## UPCOMING

- Project proposals due **today** (10/19)
- Colab 2 due on **Thursday** (10/21)

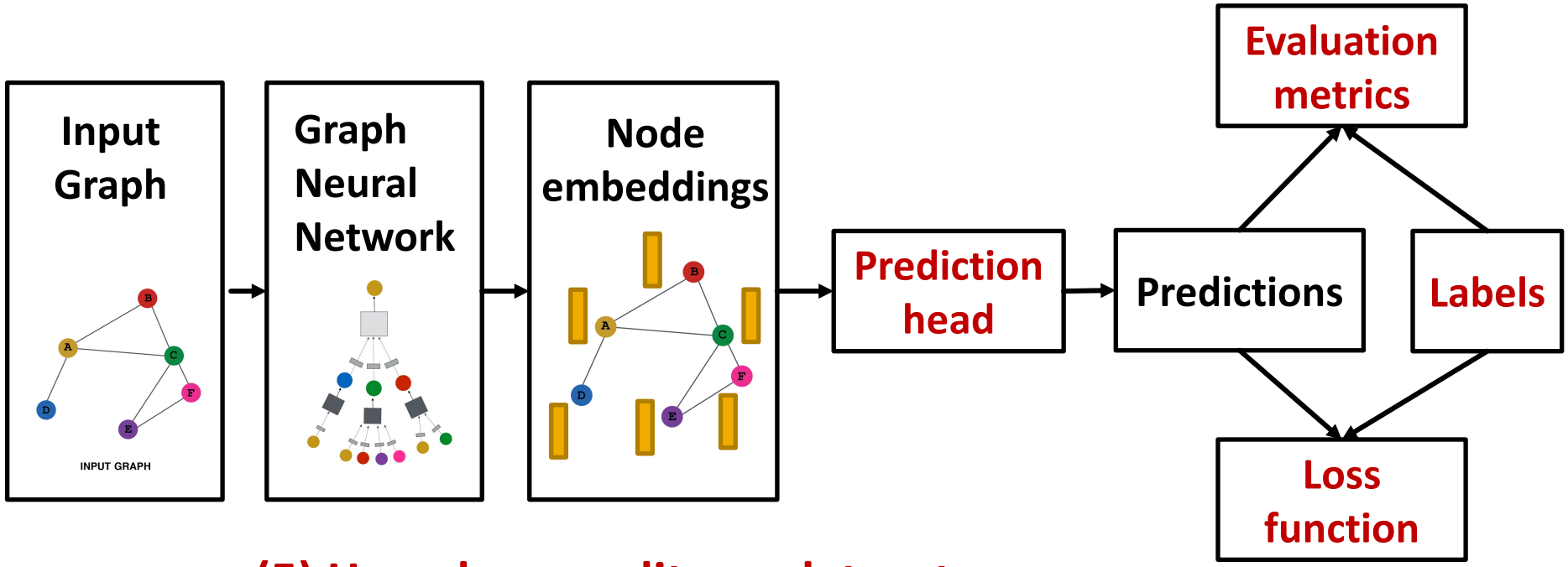
# Stanford CS224W: Setting-up GNN Prediction Tasks

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

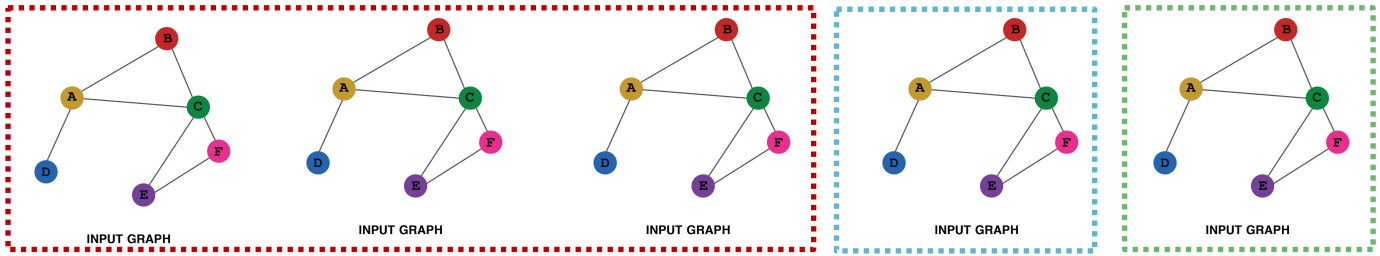
# COURSE FEEDBACK

- **Thank you** for providing thoughtful course feedback!
- **Highlights**: enjoying lectures, homeworks, Colabs, and responsiveness from teaching staff
- **Improvements**
  - Future homeworks: submitting PDF instead of writing on Gradescope, more coordination between TAs
  - OH: allowing students to join general room with 1:1 breakout rooms for TA & student
  - Lectures: more time for questions, slightly slower pacing

# GNN Training Pipeline (5)




**(5) How do we split our dataset into train / validation / test set?**




**Dataset split**

# Dataset Split: Fixed / Random Split

- **Fixed split:** We will split our dataset **once**
  - **Training set**: used for optimizing GNN parameters
  - **Validation set**: develop model/hyperparameters
  - **Test set**: held out until we report final performance
- **A concern:** sometimes we cannot guarantee that the test set will really be held out
- **Random split:** we will **randomly split** our dataset into training / validation / test
  - We report **average performance over different random seeds**

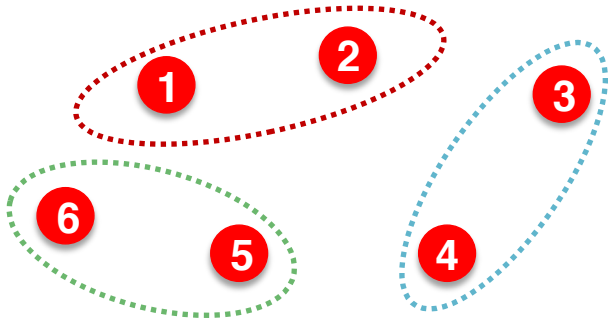
# Why Splitting Graphs is Special

- **Suppose we want to split an image dataset**
  - **Image classification: Each data point is an image**
  - Here **data points are independent**
    - **Image 5 will not affect our prediction on image 1**

Training

Validation

Test



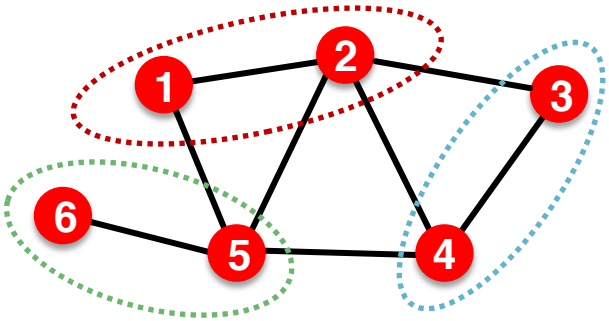
# Why Splitting Graphs is Special

- **Splitting a graph dataset is different!**
  - **Node classification: Each data point is a node**
  - Here **data points are NOT independent**
    - **Node 5 will affect our prediction on node 1,** because it will participate in message passing → affect node 1's embedding

Training

Validation

Test




- **What are our options?**

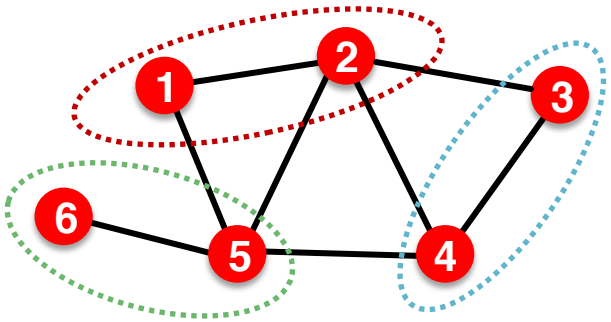
# Why Splitting Graphs is Special

- **Solution 1 (Transductive setting): The input graph can be observed in all the dataset splits (training, validation and test set).**
- **We will only split the (node) labels**
  - **At training time,** we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At validation time,** we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**

Training

Validation

Test



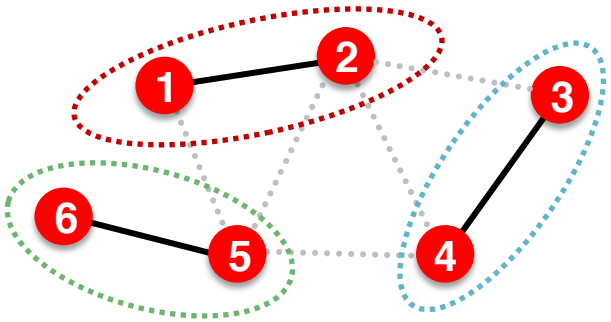
# Why Splitting Graphs is Special

- **Solution 2 (Inductive setting):** **We break the edges between splits to get multiple graphs**
  - **Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 any more**
  - **At training time,** we compute embeddings **using the graph over node 1&2**, and train **using node 1&2's labels**
  - **At validation time,** we compute embeddings **using the graph over node 3&4**, and **evaluate on node 3&4's labels**

Training

Validation

Test

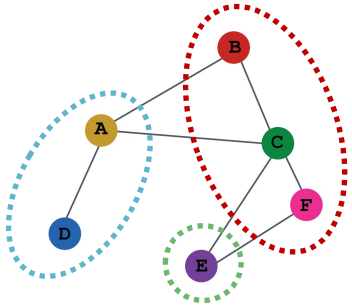


# Transductive / Inductive Settings

- **Transductive setting:** training / validation / test sets are **on the same graph**
  - The **dataset consists of one graph**
  - **The entire graph can be observed in all dataset splits, we only split the labels**
  - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / validation / test sets are **on different graphs**
  - The **dataset consists of multiple graphs**
  - Each split can **only observe the graph(s) within the split**. A successful model should **generalize to unseen graphs**
  - Applicable to **node / edge / graph** tasks

# Example: Node Classification

- **Transductive** node classification
  - **All the splits can observe the entire graph structure**, but can only observe the labels of their respective nodes

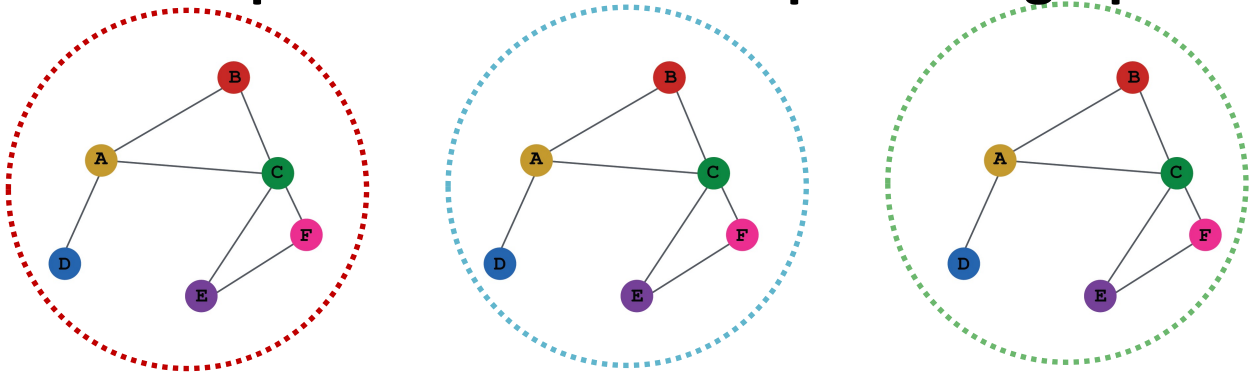



Training

Validation

Test

- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**



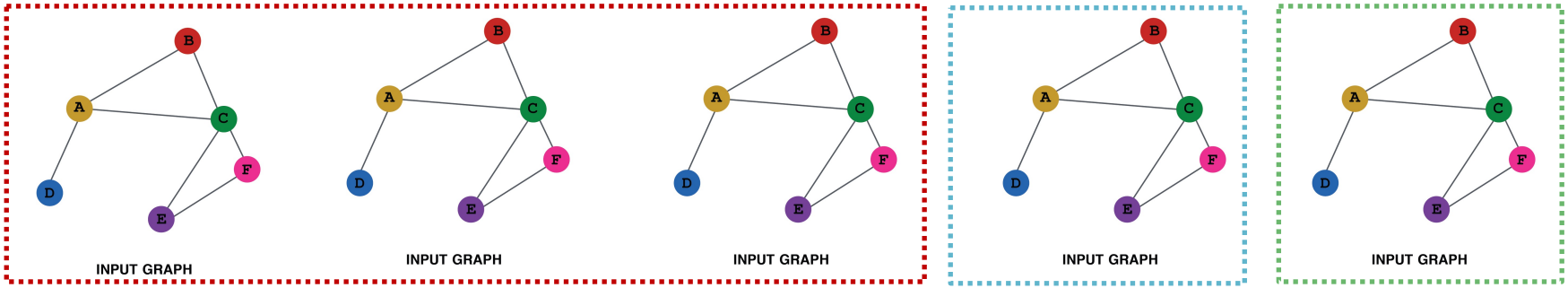

Training

Validation

Test

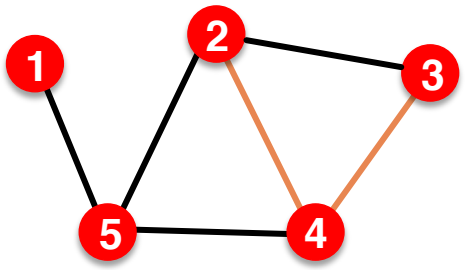
# Example: Graph Classification

- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).

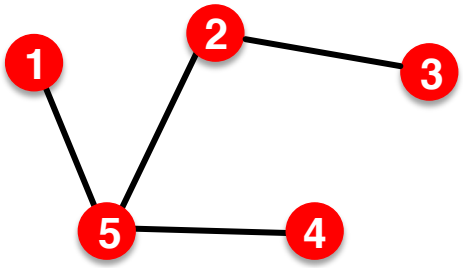


# Example: Link Prediction

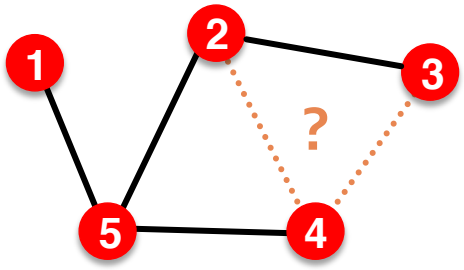
- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and the **let the GNN predict if the edges exist**




**Original graph**

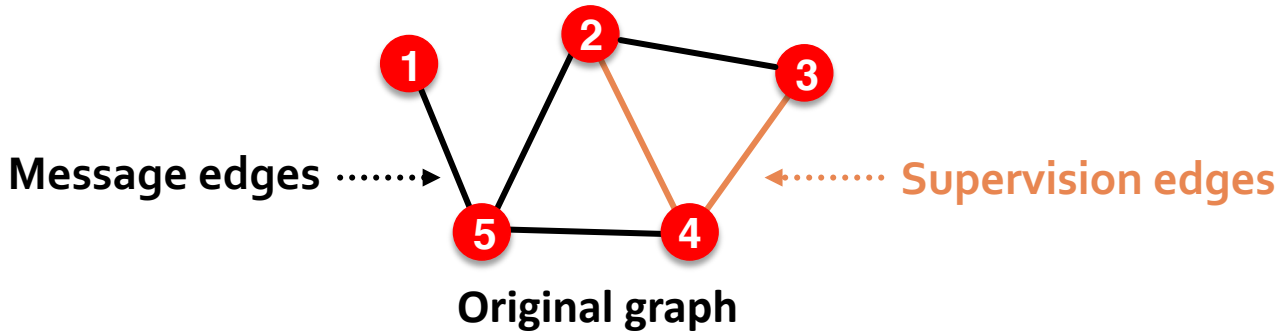



**Input graph to GNN**




**Predictions made by GNN**

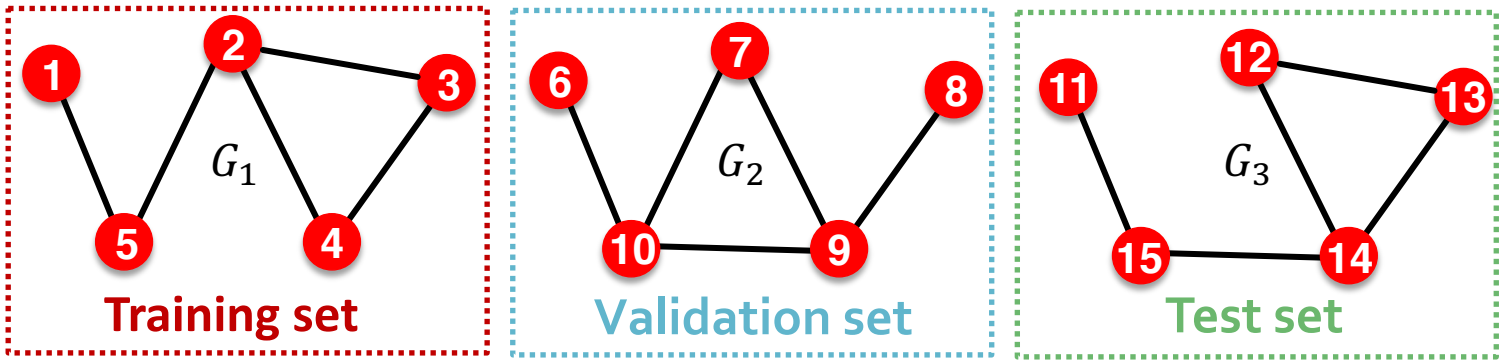
# Setting up Link Prediction




- For link prediction, we will split edges twice
- Step 1: Assign 2 types of edges in the original graph
  - Message edges: Used for GNN message passing
  - Supervision edges: Use for computing objectives
  - After step 1:
    - Only message edges will remain in the graph
    - Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN!

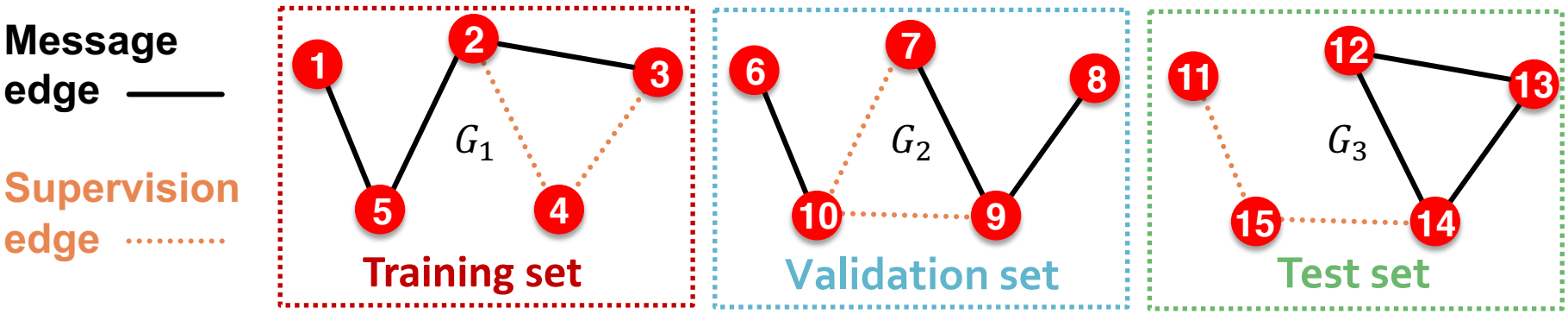
# Setting up Link Prediction

- **Step 2: Split edges into train / validation / test**
- **Option 1: Inductive link prediction split**
  - **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**



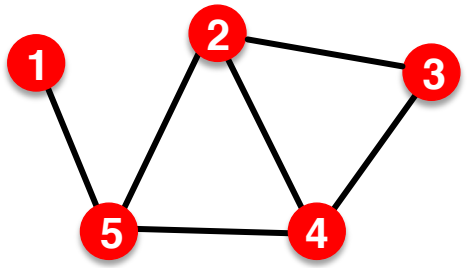
# Setting up Link Prediction

- **Step 2: Split edges into train / validation / test**
- **Option 1: Inductive link prediction split**
  - **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**
  - **In train or val or test set, each graph will have 2 types of edges: message edges + supervision edges**
    - **Supervision edges** are not the input to GNN



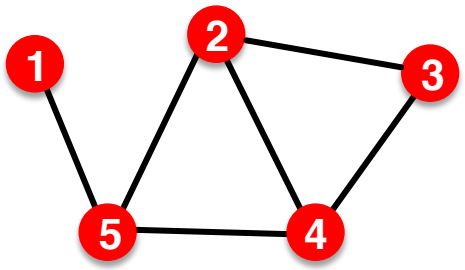
# Setting up Link Prediction

- **Option 2: Transductive link prediction split:**
  - **This is the default setting when people talk about link prediction**
  - **Suppose we have a dataset of 1 graph**



# Setting up Link Prediction

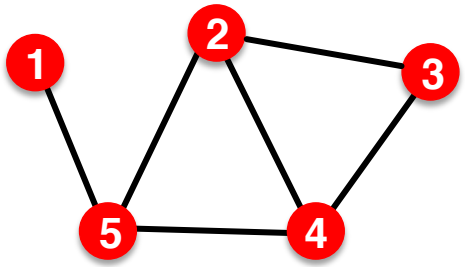
- **Option 2: Transductive link prediction split:**
  - **By definition of "transductive", the entire graph can be observed in all dataset splits**
    - **But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges**
    - **To train the training set, we further need to hold out supervision edges for the training set**



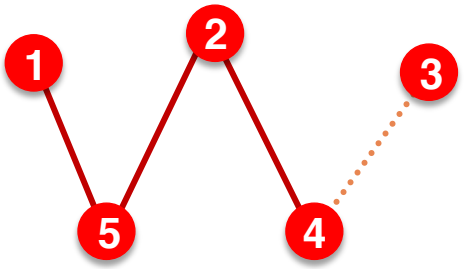

  - **Next:** we will show the exact settings

# Setting up Link Prediction

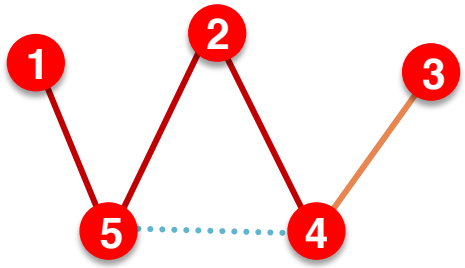
- **Option 2: Transductive link prediction split:**



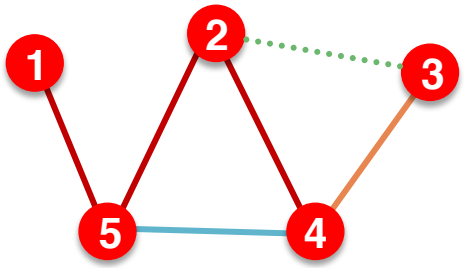

**The original graph**




**(1) At training time:**
Use **training message edges** to predict **training supervision edges**



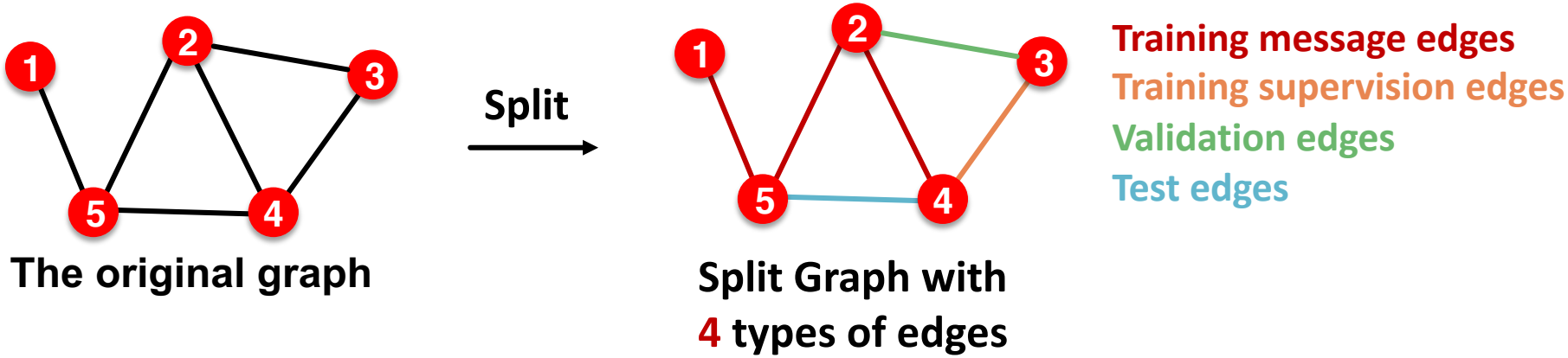

**(2) At validation time:**
Use **training message edges & training supervision edges** to predict **validation edges**




**(3) At test time:**
Use **training message edges & training supervision edges & validation edges** to predict **test edges**

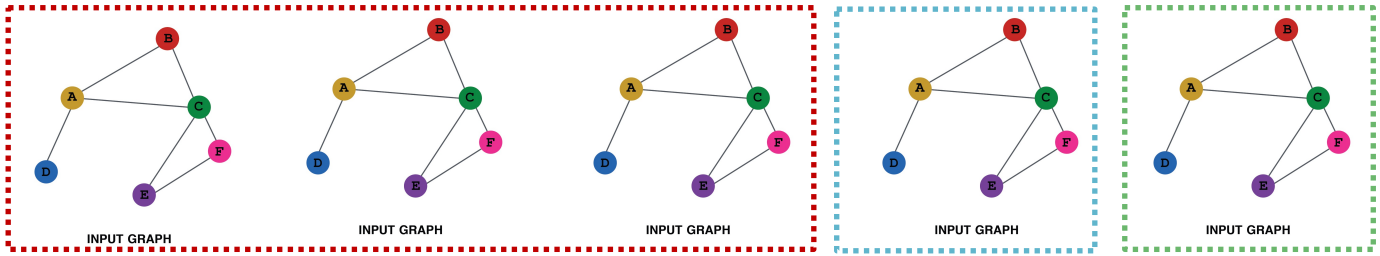
# Setting up Link Prediction

- **Summary: Transductive link prediction split:**

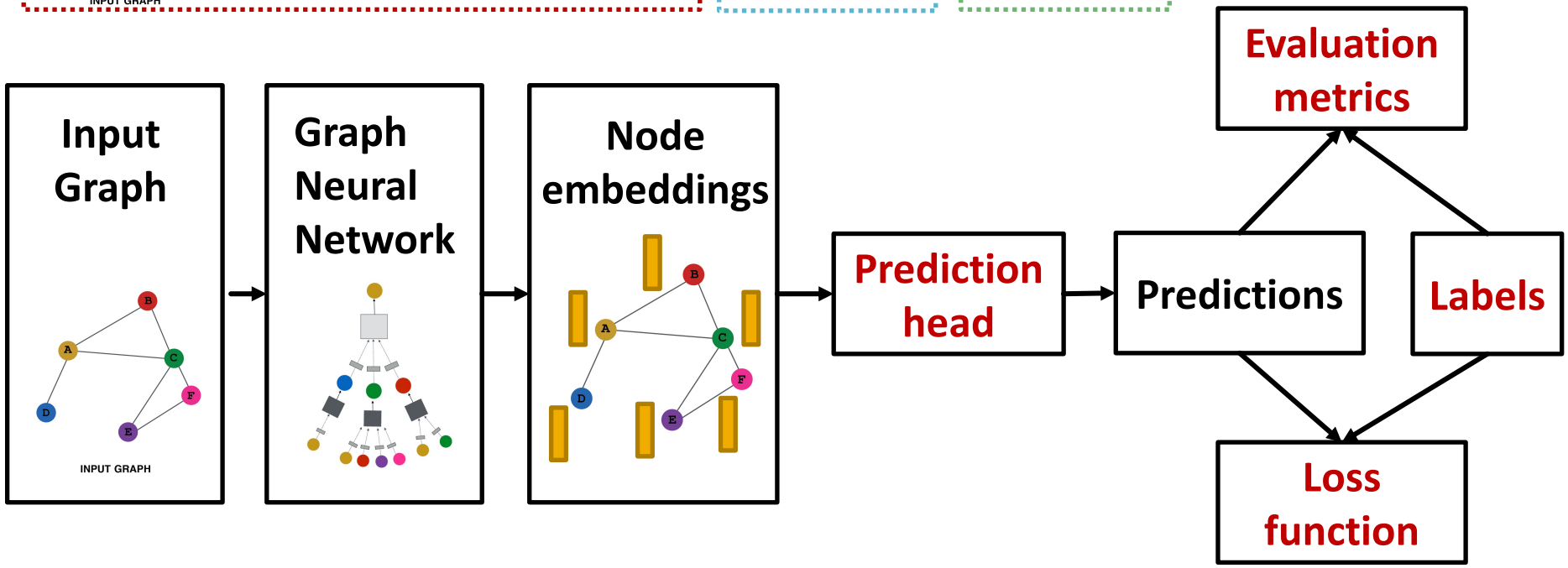



  - **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently. But if you follow our reasoning steps, **this should be the right way to implement link prediction**
  - Luckily, we have full support in **DeepSNAP and GraphGym**

# GNN Training Pipeline




Dataset split




**Implementation resources:**

DeepSNAP provides core modules for this pipeline

GraphGym further implements the full pipeline to facilitate GNN design

# Stanford CS224W: When Things Don't Go As Planned

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

# General Tips

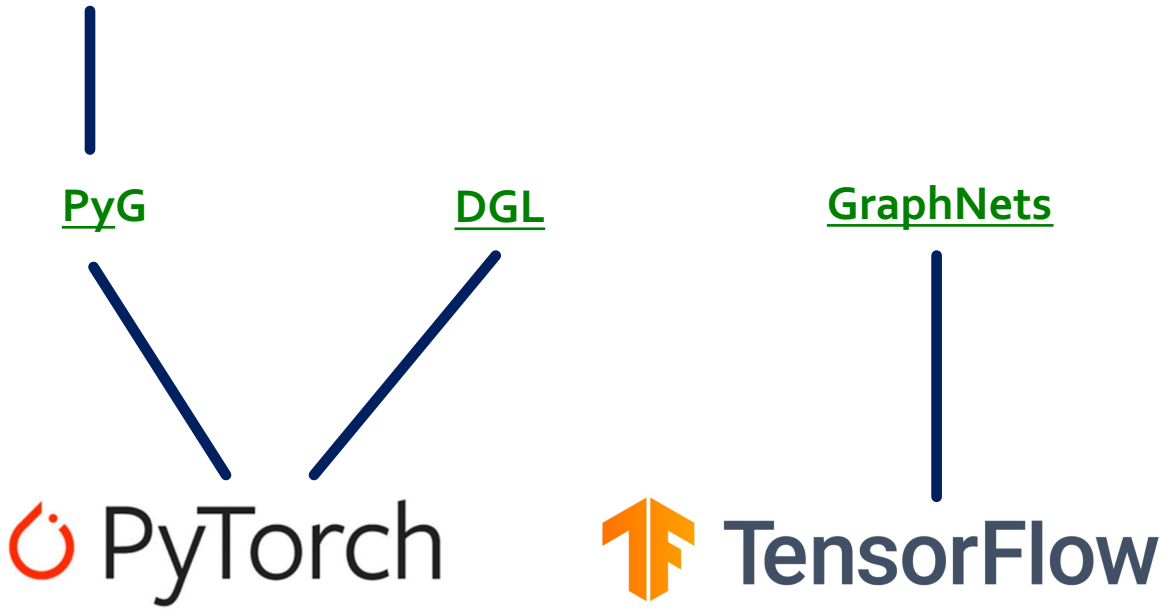
- **Data preprocessing is important:**
  - Node attributes can vary a lot!
    - E.g. probability ranges $(0,1)$, but some inputs could have much larger range, say $(-1000, 1000)$
  - Use normalization
- **Optimizer:**
  - ADAM is relatively robust to learning rate
- **Activation function**
  - ReLU activation function often works well
  - Other alternatives: LeakyReLU, SWISH, rational activation
  - No activation function at your output layer:
- **Include bias term in every layer**
- **Embedding dimensions:**
  - 32, 64 and 128 are often good starting points

# Debugging Deep Networks

- **Debug issues:** Loss/accuracy not converging during training
  - Check pipeline (e.g. in PyTorch we need zero_grad)
  - Adjust hyperparameters such as learning rate
  - Pay attention to weight parameter initialization
- **Important for model development:**
  - **Overfit on (part of) training data:**
    - With a small training dataset, loss should be essentially close to 0, with an expressive neural network
    - If neural network cannot overfit a single data point, something is wrong
  - **Scrutinize loss function!**
  - **Scrutinize visualizations!**

# Resources on Graph Neural Networks

**GraphGym**:
**Easy and flexible implementation support based on PyTorch Geometric**




**GNN framework:**
**Implements a variety of GNN architectures**

**Auto-differentiation framework**

# Resources on Graph Neural Networks

**Tutorials and overviews:**
- Relational inductive biases and graph networks (Battaglia et al., 2018)
- Representation learning on graphs: Methods and applications (Hamilton et al., 2017)

**Attention-based neighborhood aggregation:**
- Graph attention networks (Hoshen, 2017; Velickovic et al., 2018; Liu et al., 2018)

**Embedding entire graphs:**
- Graph neural nets with edge embeddings (Battaglia et al., 2016; Gilmer et. al., 2017)
- Embedding entire graphs (Duvenaud et al., 2015; Dai et al., 2016; Li et al., 2018) and graph pooling (Ying et al., 2018, Zhang et al., 2018)
- Graph generation and relational inference (You et al., 2018; Kipf et al., 2018)
- How powerful are graph neural networks(Xu et al., 2017)

**Embedding nodes:**
- Varying neighborhood: Jumping knowledge networks (Xu et al., 2018), GeniePath (Liu et al., 2018)
- Position-aware GNN (You et al. 2019)

**Spectral approaches to graph neural networks:**
- Spectral graph CNN & ChebNet (Bruna et al., 2015; Defferrard et al., 2016)
- Geometric deep learning (Bronstein et al., 2017; Monti et al., 2017)

**Other GNN techniques:**
- Pre-training Graph Neural Networks (Hu et al., 2019)
- GNNExplainer: Generating Explanations for Graph Neural Networks (Ying et al., 2019)

# Stanford CS224W: How Expressive are Graph Neural Networks?

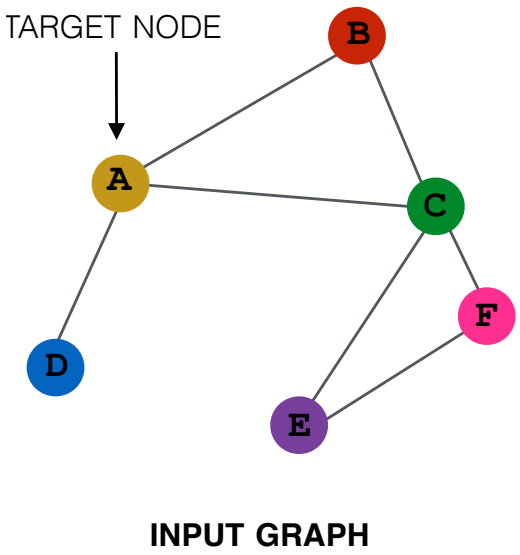
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

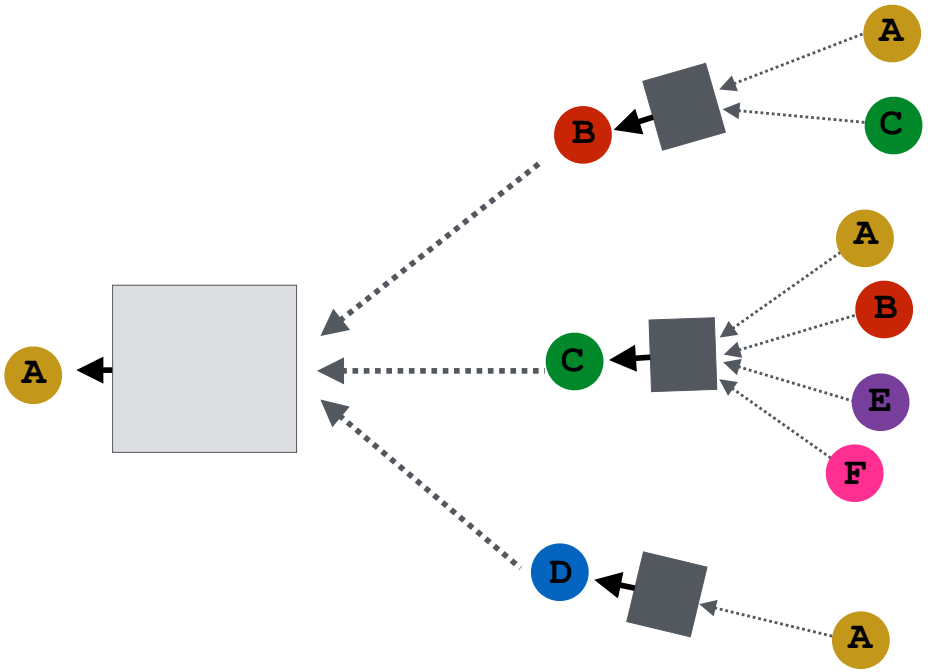
http://cs224w.stanford.edu

# Idea: Aggregate Neighbors

- **Key idea:** Generate node embeddings based on **local network neighborhoods**

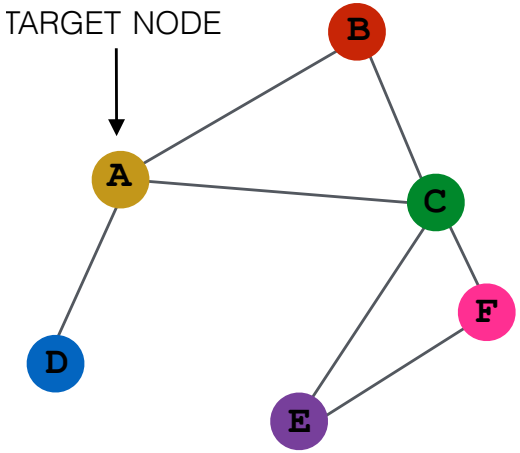



INPUT GRAPH

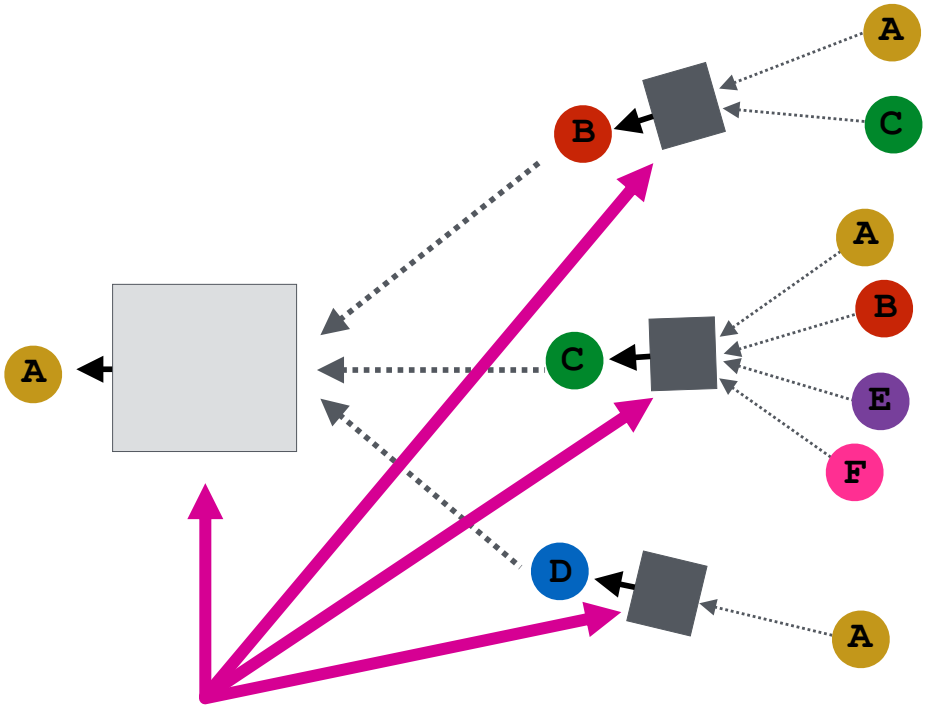


# Idea: Aggregate Neighbors

- **Intuition:** Nodes aggregate information from their neighbors using neural networks




INPUT GRAPH




Neural networks

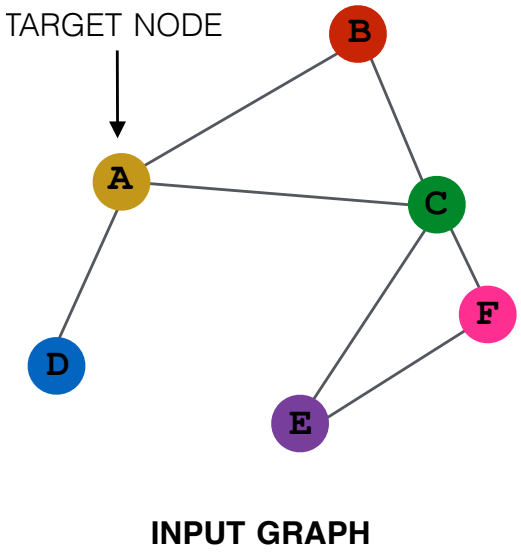
# Theory of GNNs

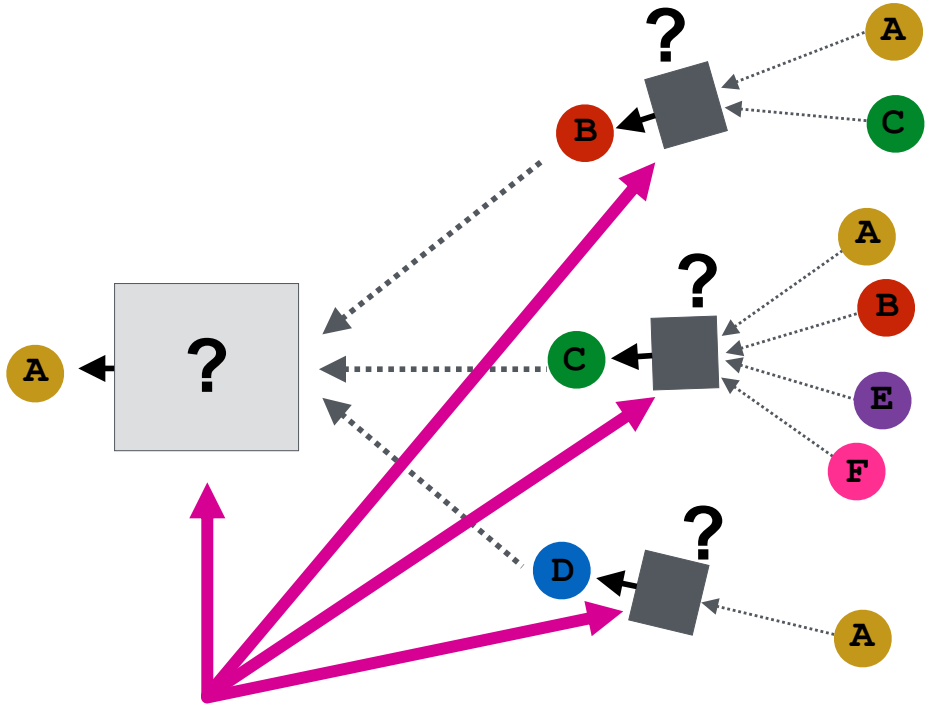
## How powerful are GNNs?

- Many GNN models have been proposed (e.g., GCN, GAT, GraphSAGE, design space).
- What is the expressive power (ability to distinguish different graph structures) of these GNN models?
- How to design a maximally expressive GNN model?

# Background: Many GNN Models

- Many GNN models have been proposed:
  - GCN, GraphSAGE, GAT, Design Space etc.

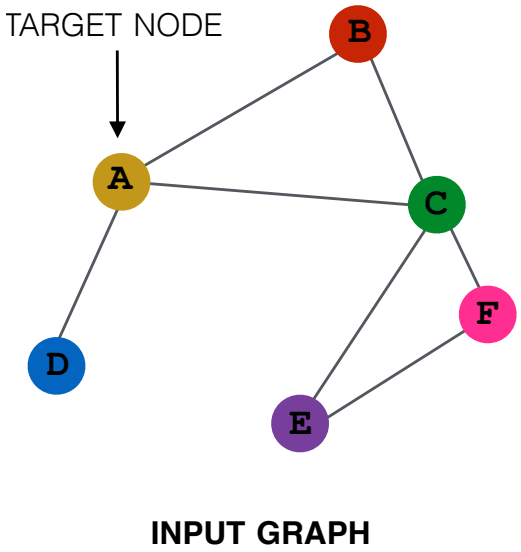


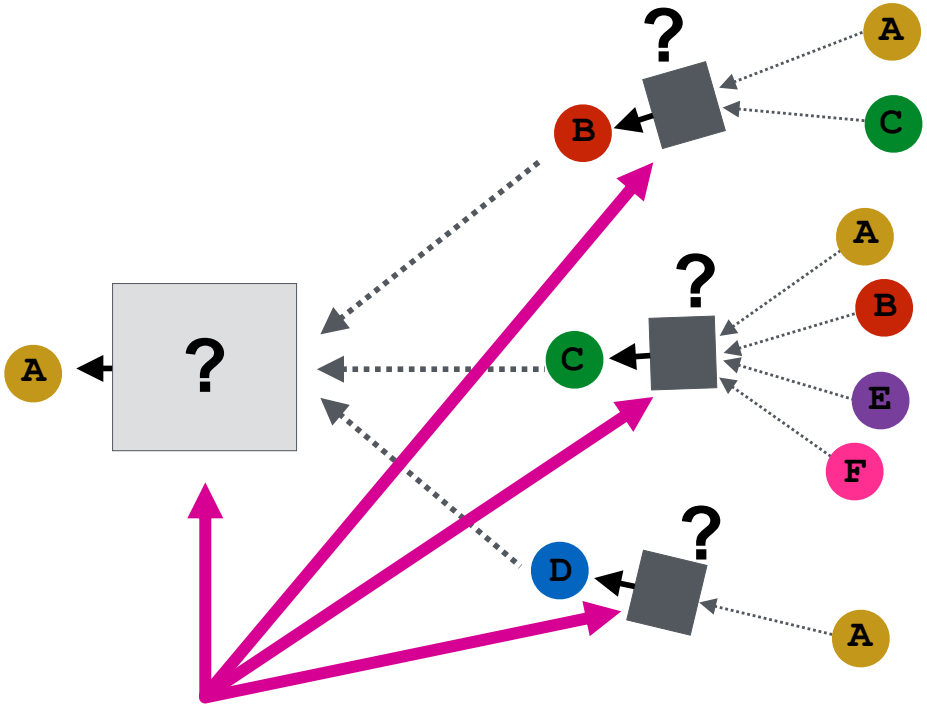

Different GNN models use different neural networks in the box

# GNN Model Example (1)

- **GCN (mean-pool)** [Kipf and Welling ICLR 2017]



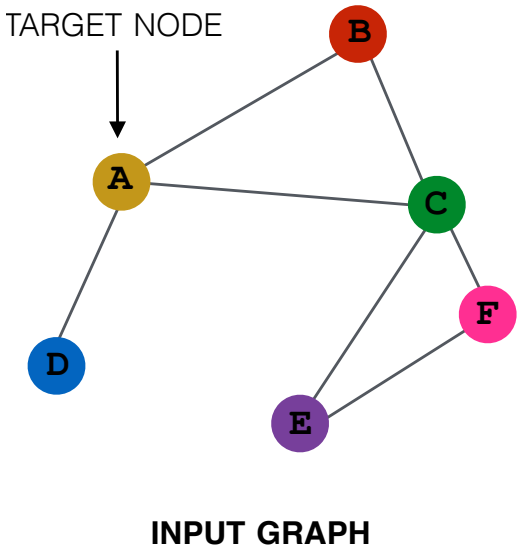

INPUT GRAPH

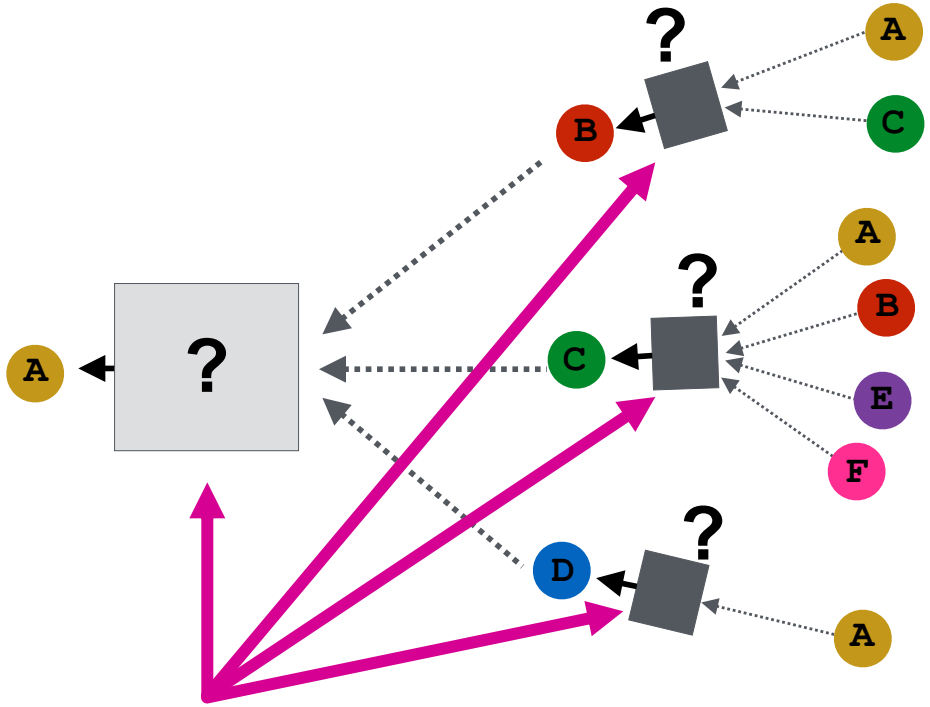



Element-wise mean pooling + Linear + ReLU non-linearity

# GNN Model Example (2)

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]

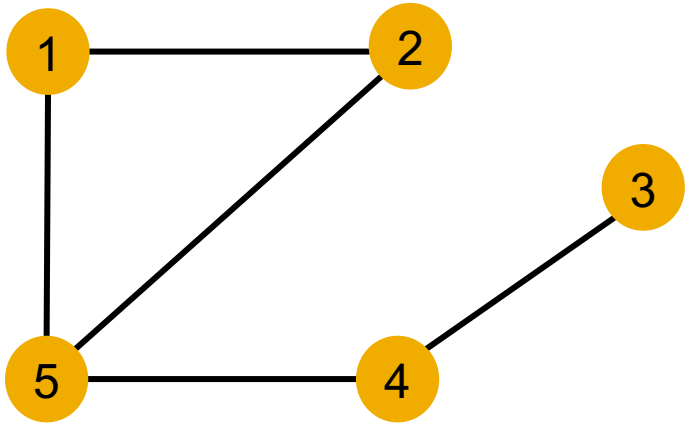



MLP + element-wise max-pooling

# Note: Node Colors

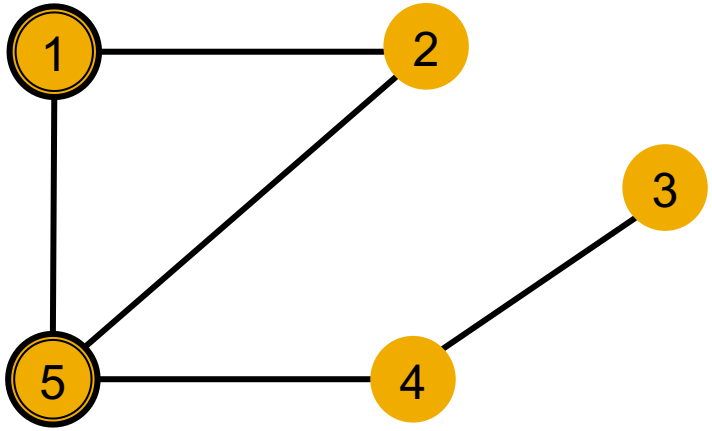
- We use node same/different **colors** to represent nodes with same/different features.
  - For example, the graph below assumes all the nodes share the same feature.




- **Key question: How well can a GNN distinguish different graph structures?**

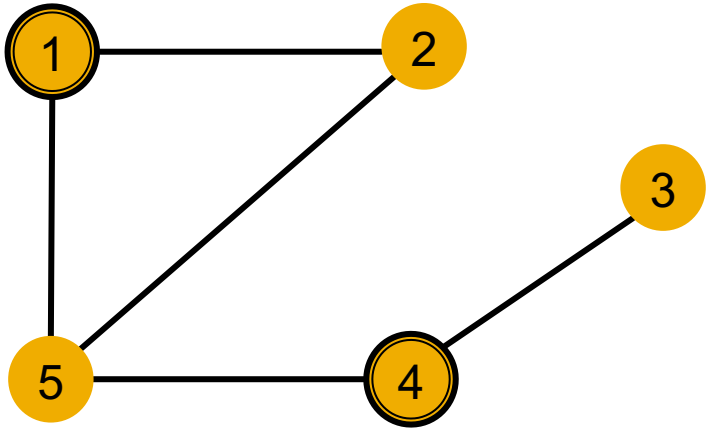
# Local Neighborhood Structures

- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 5 have **different** neighborhood structures because they have different node degrees.



# Local Neighborhood Structures

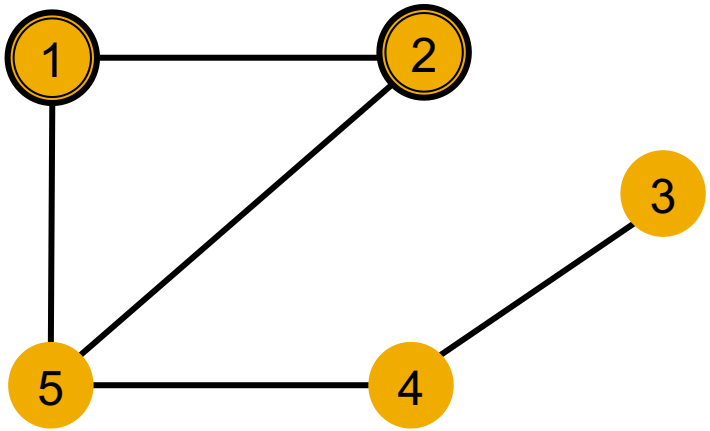
- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 4 both have the same node degree of 2. However, they still have **different** neighborhood structures because **their neighbors have different node degrees.**




Node 1 has neighbors of degrees 2 and 3.
Node 4 has neighbors of degrees 1 and 3.

# Local Neighborhood Structures

- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 2 have the **same** neighborhood structure because **they are symmetric within the graph.**



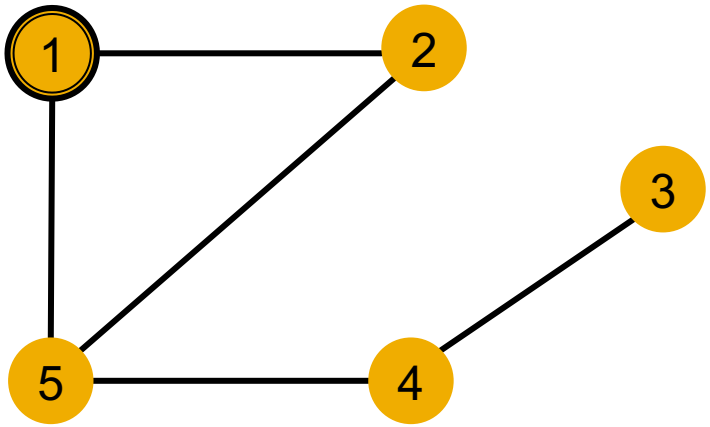

Node 1 has neighbors of degrees 2 and 3.
Node 2 has neighbors of degrees 2 and 3.
And even if we go a step deeper to 2nd hop neighbors, both nodes have the same degrees (Node 4 of degree 2)

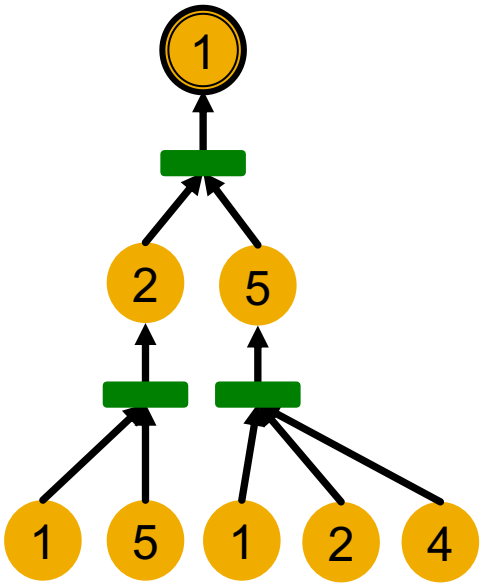
# Local Neighborhood Structures

- **Key question**: **Can GNN node embeddings distinguish different node's local neighborhood structures?**
  - If so, when? If not, when will a GNN fail?
- **Next**: **We need to understand how a GNN captures local neighborhood structures.**
  - **Key concept**: **Computational graph**

# Computational Graph (1)

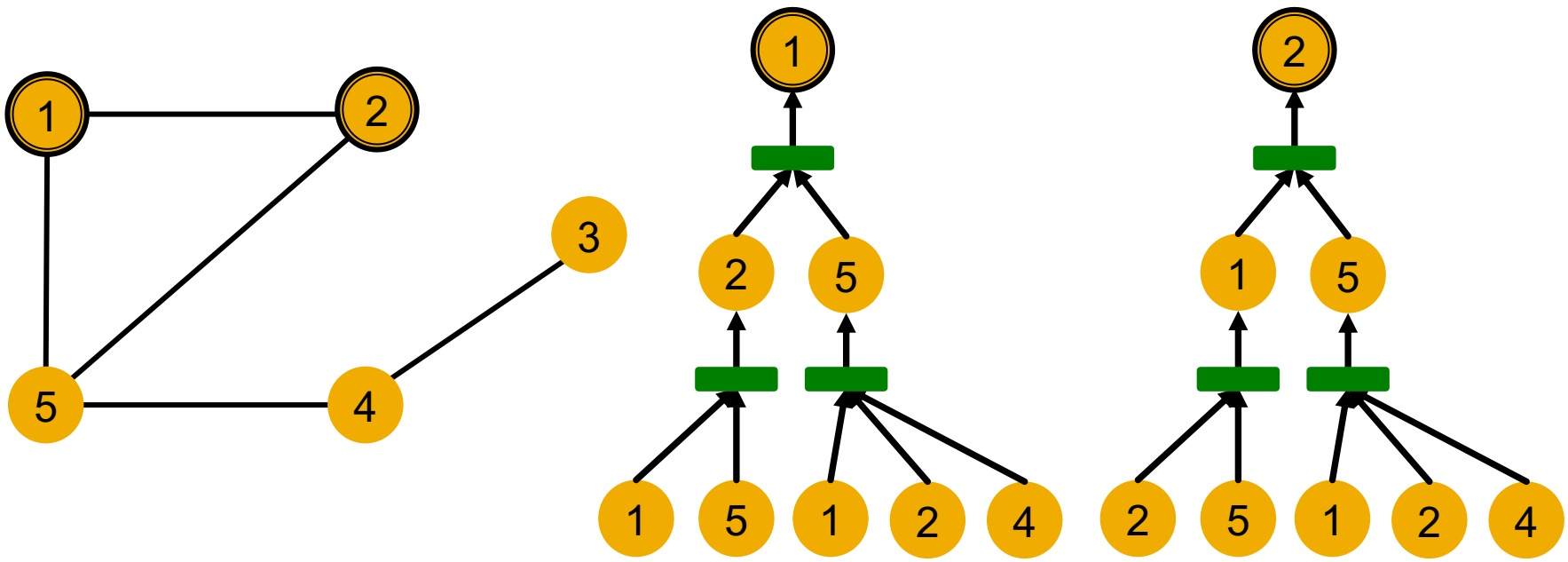
- **In each layer, a GNN aggregates neighboring node embeddings.**
- A GNN generates node embeddings through a computational graph defined by the neighborhood.
  - **Ex:** Node 1's computational graph (2-layer GNN)



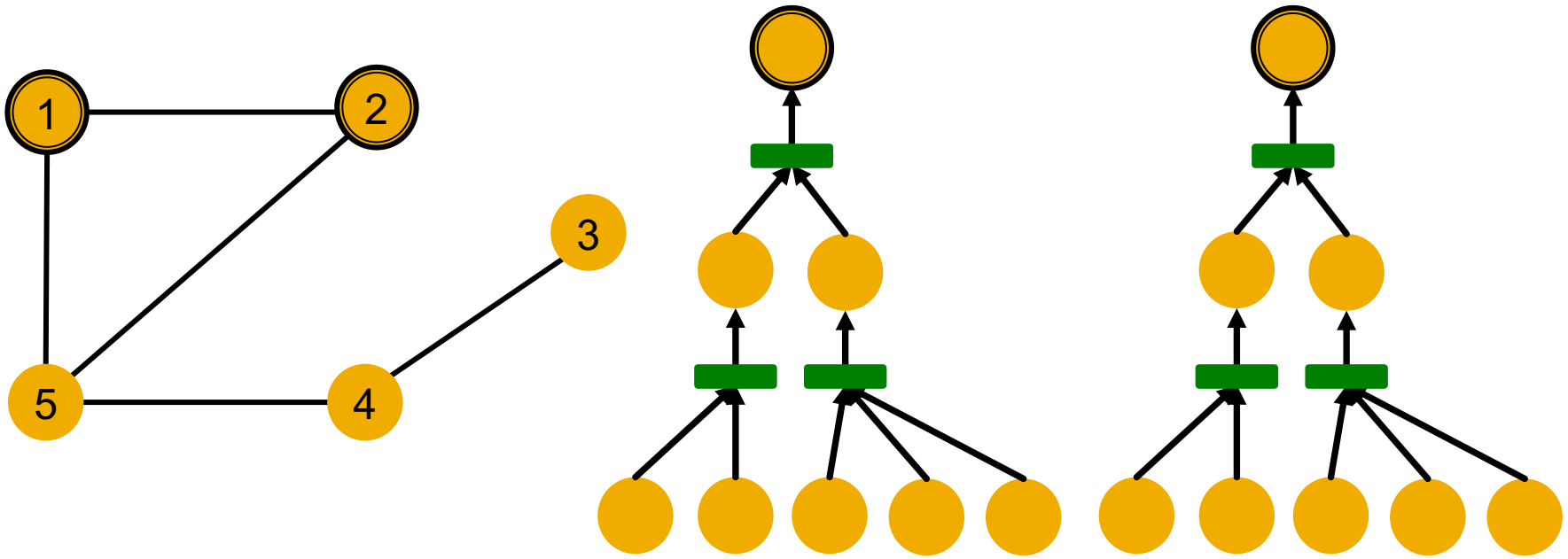
# Computational Graph (2)

- **Ex:** Nodes 1 and 2's computational graphs.



# Computational Graph (3)

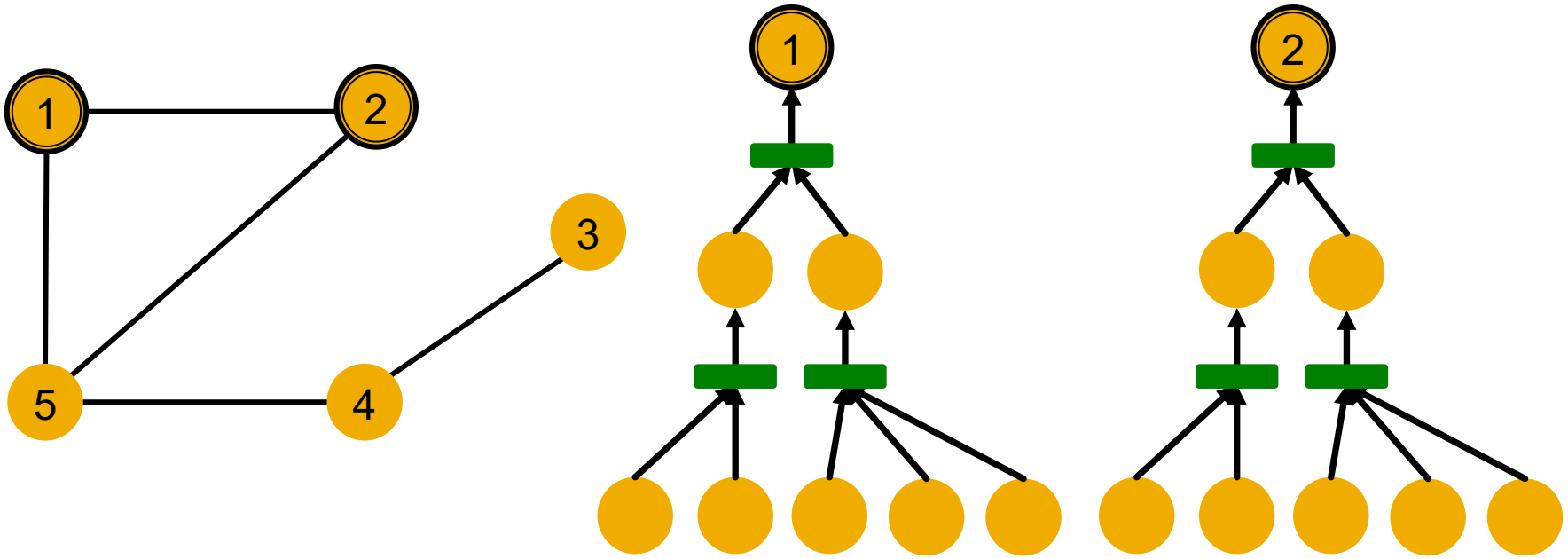
- **Ex:** Nodes 1 and 2's computational graphs.
- **But GNN only sees node features (not IDs):**



# Computational Graph (4)

- **A GNN will generate the same embedding for nodes 1 and 2 because:**
  - Computational graphs are the same.
  - Node features (colors) are identical.

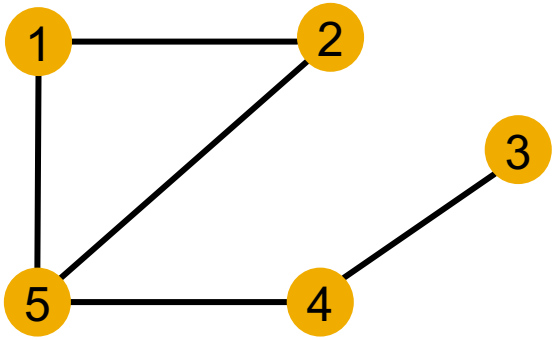
Note: GNN does not care about node ids, it just aggregates features vectors of different nodes.

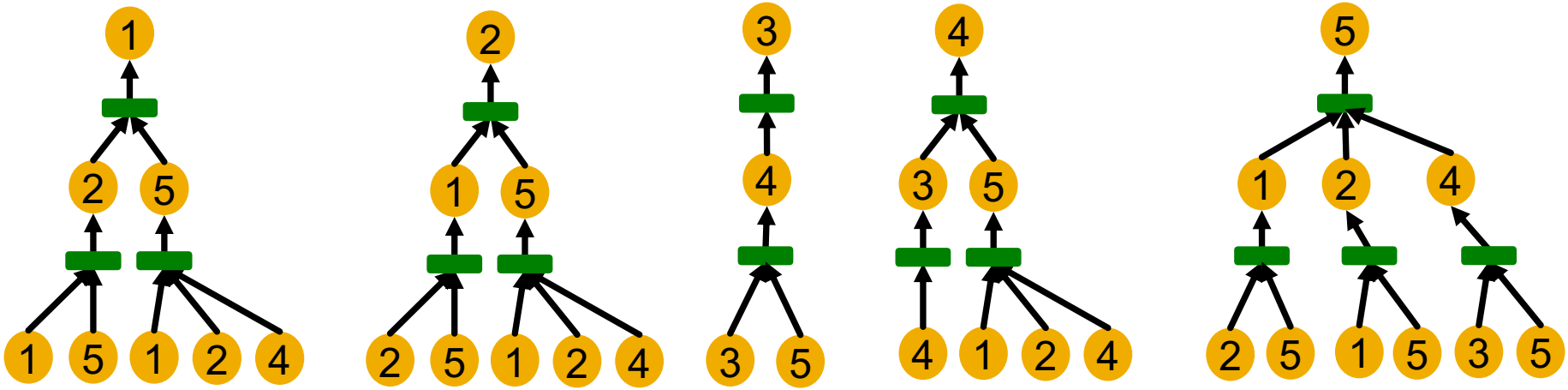



GNN won't be able to distinguish nodes 1 and 2

# Computational Graph

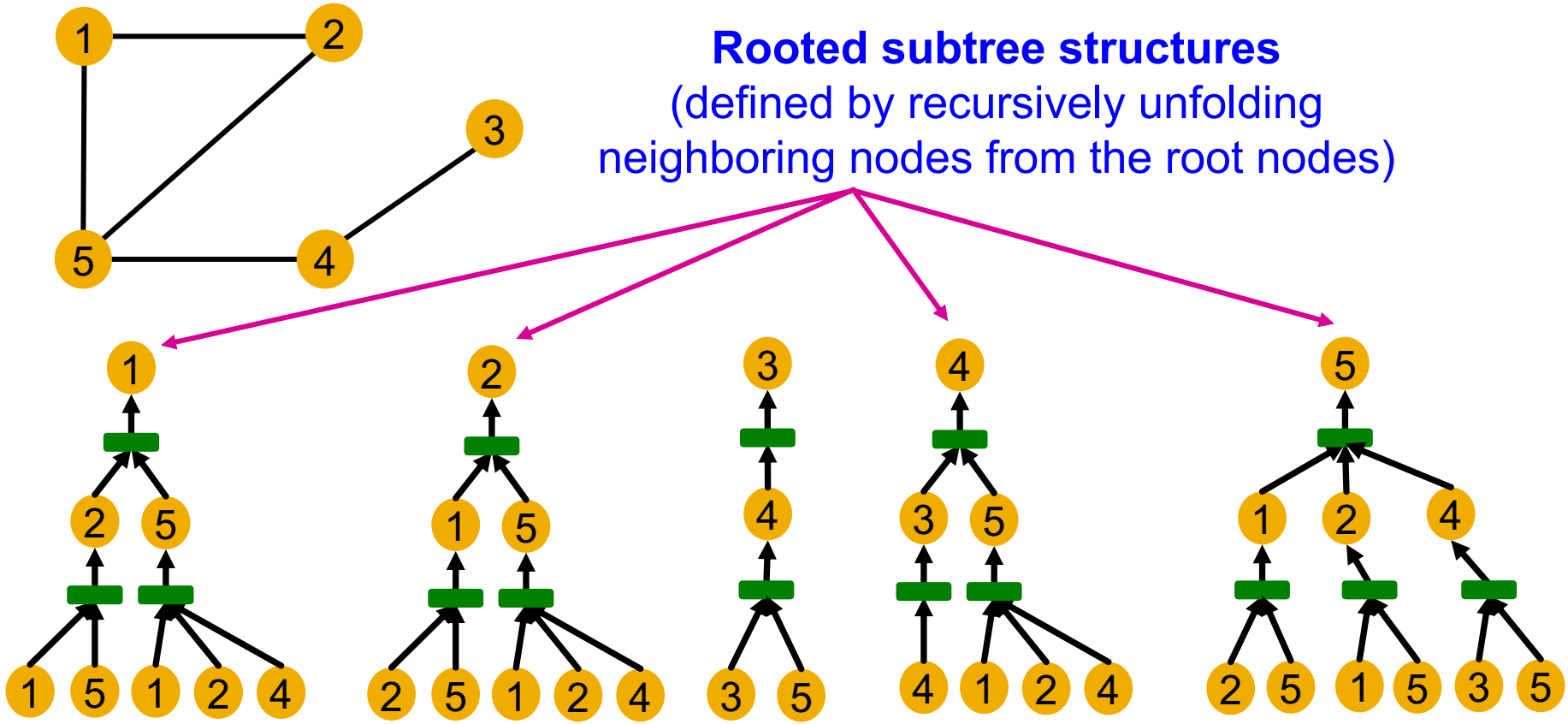
- **In general, different local neighborhoods define different computational graphs**



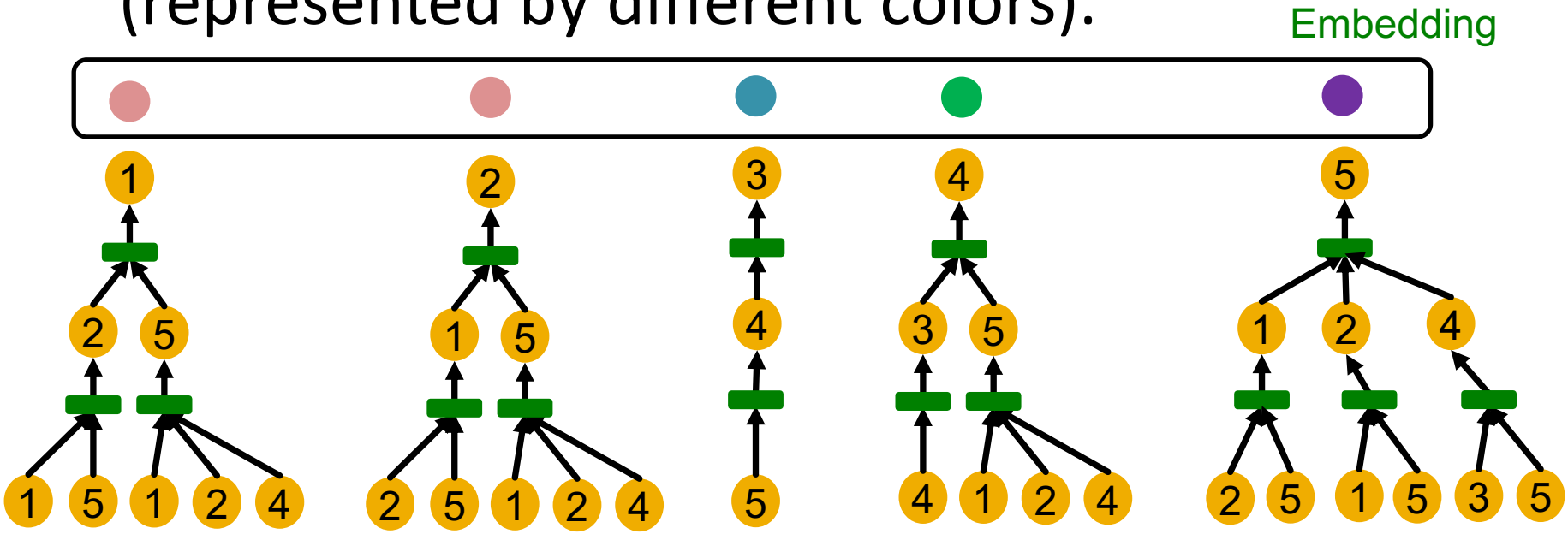
# Computational Graph

- Computational graphs are identical to **rooted subtree structures** around each node.



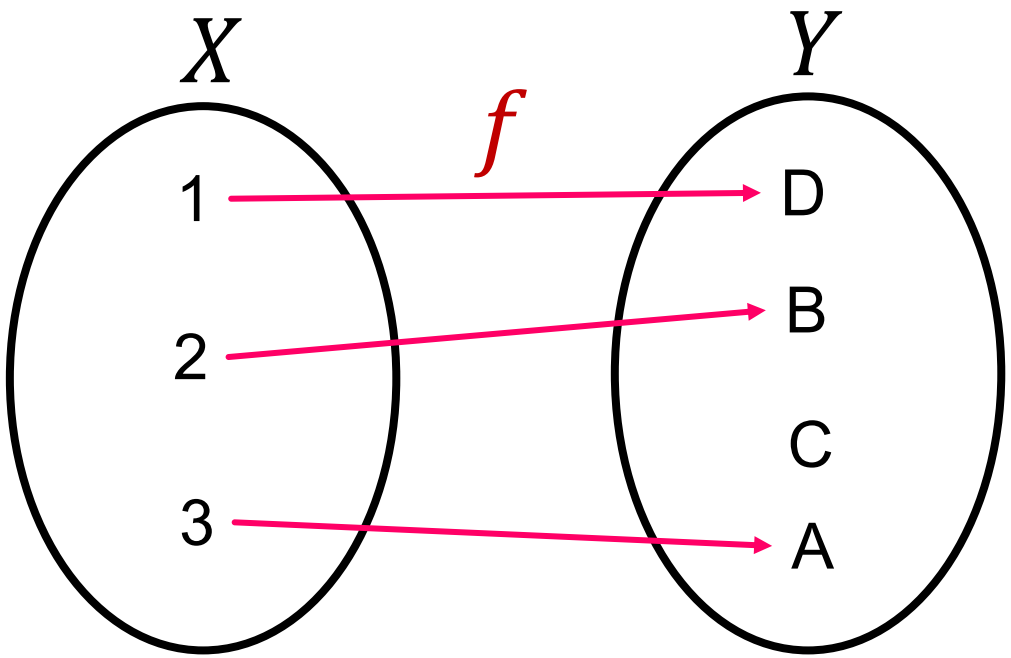
# Computational Graph

- GNN's node embeddings capture rooted subtree structures.
- Most expressive GNN maps different rooted subtrees into different node embeddings (represented by different colors).



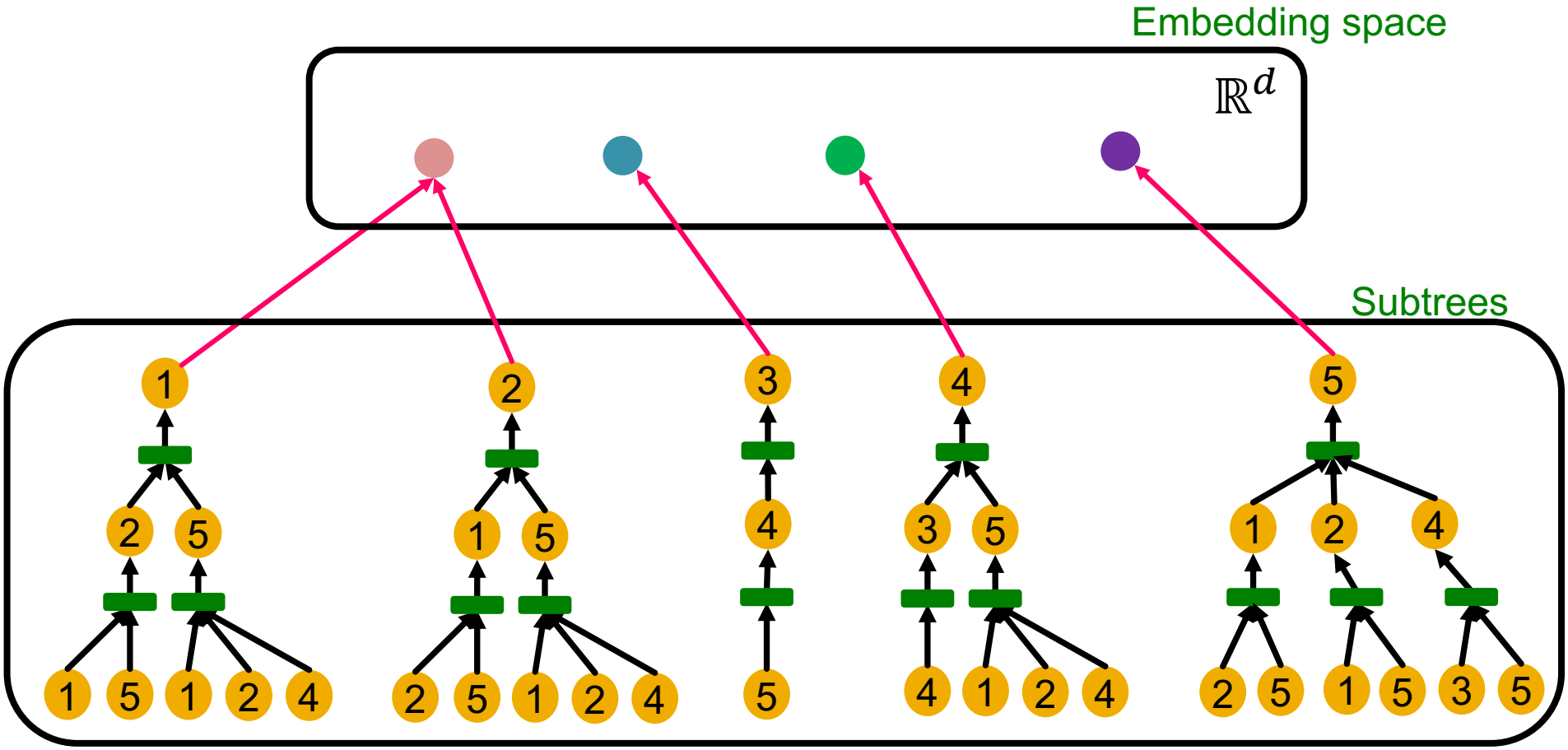
# Recall: Injective Function

- **Function** $f: X \to Y$ is **injective** if it maps different elements into different outputs.
- **Intuition:** $f$ retains all the information about input.



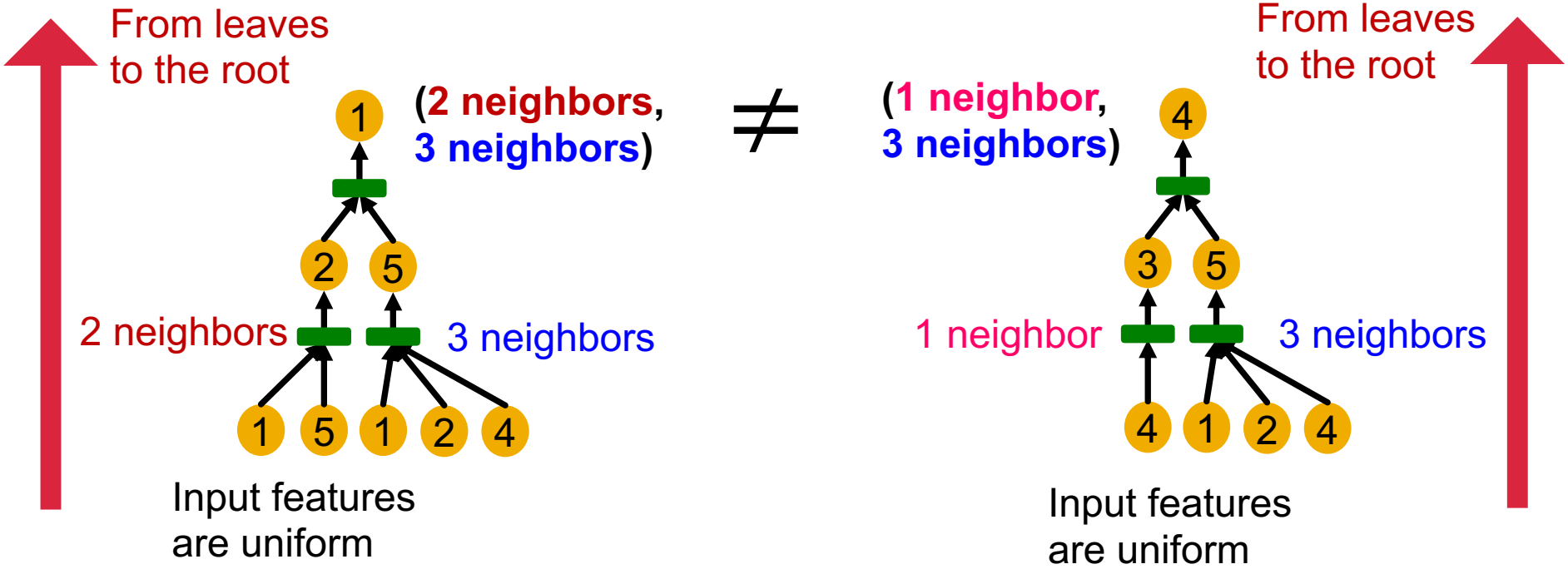
# How Expressive is a GNN?

- Most expressive GNN should map subtrees to the node embeddings **injectively**.



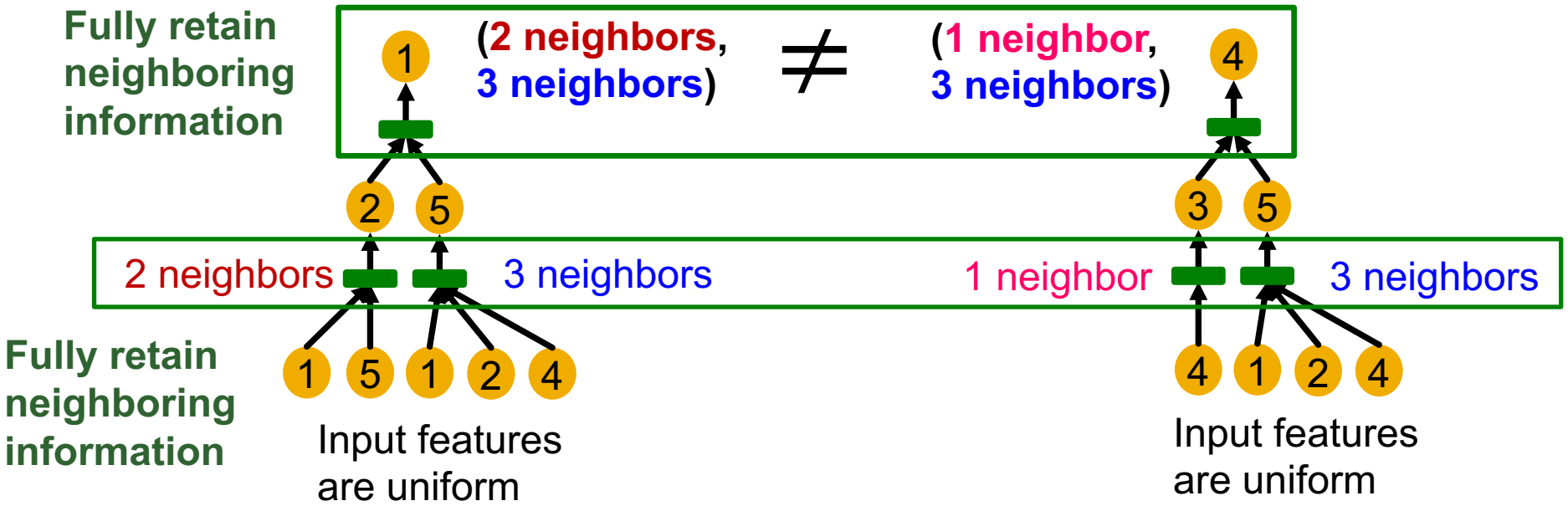
# How Expressive is a GNN?

- **Key observation**: Subtrees of the same depth can be recursively characterized from the leaf nodes to the root nodes.



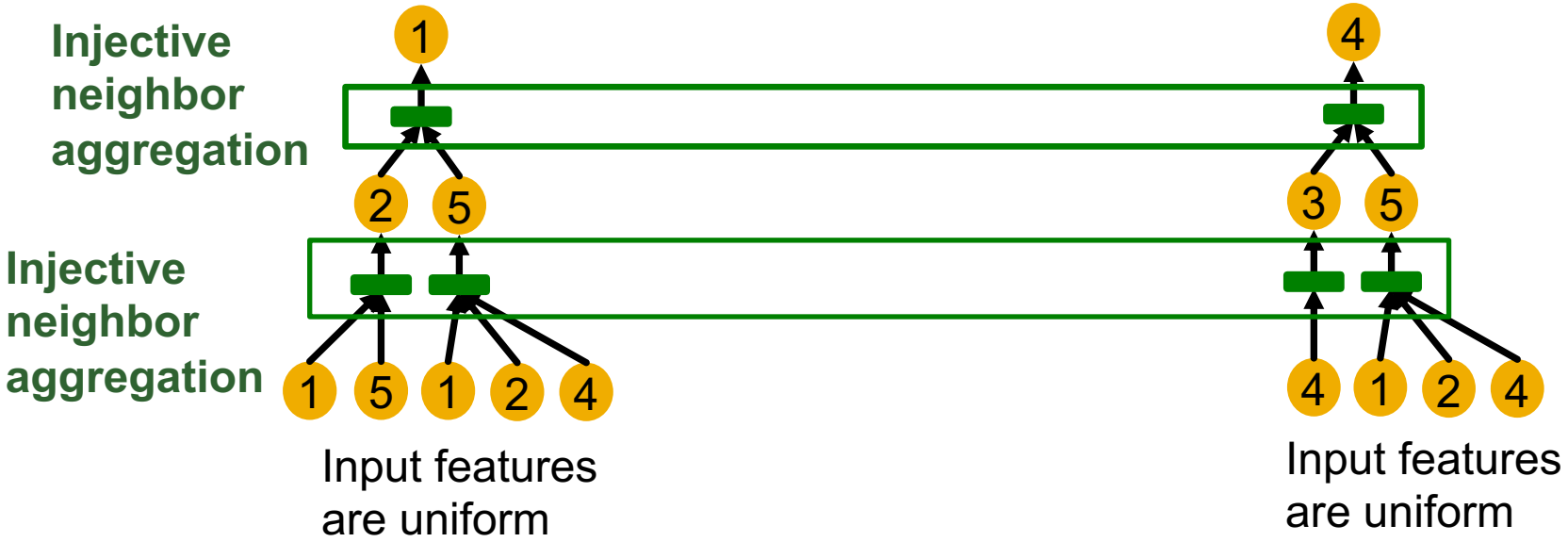
# How Expressive is a GNN?

- If each step of GNN's aggregation **can fully retain the neighboring information**, the generated node embeddings can distinguish different rooted subtrees.



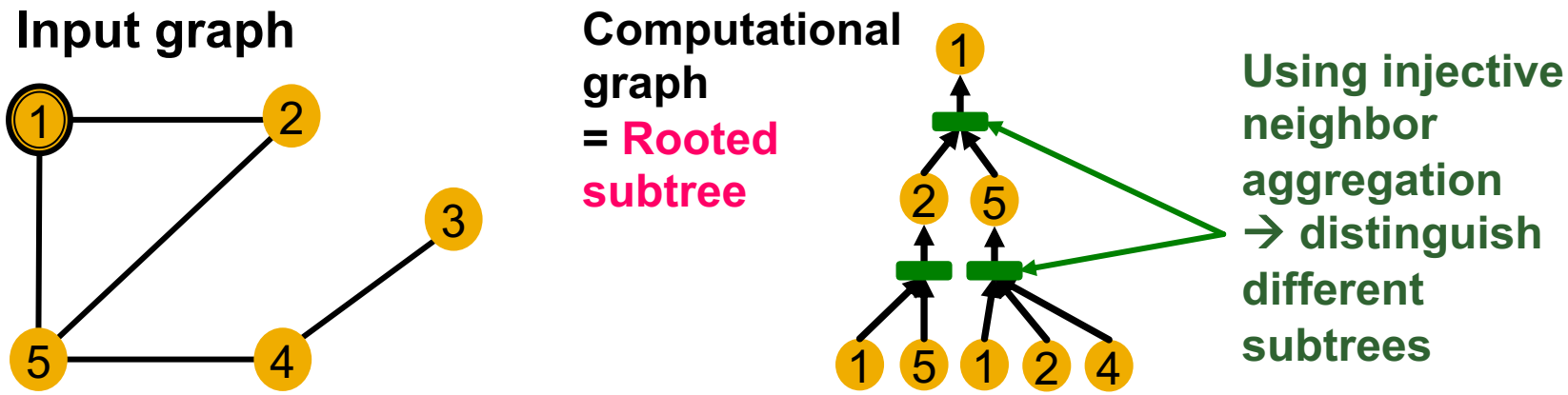
# How Expressive is a GNN?

- In other words, most expressive GNN would use an **injective neighbor aggregation** function at each step.
  - Maps different neighbors to different embeddings.



# How Expressive is a GNN?

- **Summary so far**
  - To generate a node embedding, GNNs use a computational graph corresponding to a subtree rooted around each node.




  - GNN can fully distinguish different subtree structures if **every step of its neighbor aggregation is injective.**

# Stanford CS224W: Designing the Most Powerful Graph Neural Network

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

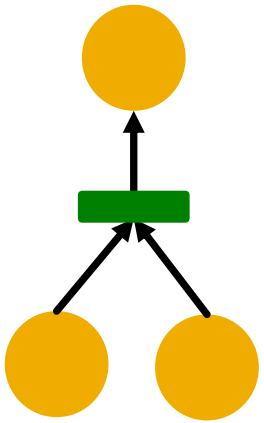
http://cs224w.stanford.edu

# Expressive Power of GNNs

- **Key observation**: **Expressive power of GNNs can be characterized by that of neighbor aggregation functions they use.**
  - A more expressive aggregation function leads to a more expressive a GNN.
  - Injective aggregation function leads to the most expressive GNN.
- **Next:**
  - Theoretically analyze expressive power of aggregation functions.

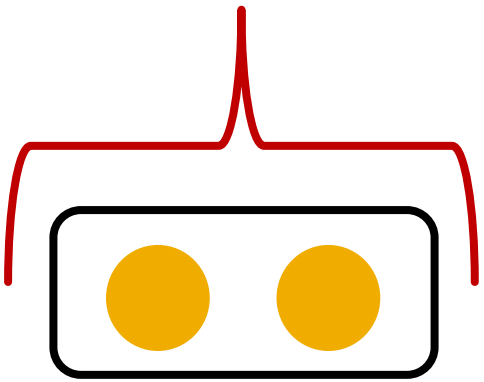
# Neighbor Aggregation

- **Observation**: **Neighbor aggregation** can be abstracted as **a function over a multi-set** (a set with repeating elements).



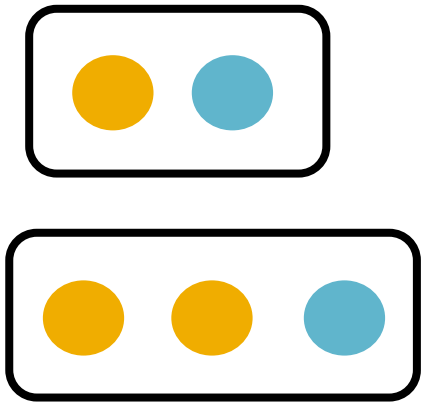
Neighbor aggregation

Equivalent



Multi-set function

Examples of multi-set



Same color indicates the same features.

# Neighbor Aggregation

- **Next**: **We analyze aggregation functions of two popular GNN models**
  - **GCN** (mean-pool) [Kipf & Welling, ICLR 2017]
    - Uses **element-wise** mean pooling over neighboring node features

$$\text{Mean}(\{x_u\}_{u \in N(v)})$$

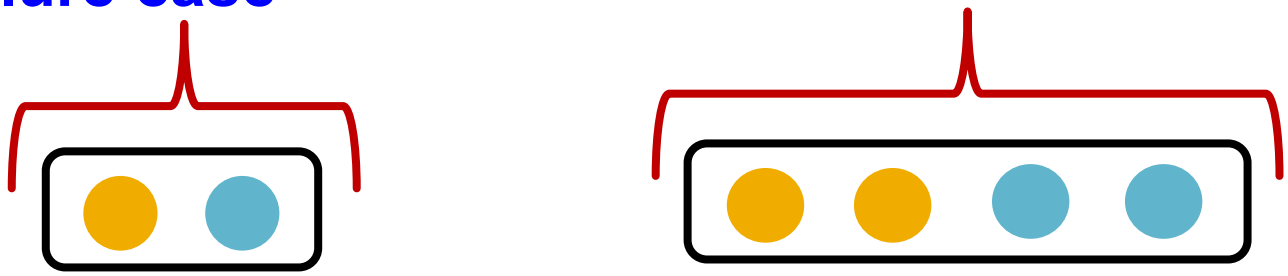
  - **GraphSAGE** (max-pool) [Hamilton et al. NeurIPS 2017]
    - Uses **element-wise** max pooling over neighboring node features

$$\text{Max}(\{x_u\}_{u \in N(v)})$$

# Neighbor Aggregation: Case Study

- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - Take **element-wise mean**, followed by linear function and ReLU activation, i.e., $\max(0, x)$.
  - **Theorem** [Xu et al. ICLR 2019]
    - GCN's aggregation **function cannot distinguish different multi-sets with the same color proportion**.

**Failure case**



- **Why?**

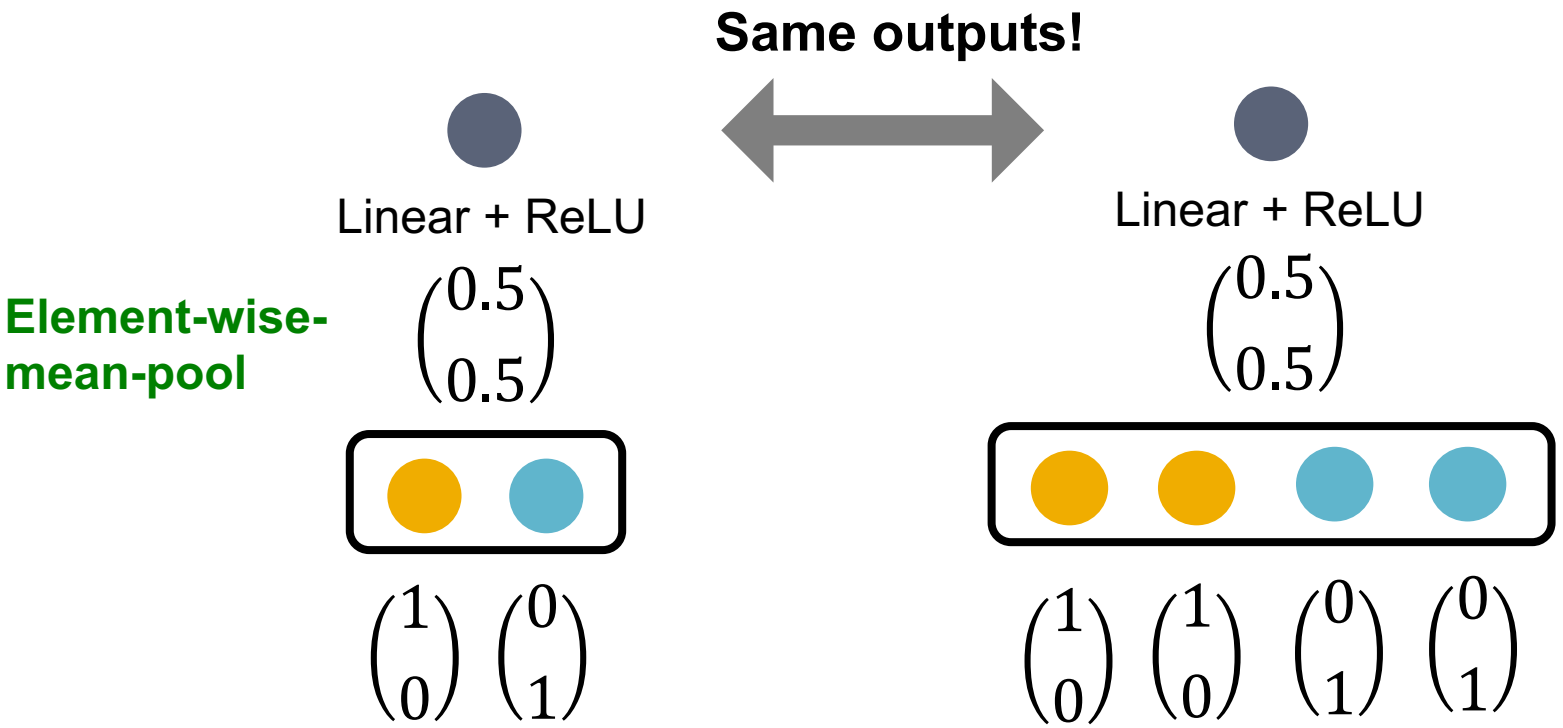
# Neighbor Aggregation

- For simplicity, we assume node colors are represented by **one-hot encoding**.
  - **Example)** If there are two distinct colors:

  - This assumption is sufficient to illustrate how GCN fails.

# Neighbor Aggregation: Case Study

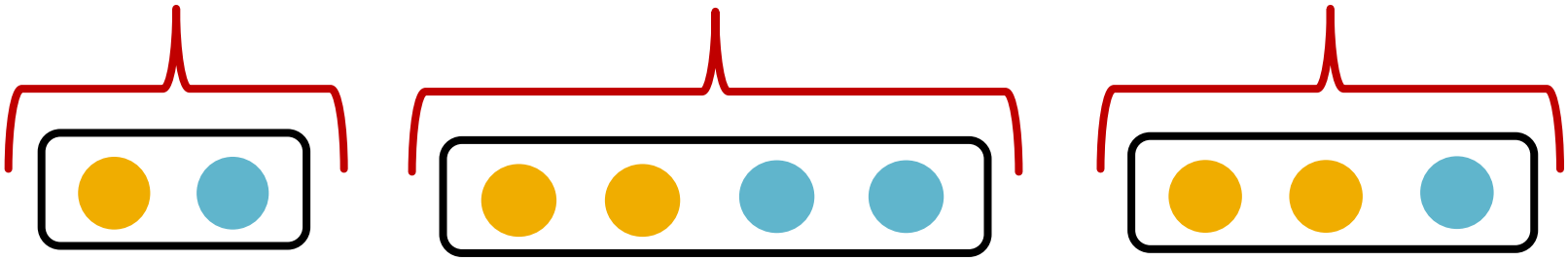
- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - **Failure case illustration**



# Neighbor Aggregation: Case Study

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - Apply an MLP, then take **element-wise max**.
  - **Theorem** [Xu et al. ICLR 2019]
    - GraphSAGE's aggregation **function cannot distinguish different multi-sets with the same set of distinct colors**.

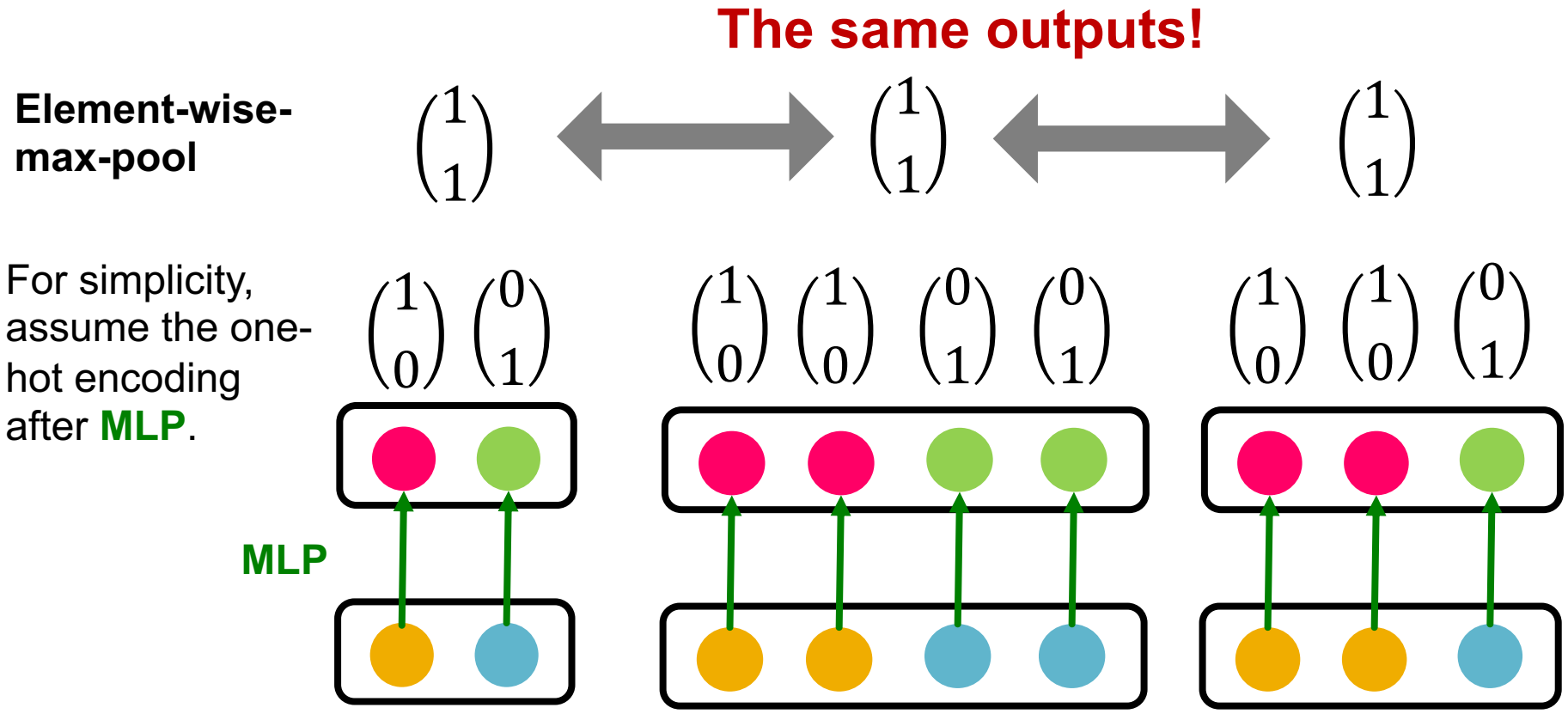
**Failure case**



- **Why?**

# Neighbor Aggregation: Case Study

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - **Failure case illustration**



# Summary So Far

- We analyzed the **expressive power of GNNs.**
- **Main takeaways**:
  - Expressive power of GNNs can be characterized by that of the neighbor aggregation function.
  - Neighbor aggregation is a function over multi-sets (sets with repeating elements)
  - GCN and GraphSAGE's aggregation functions fail to distinguish some basic multi-sets; hence **not injective**.
  - Therefore, GCN and GraphSAGE are **not** maximally powerful GNNs.

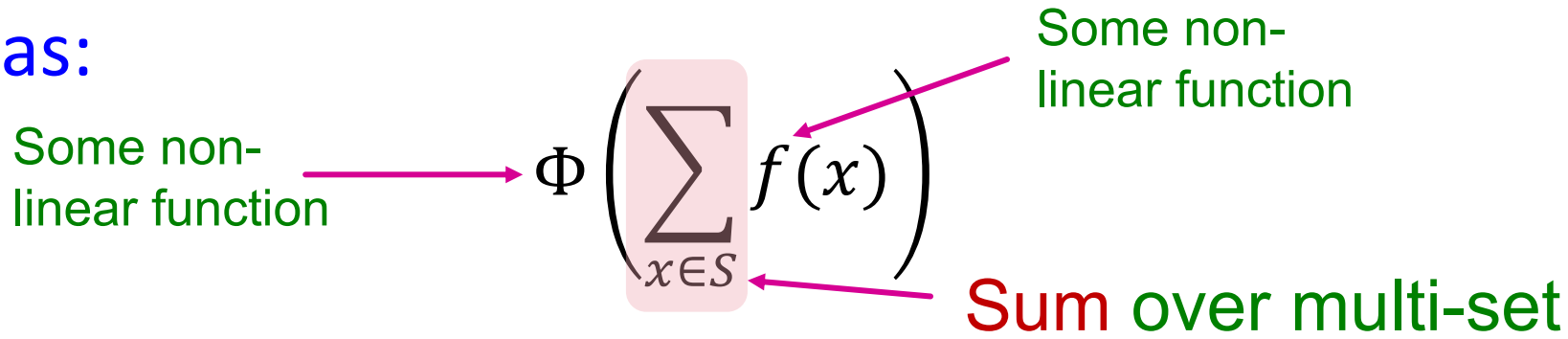
# Designing Most Expressive GNNs

- **Our goal**: **Design maximally powerful GNNs in the class of message-passing GNNs.**
- This can be achieved by designing **injective neighbor aggregation function over multi-sets**.
- **Here, we design a neural network that can model injective multiset function.**

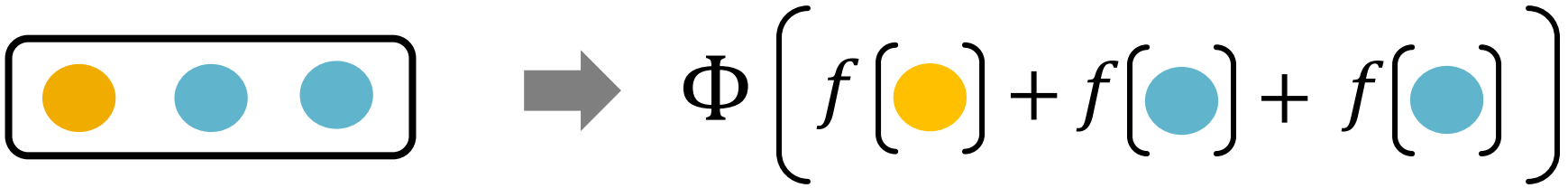
# Injective Multi-Set Function

**Theorem** [Xu et al. ICLR 2019]

Any injective multi-set function can be expressed as:




$S$ : multi-set



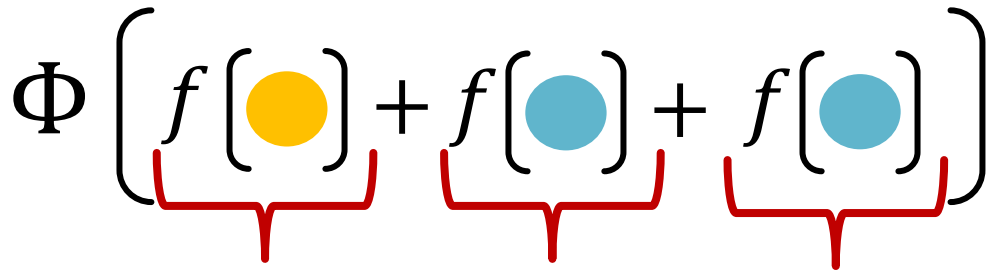
# Injective Multi-Set Function

**Proof Intuition:** [Xu et al. ICLR 2019]

$f$ produces one-hot encodings of colors. Summation of the one-hot encodings retains all the information about the input multi-set.

$$\Phi\left(\sum_{x \in S} f(x)\right)$$

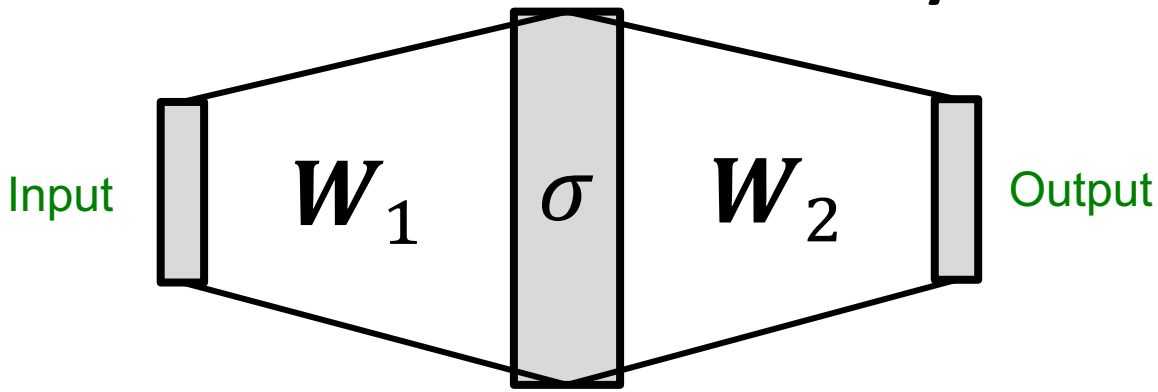
Example:



One-hot $\begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1\\2\end{pmatrix}$

# Universal Approximation Theorem

- **How to model $\Phi$ and $f$ in $\Phi\left(\sum_{x \in S} f(x)\right)$ ?**
- We use a Multi-Layer Perceptron (MLP).
- **Theorem: Universal Approximation Theorem**
  [Hornik et al., 1989]
  - 1-hidden-layer MLP with sufficiently-large hidden dimensionality and appropriate non-linearity $\sigma(\cdot)$ (including ReLU and sigmoid) can **approximate any continuous function to an arbitrary accuracy**.



# Injective Multi-Set Function

- **We have arrived at a neural network that can model any injective multiset function.**

$$\text{MLP}_\Phi\left(\sum_{x \in S} \text{MLP}_f(x)\right)$$

  - In practice, MLP hidden dimensionality of 100 to 500 is sufficient.

# Most Expressive GNN

- **Graph Isomorphism Network (GIN)** [Xu et al. ICLR 2019]
  - Apply an MLP, element-wise **sum**, followed by another MLP.

$$\mathrm{MLP}_{\Phi}\left(\sum_{x \in S} \mathrm{MLP}_{f}(x)\right)$$

- **Theorem** [Xu et al. ICLR 2019]
  - GIN's neighbor aggregation function is injective.
- **No failure cases!**
- **GIN is THE most expressive GNN in the class of message-passing GNNs!**

# Full Model of GIN

- **So far:** We have described the neighbor aggregation part of GIN.

- We now describe the full model of GIN by relating it to **WL graph kernel** (traditional way of obtaining graph-level features).
  - We will see how GIN is a "neural network" version of the WL graph kernel.

# Relation to WL Graph Kernel

**Recall: Color refinement algorithm** in WL kernel.

- **Given:** A graph $G$ with a set of nodes $V$.
  - Assign an initial color $c^{(0)}(v)$ to each node $v$.
  - Iteratively refine node colors by

$$c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right),$$

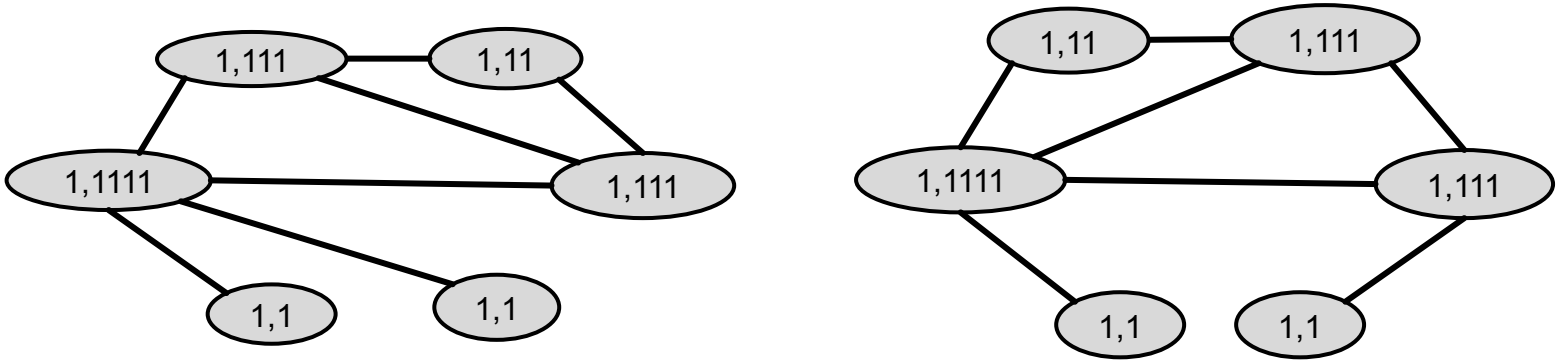
    where HASH maps different inputs to different colors.
  - After $K$ steps of color refinement, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood

# Color Refinement (1)

## Example of color refinement given two graphs

- Assign initial colors

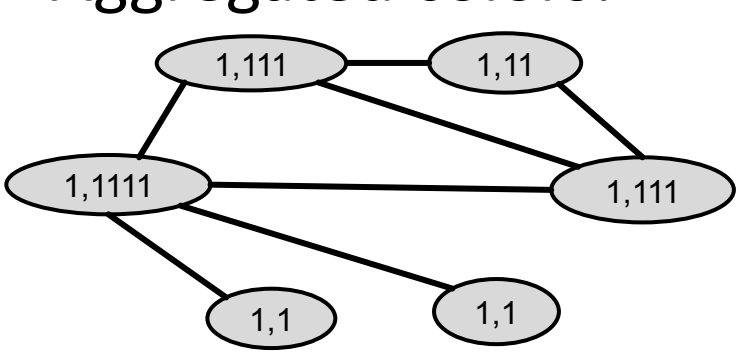
- Aggregate neighboring colors

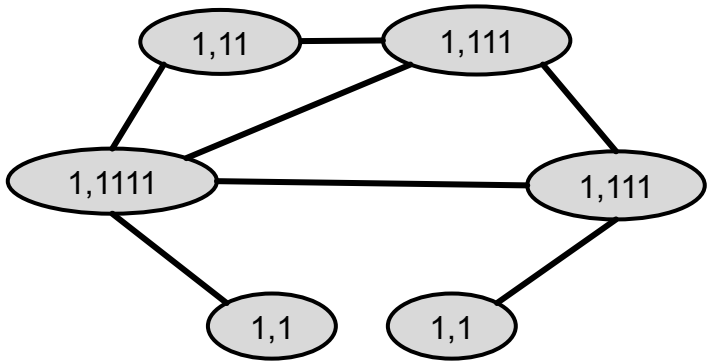


# Color Refinement (2)

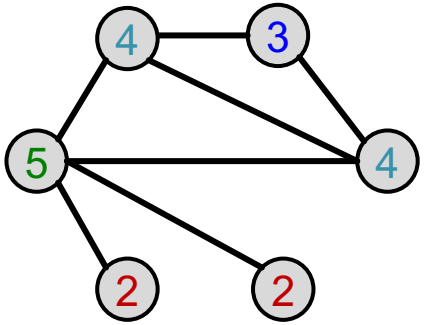
## Example of color refinement given two graphs

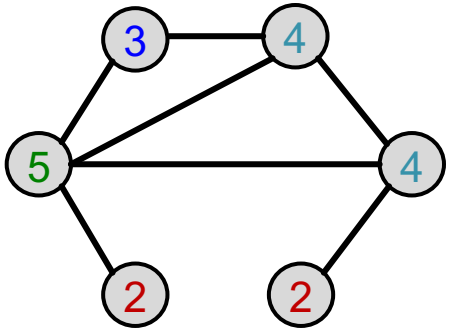
- Aggregated colors:




- **Injectively** HASH the aggregated colors



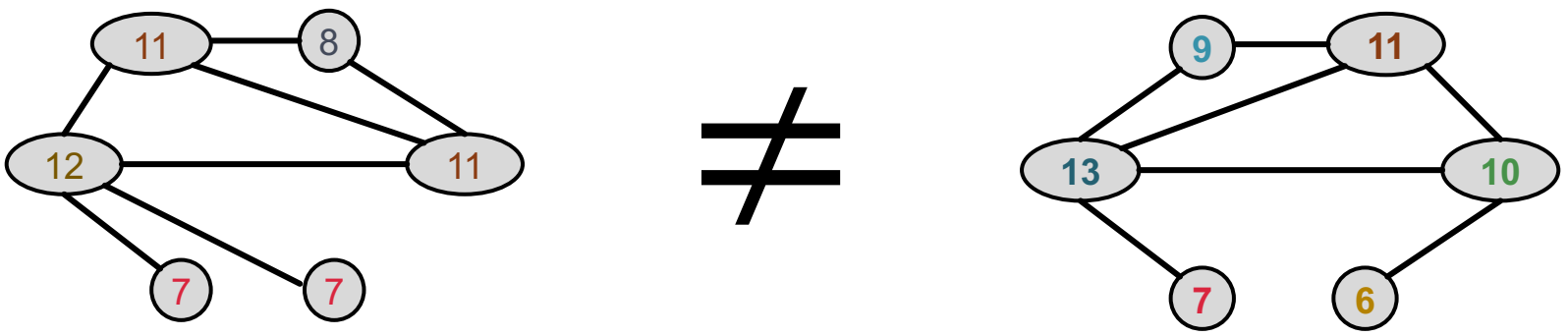

HASH table: **Injective!**

| | | |
|---|---|---|
| 1,1 | --> | 2 |
| 1,11 | --> | 3 |
| 1,111 | --> | 4 |
| 1,1111 | --> | 5 |

# Color Refinement (3)

## Example of color refinement given two graphs

- Process continues until a stable coloring is reached
- Two graphs are considered **isomorphic** if they have the same set of colors.



# The Complete GIN Model

- **GIN uses a neural network to model the injective HASH function.**

$$c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

- Specifically, we will model the injective function over the tuple:

$$\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

Root node features

Neighboring node colors

# The Complete GIN Model

**Theorem** (Xu et al. ICLR 2019)

Any injective function over the tuple

Root node feature: $c^{(k)}(v)$

Neighboring node features: $\{c^{(k)}(u)\}_{u \in N(v)}$

$$\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

can be modeled as

$$\mathrm{MLP}_\Phi\left((1+\epsilon)\cdot \mathrm{MLP}_f(c^{(k)}(v)) + \sum_{u \in N(v)} \mathrm{MLP}_f(c^{(k)}(u))\right)$$

where $\epsilon$ is a learnable scalar.

# The Complete GIN Model

- If input feature $c^{(0)}(v)$ is represented as one-hot, **direct summation is injective**.

Example: $\Phi\left(\begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix}\right) = \begin{pmatrix}1\\2\end{pmatrix}$

- We only need $\Phi$ to ensure the injectivity.

$$\text{GINConv}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u\in N(v)}\right) = \text{MLP}_\Phi\left((1+\epsilon)\cdot c^{(k)}(v) + \sum_{u\in N(v)} c^{(k)}(u)\right)$$

Root node features

Neighboring node features

This MLP can provide "one-hot" input feature for the next layer.

# The Complete GIN Model

- **GIN's node embedding updates**
- **Given:** A graph $G$ with a set of nodes $V$.
  - Assign an **initial vector** $c^{(0)}(v)$ to each node $v$.
  - Iteratively update node vectors by

$$c^{(k+1)}(v) = \text{GINConv}\left(\left\{c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right\}\right),$$

Differentiable color HASH function

where GINConv maps different inputs to different embeddings.

  - After $K$ steps of GIN iterations, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood.

# GIN and WL Graph Kernel

- **GIN can be understood as differentiable neural version of the WL graph Kernel:**

| | Update target | Update function |
|---|---|---|
| **WL Graph Kernel** | Node colors (one-hot) | HASH |
| **GIN** | Node embeddings (low-dim vectors) | GINConv |

- **Advantages of GIN over the WL graph kernel are:**
  - Node embeddings are **low-dimensional**; hence, they can capture the fine-grained similarity of different nodes.
  - Parameters of the update function can be **learned for the downstream tasks.**

# Expressive Power of GIN

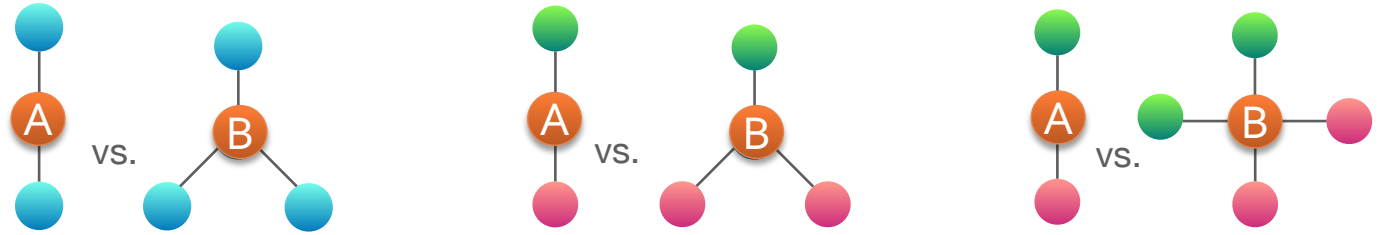
- **Because of the relation between GIN and the WL graph kernel, their expressive is exactly the same.**
  - If two graphs can be distinguished by GIN, they can be also distinguished by the WL kernel, and vice versa.
- **How powerful is this?**
  - WL kernel has been both theoretically and empirically shown to distinguish most of the real-world graphs [Cai et al. 1992].
  - Hence, GIN is also powerful enough to distinguish most of the real graphs!

# Summary of the Lecture

- We design a neural network that can model injective multi-set function.
- We use the neural network for neighbor aggregation function and arrive at **GIN**---the most expressive GNN model.
- The key is to use **element-wise sum pooling**, instead of mean-/max-pooling.
- GIN is closely related to the WL graph kernel.
- Both GIN and WL graph kernel can distinguish most of the real graphs!

# The Power of Pooling

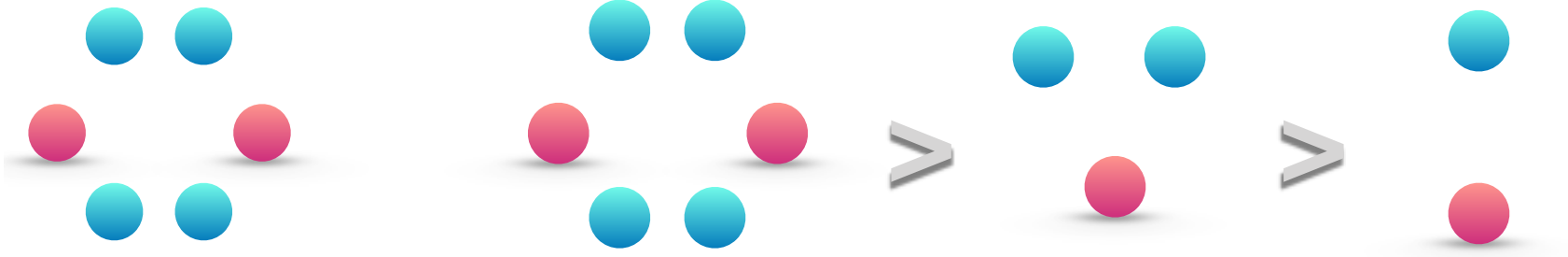
## Failure cases for mean and max pooling:




(a) Mean and Max both fail (b) Max fails (c) Mean and Max both fail

Colors represent feature values

## Ranking by discriminative power:

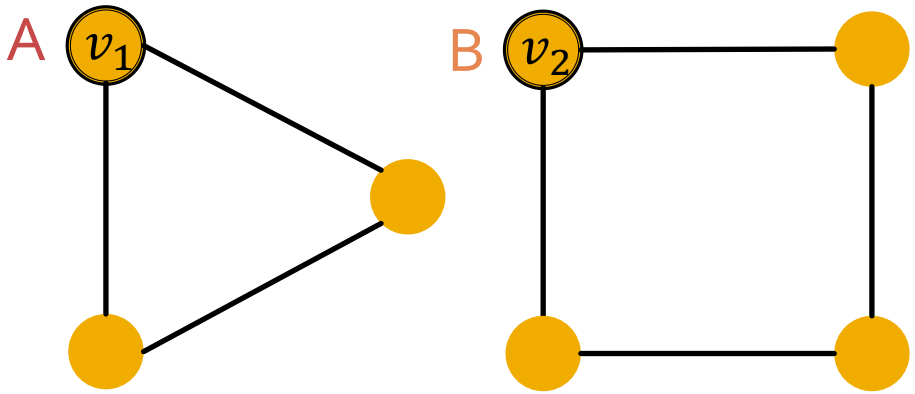



**Input** sum - multiset mean - distribution max - set

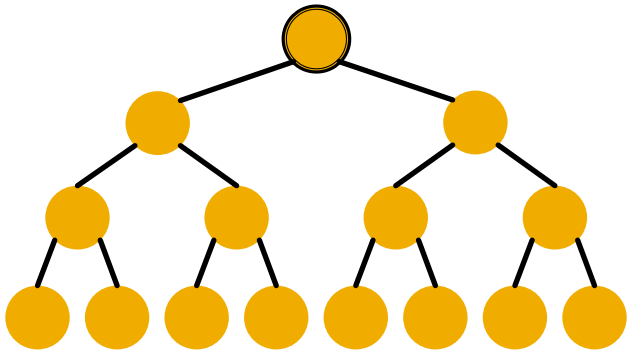
# Improving GNNs' Power

- **Can expressive power of GNNs be improved?**
  - There are basic graph structures that existing GNN framework cannot distinguish, such as difference in cycles.

Graphs




Computational graphs for nodes $v_1$ and $v_2$:



  - GNNs' expressive power **can be improved** to resolve the above problem. [You et al. AAAI 2021, Li et al. NeurIPS 2020]

# Summary

- GNNs and connection to bijective functions on sets.
- Most powerful GNN is equivalent to WL graph isomorphism test.
- GIN is the most powerful GNN.
  - Sum aggregator is more powerful than mean is more powerful than max.