# CS224W Homework 1

October 19, 2023

## 1 GNN Expressiveness (28 points)

**For Q1.1, write down number of layers needed. For Q1.2, write down the transition matrix $M$ and the limiting distribution $r$. For Q1.3 and 1.4, write down the transition matrix w.r.t $A$ and $D$. For Q1.5, write down your proof in a few sentences (equations if necessary). For Q1.6, describe the message function, aggregate function, and update rule in a few sentences or equations.**

Graph Neural Networks (GNNs) are a class of neural network architectures used for deep learning on graph-structured data. Broadly, GNNs aim to generate high-quality embeddings of nodes by iteratively aggregating feature information from local graph neighborhoods using neural networks; embeddings can then be used for recommendations, classification, link prediction, or other downstream tasks. Two important types of GNNs are GCNs (graph convolutional networks) and GraphSAGE (graph sampling and aggregation).

Let $G = (V, E)$ denote a graph with node feature vectors $X_u$ for $u \in V$. To generate the embedding for a node $u$, we use the neighborhood of the node as the computation graph. At every layer $l$, for each pair of nodes $u \in V$ and its neighbor $v \in V$, we compute a message function via neural networks, and apply a convolutional operation that aggregates the messages from the node's local graph neighborhood (Figure 1.1), and updates the node's representation at the next layer. By repeating this process through $K$ GNN layers, we capture feature and structural information from a node's local $K$-hop neighborhood. For each of the message computation, aggregation, and update functions, the learnable parameters are shared across all nodes in the same layer.

We initialize the feature vector for node $X_u$ based on its individual node attributes. If we already have outside information about the nodes, we can embed that as a feature vector. Otherwise, we can use a constant feature (vector of 1) or the degree of the node as the feature vector.

These are the key steps in each layer of a GNN:

- **Message computation**: We use a neural network to learn a message function between nodes. For each pair of nodes $u$ and its neighbor $v$, the neural network message function can be expressed as $M(h_u^k, h_v^k, e_{u,v})$. In GCN and GraphSAGE, this can simply be $\sigma(Wh_v + b)$, where $W$ and $b$

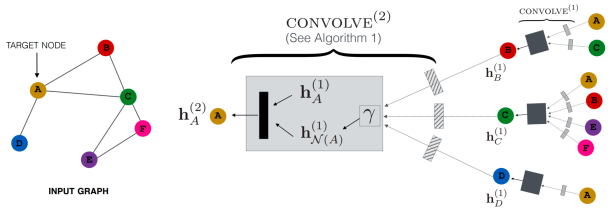



Figure 1.1: GNN architecture

are the weights and bias of a neural network linear layer. Here $h_u^k$ refers to the hidden representation of node $u$ at layer $k$, and $e_{u,v}$ denotes available information about the edge $(u, v)$, like the edge weight or other features. For GCN and GraphSAGE, the neighbors of $u$ are simply defined as nodes that are connected to $u$. However, many other variants of GNNs have different definitions of neighborhood.

- **Aggregation**: At each layer, we apply a function to aggregate information from all of the neighbors of each node. The aggregation function is usually permutation invariant, to reflect the fact that nodes' neighbors have no canonical ordering. In a GCN, the aggregation is done by a weighted sum, where the weight for aggregating from $v$ to $u$ corresponds to the $(u, v)$ entry of the normalized adjacency matrix $D^{-1/2}AD^{-1/2}$.

- **Update**: We update the representation of a node based on the aggregated representation of the neighborhood. For example, in GCNs, a multi-layer perceptron (MLP) is used; Graph-SAGE combines a skip layer with the MLP.

- **Pooling**: The representation of an entire graph can be obtained by adding a pooling layer at the end. The simplest pooling methods are just taking the mean, max, or sum of all of the individual node representations. This is usually done for the purposes of graph classification.

We can formulate the Message computation, Aggregation, and Update steps for a GCN as a layer-wise propagation rule given by:

$$h^{k+1} = \sigma(D^{-1/2}AD^{-1/2}h^k W^k) \tag{1}$$

where $h^k$ represents the matrix of activations in the $k$-th layer, $D^{-1/2}AD^{-1/2}$ is the normalized adjacency of graph $G$, $W_k$ is a layer-specific learnable matrix, and $\sigma$ is a non-linearity function. Dropout and other forms of regularization can also be used.

We provide the pseudo-code for GraphSAGE embedding generation below. This will also be relevant to the questions below.

**Algorithm 1:** Pseudo-code for forward propagation in GraphSAGE

**Input :** Graph $G(V, E)$; input features $\{x_v, \forall v \in V\}$; depth $K$;
non-linearity $\sigma$; weight matrices $\{W^k, \forall k \in [1, K]\}$;
neighborhood function $\mathcal{N} : v \rightarrow 2^V$;
aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$

**Output:** Vector representations $z_v$ for all $v \in V$

$h_v^0 \leftarrow x_v, \forall v \in V$ ;
**for** $k = 1...K$ **do**
  **for** $v \in V$ **do**
    $h_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ // aggregation
    $h_v^k \leftarrow \sigma\left(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{\mathcal{N}(v)}^k)\right)$ // MLP with skip connection
  $h_v^k \leftarrow h_v^k / \|h_v^k\|_2, \forall v \in V$ // update step
$z_v \leftarrow h_v^K, \forall v \in V$

In this question, we investigate the effect of the number of message passing layers on the expressive power of Graph Convolutional Networks. In neural networks, expressiveness refers to the set of functions (usually the loss function for classification or regression tasks) a neural network is able to compute, which depends on the structural properties of a neural network architecture.

.

## 1.1 Effect of Depth on Expressiveness (4 points)

Consider the following 2 graphs in figure 1.2, where all nodes have 1-dimensional initial feature vector $x = [1]$. We use a simplified version of GNN, with no non-linearity, no learned linear transformation, and sum aggregation. Specifically, at every layer, the embedding of node $v$ is updated as the sum over the embeddings of its neighbors ($N_v$) and its current embedding $h_v^t$ to get $h_v^{t+1}$. We run the GNN to compute node embeddings for the 2 red nodes respectively. Note that the 2 red nodes have different 5-hop neighborhood structure (note this is not the minimum number of hops for which the neighborhood structure of the 2 nodes differs). How many layers of message passing are needed so that these 2 nodes can be distinguished (i.e., have different GNN embeddings)? Explain your answer in a few sentences.

.

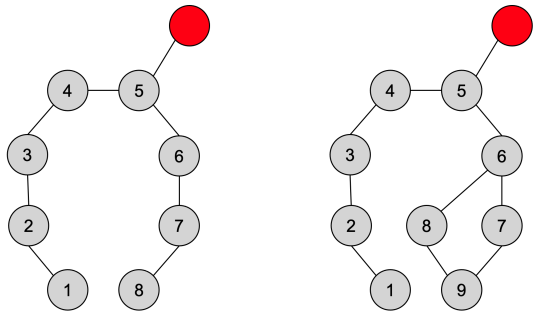



Figure 1.2: Figure for Question 1.1

★ **Solution** ★ 3 layers of message passing are needed so that the two nodes can be distinguished. This is do to the affect of the neighborhood around node 6. Node 6 on the right graph increases the sum aggregation at a greater rate than on the left, as it has more direct neighbors. This takes a few rounds to have an affect, as the sum aggregation operation uses the previous layer to calculate the next layer. I've included a table below that shows my calculations for the red node on each side at each layer.

Table 1: Sum Aggregation values per layer of red nodes

| | Left Red Node | Right Red Node |
|---|---|---|
| Layer 0 | 1 | 1 |
| Layer 1 | 2 | 2 |
| Layer 2 | 6 | 6 |
| Layer 3 | 18 | 19 |

## 1.2 Random Walk Matrix (4 points)

Consider the graph shown below (figure 1.3).

1. Assume that the current distribution over nodes is $r = [0, 0, 1, 0]$, and after the random walk, the distribution is $M \cdot r$. What is the random walk transition matrix $M$, where each row of $M$ corresponds with the node ID in the graph?

2. What is the limiting distribution $r$, namely the eigenvector of $M$ that has an eigenvalue of 1 ($r = Mr$)? Write your answer in fraction form or round it to the nearest thousandth place and in the following form, e.g. $[1.200, 0.111, 0.462, 0.000]$. Note that before reporting you should normalize $r$ (Hint: $r$ is a probability distribution).

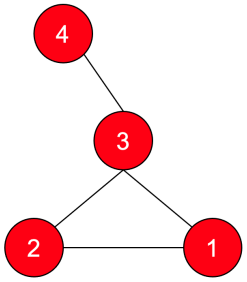



Figure 1.3: Figure for Question 1.2

★ **Solution** ★ For this question I set the nodes column wise aligning to Markov matrix as they take column vectors. I also assumed that each stage would have the same probabilities for the creation of the matrix i.e. Node 3 would have a 1/3 probability of reaching node 1, 2, or 4 as there are 3 choices. The following is the matrix with columns for the nodes:

$$\begin{bmatrix} 0 & 0.5 & 1/3 & 0 \\ 0.5 & 0 & 1/3 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1/3 & 0 \end{bmatrix}$$

For the limiting distribution r, I calculated the eigenvector of M using the following set of linear equations:

(1) $r_1 + r_2 + r_3 + r_4 = 1$
(2) $-r_1 + 0.5r_2 + 1/3r_3 = 0$
(3) $0.5r_1$ - $r_2 + 1/3r_3 = 0$
(4) $0.5r_1 + 0.5r_2$ - $r_3 + r_4 = 0$
(5) $1/3r_3$ - $r_4 = 0$

This set of equations ultimately gave me the following distribution:

$r =$ [ 0.25, 0.25, 0.375, 0.125]

## 1.3 Relation to Random Walk (i) (4 points)

Let's explore the similarity between message passing and random walks. Let $h_i^{(l)}$ be the embedding of node $i$ at layer $l$. Suppose that we are using a mean aggregator for message passing, and omit the learned linear transformation and non-linearity: $h_i^{(l+1)} = \frac{1}{|N_i|} \sum_{j \in N_i} h_j^{(l)}$. If we start at a node $u$ and take a uniform random walk for 1 step, the expectation over the layer-$l$ embeddings of nodes we can end up with is $h_u^{(l+1)}$, exactly the embedding of $u$ in the next

GNN layer. What is the transition matrix of the random walk? Describe the transition matrix using the adjacency matrix $A$, and degree matrix $D$, a diagonal matrix where $D_{i,i}$ is the degree of node $i$.

★ **Solution** ★

The equation above represents the behavior of message passing in GNNs. For a node $u$, the embedding at the next layer $(l+1)$ is computed as the average (mean aggregator) of the embeddings of its neighboring nodes at layer $l$. The average embedding that is calculated represents the "message" that is passed to node $u$. For a random walk, if you start at node $u$ and take a uniform random walk for 1 step, you'd move randomly to one of its neighbors. The embedding of this neighbor node is exactly the average embedding of node $u$ neighbors. The following is the transition matrix $T$ of the random walk:

$T = D^{-1}A$

Where:
- $A$ is the adjacency matrix
- $D$ is the degree matrix

The entries of the transition matrix are defined by dividing each entry of the adjacency matrix (the connections) by the degree, hence why $D$ in the equation above is $D^{-1}$. The transition matrix $T$ represents the probability of moving from one node to another in a single step of the random walk.

## 1.4 Relation to Random Walk (ii) (4 points)

Suppose that we add a skip connection to the aggregation from Question 1.3:

$$h_i^{(l+1)} = \frac{1}{2}h_i^{(l)} + \frac{1}{2}\frac{1}{|N_i|}\sum_{j \in N_i} h_j^{(l)}$$

What is the new corresponding transition matrix?

★ **Solution** ★ Given the above updated agreggation function, with the skip connection a node not only transitions to its neighbors, but also stays with half the probability. The probability for a node to stay is $1/2$, and to transition to a neighbor is $1/2$ * $1/degree(i)$. It adds a self-loop to each node that will pass a message to itself along with its neighbors. The new transition matrix $T'$ is as follows:

$T' = \frac{1}{2}I + \frac{1}{2}D^{-1}A$

Where:

- $I$ is the identity matrix that contributes to the self-loop
- $\frac{1}{2}D^{-1}A$ yields the reduced transition probabilities due to the skip connection

### 1.5 Over-Smoothing Effect (5 points)

In Question 1.1 we see that increasing depth could give more expressive power. On the other hand, however, a very large depth also gives rise to the undesirable effect of over smoothing. Assume we are still using the aggregation function from Question 1.3: $h_i^{(l+1)} = \frac{1}{|N_i|} \sum_{j \in N_i} h_j^{(l)}$. Show that the node embedding $h^{(l)}$ will converge as $l \to \infty$. Here we assume that the graph is connected and has no bipartite components. We also assume that the graph is undirected.

Over-smoothing thus refers to the problem of node embedding convergence. Namely, if all node embeddings converge to the same value then we can no longer distinguish them and our node embeddings become useless for downstream tasks. However, in practice, learnable weights, non-linearity, and other architecture choices can alleviate the over-smoothing effect.

**Hint: Think about the Markov Convergence Theorem: Is the Markov chain irreducible and aperiodic? You don't need to be super rigorous with your proof.**

**★ Solution ★**

The aggregation function from question 1.3 states that the new embedding of node $i$ is the average of its neighbors embeddings. Repeated application of this function will result in a node's embedding depending on embeddings of nodes further away from it.

Relating this to Markov Convergence Theorem, the aggregation scheme can be described by a Markov chain. The steps in the aggregation function relates to a step in a random walk, which defines a Markov chain where the probability of transitioning from node $i$ to node $j$ is $\frac{1}{N_i}$ if $j$ is a neighbor of $i$ and 0 if not. As stated above we can assume that the graph is connected, has no bipartite components, is undirected, and Markov chains have 2 distinct properties in this context:

1. Irreducible: A Markov chain is irreducible if there's a non-zero probability of getting from any state to any other state in a finite series of steps. The graph as stated above is connected, which meets this criteria
2. Aperiodic: As defined above the graph is undirected and has no bipartite components. Thusly,it is aperiodic - there are no integer $k$ ¿ 1 that divides the length of every cyvle of the graph. Conversely, if the graph were directed and bipartite, all cycles would have a length that is divisible by 2, and thusly, they cannot be aperiodic.

An irreducible and aperiodic chain has a unique stationary distribution to which it will converge on. Since the above aggregate function is described by a Markov Chain with the properties listed above, as $l \to \infty$ the embeddings will converge to a stationary distribution, which will result in them become more similar to one another, and result in them being indistinguishable from one another. Given a deep enough architecture, this behavior will cause and "Over-Smoothing" effect, which leads to the loss of node-specific information.

### 1.6 Learning BFS with GNN (7 points)

Next, we investigate the expressive power of GNN for learning simple graph algorithms. Consider breadth-first search (BFS), where at every step, nodes that are connected to already visited nodes become visited. Suppose that we use GNN to learn to execute the BFS algorithm. Suppose that the embeddings are 1-dimensional. Initially, all nodes have input feature 0, except a source node which has input feature 1. At every step, nodes reached by BFS have embedding 1, and nodes not reached by BFS have embedding 0. Describe a message function, an aggregation function, and an update rule for the GNN such that it learns the task perfectly.

★ **Solution** ★ Goal: describe the set of functions for the GNN to learn BFS. Initializiation of all nodes have input feature of 0, with the source node as the exception with an input value of 1:

Message Function: $M(u, v) = h_v^{(l)}$
- Function that passes the current embedding of node $v$ to its neighbor $v$.

Aggregation Function: $A(u) = max_{v \in \mathcal{N}(u)} M(u, v)$
- Goal is to detect if a neighboring node has been visited. Selecting a max function as it can sufficiently meet the requirements very easily for 1-dimensional embeddings. If any neighbor $v$ of node $u$ has an embedding of 1, it will return 1, if all neighbors have not been visited, it will return 0.

Update Rule: $h_u^{(l+1)} = \begin{cases} 1 & \text{if } A(u) = 1 \text{ and } h_u^{(l)} = 0 \\ h_u^{(l)} & \text{otherwise} \end{cases}$
- The update rule has the embeddeding node $u$ at layer $l + 1$ set to 1 if: the aggregated message function return is 1, at least one of it's neighbors was visited in a prior layer, AND the current state of the node hasn't been visited. If the conditions are not met, then the embedding maintains the same as prior layers. Starting from the source node, nodes will get visited similar to the BFS algorithm: marking nodes as visited in the next layer if the node is adjacent to an already visited node.

## 2 Node Embedding and its Relation to Matrix Factorization (24 points)

**What to submit: For Q2.1, one or more sentences/equations describing the decoder. For Q2.2, write down the objective function. For Q2.3, describe the characteristics of $W$ in one or more sentences. For Q2.4, write down the objective function. For Q2.5, characterize the embeddings, whether you think it will reflect structural similarity, and your justification. For Q2.6, one or more sentences for node2vec and struct2vec respectively. For Q2.7, one or more sentences of expla-**

**nation. For Q2.8, one or more sentences characterizing embeddings from struct2vec.**

Recall that matrix factorization and the encoder-decoder view of node embeddings are closely related. For the embeddings, when properly formulating the encoder-decoder and the objective function, we can find equivalent matrix factorization formulation approaches.

Note that in matrix factorization we are optimizing for L2 distance; in encoder-decoder examples such as DeepWalk and node2vec, we often use log-likelihood as in lecture slides. The goal to approximate A with $Z^T Z$ is the same, but for this question, stick with the L2 objective function.

## 2.1 Simple matrix factorization (3 points)

In the simple matrix factorization, the objective is to approximate adjacency matrix A by the product of embedding matrix with its transpose. The optimization objective is $\min_Z ||A - Z^T Z||_2$.

In the encoder-decoder perspective of node embeddings, what is the decoder? (Please provide a mathematical expression for the decoder)

★ **Solution** ★ In the encoder-decoder perspective of node embeddings, the encoder maps each node to a vector in the embedding space. The decoder then takes these embeddings and reconstructs the weights/similarity between nodes as represented in the adjacency matrix. Given two nodes with embeddings $z_i$ and $z_j$, the decoder produces and approximate entry in the adjacency matrix. Ultimately this is the dot product of nodes in the embedding space to approximate the adjacency matrix.
$A_{ij} \approx D(z_i, z_j) = z_i^T z_j$

## 2.2 Alternate matrix factorization (3 points)

In linear algebra, we define bilinear form as $z_i^T W z_j$, where W is a matrix. Suppose that we define the decoder as the bilinear form, what would be the objective function for the corresponding matrix factorization? (Assume that the $W$ matrix is fixed)

★ **Solution** ★ Leveraging the matrix notation of the optimization objective in 2.1, the adjusted objective function that meets this definition would be as follows:

$J = ||A - Z^T W Z||_2$

- This function minimizes the difference between the adjacency matrix and the matrix obtained using the bilinear form.

### 2.3 BONUS: Relation to eigen-decomposition (3 points)

Recall eigen-decomposition of a matrix (link). What would be the condition of W, such that the matrix factorization in the previous question (2.2) is equivalent to learning the eigen-decomposition of matrix A?

★ **Solution** ★

### 2.4 Multi-hop node similarity (3 points)

Define node similarity with the multi-hop definition: 2 nodes are similar if they are connected by at least one path of length at most k, where k is a parameter (e.g. $k = 2$). Suppose that we use the same encoder (embedding lookup) and decoder (inner product) as before. What would be the corresponding matrix factorization problem we want to solve?

★ **Solution** ★ For a single hop, the adjacency matrix $A$ defines the relationships between nodes as 1 if nodes are directly connected and 0 otherwise. For Multi-hop connections, we can raise the matrix to the power of $k$ which will capture all paths of length up to $k$. If there's one path of length up to $k$ between nodes, then the entry will be non-zero.

$J = \sum_{(i,j)}(A^k - z_i^T z_j)^2$

### 2.5 node2vec & struct2vec (i) (3 points)

Finally, we'll explore some limitations of node2vec that are introduced in the lecture, and look at algorithms that try to overcome them.

As mentioned in the lecture, due to the way random walk works, it's hard for node2vec to learn structural embedding from the graph. Think about how a new algorithm called **struct2vec** works. For this question, we define a **clique** to be a fully connected graph, where any two nodes are connected.

Given a graph $G(V, E)$, it defines K functions $g_k(u, v)$, $k = 1, 2, .., K$, which measure the structural similarity between nodes. The parameter k means that only the local structures within distance k of the node are taken into account. With all the nodes in G, regardless of the existing edges, it forms a new clique graph where any two nodes are connected by an edge whose weight is equal to the structural similarity between them. Since struct2vec defines K structural similarity functions, each edge has a set of possible weights corresponding to $g_1, g_2, ..., g_K$.

The random walks are then performed on the clique. During each step, weights

are assigned according to different $g_k$'s selected by some rule (omitted here for simplification). Then, the algorithm chooses the next node with probability proportional to the edge weights.

Characterize the vector representations (i.e. the embedding space) of the 10-node cliques after running the **node2vec** algorithm on the graph in figure 2.1. Assume through the random walk, nodes that are close to each other have similar embeddings. Do you think the node embeddings will reflect the structural similarity? Justify your answer.

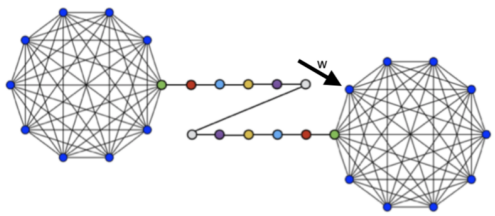



Figure 2.1: Two 10-node cliques

**★ Solution ★**

Node2vec's intent is to capture node embeddings via biased random walks. Looking at the barbell figure and running node2vec on it will result in it assign different embeddings for each of the 10 node cliques. This is due to node2vec capturing aspects of the local neighborhood due to the structure of its random walks.

The global structure of the graph will be poorly reflected due to the distance separation between the barbells. When Node2vec is run on the graph, nodes within each clique will be likely to have similar embeddings, as any node in the clique is a single step from another node. The chaining of the two cliques with the middle section in the graph introduces distance and variance in the random walk neighborhood, which will likely yield differences in the embeddings between the two cliques.

Ultimately, this means that even though the structures of the 2 10-node cliques are identical, node2vec will assign them different embeddings, and might not represent the structural similarity of the graph due to the walk-based distance/proximity.

.

### 2.6 node2vec & struct2vec (ii) (3 points)

In the above figure 2.1, suppose you arrive at node w. What are the nodes that you can reach after taking one step further with the node2vec algorithm? What about with the struct2vec algorithm (suppose that for this graph, $g_k(u, v) > 0$ for any u,v,k)?

★ **Solution** ★
Node2vec:
- When at node $w$ in the node2vec algorithm you can reach any node that is directly connected to $w$ by an edge. Looking at the graph, this indicates you'd be able to reach the green node, and any blue node in the rightmost clique.

Struct2vec:
- When at node $w$, struct2vec has the ability to form a new clique graphy where any two nodes are connnected by an edges whose weight is equal to the structural similarity between them. Given $g_k(u, v) > 0$ for any $u, v, k$, every node has some level of similarity with every other node. Struct2vec would technically allow you to reach any node in the graph after a step as they are all connected in the clique graph it forms. It incorporates the global structural context of the graph.

### 2.7 node2vec & struct2vec (iii) (3 points)

Why is it necessary to consider different $g_k$'s during the random walk?

★ **Solution** ★ Considering different $g_k$'s allows for the potential for enriched embeddings. They can capture various local structural properties of the graph at different scales/distances. For example, adjusting the values could enable the algorithm to measure direct neighbors at one layer, while the next may be neighbors of neighbors. Contrasting the scales allows for a richer set of structural features. It could also assist in lower the impact of local noise or irregularities in the local neighborhoods. Finally, it could increase the diversity sampling a broader specturm of structural information, again potentially enhancing the quality of the embeddings. The main impact of considering different $g_k$'s is the ability to yield more informative and versatile embeddings.

### 2.8 node2vec & struct2vec (iv) (3 points)

Characterize the vector representations (i.e. the embedding space) of the two 10-node cliques after running the struct2vec algorithm on the graph in the above figure (Figure 2.1).

★ **Solution** ★ Given that each clique is structurally similar (same local neighborhood), nodes within each of the 2 10-node cliques will have embeddings that are very close to each other in the embedding space. This is due to struct2vec considering their structural roles to be identical or very similar. The exact positioning

and differentiation in the embedding space will be influenced by the selected $g_k$ functions and their effects on different structural scales.

# 3 GCN (11 points)

Consider a graph $G = (V, E)$, with node features $x(v)$ for each $v \in V$. For each node $v \in V$, let $h_v^{(0)} = x(v)$ be the node's initial embedding. At each iteration $k$, the embeddings are updated as

$$h_{\mathcal{N}(v)}^{(k)} = \text{AGGREGATE}\left(\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right)$$
$$h_v^{(k)} = \text{COMBINE}\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right),$$

for some functions AGGREGATE$(\cdot)$ and COMBINE$(\cdot)$. Note that the argument to the AGGREGATE$(\cdot)$ function, $h_u^{(k-1)}, \forall u \in \mathcal{N}(v)$, is a *multi-set*. That is, since multiple nodes can have the same embedding, the same element can occur in $h_u^{(k-1)}, \forall u \in \mathcal{N}(v)$ multiple times. Finally, a graph itself may be embedded by computing some function applied to the multi-set of all the node embeddings at some final iteration $K$, which we notate as

$$\text{READOUT}\left(\left\{h_v^{(K)}, \forall v \in V\right\}\right)$$

We want to use the graph embeddings above to test whether two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic*. Recall that this is true if and only if there is some bijection $\phi : V_1 \to V_2$ between nodes of $G_1$ and nodes of $G_2$ such that for any $u, v \in V_1$,

$$(u, v) \in E_1 \Leftrightarrow (\phi(u), \phi(v)) \in E_2$$

The way we use the model above to test isomorphism is the following. For the two graphs, if their readout functions differ, that is

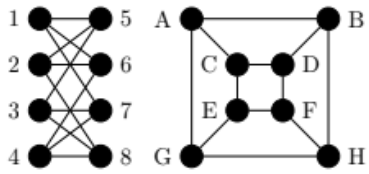
$$\text{READOUT}\left(\left\{h_v^{(K)}, \forall v \in V_1\right\}\right) \neq \text{READOUT}\left(\left\{h_v^{(K)}, \forall v \in V_2\right\}\right),$$

we conclude the graphs are *not* isomorphic. Otherwise, we conclude the graphs are isomorphic. Note that this algorithm is not perfect: graph isomorphism is thought to be hard! Below, we will explore the expressiveness of these graph embeddings.

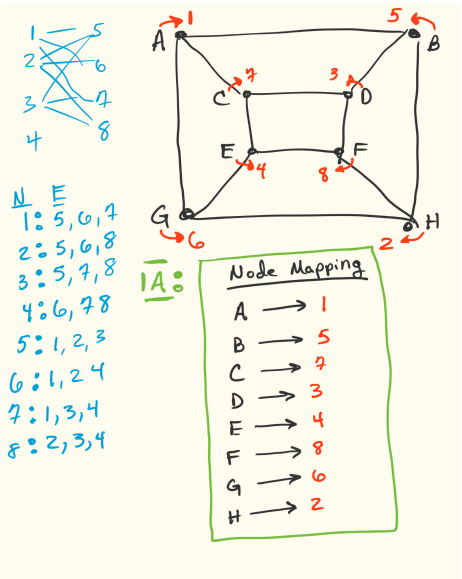
## 3.1 Isomorphism Check (2 points)

Are the following two graphs isomorphic? If so, demonstrate an isomorphism between the sets of vertices. To demonstrate an isomorphism between two graphs, you need to find a 1-to-1 correspondence between their nodes and edges.

If these two graphs are not isomorphic, prove it by finding a structure (node and/or edge) in one graph which is not present in the other.




★ **Solution** ★ Yes these 2 graphs are isomorphic. I've redrawn and compared the nodes for equivalence. See the following picture and node breakdown:




## 3.2 Aggregation Choice (3 points)

The choice of the AGGREGATE(·) is important for the expressiveness of the model above. Three common choices are:

$$\text{AGGREGATE}_{\text{max}}\left(\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right)_i = \max_{u\in\mathcal{N}(v)}\left(h_u^{(k-1)}\right)_i \text{ (element-wise max)}$$

$$\text{AGGREGATE}_{\text{mean}}\left(\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right) = \frac{1}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)}\left(h_u^{(k-1)}\right)$$

$$\text{AGGREGATE}_{\text{sum}}\left(\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right) = \sum_{u\in\mathcal{N}(v)}\left(h_u^{(k-1)}\right)$$

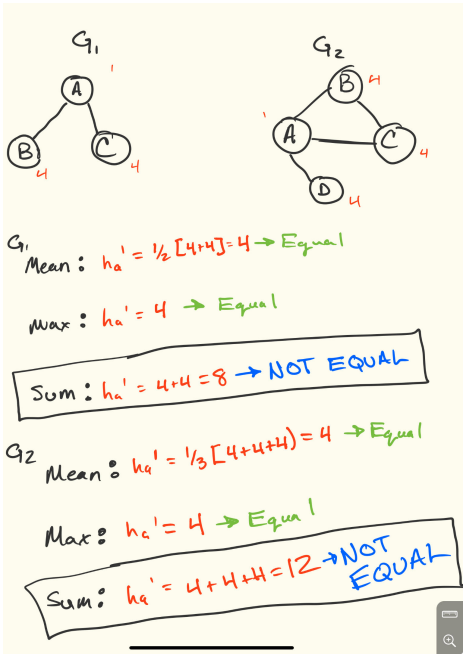
Give an example of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and their initial node features, such that for some node $v_1 \in V_1$ and some node $v_2 \in V_2$ with the same initial features $h_{v_1}^{(0)} = h_{v_2}^{(0)}$, the updated features $h_{v_1}^{(1)}$ and $h_{v_2}^{(1)}$ are equal if we use mean and max aggregation, but different if we use sum aggregation.
**Hint:** Your node features can be scalars rather than vectors, i.e. one dimensional node features instead of n-dimensional. Also, You are free to arbitrarily choose the number of nodes (e.g. 3 nodes), their connections (i.e. edges between nodes) in your example.

**★ Solution ★**

I've defined 2 graphs $G_1$ and $G_2$ in the attached image below. The graphs have the will be compared around node $A$, as they have the same starting embedding. As implemented below, the max and mean are equal for node $A$ in each graph; however, the sums are different.

$G_1$: $h_A^{(0)} = 1$, $h_B^{(0)} = 4$, $h_C^{(0)} = 4$
Max Aggregate: $h_A^{(1)} = \text{max}(4,4) = 4$
Mean Aggregate: $h_A^{(1)} = \text{mean}(4,4) = 4$
Sum Aggregate: $h_A^{(1)} = \text{sum}(4,4) = 8$

$G_2$: $h_A^{(0)} = 1$, $h_B^{(0)} = 4$, $h_C^{(0)} = 4$, $h_D^{(0)} = 4$
Max Aggregate: $h_A^{(1)} = \text{max}(4,4,4) = 4$
Mean Aggregate: $h_A^{(1)} = \text{mean}(4,4,4) = 4$
Sum Aggregate: $h_A^{(1)} = \text{sum}(4,4,4) = 12$




### 3.3 Weisfeiler-Lehman Test (6 points)

Our isomorphism-test algorithm is known to be at most as powerful as the well-known *Weisfeiler-Lehman test* (WL test). At each iteration, this algorithm updates the representation of each node to be the set containing its previous representation and the previous representations of all its neighbors. The full algorithm is below.

**Algorithm 3:** Weisfeiler-Lehman Test

**Data:** $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, initial features $x(\cdot)$, number of iterations $K$
**Result:** Prediction of whether $G_1$ and $G_2$ are isomorphic
**for** $v \in V_1 \cup V_2$ **do**
  $l_v^{(0)} \leftarrow x(v)$
**end**
**for** $k = 1, \ldots, K$ **do**
  **for** $v \in V_1 \cup V_2$ **do**
    $l_v^{(k)} \leftarrow \text{HASH}\left(l_v^{(k-1)}, \{l_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)$
  **end**
**end**
**return** $\left\{l_v^{(K)}, \forall v \in V_1\right\} = \left\{l_v^{(K)}, \forall v \in V_2\right\}$

Prove that our neural model is at most as powerful as the WL test. More precisely, let $G_1$ and $G_2$ be non-isomorphic, and suppose that their node embeddings are updated over $K$ iterations with the same AGGREGATE$(\cdot)$ and COMBINE$(\cdot)$ functions. Show that if

$$\text{READOUT}\left(\left\{h_v^{(K)}, \forall v \in V_1\right\}\right) \neq \text{READOUT}\left(\left\{h_v^{(K)}, \forall v \in V_2\right\}\right),$$

then the WL test also decides the graphs are not isomorphic.

Note: The proof has to be generic to any AGGREGATE, COMBINE, READOUT functions. Namely, it's not sufficient to show this for a specific instance of the GNN model.

**Hint:** You can use proof by contradiction by first assuming that *Weisfeiler-Lehman* test cannot decide whether $G_1$ and $G_2$ are isomorphic at the end of $K$'th iteration.

★ **Solution** ★