Stanford CS224W: A General Perspective on Graph Neural Networks

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

- Colab 1 due this Thursday
- Colab 2 out on Thursday (same day)
- Thank you for the suggestion for in-person & group office hours!

We will be hosting 1 group OH in person a week. CA Zhuoyi Huang will lead these.
**Recap: Deep Graph Encoders**

Output: Node embeddings. Also, we can embed subgraphs, graphs.
Idea: Node’s neighborhood defines a computation graph

- Determine node computation graph
- Propagate and transform information

Learn how to propagate information across the graph to compute node features
Intuition: Nodes aggregate information from their neighbors using neural networks.
**Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!
Today: A General GNN Framework

(1) Message
(2) Aggregation
(3) Layer connectivity
(4) Graph augmentation
(5) Learning objective

J. You, R. Ying, J. Leskovec. *Design Space of Graph Neural Networks*, NeurIPS 2020
A General GNN Framework (1)

GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

(1) Message

(2) Aggregation

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections
A General GNN Framework (3)

Idea: Raw input graph $\neq$ computational graph

- Graph feature augmentation
- Graph structure augmentation

(4) Graph augmentation
How do we train a GNN

- Supervised/Unsupervised objectives
- Node/Edge/Graph level objectives
(We will discuss all of these later in class)
GNN Framework: Summary

(1) Message

(2) Aggregation

(3) Layer connectivity

(4) Graph augmentation

(5) Learning objective
Stanford CS224W: A Single Layer of a GNN
A GNN Layer

GNN Layer = Message + Aggregation
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

J. You, R. Ying, J. Leskovec. *Design Space of Graph Neural Networks*, NeurIPS 2020
### Idea of a GNN Layer:

- Compress a set of vectors into a single vector
- **Two-step process:**
  - (1) Message
  - (2) Aggregation

#### Diagram:

- **Input node embedding** $\mathbf{h}_v^{(l-1)}$, $\mathbf{h}_u^{(l-1)}$ (from node itself + neighboring nodes)
- **Output node embedding** $\mathbf{h}_v^{(l)}$
- **$l$-th GNN Layer**

Node $v$
(1) Message computation

Message function: \( m_u^{(l)} = MSG^{(l)} \left( h_u^{(l-1)} \right) \)

- **Intuition:** Each node will create a message, which will be sent to other nodes later.
- **Example:** A Linear layer \( m_u^{(l)} = W^{(l)} h_u^{(l-1)} \)
  - Multiply node features with weight matrix \( W^{(l)} \)

(2) Aggregation

(1) Message

Node \( v \)

(2) Aggregation

INPUT GRAPH

TARGET NODE

A

B

C

D

E

F

2/16/2023

(2) Aggregation

- **Intuition:** Each node will aggregate the messages from node $v$'s neighbors

$$h_v^{(l)} = AGG^{(l)} \left( \{m_u^{(l)}, u \in N(v) \} \right)$$

- **Example:** $\text{Sum}(\cdot)$, $\text{Mean}(\cdot)$ or $\text{Max}(\cdot)$ aggregator

$$h_v^{(l)} = \text{Sum}(\{m_u^{(l)}, u \in N(v)\})$$
**Issue:** Information from node \( v \) itself **could get lost**

- Computation of \( h_v^{(l)} \) does not directly depend on \( h_v^{(l-1)} \)

**Solution:** Include \( h_v^{(l-1)} \) when computing \( h_v^{(l)} \)

(1) **Message:** compute message from node \( v \) itself

- Usually, a **different message computation** will be performed

\[
\begin{align*}
\mathbf{m}_u^{(l)} &= \mathbf{W}^{(l)} h_u^{(l-1)} \\
\mathbf{m}_v^{(l)} &= \mathbf{B}^{(l)} h_v^{(l-1)}
\end{align*}
\]

(2) **Aggregation:** After aggregating from neighbors, we can aggregate the message from node \( v \) itself

- Via **concatenation** or **summation**

\[
h_v^{(l)} = \text{CONCAT} \left( \text{AGG} \left( \{ \mathbf{m}_u^{(l)}, u \in N(v) \} \right), \mathbf{m}_v^{(l)} \right)
\]

Then aggregate from node itself

First aggregate from neighbors
Putting things together:

1. **Message**: each node computes a message
   \[ m_u^{(l)} = \text{MSG}^{(l)} \left( h_u^{(l-1)} \right), \, u \in \{ N(v) \cup v \} \]

2. **Aggregation**: aggregate messages from neighbors
   \[ h_v^{(l)} = \text{AGG}^{(l)} \left( \{ m_u^{(l)}, \, u \in N(v) \}, \, m_v^{(l)} \right) \]

**Nonlinearity (activation)**: Adds expressiveness

- Often written as \( \sigma(\cdot) \): ReLU(\cdot), Sigmoid(\cdot), ...
- Can be added to message or aggregation
(1) Graph Convolutional Networks (GCN)

\[
h_{v}^{(l)} = \sigma \left( W^{(l)} \sum_{u \in N(v)} \frac{h_{u}^{(l-1)}}{|N(v)|} \right)
\]

- How to write this as Message + Aggregation?

\[
h_{v}^{(l)} = \sigma \left( \sum_{u \in N(v)} \frac{W^{(l)} h_{u}^{(l-1)}}{|N(v)|} \right)
\]
(1) Graph Convolutional Networks (GCN)

\[ h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} W^{(l)} \frac{h^{(l-1)}_u}{|N(v)|} \right) \]

- **Message:**
  - Each Neighbor: \( m^{(l)}_u = \frac{1}{|N(v)|} W^{(l)} h^{(l-1)}_u \)

- **Aggregation:**
  - Sum over messages from neighbors, then apply activation
  - \( h^{(l)}_v = \sigma \left( \text{Sum} \left( \{ m^{(l)}_u, u \in N(v) \} \right) \right) \)

Normalized by node degree
(In the GCN paper they use a slightly different normalization)

In GCN graph is assumed to have self-edges that are included in the summation.
(2) GraphSAGE

\[ h_v^{(l)} = \sigma \left( W^{(l)} \cdot \text{ CONCAT} \left( h_v^{(l-1)} , \text{ AGG} \left( \{ h_u^{(l-1)} \} \right), \forall u \in N(v) \right) \right) \]

- **How to write this as Message + Aggregation?**
  - **Message** is computed within the \( \text{AGG}(\cdot) \)
  - **Two-stage aggregation**
    - **Stage 1:** Aggregate from node neighbors
      \[ h_{N(v)}^{(l)} \leftarrow \text{AGG} \left( \{ h_u^{(l-1)} \}, \forall u \in N(v) \right) \]
    - **Stage 2:** Further aggregate over the node itself
      \[ h_v^{(l)} \leftarrow \sigma \left( W^{(l)} \cdot \text{ CONCAT} \left( h_v^{(l-1)} , h_{N(v)}^{(l)} \right) \right) \]
GraphSAGE Neighbor Aggregation

- **Mean:** Take a weighted average of neighbors

\[
AGG = \sum_{u \in N(v)} \frac{h_{u}^{(l-1)}}{|N(v)|}
\]

- **Pool:** Transform neighbor vectors and apply symmetric vector function Mean(·) or Max(·)

\[
AGG = \text{Mean}(\{\text{MLP}(h_{u}^{(l-1)}), \forall u \in N(v)\})
\]

- **LSTM:** Apply LSTM to reshuffled of neighbors

\[
AGG = \text{LSTM}([h_{u}^{(l-1)}, \forall u \in \pi(N(v))])
\]
**GraphSAGE: L2 Normalization**

- **$\ell_2$ Normalization:**
  - **Optional:** Apply $\ell_2$ normalization to $h_v^{(l)}$ at every layer
  
  $h_v^{(l)} \leftarrow \frac{h_v^{(l)}}{\|h_v^{(l)}\|_2} \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ ($\ell_2$-norm)

- Without $\ell_2$ normalization, the embedding vectors have different scales ($\ell_2$-norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After $\ell_2$ normalization, all vectors will have the same $\ell_2$-norm
### Classical GNN Layers: GAT (1)

- **(3) Graph Attention Networks**

\[
    h_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)} \right)
\]

- **In GCN / GraphSAGE**
  - \( \alpha_{vu} = \frac{1}{|N(v)|} \) is the **weighting factor (importance)** of node \( u \)'s message to node \( v \)
  - \( \Rightarrow \alpha_{vu} \) is defined **explicitly** based on the structural properties of the graph (node degree)
  - \( \Rightarrow \) All neighbors \( u \in N(v) \) are equally important to node \( v \)
Graph Attention Networks

$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)} \right)$

Not all node’s neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** $\alpha_{vu}$ focuses on the important parts of the input data and fades out the rest.
  - **Idea**: the NN should devote more computing power on that small but important part of the data.
  - Which part of the data is more important depends on the context and is learned through training.
Can we do better than simple neighborhood aggregation?

Can we let weighting factors $\alpha_{vu}$ to be learned?

- **Goal:** Specify arbitrary importance to different neighbors of each node in the graph
- **Idea:** Compute embedding $h^{(l)}_v$ of each node in the graph following an attention strategy:
  - Nodes attend over their neighborhoods’ message
  - Implicitly specifying different weights to different nodes in a neighborhood

[Velickovic et al., ICLR 2018; Vaswani et al., NIPS 2017]
Let $\alpha_{vu}$ be computed as a byproduct of an attention mechanism $a$:

1. Let $a$ compute attention coefficients $e_{vu}$ across pairs of nodes $u, v$ based on their messages:
   \[ e_{vu} = a(W^{(l)}h_{u}^{(l-1)}, W^{(l)}h_{v}^{(l-1)}) \]
   - $e_{vu}$ indicates the importance of $u'$s message to node $v$

2. $e_{AB} = a(W^{(l)}h_{A}^{(l-1)}, W^{(l)}h_{B}^{(l-1)})$
Attention Mechanism (2)

- **Normalize** $e_{vu}$ into the **final attention weight** $\alpha_{vu}$
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:
    \[
    \alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}
    \]

- **Weighted sum** based on the **final attention weight** $\alpha_{vu}$
  
  \[
  h_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)})
  \]

Weighted sum using $\alpha_{AB}, \alpha_{AC}, \alpha_{AD}$:

\[
    h_A^{(l)} = \sigma(\alpha_{AB} W^{(l)} h_B^{(l-1)} + \alpha_{AC} W^{(l)} h_C^{(l-1)} + \alpha_{AD} W^{(l)} h_D^{(l-1)})
\]
What is the form of attention mechanism $a$?

- The approach is agnostic to the choice of $a$
  - E.g., use a simple single-layer neural network
    - $a$ have trainable parameters (weights in the Linear layer)

Parameters of $a$ are trained jointly:

- Learn the parameters together with weight matrices (i.e., other parameter of the neural net $W^{(l)}$) in an end-to-end fashion

\[ e_{AB} = a \left( W^{(l)} h_A^{(l-1)}, W^{(l)} h_B^{(l-1)} \right) \]
\[ = \text{Linear} \left( \text{Concat} \left( W^{(l)} h_A^{(l-1)}, W^{(l)} h_B^{(l-1)} \right) \right) \]
Multi-head attention: Stabilizes the learning process of attention mechanism

- Create multiple attention scores (each replica with a different set of parameters):
  \[ h_v^{(l)}[1] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^1 W^{(l)} h_u^{(l-1)}) \]
  \[ h_v^{(l)}[2] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 W^{(l)} h_u^{(l-1)}) \]
  \[ h_v^{(l)}[3] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 W^{(l)} h_u^{(l-1)}) \]

- Outputs are aggregated:
  - By concatenation or summation
  \[ h_v^{(l)} = AGG(h_v^{(l)}[1], h_v^{(l)}[2], h_v^{(l)}[3]) \]
Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying different importance values ($\alpha_{vu}$) to different neighbors.

- **Computationally efficient:**
  - Computation of attentional coefficients can be parallelized across all edges of the graph.
  - Aggregation may be parallelized across all nodes.

- **Storage efficient:**
  - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored.
  - **Fixed** number of parameters, irrespective of graph size.

- **Localized:**
  - Only attends over local network neighborhoods.

- **Inductive capability:**
  - It is a shared edge-wise mechanism.
  - It does not depend on the global graph structure.
Stanford CS224W: GNN Layers in Practice

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
In practice, these classic GNN layers are a great starting point.

- We can often get better performance by considering a general GNN layer design.
- Concretely, we can include modern deep learning modules that proved to be useful in many domains.
Many modern deep learning modules can be incorporated into a GNN layer

- **Batch Normalization:**
  - Stabilize neural network training

- **Dropout:**
  - Prevent overfitting

- **Attention/Gating:**
  - Control the importance of a message

- **More:**
  - Any other useful deep learning modules
Batch Normalization

- **Goal**: Stabilize neural networks training
- **Idea**: Given a batch of inputs (node embeddings)
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance

**Input**: \( \mathbf{X} \in \mathbb{R}^{N \times D} \)
\( N \) node embeddings

**Trainable Parameters**: \( \gamma, \beta \in \mathbb{R}^{D} \)

**Output**: \( \mathbf{Y} \in \mathbb{R}^{N \times D} \)
Normalized node embeddings

**Step 1**: Compute the mean and variance over \( N \) embeddings

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

**Step 2**: Normalize the feature using computed mean and variance

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}
\]

\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]
Goal: Regularize a neural net to prevent overfitting.

Idea:

- **During training**: with some probability $p$, randomly set neurons to zero (turn off)
- **During testing**: Use all the neurons for computation
In GNN, Dropout is applied to the linear layer in the message function.

A simple message function with linear layer:

\[ m_u^{(l)} = W^{(l)} h_u^{(l-1)} \]
Activation (Non-linearity)

Apply activation to $i$-th dimension of embedding $x$

- **Rectified linear unit (ReLU)**
  \[
  \text{ReLU}(x_i) = \max(x_i, 0)
  \]
  - Most commonly used

- **Sigmoid**
  \[
  \sigma(x_i) = \frac{1}{1 + e^{-x_i}}
  \]
  - Used only when you want to restrict the range of your embeddings

- **Parametric ReLU**
  \[
  \text{PReLU}(x_i) = \max(x_i, 0) + a_i \min(x_i, 0)
  \]
  - $a_i$ is a trainable parameter
  - Empirically performs better than ReLU
Summary: Modern deep learning modules can be included into a GNN layer for better performance.

Designing novel GNN layers is still an active research frontier!

Suggested resources: You can explore diverse GNN designs or try out your own ideas in GraphGym.
Stanford CS224W: Stacking Layers of a GNN

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity
How to construct a Graph Neural Network?

- The standard way: Stack GNN layers sequentially
- Input: Initial raw node feature $x_v$
- Output: Node embeddings $h_v^{(L)}$ after $L$ GNN layers

$h_v^{(0)} = x_v$

GNN Layer

$h_v^{(1)}$

GNN Layer

$h_v^{(2)}$

GNN Layer

$h_v^{(3)}$
The Issue of stacking many GNN layers
- GNN suffers from the over-smoothing problem

The over-smoothing problem: all the node embeddings converge to the same value
- This is bad because we want to use node embeddings to differentiate nodes

Why does the over-smoothing problem happen?
**Receptive field**: the set of nodes that determine the embedding of a node of interest

- In a $K$-layer GNN, each node has a receptive field of $K$-hop neighborhood.

---

**Receptive field for 1-layer GNN**

- Node of interest
- Receptive field
- Other nodes

**Receptive field for 2-layer GNN**

- Node of interest
- Receptive field
- Other nodes

**Receptive field for 3-layer GNN**

- Node of interest
- Receptive field
- Other nodes
Receptive field overlap for two nodes

- The shared neighbors quickly grows when we increase the number of hops (num of GNN layers)

1-hop neighbor overlap
Only 1 node

2-hop neighbor overlap
About 20 nodes

3-hop neighbor overlap
Almost all the nodes!
We can explain over-smoothing via the notion of receptive field

- We knew the embedding of a node is determined by its receptive field
  - If two nodes have highly-overlapped receptive fields, then their embeddings are highly similar

Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem

Next: how do we overcome over-smoothing problem?
What do we learn from the over-smoothing problem?

Lesson 1: Be cautious when adding GNN layers

- Unlike neural networks in other domains (CNN for image classification), adding more GNN layers do not always help
- **Step 1:** Analyze the necessary receptive field to solve your problem. E.g., by computing the diameter of the graph
- **Step 2:** Set number of GNN layers $L$ to be a bit more than the receptive field we like. **Do not set $L$ to be unnecessarily large!**

Question: How to enhance the expressive power of a GNN, if the number of GNN layers is small?
How to make a shallow GNN more expressive?

**Solution 1**: Increase the expressive power within each GNN layer

- In our previous examples, each transformation or aggregation function only include one linear layer
- We can make aggregation / transformation become a deep neural network!

If needed, each box could include a 3-layer MLP

(1) Transformation

(2) Aggregation
How to make a shallow GNN more expressive?

Solution 2: Add layers that do not pass messages

- A GNN does not necessarily only contain GNN layers
  - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**

**Pre-processing layers**: Important when encoding node features is necessary. E.g., when nodes represent images/text

**Post-processing layers**: Important when reasoning / transformation over node embeddings are needed E.g., graph classification, knowledge graphs

In practice, adding these layers works great!
What if my problem still requires many GNN layers?

Lesson 2: Add skip connections in GNNs

- Observation from over-smoothing: Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- Solution: We can increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNN

Idea of skip connections:
Before adding shortcuts: \( F(x) \)
After adding shortcuts: \( F(x) + x \)

He et al. Deep Residual Learning for Image Recognition, CVPR 2015
Why do skip connections work?

- **Intuition:** Skip connections create a mixture of models
- $N$ skip connections $\rightarrow 2^N$ possible paths
- Each path could have up to $N$ modules
- We automatically get a mixture of shallow GNNs and deep GNNs

**Path 1:** include this module

(a) Conventional 3-block residual network

**Path 2:** skip this module

(b) Unraveled view of (a)

Veit et al. Residual Networks Behave Like Ensembles of Relatively Shallow Networks, ArXiv 2016
Example: GCN with Skip Connections

- **A standard GCN layer**

  \[ h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} \frac{W^{(l)} h^{(l-1)}_u}{|N(v)|} \right) \]

  This is our \( F(x) \)

- **A GCN layer with skip connection**

  \[ h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} \frac{W^{(l)} h^{(l-1)}_u}{|N(v)|} \right) + h^{(l-1)}_v \]

  \[ F(x) + x \]
- **Other options:** Directly skip to the last layer
  - The final layer directly aggregates from the **all the node embeddings** in the previous layers
Stanford CS224W: Graph Manipulation in GNNs

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu
Idea: Raw input graph $\neq$ computational graph

- Graph feature augmentation
- Graph structure manipulation

(4) Graph manipulation
Our assumption so far has been

- **Raw input graph = computational graph**

Reasons for breaking this assumption

- **Feature level:**
  - The input graph **lacks features** → feature augmentation

- **Structure level:**
  - The graph is **too sparse** → inefficient message passing
  - The graph is **too dense** → message passing is too costly
  - The graph is **too large** → cannot fit the computational graph into a GPU

- It’s just **unlikely that the input graph happens to be the optimal computation graph** for embeddings
Graph Manipulation Approaches

- **Graph Feature manipulation**
  - The input graph lacks features → feature augmentation

- **Graph Structure manipulation**
  - The graph is too sparse → Add virtual nodes / edges
  - The graph is too dense → Sample neighbors when doing message passing
  - The graph is too large → Sample subgraphs to compute embeddings
    - Will cover later in lecture: Scaling up GNNs
Why do we need feature augmentation?

- (1) Input graph does not have node features
  - This is common when we only have the adj. matrix
- Standard approaches:
- a) Assign constant values to nodes
Why do we need feature augmentation?

- (1) Input graph does not have node features
  - This is common when we only have the adj. matrix
- Standard approaches:
  - b) Assign unique IDs to nodes
    - These IDs are converted into one-hot vectors

One-hot vector for node with ID=5

\[ [0, 0, 0, 0, 1, 0] \]

Total number of IDs = 6
## Feature Augmentation on Graphs

### Feature augmentation: constant vs. one-hot

<table>
<thead>
<tr>
<th></th>
<th>Constant node feature</th>
<th>One-hot node feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressive power</strong></td>
<td><strong>Medium.</strong> All the nodes are identical, but GNN can still learn from the graph structure</td>
<td><strong>High.</strong> Each node has a unique ID, so node-specific information can be stored</td>
</tr>
<tr>
<td><strong>Inductive learning (Generalize to unseen nodes)</strong></td>
<td><strong>High.</strong> Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN</td>
<td><strong>Low.</strong> Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn’t know how to embed unseen IDs</td>
</tr>
<tr>
<td><strong>Computational cost</strong></td>
<td><strong>Low.</strong> Only 1 dimensional feature</td>
<td><strong>High.</strong> High dimensional feature, cannot apply to large graphs</td>
</tr>
<tr>
<td><strong>Use cases</strong></td>
<td>Any graph, inductive settings (generalize to new nodes)</td>
<td>Small graph, transductive settings (no new nodes)</td>
</tr>
</tbody>
</table>
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Example: Cycle count feature
  - Can GNN learn the length of a cycle that $v_1$ resides in?
  - Unfortunately, no

$v_1$ resides in a cycle with length 3

$v_1$ resides in a cycle with length 4
- $v_1$ cannot differentiate which graph it resides in
  - Because all the nodes in the graph have degree of 2
  - The computational graphs will be the same binary tree

$v_1$ resides in a cycle with length 3

$v_1$ resides in a cycle with length 4

$v_1$ resides in a cycle with infinite length

The computational graphs for node $v_1$ are always the same
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- **Solution:**
  - We can use *cycle count* as augmented node features

Augmented node feature for $v_1$

- $[0, 0, 0, 1, 0, 0]$  
  - $v_1$ resides in a cycle with length 3

Augmented node feature for $v_1$

- $[0, 0, 0, 0, 1, 0]$  
  - $v_1$ resides in a cycle with length 4
Why do we need feature augmentation?

- (2) Certain structures are hard to learn by GNN
- Other commonly used augmented features:
  - Clustering coefficient
  - PageRank
  - Centrality
  - ...
- Any feature we have introduced in Lecture 2 can be used!
Add Virtual Nodes / Edges

- **Motivation:** Augment sparse graphs
- **(1) Add virtual edges**
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix $A$ for GNN computation, use $A + A^2$

- **Use cases:** Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph
Add Virtual Nodes / Edges

- **Motivation:** Augment sparse graphs
- (2) Add virtual nodes
  - The virtual node will connect to all the nodes in the graph
    - Suppose in a sparse graph, two nodes have shortest path distance of 10
    - After adding the virtual node, all the nodes will have a distance of 2
      - Node A – Virtual node – Node B
- **Benefits:** Greatly improves message passing in sparse graphs
Previously:

- All the nodes are used for message passing

New idea: (Randomly) sample a node’s neighborhood for message passing
For example, we can randomly choose 2 neighbors to pass messages

- Only nodes $B$ and $D$ will pass message to $A$
Next time when we compute the embeddings, we can sample different neighbors

- Only nodes $C$ and $D$ will pass message to $A$
In expectation, we can get embeddings similar to the case where all the neighbors are used

- **Benefits**: Greatly reduce computational cost
- And in practice it works great!
**Recap:** A general perspective for GNNs

- **GNN Layer:**
  - Transformation + Aggregation
  - Classic GNN layers: GCN, GraphSAGE, GAT

- **Layer connectivity:**
  - Deciding number of layers
  - Skip connections

- **Graph Manipulation:**
  - Feature augmentation
  - Structure manipulation

**Next:** GNN objectives, GNN in practice