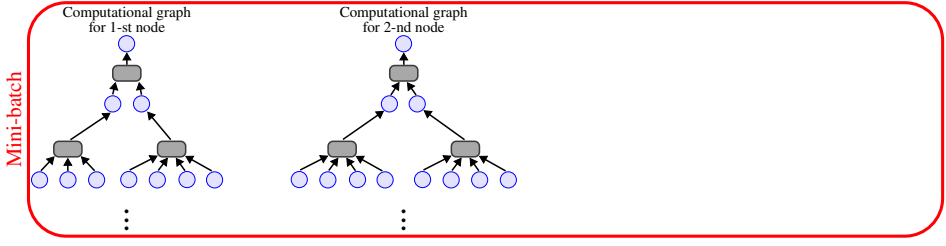
- **Key insight**: We need the K-hop neighborhood to compute the embedding of a single node
- Given a set of M different nodes in a mini-batch, we can generate their embeddings using M computational graphs (we can parallelize this using a GPU)

- **Key insight**: We need the K-hop neighborhood to compute the embedding of a single node
- Given a set of M different nodes in a mini-batch, we can generate their embeddings using M computational graphs (we can parallelize this using a GPU)

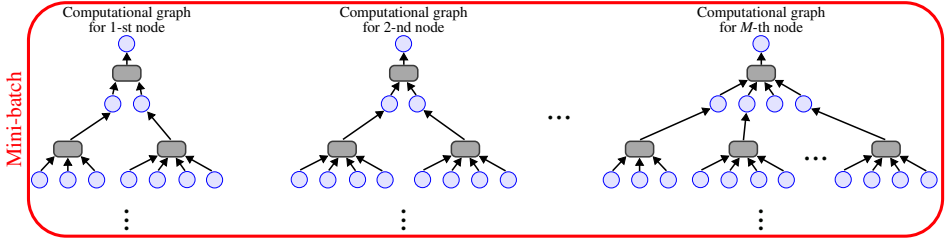
GraphSAGE Neighbor Sampling

Computing Node Embeddings

- **Key insight**: We need the K-hop neighborhood to compute the embedding of a single node
- Given a set of M different nodes in a mini-batch, we can generate their embeddings using M computational graphs (we can parallelize this using a GPU)



- **Key insight**: We need the K-hop neighborhood to compute the embedding of a single node
- Given a set of M different nodes in a mini-batch, we can generate their embeddings using M computational graphs (we can parallelize this using a GPU)



- We can now consider a SGD strategy to train K-layer GNNs:

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes
  - For each sample node v_i:

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes
  - For each sample node v_i:
    - Get the K-hop neighborhood and construct the computation graph

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes
  - For each sample node v_i:
    - Get the K-hop neighborhood and construct the computation graph
    - Use the computation graph to generate the v_i embedding

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes
  - For each sample node v_i:
    - Get the K-hop neighborhood and construct the computation graph
    - Use the computation graph to generate the v_i embedding
  - Compute the loss $\mathcal{L}(\mathbf{\Theta})$ averaged over the M nodes

- We can now consider a SGD strategy to train K-layer GNNs:
  - Randomly sample M ($<< N$) nodes
  - For each sample node v_i:
    - Get the K-hop neighborhood and construct the computation graph
    - Use the computation graph to generate the v_i embedding
  - Compute the loss $\mathcal{L}(\mathbf{\Theta})$ averaged over the M nodes
  - Update the parameters according to the gradient of the loss

GraphSAGE Neighbor Sampling

Issues with Stochastic Training of GNNs

GraphSAGE Neighbor Sampling

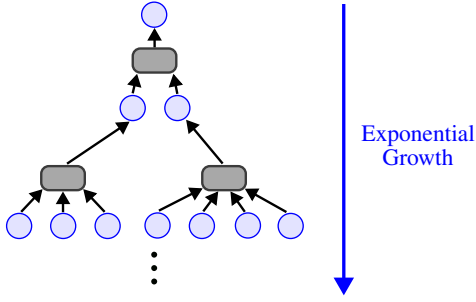
Issues with Stochastic Training of GNNs

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

GraphSAGE Neighbor Sampling

Issues with Stochastic Training of GNNs

- For each node, we need to get the entire K-hop neighborhood and pass it through the computation graph. We need to aggregate a lot of information just to compute one node embedding
- Computation graph becomes exponentially large w.r.t the layer size K

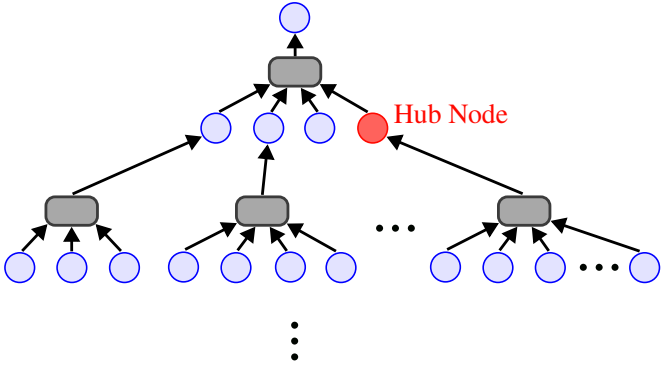


GraphSAGE Neighbor Sampling

Issues with Stochastic Training of GNNs

- What other problem do you see?

- What other problem do you see?
- Computation graph explodes when it hits a hub node (a node with a high degree, *e.g.* Cristiano Ronaldo has many followers on Instagram)



GraphSAGE Neighbor Sampling

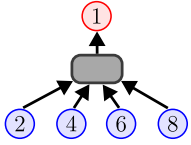
Neighborhood Sampling

- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)

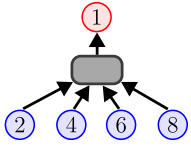
GraphSAGE Neighbor Sampling

Neighborhood Sampling

- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)

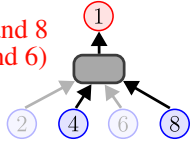


- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)



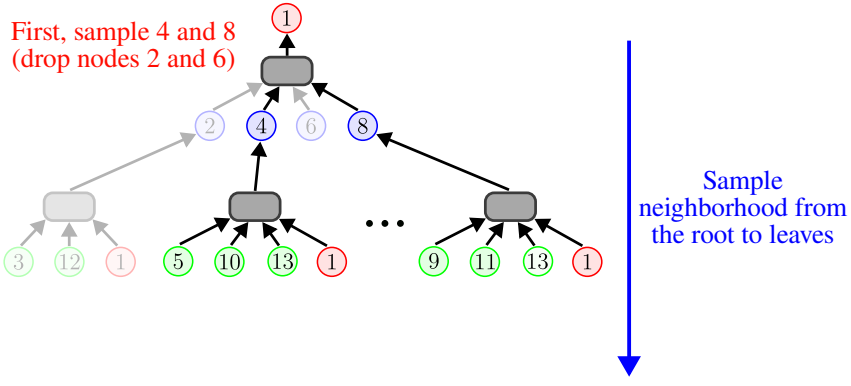
- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)

First, sample 4 and 8
(drop nodes 2 and 6)

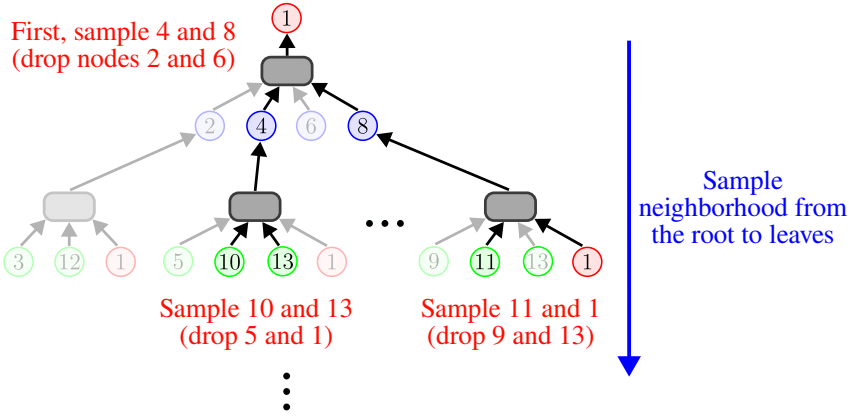


Sample
neighborhood from
the root to leaves

- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)



- What idea do you have to solve the problems we mentioned before? (**Hint**: see the subtitle and try to come up with an algorithm)



GraphSAGE Neighbor Sampling

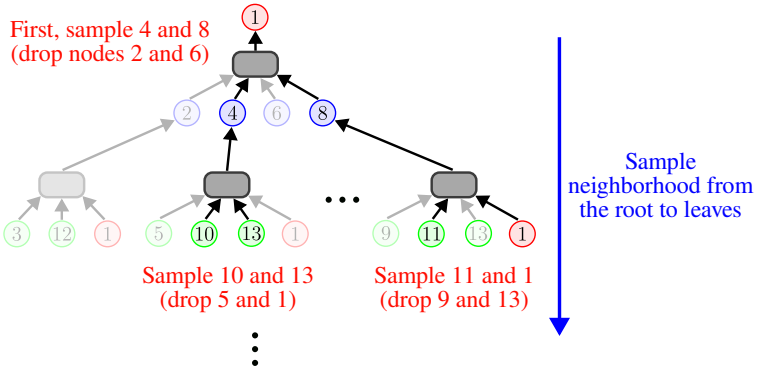
Neighborhood Sampling

- **Key idea**: Construct the computational graph by randomly sampling at most H neighbors at each hop

GraphSAGE Neighbor Sampling

Neighborhood Sampling

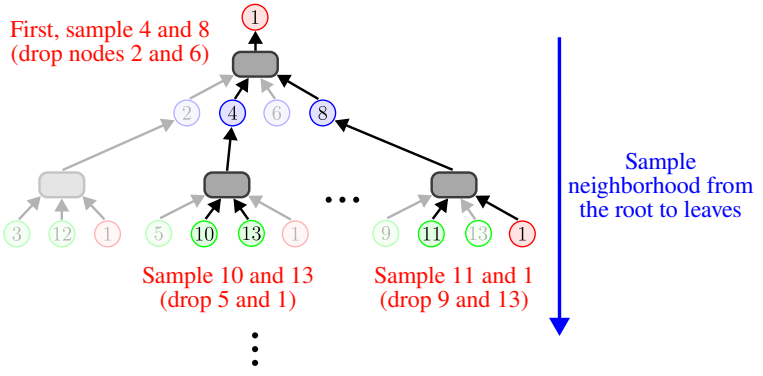
- **Key idea**: Construct the computational graph by randomly sampling at most H neighbors at each hop
- The example we presented before is for $H = 2$



GraphSAGE Neighbor Sampling

Neighborhood Sampling

- **Key idea**: Construct the computational graph by randomly sampling at most H neighbors at each hop
- The example we presented before is for $H = 2$
- Now we can use the pruned computational graph to do message passing



- **Remark 1**: A K-layer GNN will involve at most H^K leaf nodes in the computational graph

- **Remark 1**: A K-layer GNN will involve at most H^K leaf nodes in the computational graph
- **Remark 2**: 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 neighborhood aggregation

- **Remark 1**: A K-layer GNN will involve at most H^K leaf nodes in the computational graph
- **Remark 2**: 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 neighborhood aggregation
- **Remark 3**: The size of the computational graph is still exponential w.r.t. the number of GNN layers K, *i.e.*, adding one more GNN layer will make computation H times more expensive

- **Remark 1**: A K-layer GNN will involve at most H^K leaf nodes in the computational graph
- **Remark 2**: 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 neighborhood aggregation
- **Remark 3**: The size of the computational graph is still exponential w.r.t. the number of GNN layers K, *i.e.*, adding one more GNN layer will make computation H times more expensive
- **Remark 4**: We said that we can randomly sample the nodes, this strategy is fast but many times not optimal (we may sample many "irrelevant" nodes)

- **Remark 1**: A K-layer GNN will involve at most H^K leaf nodes in the computational graph
- **Remark 2**: 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 neighborhood aggregation
- **Remark 3**: The size of the computational graph is still exponential w.r.t. the number of GNN layers K, *i.e.*, adding one more GNN layer will make computation H times more expensive
- **Remark 4**: We said that we can randomly sample the nodes, this strategy is fast but many times not optimal (we may sample many "irrelevant" nodes)
  - We can use the random walk matrix $\mathbf{D}^{-1}\mathbf{A}$ to sample more "relevant" nodes in the neighborhood. This works better in practice

- A computational graph is constructed for each node in a mini-batch

- A computational graph is constructed for each node in a mini-batch
- In neighbor sampling, the computational graph is sampled/pruned to reduce computational complexity

- A computational graph is constructed for each node in a mini-batch
- In neighbor sampling, the computational graph is sampled/pruned to reduce computational complexity
- This pruned graph is used to generate a node embedding

- A computational graph is constructed for each node in a mini-batch
- In neighbor sampling, the computational graph is sampled/pruned to reduce computational complexity
- This pruned graph is used to generate a node embedding
- The computational graph can still become large, especially for GNNs with many message-passing layers. Whether we need deep GNNs or not is still an active field of research as we mentioned in previous lectures

Cluster-GCN

Cluster-GCN

Issues with Neighbor Sampling

- For $H \geq 2$ the size of the computational graph becomes exponentially large w.r.t. the number of GNN layers

Cluster-GCN

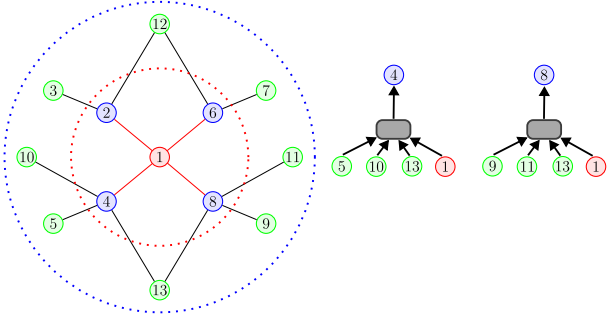
Issues with Neighbor Sampling

- For $H \geq 2$ the size of the computational graph becomes exponentially large w.r.t. the number of GNN layers
- Computation is redundant, especially when we have nodes that share many neighbors in a mini-batch

Cluster-GCN

Issues with Neighbor Sampling

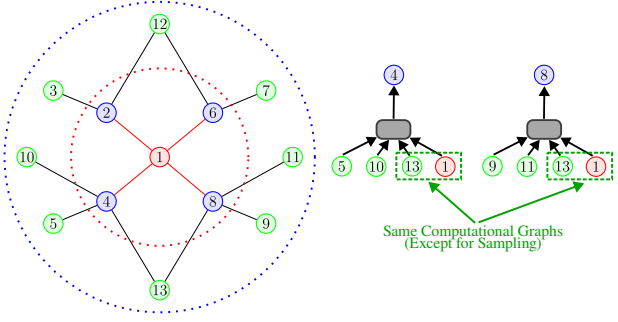
- For $H \geq 2$ the size of the computational graph becomes exponentially large w.r.t. the number of GNN layers
- Computation is redundant, especially when we have nodes that share many neighbors in a mini-batch
- What do you notice in the computational graph of these two nodes?



Cluster-GCN

Issues with Neighbor Sampling

- For $H \geq 2$ the size of the computational graph becomes exponentially large w.r.t. the number of GNN layers
- Computation is redundant, especially when we have nodes that share many neighbors in a mini-batch
- What do you notice in the computational graph of these two nodes?

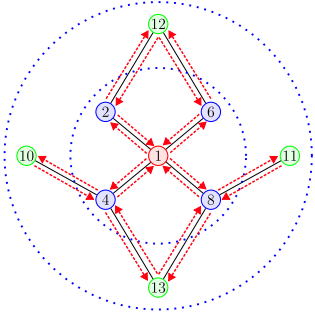


- In full-batch GNN, all the node embeddings are updated together using the embeddings of the previous layer:

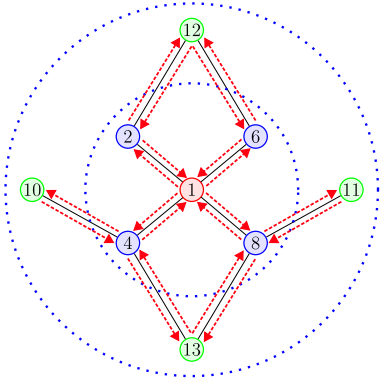
$$\mathbf{h}_i^{(l+1)} = \phi_l \left(\mathbf{h}_i^{(l)}, \sum_{j \in \mathcal{N}_i} \mathbf{S}(i,j) \psi_l(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}) \right)$$

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

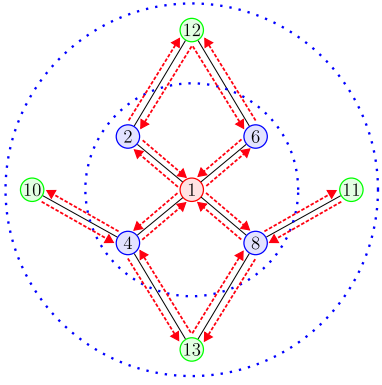
$$\mathbf{h}_i^{(l+1)} = \phi_l \left(\mathbf{h}_i^{(l)}, \sum_{j \in \mathcal{N}_i} \mathbf{S}(i,j) \psi_l(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}) \right)$$



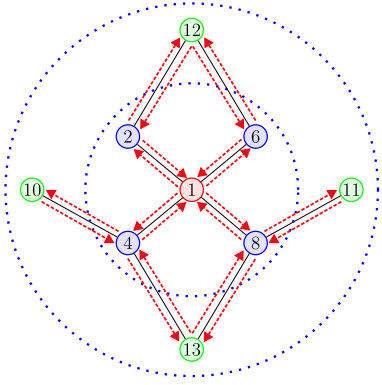
- In each layer, we only need to compute $2 \times$ (number of edges) messages for the full batch (for undirected graphs, for directed graphs it's usually less than that)



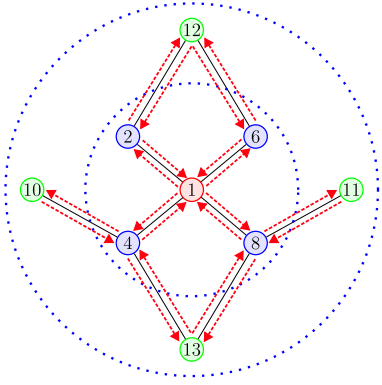
- In each layer, we only need to compute $2 \times$ (number of edges) messages for the full batch (for undirected graphs, for directed graphs it's usually less than that)
- For a K-layers GNN, we need $2 \times$ (number of edges) $\times K$ messages



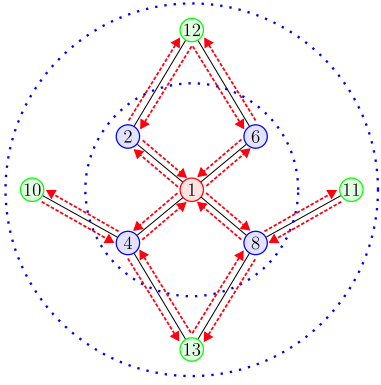
- In each layer, we only need to compute $2 \times$ (number of edges) messages for the full batch (for undirected graphs, for directed graphs it's usually less than that)
- For a K-layers GNN, we need $2 \times$ (number of edges) $\times K$ messages
- The entire computation of the GNN is linear in the number of edges and the number of GNN layers! This is pretty fast



- The layer-wise node embedding update allows the re-use of embeddings from the previous layer



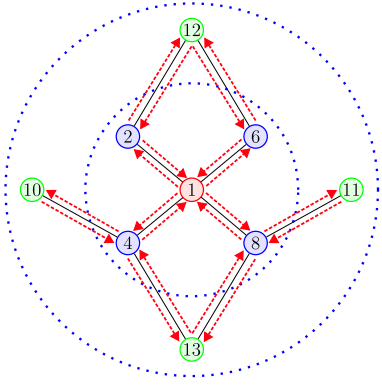
- The layer-wise node embedding update allows the re-use of embeddings from the previous layer
- This significantly reduces the computational redundancy in neighbor sampling



Cluster-GCN

Insights 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 in neighbor sampling
- Unfortunately, full-batch is not possible when the whole graph does not fit in memory (that's the whole reason why we introduced neighbor sampling)



Cluster-GCN

Insights from Full-batch GNN

- What idea we can use from full-batch for scaling up GNNs?

Cluster-GCN

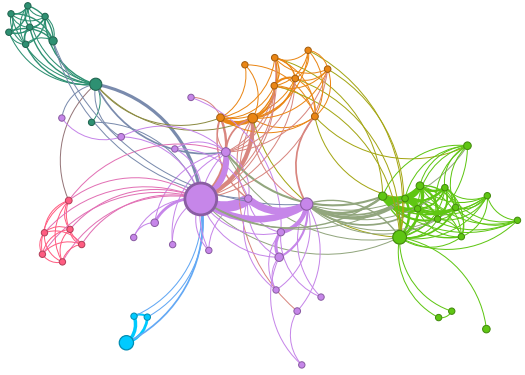
Insights from Full-batch GNN

- What idea we can use from full-batch for scaling up GNNs?
- What if we take some sub-graphs from the full graph and we apply message passing there?

Cluster-GCN

Insights from Full-batch GNN

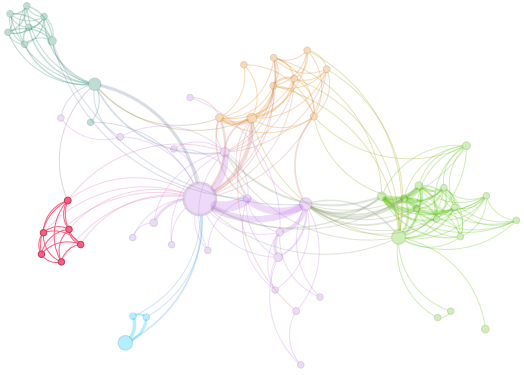
- What idea we can use from full-batch for scaling up GNNs?
- What if we take some sub-graphs from the full graph and we apply message passing there?



Cluster-GCN

Insights from Full-batch GNN

- What idea we can use from full-batch for scaling up GNNs?
- What if we take some sub-graphs from the full graph and we apply message passing there?



Cluster-GCN

Insights from Full-batch GNN

- What idea we can use from full-batch for scaling up GNNs?
- What if we take some sub-graphs from the full graph and we apply message passing there?

Cluster-GCN

Insights from Full-batch GNN

- What idea we can use from full-batch for scaling up GNNs?
- What if we take some sub-graphs from the full graph and we apply message passing there?

- Question: What sub-graphs are good for training GNNs?

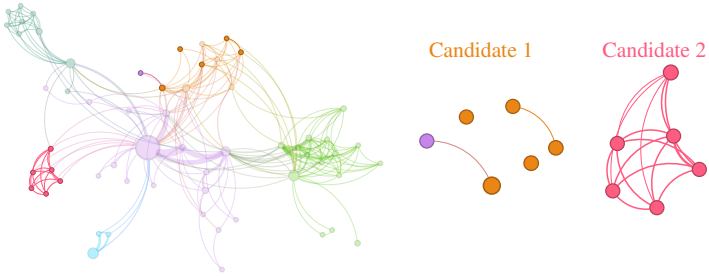
- Question: What sub-graphs are good for training GNNs?
- Remember that GNNs perform node embedding by passing messages via the edges

- Question: What sub-graphs are good for training GNNs?
- Remember that GNNs perform node embedding by passing messages via the edges
- Sub-graphs should retain the edge connectivity structure of the original graph as much as possible

- Question: What sub-graphs are good for training GNNs?
- Remember that GNNs perform node embedding by passing messages via the edges
- Sub-graphs should retain the edge connectivity structure of the original graph as much as possible
- As a consequence, the GNN over the sub-graph generates embeddings closer to the GNN over the original graph

Cluster-GCN

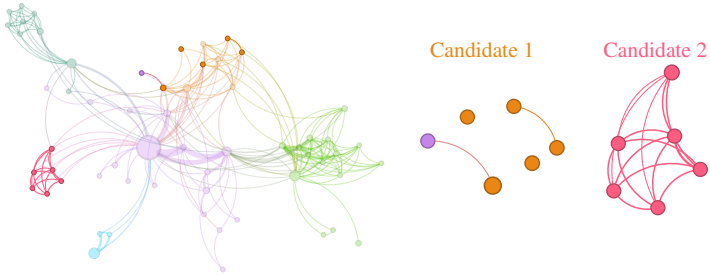
Good Sub-graphs for GNN Training



- Which sub-graph is good for training a GNN?

Cluster-GCN

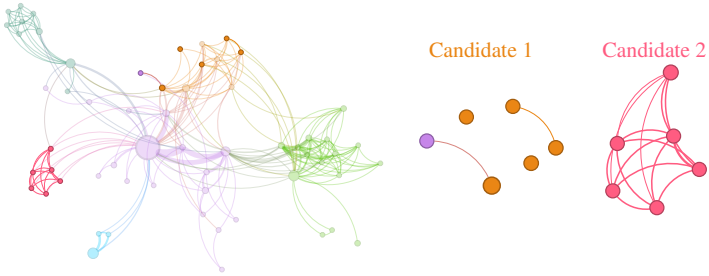
Good Sub-graphs for GNN Training



- Which sub-graph is good for training a GNN?
- The first candidate drops many connectivity patterns, even leading to isolated nodes, this is bad

Cluster-GCN

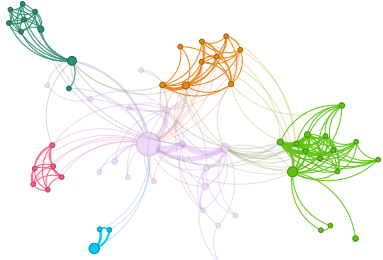
Good Sub-graphs for GNN Training



- Which sub-graph is good for training a GNN?
- The first candidate drops many connectivity patterns, even leading to isolated nodes, this is bad
- The second candidate retains the essential community structure among the 6 nodes, this is good

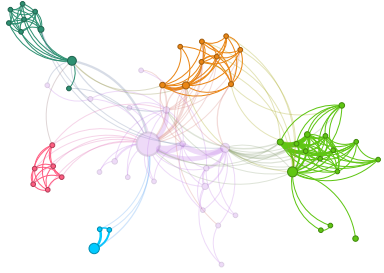
Cluster-GCN

Exploiting Community Structure



Cluster-GCN

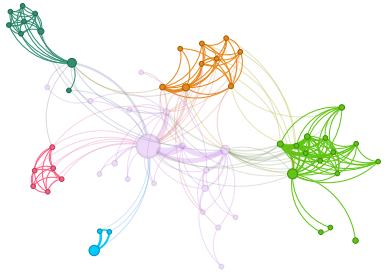
Exploiting Community Structure



- Many real-world graphs exhibit community structure, think of social networks, recommender systems, etc

Cluster-GCN

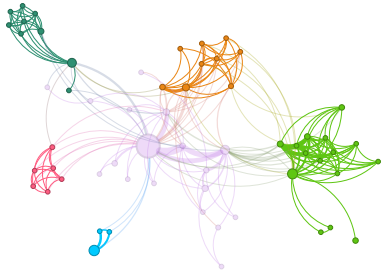
Exploiting Community Structure



- Many real-world graphs exhibit community structure, think of social networks, recommender systems, etc
- As a consequence a large graph can be decomposed into many small communities, as the graph in the figure

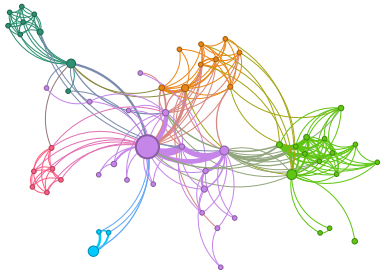
Cluster-GCN

Exploiting Community Structure

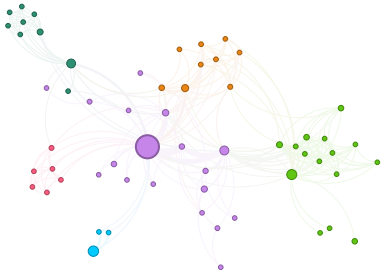


- Many real-world graphs exhibit community structure, think of social networks, recommender systems, etc
- As a consequence a large graph can be decomposed into many small communities, as the graph in the figure
- Key idea: Sample a community as a sub-graph, where each sub-graph retains essential local connectivity of the original graph

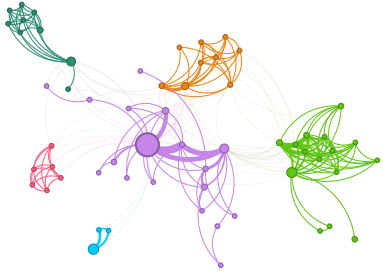
- **Pre-processing**: Given a large graph $G = (\mathcal{V}, \mathcal{E})$, we divide it into C sub-graphs $G_1, G_2, \ldots, G_C$



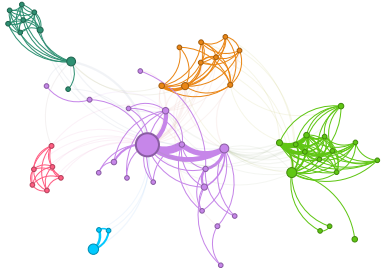
- **Pre-processing**: Given a large graph $G = (\mathcal{V}, \mathcal{E})$, we divide it into C sub-graphs $G_1, G_2, \ldots, G_C$
- For doing so, we divide the set of nodes $\mathcal{V}$ into C partitions $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_C$ such that $\bigcup_{i=1}^{C} \mathcal{V}_i = \mathcal{V}$



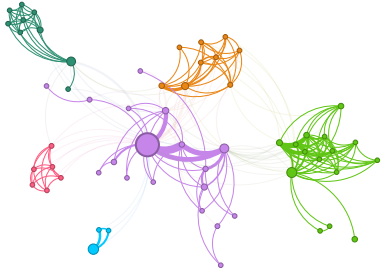
- **Pre-processing**: Given a large graph $G = (\mathcal{V}, \mathcal{E})$, we divide it into C sub-graphs $G_1, G_2, \ldots, G_C$
- For doing so, we divide the set of nodes $\mathcal{V}$ into C partitions $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_C$ such that $\bigcup_{i=1}^{C} \mathcal{V}_i = \mathcal{V}$
- Each $\mathcal{V}_i$ induces a sub-graph $G_i = (\mathcal{V}_i, \mathcal{E}_i)$



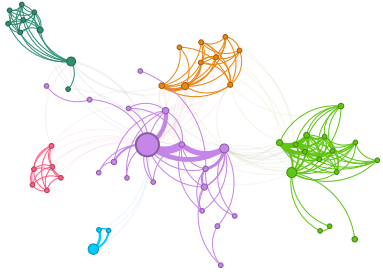
- **Pre-processing**: Given a large graph $G = (\mathcal{V}, \mathcal{E})$, we divide it into C sub-graphs $G_1, G_2, \ldots, G_C$
- For doing so, we divide the set of nodes $\mathcal{V}$ into C partitions $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_C$ such that $\bigcup_{i=1}^{C} \mathcal{V}_i = \mathcal{V}$
- Each $\mathcal{V}_i$ induces a sub-graph $G_i = (\mathcal{V}_i, \mathcal{E}_i)$
- Notice that in general $\bigcup_{i=1}^{C} \mathcal{E}_i \neq \mathcal{E}$ since the between-group edges are not included in $G_1, \ldots, G_C$



- Once you have the C induced sub-graphs, you apply the regular GNN's layer-wise node update over G_i to obtain the node embedding $\mathbf{h}_v$ for each node $v \in \mathcal{V}_i$



- Once you have the C induced sub-graphs, you apply the regular GNN's layer-wise node update over G_i to obtain the node embedding $\mathbf{h}_v$ for each node $v \in \mathcal{V}_i$
- For doing so, your treat each sub-graph as a mini-batch to update the parameters of your GNN



Cluster-GCN

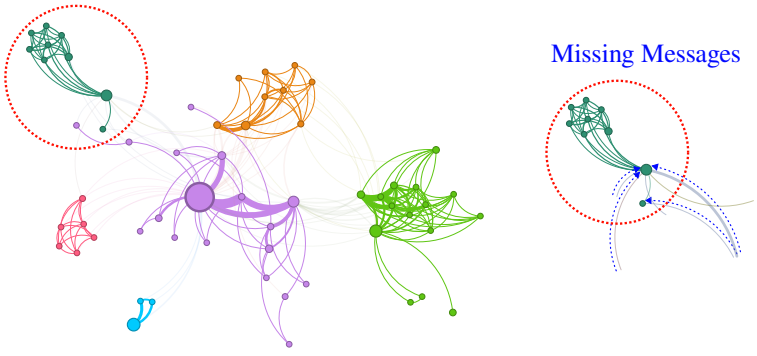
Issues with Cluster-GCN

- Notice that the induced sub-graph removes between-group connections

Cluster-GCN

Issues with Cluster-GCN

- Notice that the induced sub-graph removes between-group connections
- As a consequence, messages from other groups will be lost during message passing, which could hurt the GNN's performance



- Sampled nodes on each sub-graph are not diverse enough to be representative of the entire graph structure

- Sampled nodes on each sub-graph are not diverse enough to be representative of the entire graph structure
- As a consequence, the gradient averaged over the mini-batch becomes unreliable

- Sampled nodes on each sub-graph are not diverse enough to be representative of the entire graph structure
- As a consequence, the gradient averaged over the mini-batch becomes unreliable
  - It means that the gradient fluctuates a lot from one mini-batch to another

- Sampled nodes on each sub-graph are not diverse enough to be representative of the entire graph structure
- As a consequence, the gradient averaged over the mini-batch becomes unreliable
  - It means that the gradient fluctuates a lot from one mini-batch to another
  - In other words, the gradient has a high variance

- Sampled nodes on each sub-graph are not diverse enough to be representative of the entire graph structure
- As a consequence, the gradient averaged over the mini-batch becomes unreliable
  - It means that the gradient fluctuates a lot from one mini-batch to another
  - In other words, the gradient has a high variance
- This issue leads to slow convergence of the SGD algorithm to train the model

Summary: Cluster-GCN

- Cluster-GCN divides the entire set of nodes into small subsets of nodes

Summary: Cluster-GCN

- Cluster-GCN divides the entire set of nodes into small subsets of nodes
- Each subset of nodes induces a sub-graph that is treated as a mini-batch

Summary: Cluster-GCN

- Cluster-GCN divides the entire set of nodes into small subsets of nodes
- Each subset of nodes induces a sub-graph that is treated as a mini-batch
- We apply regular GNN layer-wise message passing over the induced sub-graphs

Summary: Cluster-GCN

- Cluster-GCN divides the entire set of nodes into small subsets of nodes
- Each subset of nodes induces a sub-graph that is treated as a mini-batch
- We apply regular GNN layer-wise message passing over the induced sub-graphs
- Cluster-GCN is more efficient than neighbor sampling since we avoid redundant calculations, especially for deep GNNs

Summary: Cluster-GCN

- Cluster-GCN divides the entire set of nodes into small subsets of nodes
- Each subset of nodes induces a sub-graph that is treated as a mini-batch
- We apply regular GNN layer-wise message passing over the induced sub-graphs
- Cluster-GCN is more efficient than neighbor sampling since we avoid redundant calculations, especially for deep GNNs
- Cluster-GCN could be useful in some datasets, but it leads to systematically biased gradient estimates because of the missing cross-community edges

Simplifying GNN Architecture

- We start from GCN [Kipf and Welling(2017)]

- We start from GCN [Kipf and Welling(2017)]
- We simplify the GCN (SGC) model by removing the non-linear activation from the GCN [Wu et al.(2019)]

- We start from GCN [Kipf and Welling(2017)]
- We simplify the GCN (SGC) model by removing the non-linear activation from the GCN [Wu et al.(2019)]
  - We've observerd that SGC performance on benchmark datasets are not much lower than other more complicated GNNs

- We start from GCN [Kipf and Welling(2017)]
- We simplify the GCN (SGC) model by removing the non-linear activation from the GCN [Wu et al.(2019)]
  - We've observerd that SGC performance on benchmark datasets are not much lower than other more complicated GNNs
  - SGC is extremely scalable by its model design

- We start from GCN [Kipf and Welling(2017)]
- We simplify the GCN (SGC) model by removing the non-linear activation from the GCN [Wu et al.(2019)]
  - We've observerd that SGC performance on benchmark datasets are not much lower than other more complicated GNNs
  - SGC is extremely scalable by its model design
- A very similar idea, called LightGCN, has been successfully applied in large-scale recommendation systems [He et al.(2020)]

- Let's recap the propagation rule of the GCN model:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right),$$

where $\mathbf{H}^{(l)}$ is the embedding matrix at lth layer, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix used as shift operator, and $\mathbf{W}^{(l)}$ is the matrix of learnable parameters

- Let's recap the propagation rule of the GCN model:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right),$$

  where $\mathbf{H}^{(l)}$ is the embedding matrix at lth layer, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix used as shift operator, and $\mathbf{W}^{(l)}$ is the matrix of learnable parameters
- How look like the full mathematical expression if we unfold the propagation rule for l layers?

- Let's recap the propagation rule of the GCN model:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right),$$

where $\mathbf{H}^{(l)}$ is the embedding matrix at lth layer, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix used as shift operator, and $\mathbf{W}^{(l)}$ is the matrix of learnable parameters

- How look like the full mathematical expression if we unfold the propagation rule for l layers?

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\ldots\mathbf{W}^{(l)}\right)$$

- Let's recap the propagation rule of the GCN model:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right),$$

where $\mathbf{H}^{(l)}$ is the embedding matrix at lth layer, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix used as shift operator, and $\mathbf{W}^{(l)}$ is the matrix of learnable parameters

- How look like the full mathematical expression if we unfold the propagation rule for l layers?

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\ldots\mathbf{W}^{(l-1)}\right)\mathbf{W}^{(l)}\right)$$

- Let's recap the propagation rule of the GCN model:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right),$$

where $\mathbf{H}^{(l)}$ is the embedding matrix at lth layer, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix used as shift operator, and $\mathbf{W}^{(l)}$ is the matrix of learnable parameters

- How look like the full mathematical expression if we unfold the propagation rule for l layers?

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\ldots\mathbf{W}^{(l-2)}\right)\mathbf{W}^{(l-1)}\right)\mathbf{W}^{(l)}\right)$$

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\ldots\mathbf{W}^{(l-2)}\right)\mathbf{W}^{(l-1)}\right)\mathbf{W}^{(l)}\right)$$

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\sigma\left(\hat{\mathbf{A}}\ldots\mathbf{W}^{(l-2)}\right)\mathbf{W}^{(l-1)}\right)\mathbf{W}^{(l)}\right)$$

- Let's remove the non-linear activation functions

$$\mathbf{H}^{(l+1)} = \hat{\mathbf{A}} \left(\hat{\mathbf{A}} \left(\hat{\mathbf{A}} \ldots \mathbf{W}^{(l-2)} \right) \mathbf{W}^{(l-1)} \right) \mathbf{W}^{(l)}$$

- Let's remove the non-linear activation functions
- Let's simplify a little bit

$$\mathbf{H}^{(l+1)} = \hat{\mathbf{A}}^{l}\mathbf{X}\mathbf{W}^{(1)}\ldots\mathbf{W}^{(l-2)}\mathbf{W}^{(l-1)}\mathbf{W}^{(l)}$$

- Let's remove the non-linear activation functions
- Let's simplify a little bit
- $\mathbf{X}$ is the input features

$$\mathbf{H}^{(l+1)} = \hat{\mathbf{A}}^{l}\mathbf{X}\mathbf{W}^{(1)}\ldots\mathbf{W}^{(l-2)}\mathbf{W}^{(l-1)}\mathbf{W}^{(l)}$$

- Let's remove the non-linear activation functions
- Let's simplify a little bit
- $\mathbf{X}$ is the input features
- This expression can be simplified $\mathbf{W}^{(1)}\ldots\mathbf{W}^{(l-2)}\mathbf{W}^{(l-1)}\mathbf{W}^{(l)}$

$$\mathbf{H}^{(l+1)} = \hat{\mathbf{A}}^{l}\mathbf{X}\mathbf{W}$$

- Let's remove the non-linear activation functions
- Let's simplify a little bit
- $\mathbf{X}$ is the input features
- This expression can be simplified $\mathbf{W}^{(1)}\ldots\mathbf{W}^{(l-2)}\mathbf{W}^{(l-1)}\mathbf{W}^{(l)}$
- Finally, we just need our final activation (like $\mathrm{softmax}$ for node classification)

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X}\mathbf{W}\right)$$

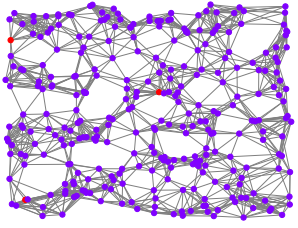
- Let's remove the non-linear activation functions
- Let's simplify a little bit
- $\mathbf{X}$ is the input features
- This expression can be simplified $\mathbf{W}^{(1)} \ldots \mathbf{W}^{(l-2)}\mathbf{W}^{(l-1)}\mathbf{W}^{(l)}$
- Finally, we just need our final activation (like $\mathrm{softmax}$ for node classification)

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

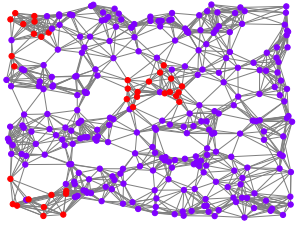
- $\hat{\mathbf{A}}^l \mathbf{X}$ diffuses node embeddings along the graph

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

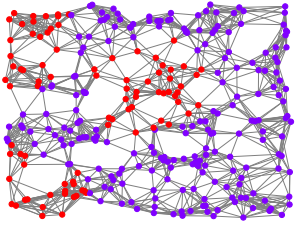
- $\hat{\mathbf{A}}^l \mathbf{X}$ diffuses node embeddings along the graph
- Remember the diffusion sequence in our last lecture, where the shift operator $\mathbf{S}$ is $\hat{\mathbf{A}}$ in this case



$\mathbf{x}^{(0)} = \mathbf{x} = \mathbf{S}^0\mathbf{x}$



$\mathbf{x}^{(1)} = \mathbf{S}\mathbf{x}^{(0)} = \mathbf{S}^1\mathbf{x}$



$\mathbf{x}^{(2)} = \mathbf{S}\mathbf{x}^{(1)} = \mathbf{S}^2\mathbf{x}$

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- We can efficiently compute $\hat{\mathbf{A}}^l \mathbf{X}$:

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- We can efficiently compute $\hat{\mathbf{A}}^l \mathbf{X}$:
  - Start from the input feature matrix $\mathbf{X}$

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- We can efficiently compute $\hat{\mathbf{A}}^l \mathbf{X}$:
  - Start from the input feature matrix $\mathbf{X}$
  - Apply $\mathbf{X} \leftarrow \hat{\mathbf{A}}\mathbf{X}$ for l times

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- We can efficiently compute $\hat{\mathbf{A}}^l \mathbf{X}$:
  - Start from the input feature matrix $\mathbf{X}$
  - Apply $\mathbf{X} \leftarrow \hat{\mathbf{A}}\mathbf{X}$ for l times
  - Notice that $\hat{\mathbf{A}}$ is usually sparse so we can leverage sparse matrix multiplications packages

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- We can efficiently compute $\hat{\mathbf{A}}^l \mathbf{X}$:
  - Start from the input feature matrix $\mathbf{X}$
  - Apply $\mathbf{X} \leftarrow \hat{\mathbf{A}}\mathbf{X}$ for l times
  - Notice that $\hat{\mathbf{A}}$ is usually sparse so we can leverage sparse matrix multiplications packages
- The matrix of learnable parameters $\mathbf{W}$ acts as a linear classifier over the diffused node embeddings $\hat{\mathbf{A}}^l \mathbf{X}$

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X}\mathbf{W}\right)$$

- The only trick that we're missing to get the original SGC is to add self-loops to the graph as in GCN, *i.e.*, $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- The only trick that we're missing to get the original SGC is to add self-loops to the graph as in GCN, *i.e.*, $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$
- SGC assumes the input node embeddings $\mathbf{X}$ to be given as features

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- The only trick that we're missing to get the original SGC is to add self-loops to the graph as in GCN, *i.e.*, $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$
- SGC assumes the input node embeddings $\mathbf{X}$ to be given as features
- It means that the input matrix $\mathbf{X}$ is fixed rather than learned

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- The only trick that we're missing to get the original SGC is to add self-loops to the graph as in GCN, *i.e.*, $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$
- SGC assumes the input node embeddings $\mathbf{X}$ to be given as features
- It means that the input matrix $\mathbf{X}$ is fixed rather than learned
- As a consequence $\hat{\mathbf{A}}^l \mathbf{X}$ needs to be computed only once

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- The only trick that we're missing to get the original SGC is to add self-loops to the graph as in GCN, *i.e.*, $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$
- SGC assumes the input node embeddings $\mathbf{X}$ to be given as features
- It means that the input matrix $\mathbf{X}$ is fixed rather than learned
- As a consequence $\hat{\mathbf{A}}^l \mathbf{X}$ needs to be computed only once
- SGC is highly efficient since you need to compute $\hat{\mathbf{A}}^l \mathbf{X}$ as a pre-processing step, and you can even do it in the CPU!

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X}\mathbf{W}\right)$$

- Let $\tilde{\mathbf{X}} = \hat{\mathbf{A}}^l \mathbf{X}$ be the pre-processed feature matrix

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \text{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- Let $\tilde{\mathbf{X}} = \hat{\mathbf{A}}^l \mathbf{X}$ be the pre-processed feature matrix
- Each row stores the pre-processed feature for each node

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

- Let $\tilde{\mathbf{X}} = \hat{\mathbf{A}}^l \mathbf{X}$ be the pre-processed feature matrix
- Each row stores the pre-processed feature for each node
- We come back to the fundamental setting of classification in Machine Learning, we have $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_N]^\mathsf{T}$, and some label $\mathbf{Y}$

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

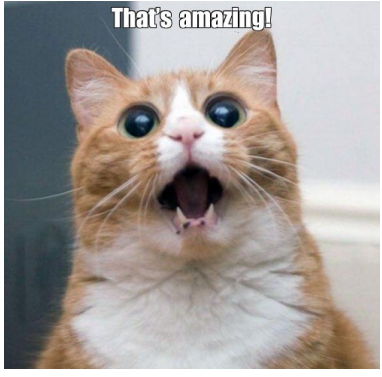
- Let $\tilde{\mathbf{X}} = \hat{\mathbf{A}}^l \mathbf{X}$ be the pre-processed feature matrix
- Each row stores the pre-processed feature for each node
- We come back to the fundamental setting of classification in Machine Learning, we have $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_N]^\mathsf{T}$, and some label $\mathbf{Y}$
- You can now use your favorite scalable ML model (*e.g.* linear model, MLP, etc)

$$\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \boldsymbol{\Theta}) = \mathrm{softmax}\left(\hat{\mathbf{A}}^l \mathbf{X} \mathbf{W}\right)$$

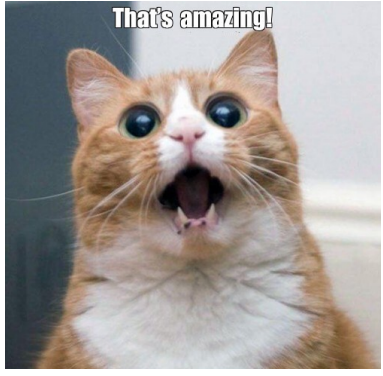
- Let $\tilde{\mathbf{X}} = \hat{\mathbf{A}}^l \mathbf{X}$ be the pre-processed feature matrix
- Each row stores the pre-processed feature for each node
- We come back to the fundamental setting of classification in Machine Learning, we have $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_N]^\mathsf{T}$, and some label $\mathbf{Y}$
- You can now use your favorite scalable ML model (*e.g.* linear model, MLP, etc)
- You can even do regular mini-batching if required because you already have the diffused information from your graph inside $\tilde{\mathbf{X}}$

- SGC empirically shows learning a linear model over $\tilde{\mathbf{X}}$ often gives performance comparable to GCN!!! (I confirm this)

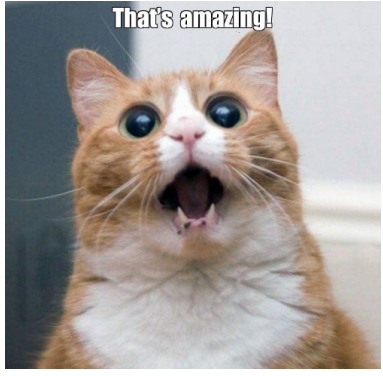
- SGC empirically shows learning a linear model over $\tilde{\mathbf{X}}$ often gives performance comparable to GCN!!! (I confirm this)



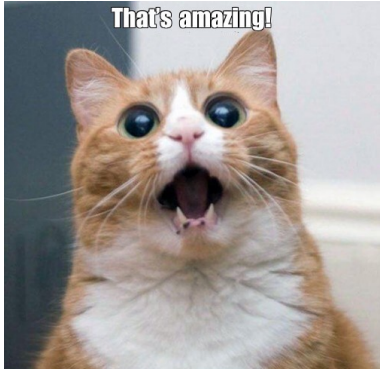
- SGC empirically shows learning a linear model over $\tilde{\mathbf{X}}$ often gives performance comparable to GCN!!! (I confirm this)
- SGC is much more efficient than other methods for scaling GNNs like neighbor sampling



- SGC empirically shows learning a linear model over $\tilde{\mathbf{X}}$ often gives performance comparable to GCN!!! (I confirm this)
- SGC is much more efficient than other methods for scaling GNNs like neighbor sampling
- SGC computes $\tilde{\mathbf{X}}$ once at the beginning, this is usually a sparse matrix multiplication that can be performed efficiently on the CPU



- SGC empirically shows learning a linear model over $\tilde{\mathbf{X}}$ often gives performance comparable to GCN!!! (I confirm this)
- SGC is much more efficient than other methods for scaling GNNs like neighbor sampling
- SGC computes $\tilde{\mathbf{X}}$ once at the beginning, this is usually a sparse matrix multiplication that can be performed efficiently on the CPU
- Once $\tilde{\mathbf{X}}$ is obtained, getting an embedding for the ith nodes takes constant time!, just look up the ith row of $\tilde{\mathbf{X}}$. You don't need to build the computational graph



- In SGC we're simplifying our GNN for the sake of scalability, and this comes with a price we have to pay

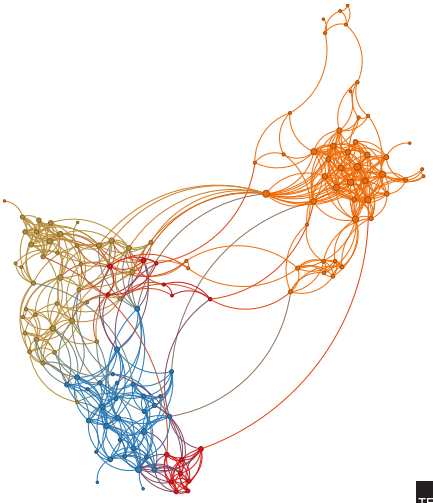
- In SGC we're simplifying our GNN for the sake of scalability, and this comes with a price we have to pay
- Compared to original GNN models, SGC's expressive power is limited due to the lack of non-linearity in generating node embeddings

- In SGC we're simplifying our GNN for the sake of scalability, and this comes with a price we have to pay
- Compared to original GNN models, SGC's expressive power is limited due to the lack of non-linearity in generating node embeddings
- So why SGC works comparably to the original GNNs despite being less expressive?

Simplifying GNN Architecture

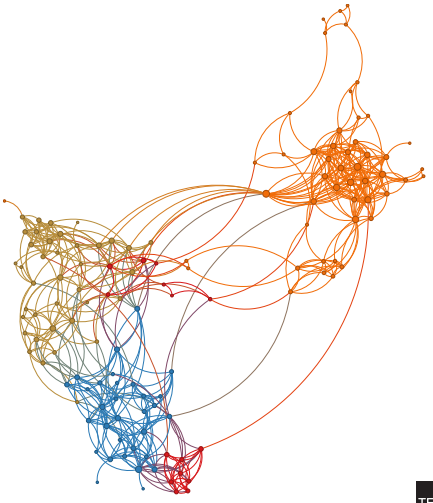
Homophily

- Remember when we talked about homophily, many node classification tasks exhibit homophily structure, *i.e.*, nodes connected by edges tend to share the same target labels



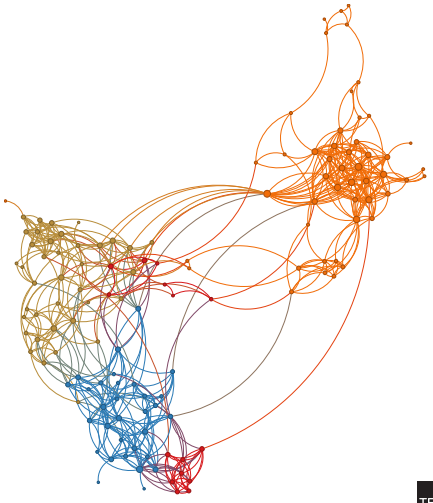
Simplifying GNN Architecture
Homophily

- Remember when we talked about homophily, many node classification tasks exhibit homophily structure, *i.e.*, nodes connected by edges tend to share the same target labels
- Example 1: For paper category classification in citation networks, two papers tend to share the same category if one cites another



Simplifying GNN Architecture
Homophily

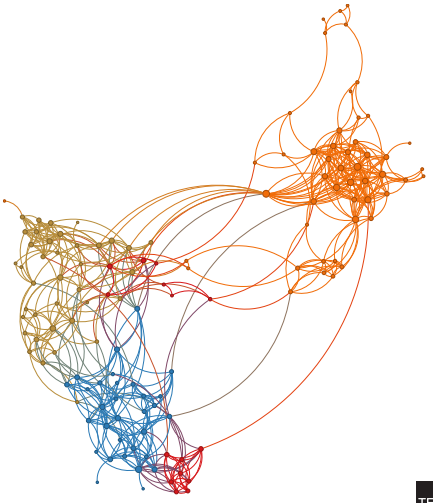
- Remember when we talked about homophily, many node classification tasks exhibit homophily structure, *i.e.*, nodes connected by edges tend to share the same target labels
- Example 1: For paper category classification in citation networks, two papers tend to share the same category if one cites another
- Example 2: For movie recommendations for users in social networks, two users tend to like the same movie if they are friends



Simplifying GNN Architecture

When does SGC work?

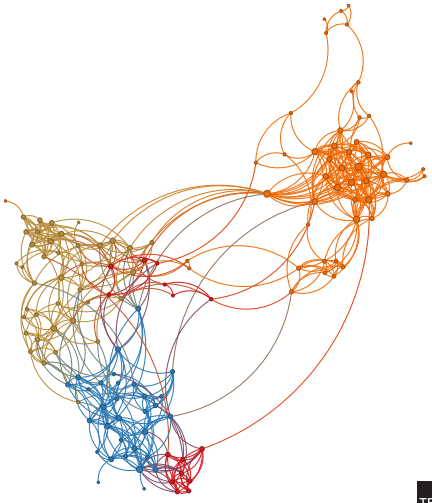
- Recall the pre-processing step of SGC: do $\mathbf{X} \leftarrow \tilde{\mathbf{A}}\mathbf{X}$ for l times



Simplifying GNN Architecture

When does SGC work?

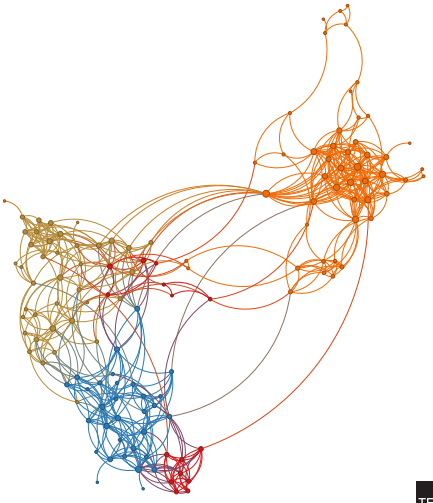
- Recall the pre-processing step of SGC: do $\mathbf{X} \leftarrow \tilde{\mathbf{A}}\mathbf{X}$ for l times
- From the diffusion sequence we know that the pre-processed features are obtained by iteratively averaging the neighboring node features of each node



Simplifying GNN Architecture

When does SGC work?

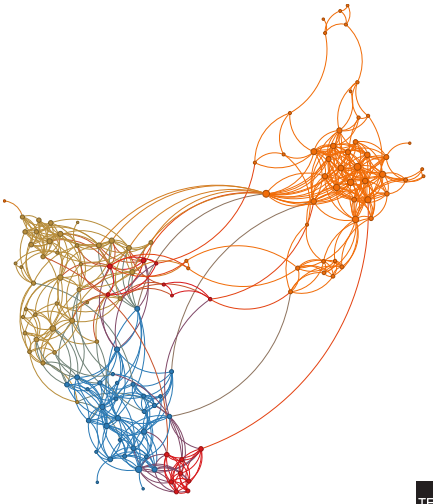
- Recall the pre-processing step of SGC: do $\mathbf{X} \leftarrow \tilde{\mathbf{A}}\mathbf{X}$ for l times
- From the diffusion sequence we know that the pre-processed features are obtained by iteratively averaging the neighboring node features of each node
- It means that nodes connected by edges tend to have similar pre-processed features



Simplifying GNN Architecture

When does SGC work?

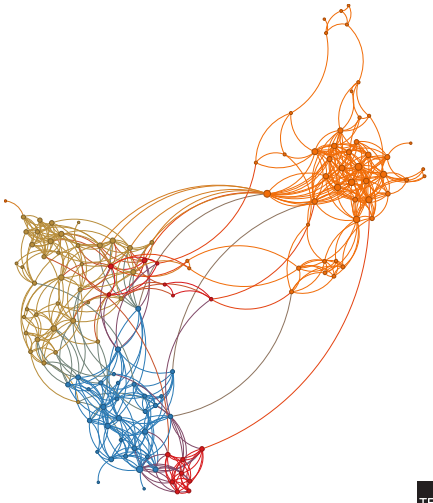
- Recall the pre-processing step of SGC: do $\mathbf{X} \leftarrow \tilde{\mathbf{A}}\mathbf{X}$ for l times
- From the diffusion sequence we know that the pre-processed features are obtained by iteratively averaging the neighboring node features of each node
- It means that nodes connected by edges tend to have similar pre-processed features
- Therefore, SGC tends to predict the same labels for nodes connected by edges



Simplifying GNN Architecture

When does SGC work?

- Recall the pre-processing step of SGC: do $\mathbf{X} \leftarrow \tilde{\mathbf{A}}\mathbf{X}$ for l times
- From the diffusion sequence we know that the pre-processed features are obtained by iteratively averaging the neighboring node features of each node
- It means that nodes connected by edges tend to have similar pre-processed features
- Therefore, SGC tends to predict the same labels for nodes connected by edges
- This aligns well with the homophily assumption in GNNs.



- SGC removes the non-linearities in GCNs, and thus it reduces to a simple pre-processing of the node features.

- SGC removes the non-linearities in GCNs, and thus it reduces to a simple pre-processing of the node features.
- Once the pre-processed features are obtained, regular mini-batch SGD can be directly applied (we already did the "message passing" part in the pre-processing step)

- SGC removes the non-linearities in GCNs, and thus it reduces to a simple pre-processing of the node features.
- Once the pre-processed features are obtained, regular mini-batch SGD can be directly applied (we already did the "message passing" part in the pre-processing step)
- SGC works well in node classification benchmarks because it aligns well with graph homophily in real-world prediction tasks

Main takeaways

- Graphs in modern applications are extremely large

Main takeaways

- Graphs in modern applications are extremely large
- Regular GNN algorithms don't scale up to these massive graphs

Main takeaways

- Graphs in modern applications are extremely large
- Regular GNN algorithms don't scale up to these massive graphs
- Regular mini-batching is not useful in these cases since we end up with isolated nodes

Main takeaways

- Graphs in modern applications are extremely large
- Regular GNN algorithms don't scale up to these massive graphs
- Regular mini-batching is not useful in these cases since we end up with isolated nodes
- We can create computational graphs for each node and do regular message passing. However, we need to sample some nodes at each layer with neighbor sampling, otherwise, the size of the computation graph grows exponentially

Main takeaways

- The neighbor sampling approach has a lot of redundant computations, so we could instead create clustered sub-graphs to apply GNN message passing there with Cluster-GCN

Main takeaways

- The neighbor sampling approach has a lot of redundant computations, so we could instead create clustered sub-graphs to apply GNN message passing there with Cluster-GCN
- Cluster-GCN loose information of between-groups link, leading to biased gradient estimates

Main takeaways

- The neighbor sampling approach has a lot of redundant computations, so we could instead create clustered sub-graphs to apply GNN message passing there with Cluster-GCN
- Cluster-GCN loose information of between-groups link, leading to biased gradient estimates
- We can design an extremely efficient GNN architecture called SGC by removing the non-linearities between layers in the regular GCN

Main takeaways

- The neighbor sampling approach has a lot of redundant computations, so we could instead create clustered sub-graphs to apply GNN message passing there with Cluster-GCN
- Cluster-GCN loose information of between-groups link, leading to biased gradient estimates
- We can design an extremely efficient GNN architecture called SGC by removing the non-linearities between layers in the regular GCN
- SGC works surprisingly well in several node classification benchmarks, but we need the homophily assumption to be true

Thank you!

Jhony H. Giraldo: jhony.giraldo@telecom-paris.fr
https://jhonygiraldo.github.io/

References I

X. He et al.
LightGCN: Simplifying and powering graph convolution network for recommendation.
In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

T. N. Kipf and M. Welling.
Semi-supervised classification with graph convolutional networks.
In *International Conference on Learning Representations*, 2017.

P. Veličković et al.
Graph attention networks.
In *International Conference on Learning Representations*, 2018.

References II


F. Wu et al.
Simplifying graph convolutional networks.
In *International Conference on Machine Learning*, 2019.