An introduction to

Graph Neural Network(GNN)

Part 2

Dr. Jamshaid Ul Rahman

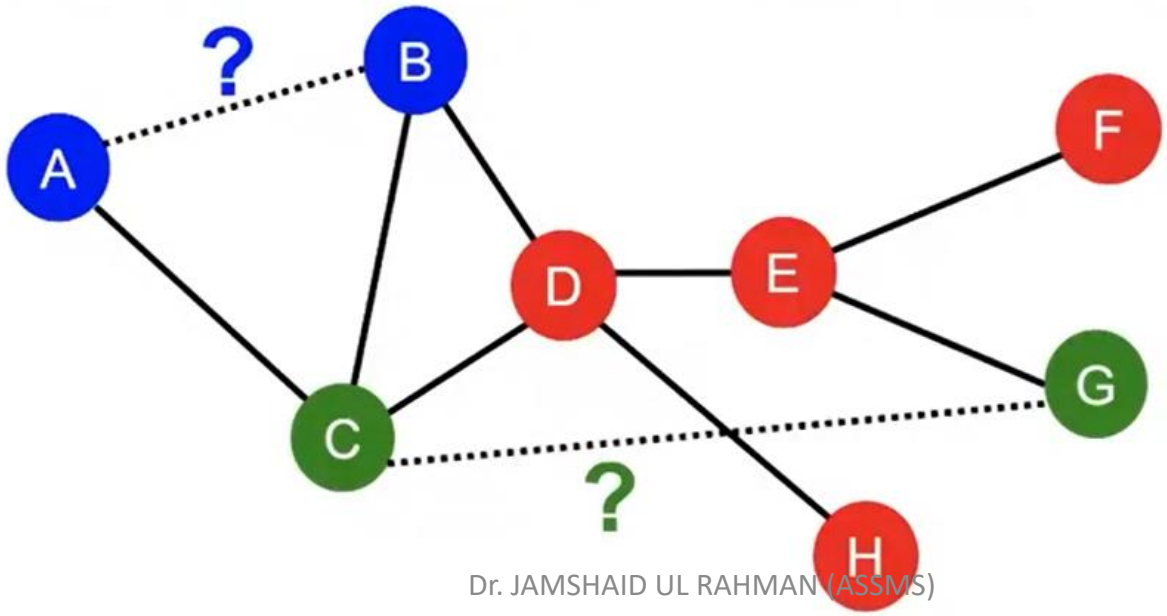
Abdus Salam School of Mathematical Sciences,

GC University, Lahore

Link Prediction Task and Features

Link-Level Tasks

- **Edge Property Prediction:** Predicting properties of edges, such as the strength of a relationship or the likelihood of an interaction.
- **Link Prediction:** Predicting whether a link (edge) should exist between two nodes, often used in recommendation systems.



Link Prediction as a Task

Two formulations of link prediction task:

1) Links missing at random:

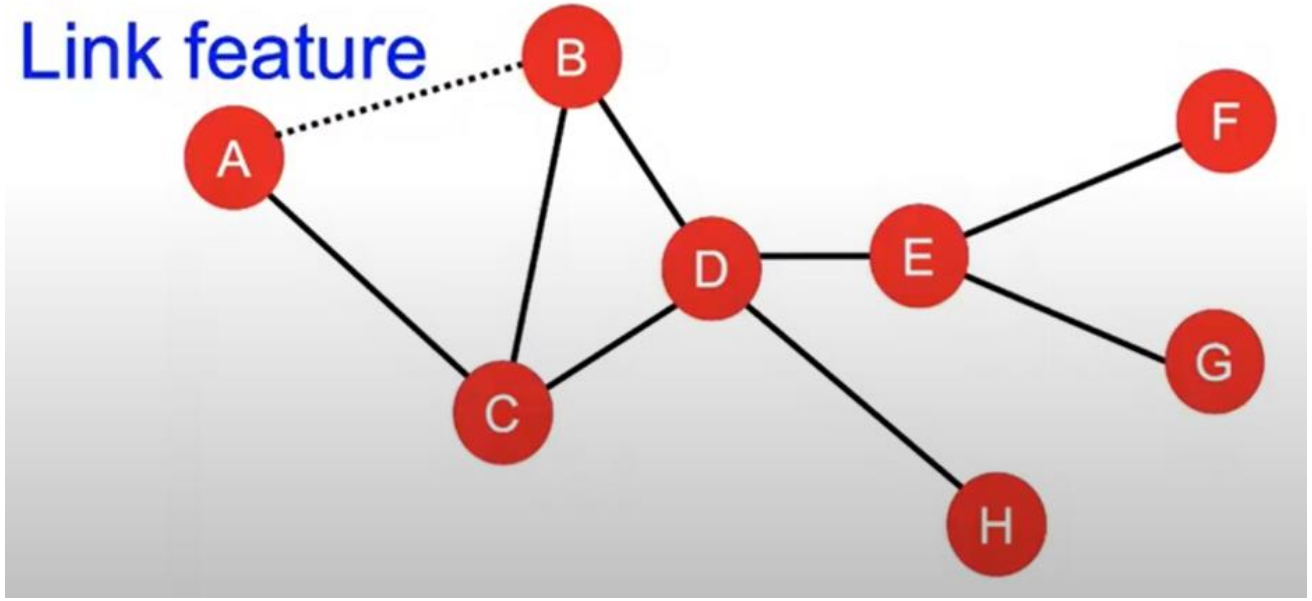
- Remove a random set of link and then aim to predict them

2) Links over time :

- This process involves observing how the connections (edges) between nodes change over different time periods.
- Given $G[t_0, t_0']$ a graph on edges up to time t_0', output a rank list L of link (not in $G[t_0, t_0']$) that are predicted to appear in $G[t_1, t_1']$

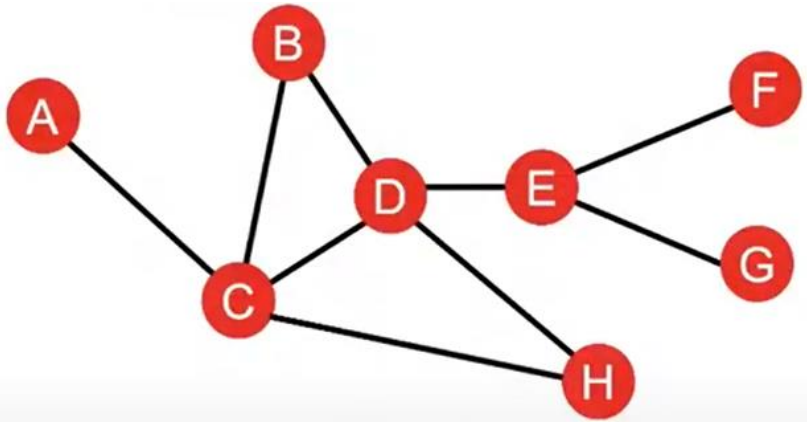
Links Level Features

- Distance based feature
- Local neighborhood overlap
- Global neighborhood overlap



Distance- Based Features

- **Shortest path distance between two nodes**
- Example:



$$S_{BH} = S_{BE} = S_{AB} = 2$$

$$S_{BG} = S_{BF} = 3$$

However, this does not capture the degree of neighborhood overlap:

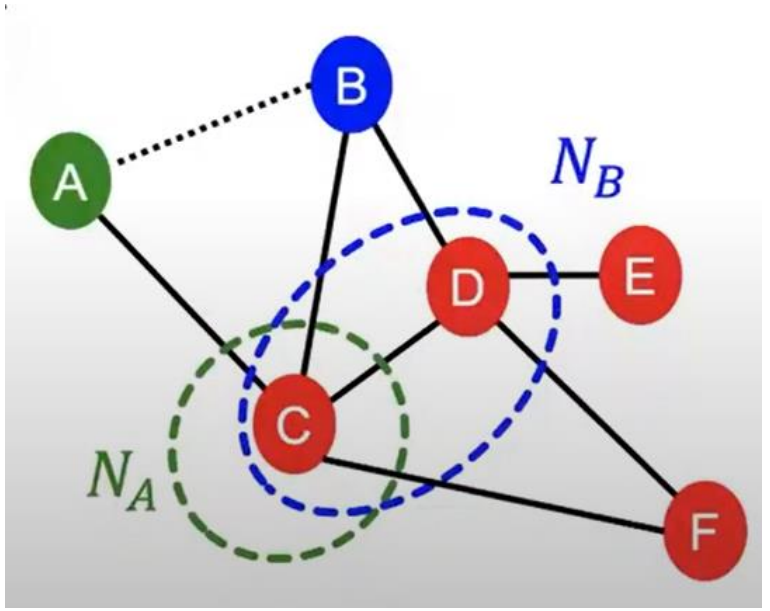
- Node pair (B, H) has 2 shared neighboring nodes, while pairs (B, E) and (A, B) only have 1 such node.
- The idea is that nodes that are closer to each other (in terms of graph distance) are more likely to form a link.

Local Neighborhood Overlap

- **Local neighborhood overlap** is a feature used in tasks like link prediction and community detection. It measures the extent to which the neighborhoods (sets of neighboring nodes) of two nodes overlap.
- Captures neighboring nodes shared between two nodes v_1 and v_2.

Local Neighborhood Overlap

- **Common neighbors** in graph theory refer to the shared neighbors between two nodes. If two nodes, say A and B, have a common neighbor, it means there exists another node C that is connected to both A and B.
- $|N(v_1) \cap N(v_2)|$
- Example :
- $|N(A) \cap N(B)| = |\{c\}| = 1$



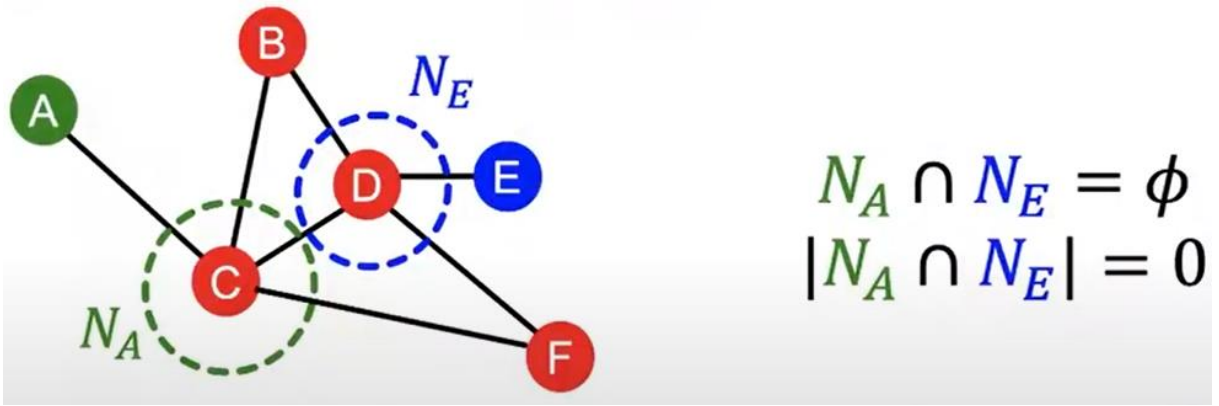
Local Neighborhood Overlap

- The **Jaccard Coefficient**, also known as the Jaccard Index or Jaccard Similarity Coefficient, is a measure used to compare the similarity and diversity of sample sets.
- Divided by the size of the union of two sets. For two sets A and B, it is calculated as: $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
- Example:
- $\frac{|N(v_A) \cap N(v_B)|}{|N(v_A) \cup N(v_B)|} = \frac{|\{C\}|}{|\{C,D\}|} = 1/2$

- The **Jaccard Coefficient**, also known as the Jaccard Index or Jaccard Similarity Coefficient, is a measure used to compare the similarity and diversity of sample sets.
- Divided by the size of the union of two sets. For two sets A and B, it is calculated as: $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
- Example:
- $\frac{|N(v_A) \cap N(v_B)|}{|N(v_A) \cup N(v_B)|} = \frac{|\{C\}|}{|\{C,D\}|} = 1/2$

Local Neighborhood Overlap

- Limitation of local neighborhood features:
- Metric is always zero if the two nodes do not have any neighbors in common.



- However, the two nodes may still potentially be connected.

Global Neighborhood Overlap

- **Gobal neighborhood overlap** metrics resolve the limitation by considering the entire graph.
- **Katz index:** count the number of paths of all lengths between a given pairs of nodes.
- **Q: how to compute paths between two nodes?**
- **Ans:** use powers of the graph adjacency matrix.

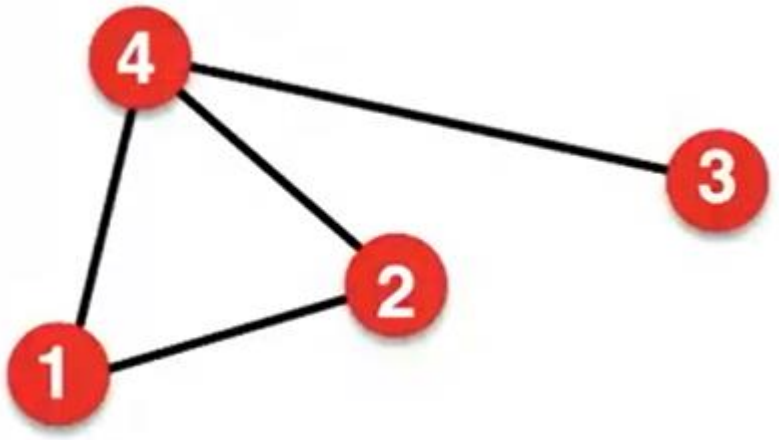
  i) A_{uv} specifies paths of length 1 between u and v.

  ii) A_{uv}^2 specifies path of length 2 between u and v.

  iii) A_{uv}^l specifies path of length l between u and v.

Power of Adjacency Matrices

- Let P_{uv}^{k} be path of length k between u and v.
- $P_{uv}^{k} = A^{k}$
- P_{uv}^{1} be the path of length 1 between u and v.

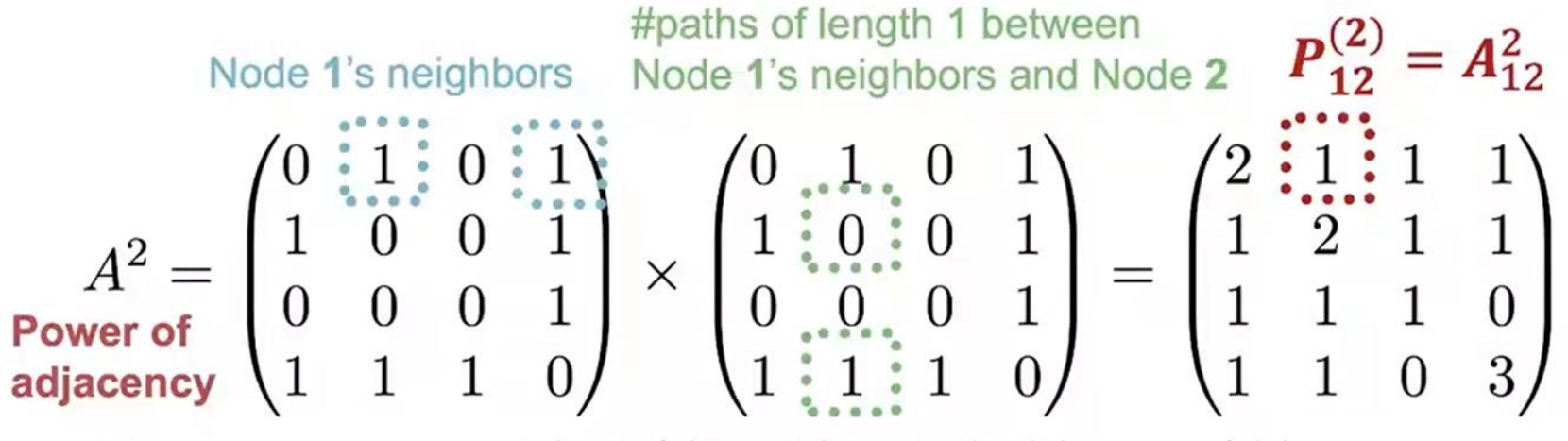


$$P_{12}^{(1)} = A_{12}$$

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Power of Adjacency Matrices

- **How to compute P^2_{uv}**
- **Step 1 :** compute paths of length 1 between each of u's neighbors and v
- **Step 2 :** $P^2_{uv} = A^2 = A * A$



Link level feature summary

- **Distance based feature:**
- uses the shortest path length between two nodes but does not capture how neighborhood overlaps.
- **Local neighborhood overlaps:**
- Capture how many neighboring nodes are shared by two nodes.
- Become zero when no neighbor nodes shared.
- **Global neighborhood overlap:**
- Uses global structure to score two nodes.
- Katz index counts path of all length between two nodes.

Graph-Level Predictions and Features

Graph-Level Predictions

- **Graph Classification:** Classifying entire graphs into different categories, such as identifying whether a molecule is toxic or not.
- **Graph Regression:** Predicting continuous values for entire graphs, such as estimating the solubility of a molecule.

Graph Level Features

- **Graph-level features** are characteristics that describe the entire graph, providing insights into its overall structure and properties.
- We want features that characterize the structure of an entire graph

Kernel Methods

A **Graph Kernel** is a mathematical function that computes the similarity between two graphs.

1. **Graphlet kernel**
2. **Weisfeiler-Lehman kernel**

- Both Graphlet Kernel and Weisfeiler-Lehman(WL) kernel use Bag of * representation graph , where * is more sophisticated than node degree.

Graphlet kernel

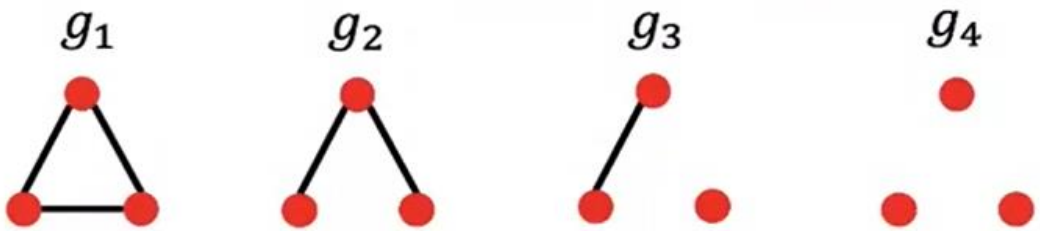
- Count the number of different graphlets in a graph.
- Defination of graphlets here is slightly different from node level features.
- **The two difference are:**

1. Nodes in graphlets here do not need to be connected
2. The graphlets here not rooted.

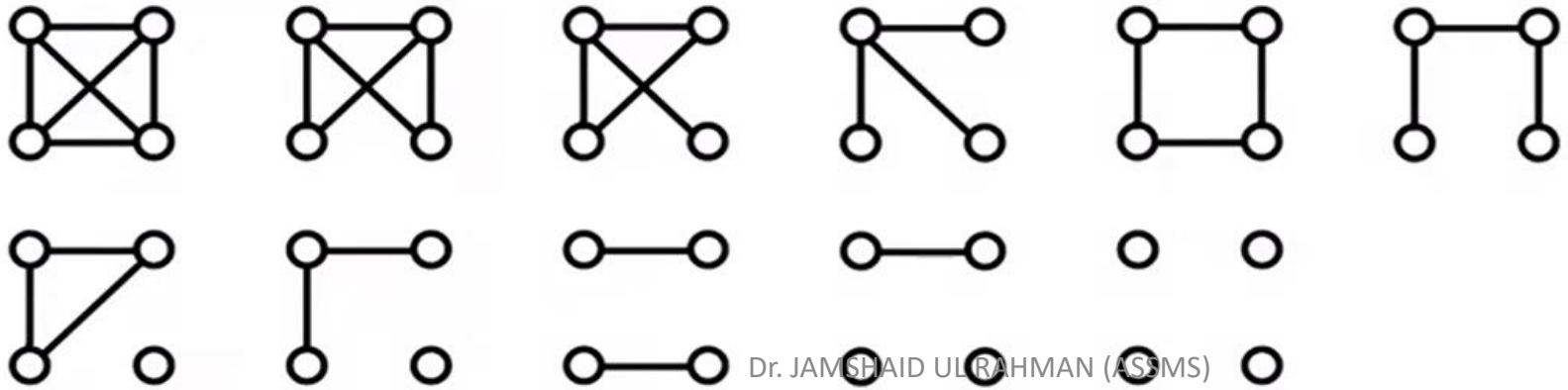
Graphlet Features

- Let $g_k = (g_1, g_2, \ldots, g_{nk})$ be a list graphlets of size k.

For $k = 3$, there are 4 graphlets.



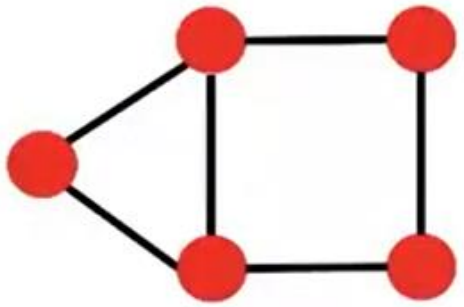
For $k = 4$, there are 11 graphlets.



Graphlet Features

Example for $k = 3$

G



g_1 g_2 g_3 g_4

$$f_G = (1, \quad 3, \quad 6, \quad 0)^{\mathrm{T}}$$

Graphlet Kernel

- Given two graphs, G and G', graphlet kernel is computed as

$$K(G, G') = f_G^T f_{G'}$$

- **Problem:** If G and G' have different sizes?
- Solution: Normalize each feature vector
- $h_G = \frac{f_G}{sum(f_G)}$

$$K(G, G') = h_G^T h_{G'}$$

Graphlet Kernel

“Different sizes" mean?

- When comparing two graphs G and G’, the **size** of a graph often refers to its number of nodes or edges.
- Larger graphs naturally tend to have **more features** or higher counts of certain structures (like subgraphs or "graphlets") compared to smaller graphs.
- This size difference causes an **imbalance** in the feature vectors f_G and f'_G.
- **For example:**
- A large graph might have $f_G = [100,50,20]$ (high counts of features).
- A small graph might have $f'_G = [10,5,2]$ (low counts of features).
- The **problem**: If we directly compute the kernel $K(G, G') = f_G^T f'_G$, the large graph's features will dominate due to their larger values, making the comparison unfair.

Graphlet Kernel

- Before normalization: $f_G = [100, 50, 20]$ → Sum = 170
- After normalization:

$$h_G = \left[\frac{100}{170}, \frac{50}{170}, \frac{20}{170}\right] = [0.588, 0.294, 0.118]$$

- Similarly, for $f_{G'} = [10, 5, 2]$:

$$h_{G'} = \left[\frac{10}{17}, \frac{5}{17}, \frac{2}{17}\right] = [0.588, 0.294, 0.118]$$

Graphlet Kernel

High Kernel Value:

- $K(G, G')$ is large if the two graphs have similar patterns of local substructures (graphlets).
- This means the graphs are likely to be structurally similar.

Low Kernel Value:

- $K(G, G')$ is small (or close to zero) if the two graphs have very different substructures.
- This indicates the graphs are dissimilar in their local patterns.

Graphlet Kernel

Limitations:

- Counting graphlets is expensive
- Counting size-K graphlets for a graph with size n by enumeration takes n^k.
- **Can we design a more efficient graph kernel?**

Weisfeiler-Lehman Kernel

- The Weisfeiler-Lehman Kernel uses the neighborhood structure of a graph to iteratively enrich the "vocabulary" of nodes.
- This method is a generalized version of the Bag of Node Degrees approach since node degrees represent one-hop neighborhood information.
- Algorithm to achieve this:

Color refinement

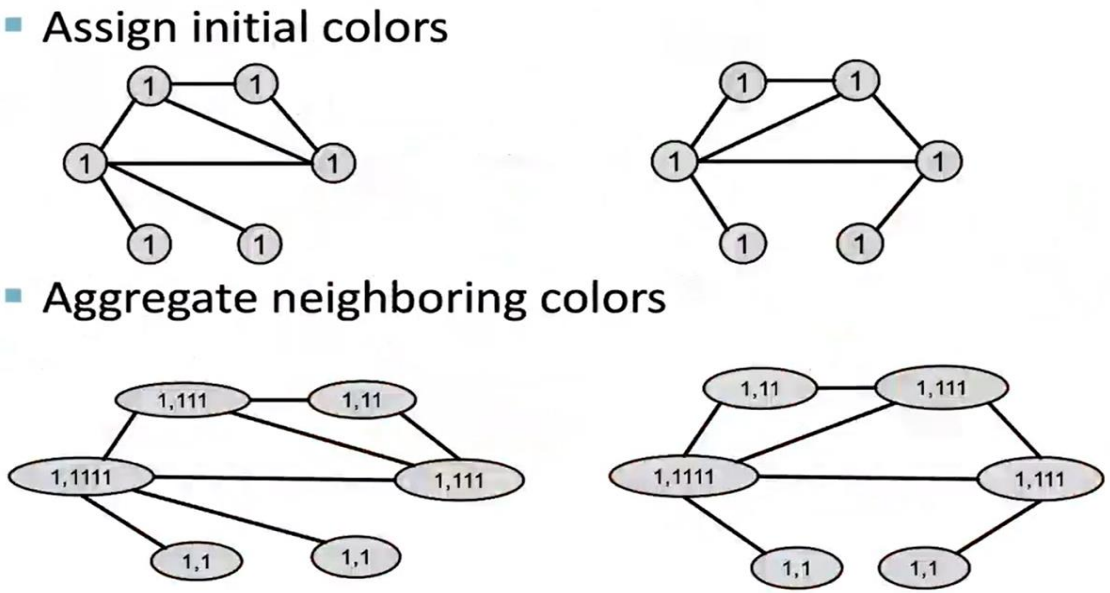
Weisfelier-Lehman kernel

Color refinement:

- Each node v is assigned an initial color $c^0(v)$. This can be based on node properties, such as node degree.
- The color of each node is updated iteratively.
- The new color $c^{k+1}(v)$ of a node v at iteration $k+1$ is determined by hashing the current color of the node $c^k(v)$ and the multiset of colors of its neighbors $\{c^k(u)\} \in N(v)$.
- This process continues for K iterations, where K is a predefined number of steps.

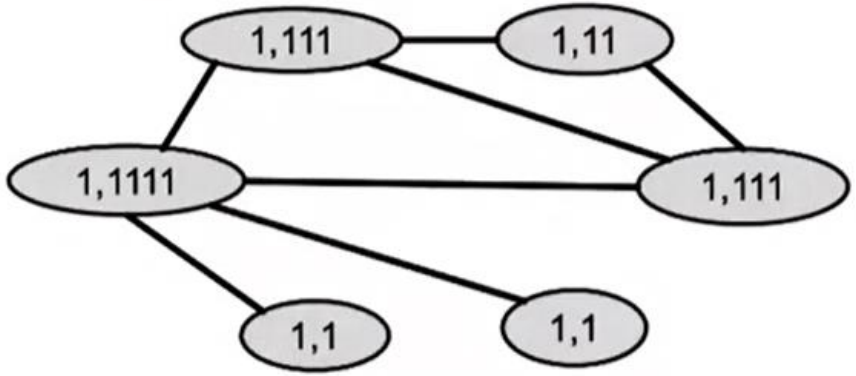
Weisfelier-Lehman kernel

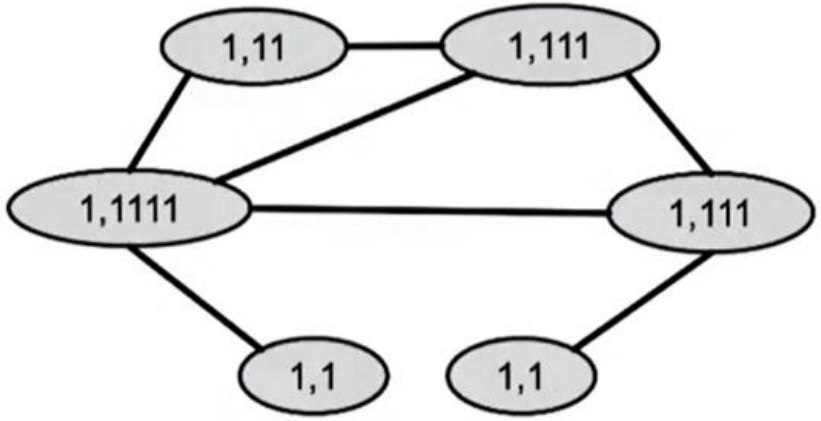
- **Example:**



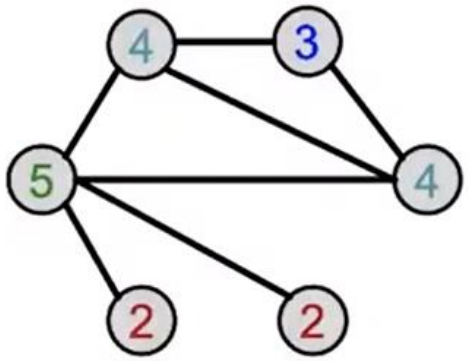
Weisfelier-Lehman kernel

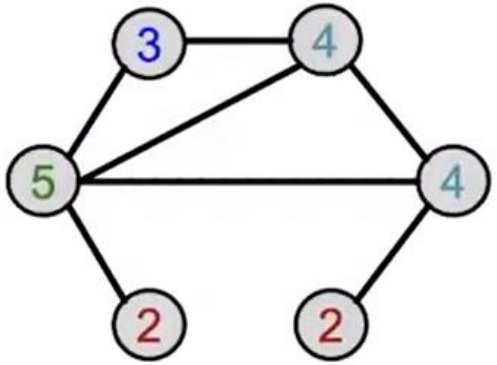
- Aggregated colors





- Hash aggregated colors

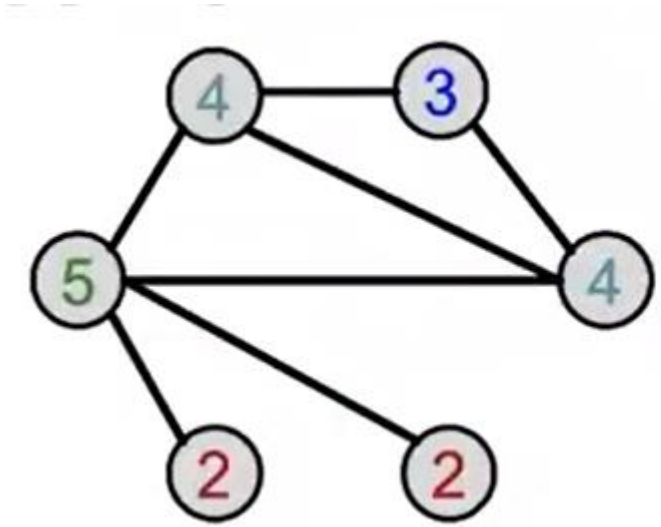


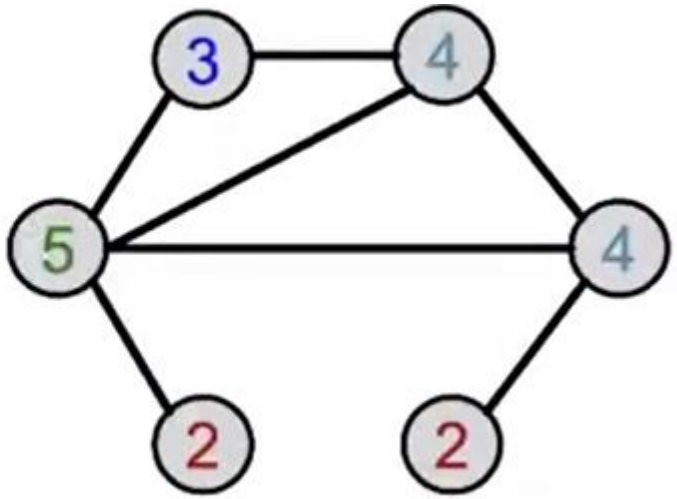


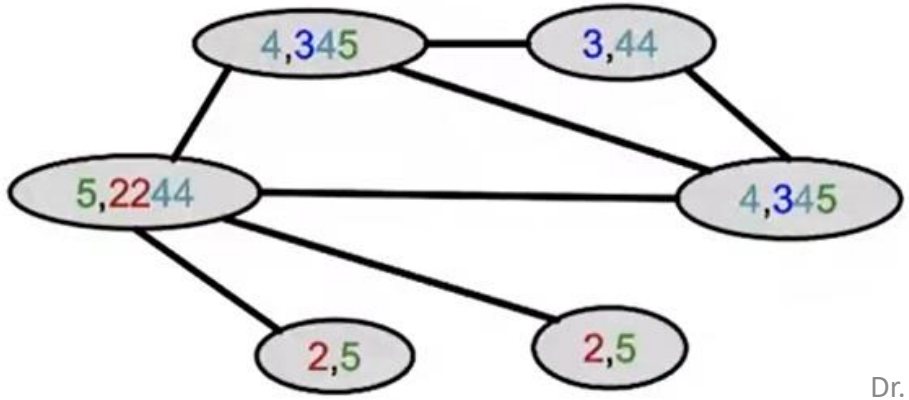
Hash table

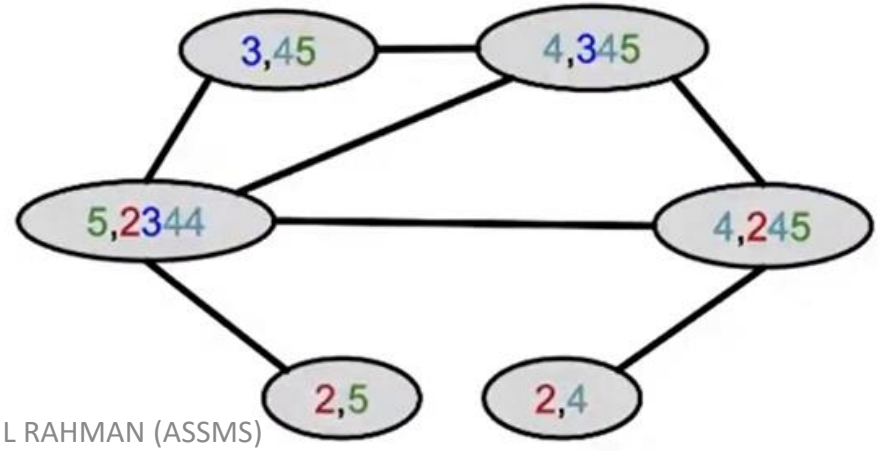
1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

Weisfelier-Lehman kernel

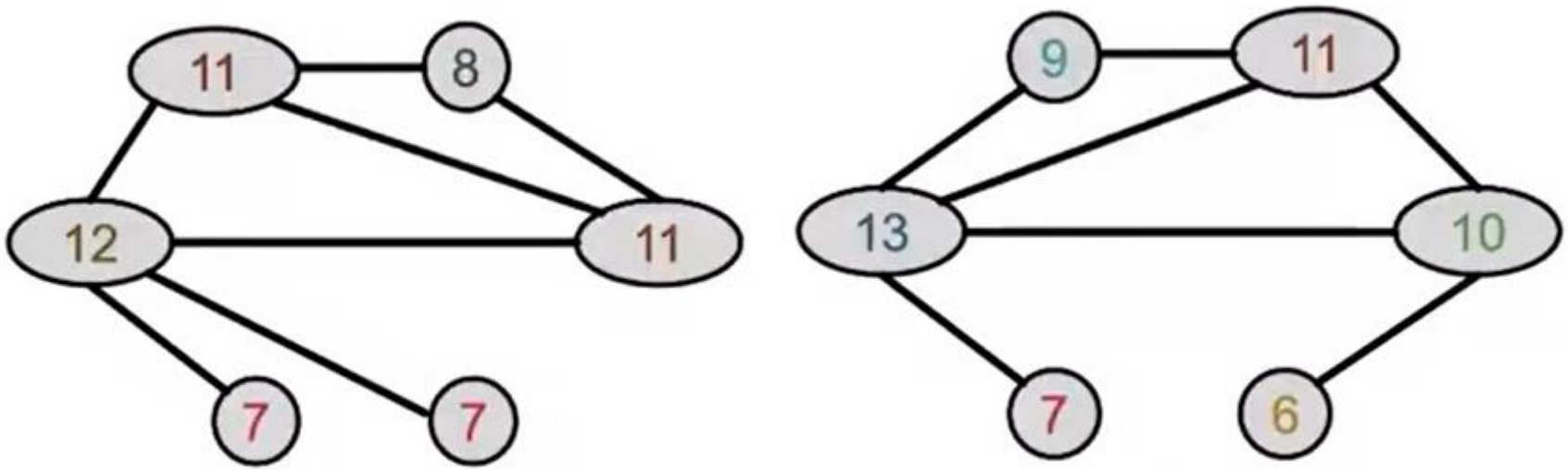






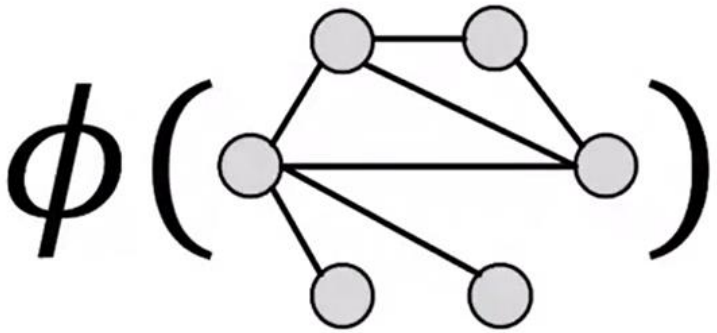


Weisfelier-Lehman kernel



Weisfelier-Lehman kernel

- After color refinement, WL kernel counts number of nodes with a given color.

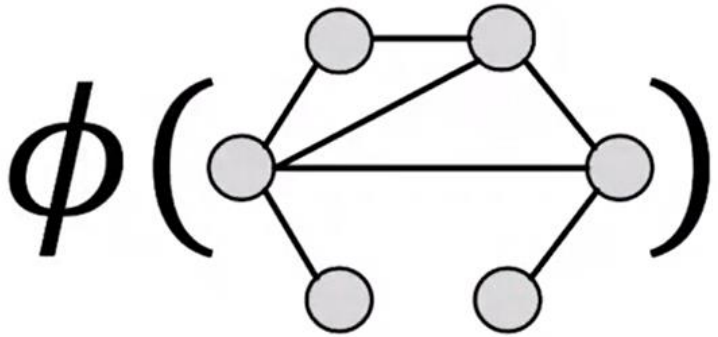


Colors

1,2,3,4,5,6,7,8,9,10,11,12,13

= [6,2,1,2,1,0,2,1,0, 0, 0, 2, 1]

Counts

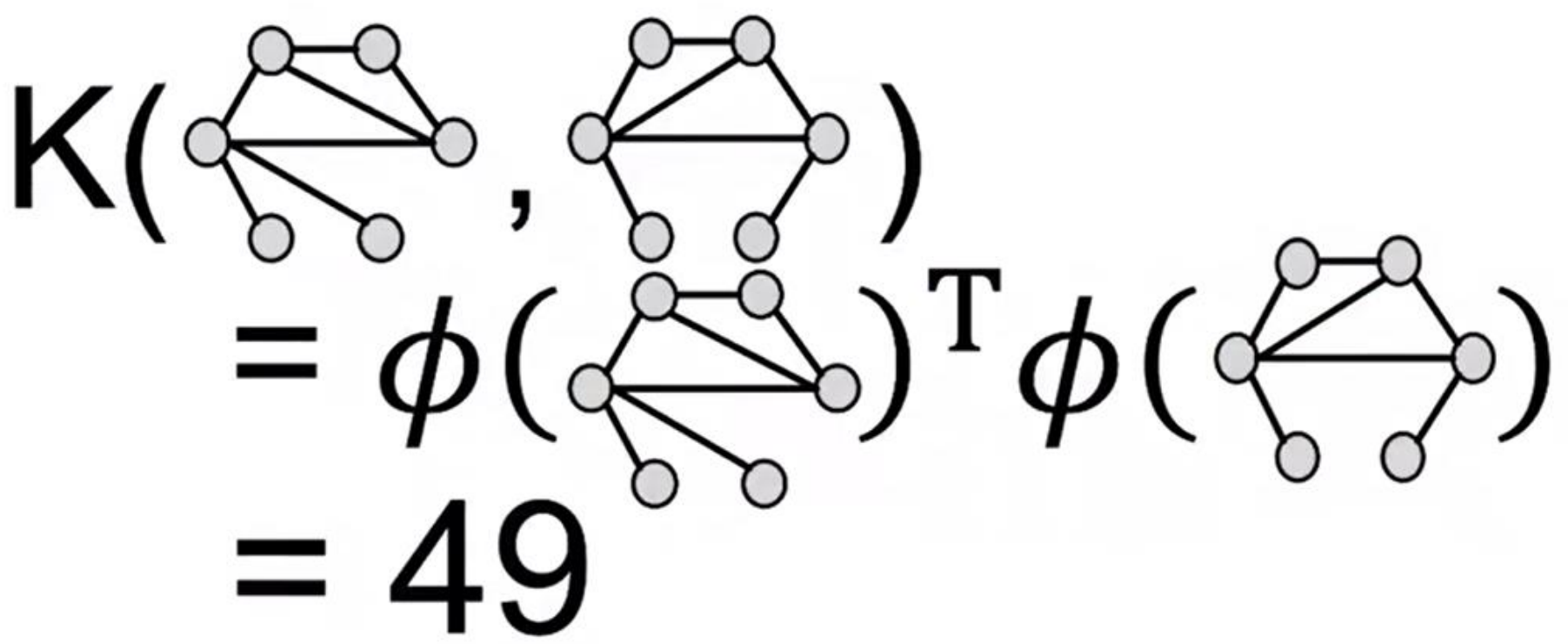


1,2,3,4,5,6,7,8,9,10,11,12,13

= [6,2,1,2,1,1,1,0,1, 1, 1, 0, 1]

Weisfelier-Lehman kernel

- The WL kernel value is computed by the inner product of the color vectors:



Weisfelier-Lehman Kernel

- WL kernel is computationally efficient (the time complexity for color refinement at each step is linear in edges, since it move aggregating neighboring colors)
- When computing a kernel value, only colors appeared in the two graphs need to be tackled.
- Counting colors takes linear time w.r.t nodes.

Graph Level Features : Summary

Graphlet kernel:

- Graph is represented as Bag-of-graphlets
- Computationally expensive

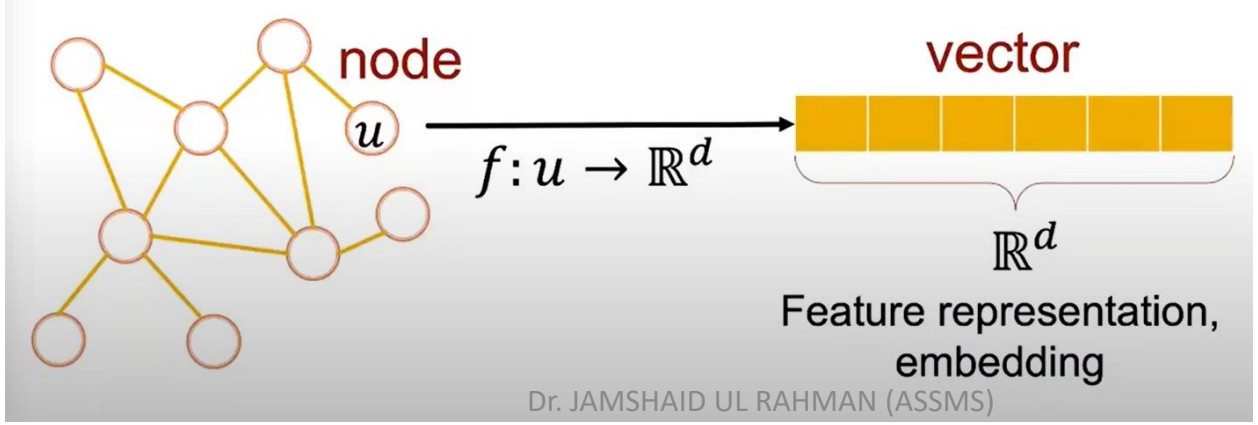
Weisfeiler-Lehman Kernel:

- Apply k step color refinement algorithm to enrich node colors.
- Graph is represented as Bag-of-colors
- Computationally efficient

Graph Embedding

Graph Embedding

- **Graph embedding** is a technique used to transform graphs into a lower-dimensional vector space while preserving the graph's structure and properties.
- This process enables the application of traditional machine learning algorithms to graph data by converting the complex relationships and structures within the graph into numerical representations.
- This transformation allows for the operations such as similarity measurement, clustering, and prediction.



Graph Embedding

- **A node in a graph can be viewed from two domains:**

1. The original graph domain, where nodes are connected via edges (or the graph structure)
2. The embedding domain, where each node is represented as a continuous vector.

- **Two key questions naturally arise:**

  1) What information to preserve?

  2) How to preserve this information?

- Different graph embedding algorithms often provide different answers to these two questions.

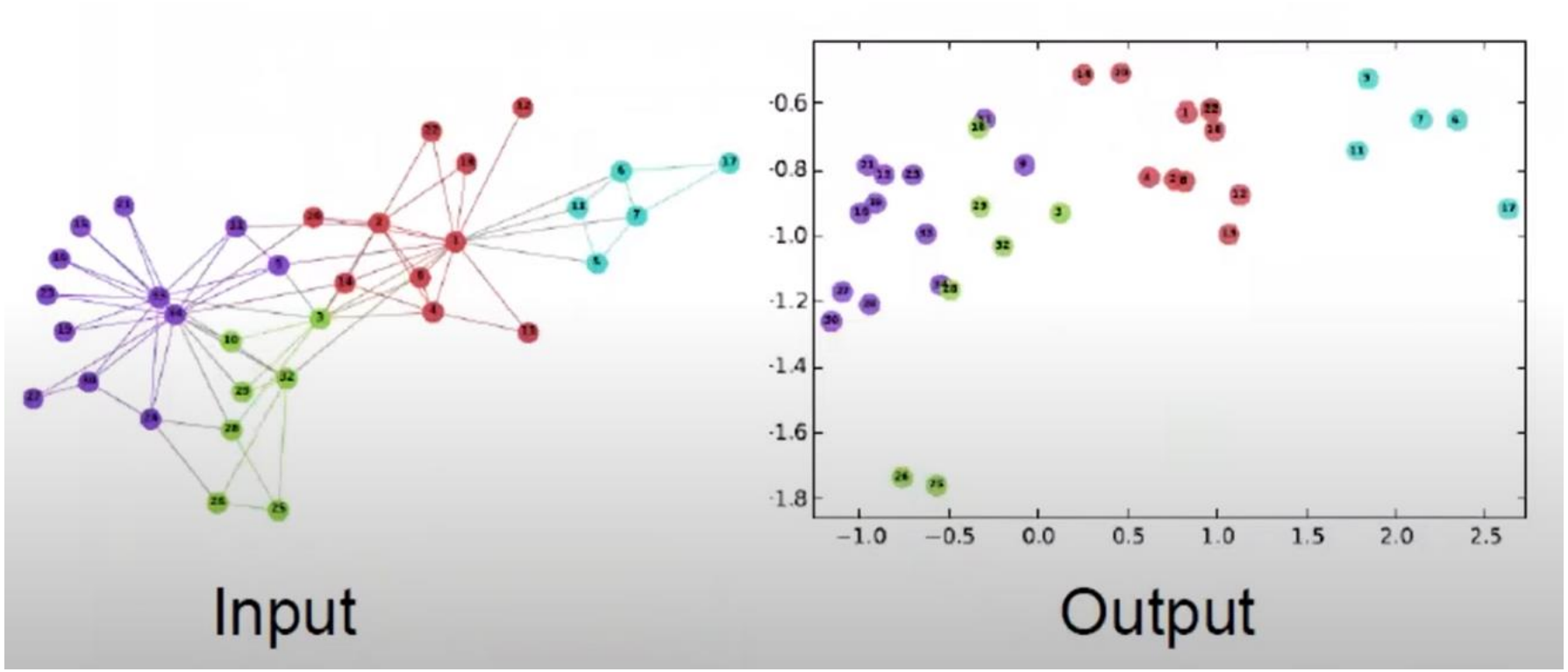
Why Embedding :

- Similarity of embedding between nodes indicates their similarity in the network. (both nodes are close to each other connected by an edge)
- Encode network information
- Use for many predictions:

1. Node classification
2. Link prediction
3. Graph classification

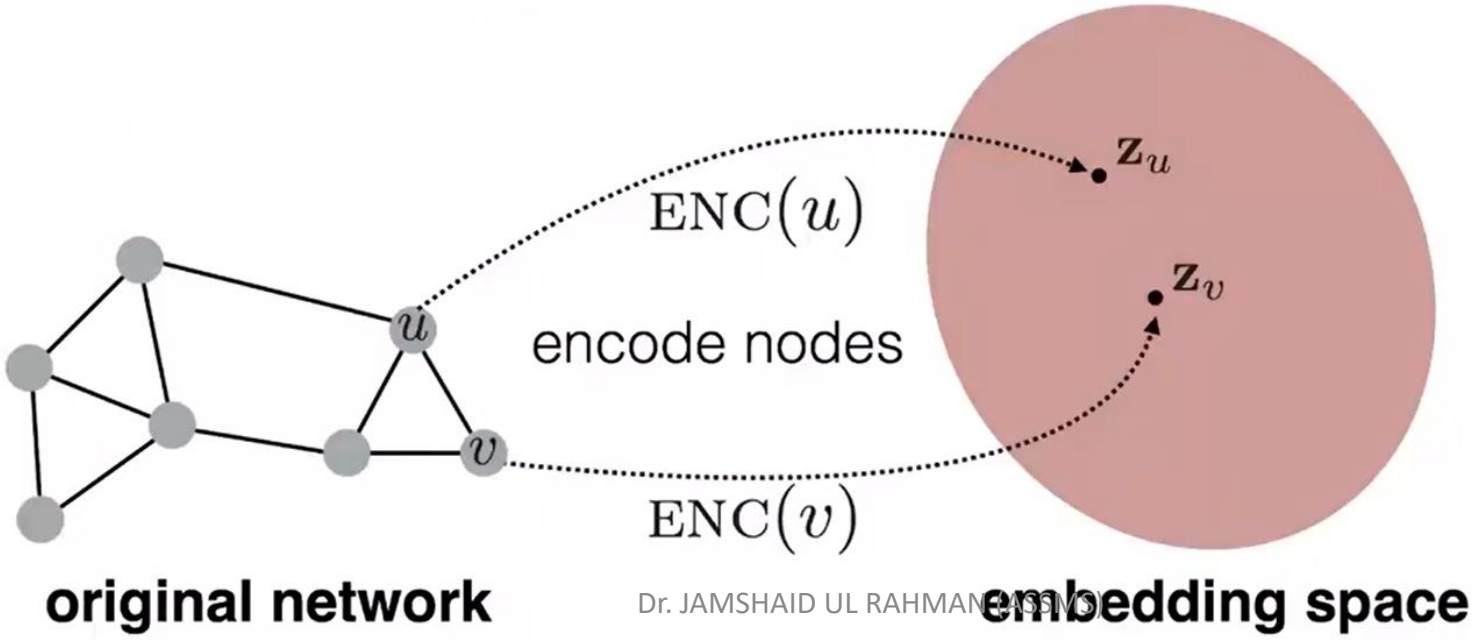
Example:

- 2D embedding of nodes



Node Embedding:

- Transforms each node in the graph into a low-dimensional vector that captures the node's structural information and relationships.
- To encode nodes so that similarity in the embedding space approximates similarity in the graph



Node Embedding:

Encoder:

- Maps nodes to embedding.
- This is the function that converts each node into a vector representation. Various techniques can be used for this encoding, such as neural networks.

Decoder (DEC):

- Maps from embedding to the similarity score.
- The decoder takes the vector representations of nodes (embedding) and computes a similarity score. This step ensures that the learned embedding reflect the actual similarities between nodes.

Learning Node Embedding

1. Encoder maps from nodes to embedding
2. Define a node similarity function (i.e a measure of similarity in the original network)
3. Decoder map from embedding to the similarity score
4. Optimize the parameter of the encoder so that:
   $similarity(u, v)(original\ network) \approx Z_v^T Z_u (embedding)$

Shallow Encoding

- Simplest encoding approach: Encoder is just an embedding-lookup

$$ENC(v) = Z.v$$

- $Z \in R^{d*|V|}$ (matrix each column is a node embedding)
- $v \in I^{|V|}$ (indicator vector, all zeroes except a one in column indicating node v)
- Each node is assigned a unique embedding vector

Example:

- Assume we have three nodes A, B, and C, each represented by a 2-dimensional embedding vector. Where the embedding matrix

$$Z = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix}$$

- The indicator vector for node $A = [1 \;\; 0 \;\; 0], B = [0 \;\; 1 \;\; 0], C = [0 \;\; 0 \;\; 1]$

- Embedding for node $A = Z.v_A = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$

- Embedding for node $\mathrm{B} = Z.v_B = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}$

- Embedding for node $C = Z.v_C = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix}$

How to Define Node Similarity?

Key choice of method is how they define node similarity.

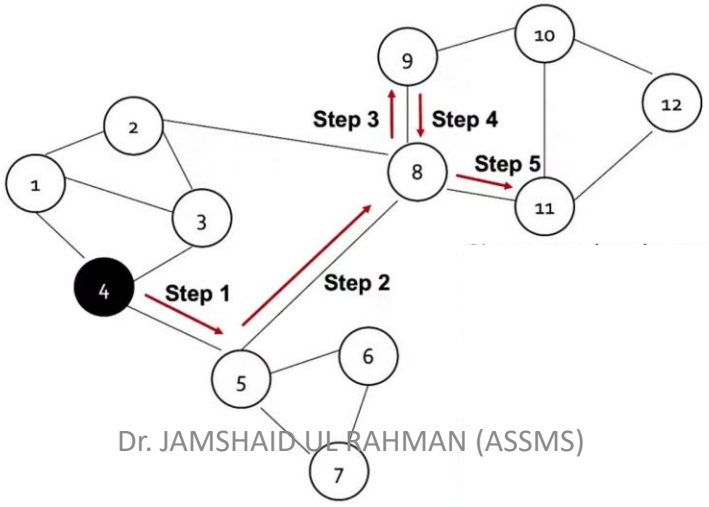
- Should two nodes have similar embedding if they

1. are linked?
2. shared neighbors?
3. have similar structure roles?

- We will now learn node similarity definition that uses random walks, and how to optimize embedding for such a similarity measure.

Random walk :

- A **random walk** on a graph is a sequence of nodes where each node is chosen randomly from the neighbors of the current node.
- Given a graph and a starting point, we select a neighbor of it at random, and move to this neighbor; than we select a neighbor of this point at random, and move to it, ect.
- The random sequence of points visited this way is a random walk on the graph.



Random Walk Embeddings

- Random walk embeddings are a technique used to capture the structure of a graph by performing random walks on it and then using the sequences generated to learn vector representations (embeddings) of the nodes.
- Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R
- Optimize embedding to encode these random walk statistic
- **Similarity in embedding space (dot product) encodes random walk similarity**

Why Random Walk

- If random walk starting from node u visits v with high probability, u and v are similar.
- Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks.

Random Walk Optimization

- Run Short Fixed-Length Random Walks: Start from each node u in the graph and perform multiple short, fixed-length random walks using a specific random walk strategy R.
- For each node u, collect $N_R(u)$, the multiset of nodes visited during the random walks starting from u. The multiset $N_R(u)$ includes the nodes visited and accounts for their frequency.
- Optimize embedding according to: given node u, predict its neighbors $N_R(U)$

Random Walk Optimization

After we obtained the objective function, how do we optimize it ?

1. Gradient descent method
2. Stochastic gradient descent

Random Walk Optimization

Gradient Descent:

- Initialize: z_i at some randomized value for all nodes i in the graph.
- For all nodes i, compute the derivative of the objective function L with respect to each embedding $z_i . \frac{dL}{dz_i}$
- For all nodes i, update the embedding z_i. Such that $\mathrm{z_i} \Leftarrow z_i - \eta \frac{dL}{dz_i}$ where η is the leaning rate.
- Iterate until the objective function converges

Random Walk Optimization

Stochastic Gradient Descent (SGD)

- Instead of calculating the gradient over the entire dataset, SGD evaluates the gradient for each individual training example or a small batch of examples.
- Start by initializing the embeddings z_i for each node i with some random values.
- Randomly sample a node i from the graph and Calculate $\frac{dL}{dz_i}$
- For all j Update : $z_i \Leftarrow z_i - \eta \frac{dL}{dz_i}$ where η is the leaning rate.
- Repeat the following steps until the embeddings converge, meaning that further updates result in negligible changes in the embeddings.

Node2vec

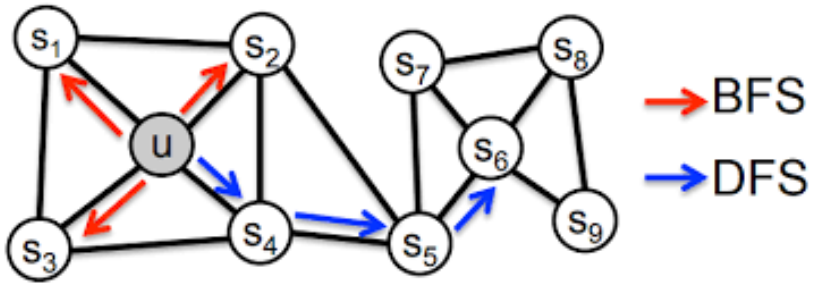
- Node2Vec was introduced in 2016 by Grover and Leskovec.
- The main objective is to embed nodes (i.e., represent them in a feature space) such that nodes with similar network neighborhoods are close to each other in this space.
- The difference is that instead of obtaining sequences of nodes with a uniform distribution, the random walks are carefully biased in Node2Vec.
- Biased random walks perform better and how to implement them in the two following sections:

  i) Defining a neighborhood

  ii) Introducing biases in random walks

Biased Random Walk

- Use flexible, biased random walks that can trade off between local and global views of the network.
- **A biased random walk** introduces preferences in how the walker chooses the next node, influenced by certain parameters or properties.
- **Breadth-First Search (BFS)**: Illustrated with red arrows, BFS focuses on exploring the local neighborhood of a node, capturing local connectivity.
- **Depth-First Search (DFS)**: Illustrated with blue arrows, DFS explores deeper into the network, capturing more global structures.



Walk of length 3 ($N_R(u)$ of size 3):

$$N_{BFS}(u) = \{s_1, s_2, s_3\}$$
$$N_{DFS}(u) = \{s_4, s_5, s_6\}$$

Biased Random Walk

Biased fixed length walk R that give a node u generate neighborhood $N_R(u)$

Two parameter :

- Return parameter p

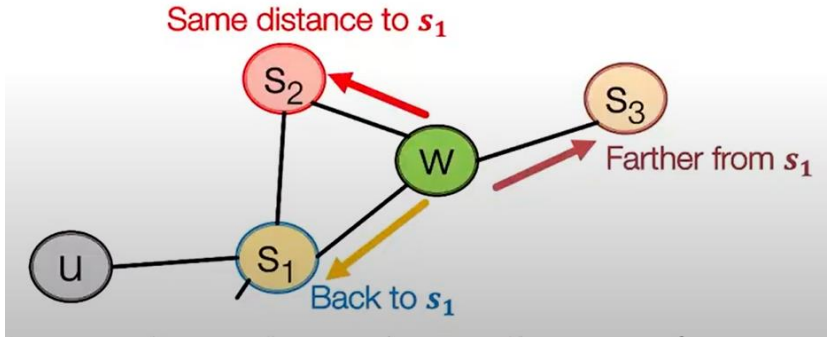
  Return back to the previous node

- In out parameter q

  Moving outwards

Biased Random Walk

Example : Neighbors of w can only be:

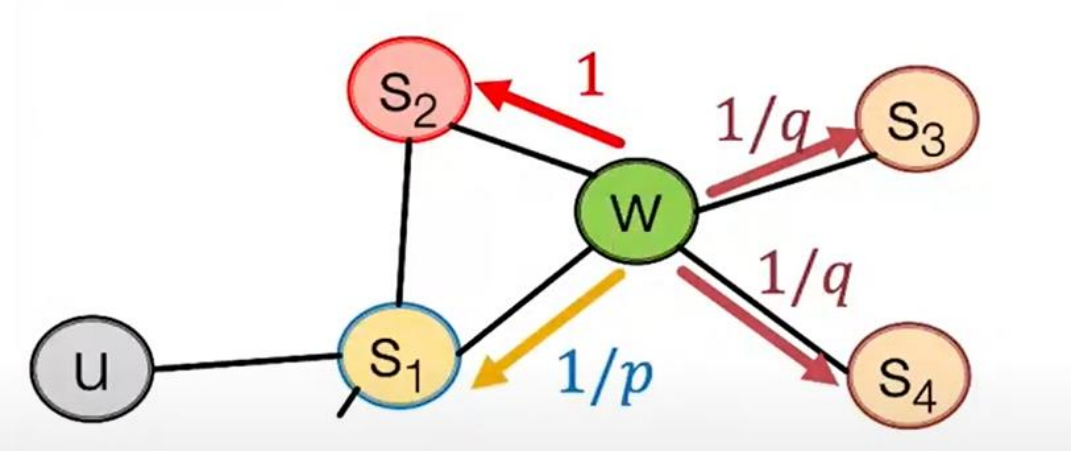


Where to go next?

- In Node2Vec, the value of $\alpha(a, b)$ is defined based on the distance between the nodes and two additional parameters: p, the return parameter, and q, the in-out parameter.
- Here is how to define value of $\alpha(a, b)$ is define as:

$$\alpha(a,b) = \begin{cases} \frac{1}{p} \; if \; d_{ab} = 0 \\ 1 \; if \; d_{ab} = 1 \\ \frac{1}{q} \; if \; d_{ab} = 2 \end{cases}$$

Biased Random Walk



$$\alpha(a,b) = \begin{cases} \frac{1}{p} \; if \; d_{ab} = 0 \\ 1 \; if \; d_{ab} = 1 \\ \frac{1}{q} \; if \; d_{ab} = 2 \end{cases}$$

- For low value of p : BFS like walk
- For low value of q : DFS like walk
- $N_R(u)$ are nodes visited by the biased walk

Node2vec Algorithm

1. Compute random walk probabilities
2. Simulate r random walk of length l starting from each node u
3. Optimize the node2vec objective using stochastic gradient descent

Ranking Nodes on the Graph

- All web pages are not equally important
- There is large diversity in the web-graph node connectivity
- So, let’s rank the pages using the web graph link structure

- Page is more important if it has more links
- In-coming links

PageRank

- PageRank is an algorithm originally developed by Larry Page and Sergey Brin to rank web pages in search engine results.
- PageRank measures the importance of each node (such as web pages) within a graph, based on the structure of the incoming links.

PageRank

- Let A be the state transition probability matrix. Each element A_{ij} indicates the probability of moving from node i to node j.

$$A = \begin{pmatrix} A_{11} & A_{12} & . & . & . & A_{1n} \\ A_{21} & A_{22} & . & . & . & A_{2n} \\ . & . & & & & . \\ . & & & & & \\ . & . & & & & . \\ . & . & & & & . \\ A_{n1} & A_{n2} & . & . & . & A_{nn} \end{pmatrix}$$

$$A_{ij} = \begin{cases} \frac{1}{O_i} & if(i,j) = E \\ 0 & otherwise \end{cases}$$

- We have $\sum_{i=1}^{n} P_0(i) = 1$
- A_{ij} represent the transition probability that sufer in state i (page i) will move to state j(page j)

PageRank : How to Solve?

Given a graph with n nodes, we use an iterative procedure:

- Assign each node an initial page rank
- Repeat until convergence $(\sum_i |P_{k+1}^i - P_k^i| < \varepsilon)$
- Calculate the page rank of each node

$$P_{k+1}^i = \sum_{(j,i)\in E} \frac{P_k^i}{d_i}$$

d_i =out-degree of node i

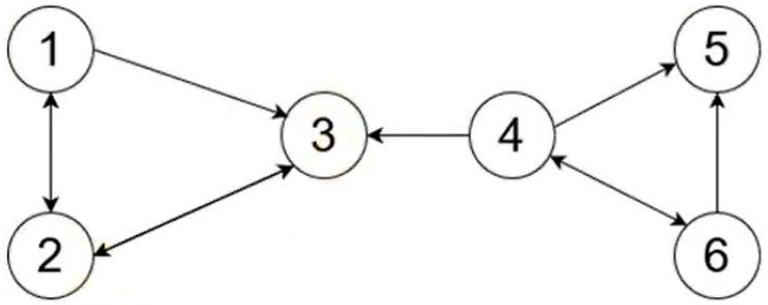
This can be solve by power iteration method

Power Iteration Method

- Power iteration : a simple iterative scheme
- Initialize : P_0
- Iterate: $P_{i+1} = A^T P_i$
- Stop when $|P_{i+1} - P_i| < \varepsilon$

PageRank: EXAMPLE

- Power iteration method :



$$A_{ij} = \begin{cases} \frac{1}{O_i} & if(i,j) = E \\ 0 & otherwise \end{cases}$$

$$A = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}$$

PageRank : Problems

Two problems

- Some pages are dead ends(have no out links). Such pages cause importance to leak out
- Spider traps(all out-links are within the group). Eventually spider traps absorb all importance.

PageRank

- Fix the problem : Two possible ways
- Remove those pages with no out-link during the pagerank computation as these pages do not affect the ranking of any other page directly.
- Add a complete set of outgoing links from each such page i to all pages on the web.
- Let us use the 2nd way

$$\overline{A} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}$$

PageRank

- By takiny the transpose

$$A^T = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/6 & 0 \\ 1/2 & 0 & 1 & 0 & 1/6 & 0 \\ 1/2 & 1/2 & 0 & 1/3 & 1/6 & 0 \\ 0 & 0 & 0 & 0 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 0 \end{pmatrix}$$

- And let $P_o = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix}$

PageRank

- $P_1 = A^T P_0$

$$P_1 = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/6 & 0 \\ 1/2 & 0 & 1 & 0 & 1/6 & 0 \\ 1/2 & 1/2 & 0 & 1/3 & 1/6 & 0 \\ 0 & 0 & 0 & 0 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 1.833 \\ 4.333 \\ 3.666 \\ 3.833 \\ 5.166 \\ 2.166 \end{pmatrix}$$

- $P_2 = A^T P_1$

$$P_2 = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/6 & 0 \\ 1/2 & 0 & 1 & 0 & 1/6 & 0 \\ 1/2 & 1/2 & 0 & 1/3 & 1/6 & 0 \\ 0 & 0 & 0 & 0 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 0 \end{pmatrix} \begin{pmatrix} 1.833 \\ 4.333 \\ 3.666 \\ 3.833 \\ 5.166 \\ 2.166 \end{pmatrix} = \begin{pmatrix} 3.021 \\ 5.433 \\ 5.212 \\ 1.937 \\ 3.213 \\ 2.132 \end{pmatrix}$$

PageRank

- $P_3 = A^T P_2$

$$P_3 = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/6 & 0 \\ 1/2 & 0 & 1 & 0 & 1/6 & 0 \\ 1/2 & 1/2 & 0 & 1/3 & 1/6 & 0 \\ 0 & 0 & 0 & 0 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 0 \end{pmatrix} \begin{pmatrix} 3.021 \\ 5.433 \\ 5.212 \\ 1.937 \\ 3.213 \\ 2.132 \end{pmatrix} = \begin{pmatrix} 3.250 \\ 7.256 \\ 5.406 \\ 1.599 \\ 2.244 \\ 1.178 \end{pmatrix}$$

- …
- $P_9 = A^T P_8$

$$P_9 = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/6 & 0 \\ 1/2 & 0 & 1 & 0 & 1/6 & 0 \\ 1/2 & 1/2 & 0 & 1/3 & 1/6 & 0 \\ 0 & 0 & 0 & 0 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 1/2 \\ 0 & 0 & 0 & 1/3 & 1/6 & 0 \end{pmatrix} \begin{pmatrix} 4.518 \\ 8.898 \\ 6.784 \\ 0.210 \\ 0.315 \\ 0.182 \end{pmatrix} = \begin{pmatrix} 4.501 \\ 9.096 \\ 6.830 \\ 0.143 \\ 0.213 \\ 0.122 \end{pmatrix}$$

- So Rank: 2,3,1,5,4,6

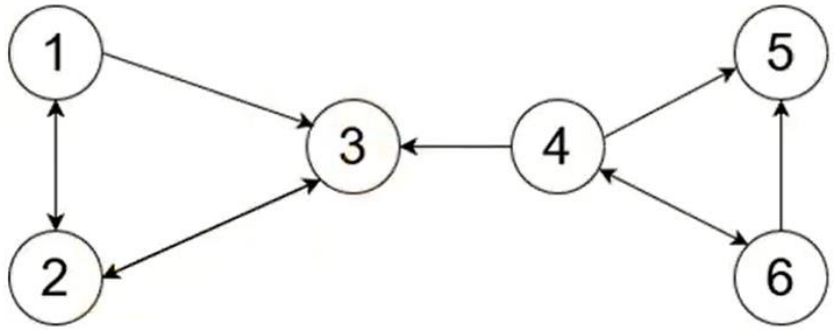
PageRank

- **Experiment 1**
- $P_0 = [1, 2, 3, 4, 5, 6]$
- $P_9 = [4.501,\ 9.096,\ 6.830, 0.143,\ 0.213,\ 0.122]$
- Ranking: [2,3,1,5,4,6]
- **Experiment 2**
- $P_0 = [4, , 3, 6, 1, 5, 2]$
- $P_9 = [4.559,\ 9.247,\ 6.914, 0.067,\ 0.100,\ 0.057]$
- Ranking: [2,3,1,5,4,6]
- **Experiment 3**
- $P_0 = [100, 100, 100, 100, 100, 100]$
- $P_9 = [130.772, 261.197, 196.792,\ 2.810, 4.188, 2.405]$
- Ranking: [2,3,1,5,4,6]

Improved PageRank

- After this augmentation, at a page, the random surfer has two options
- With probability d, he randomly choose an out-link to follow
- d is called the damping factor , $d \in [0,1]$
- $P(i) = (1-d) + d \sum_{(j,i)\epsilon E} \frac{P(j)}{O_j}$

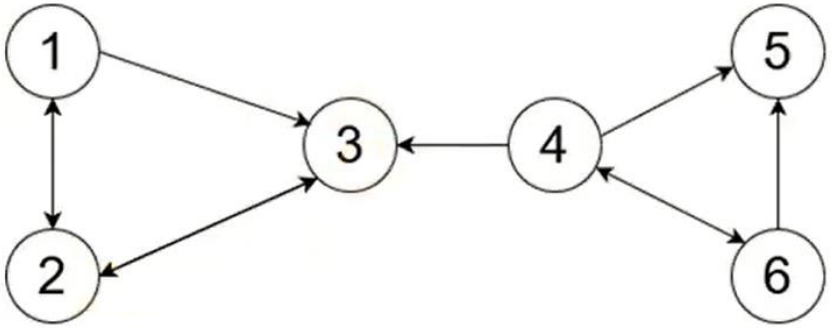
Improved PageRank



- $P(i) = (1-d) + d\sum_{(j,i)\epsilon E}\frac{P(j)}{O_j}$
- Let $d = 0.8$

- $P(1) = 1,\ \ P(2) = 2,\ \ P(3) = 3,\ P(4) = 4,\ P(5) = 5,\ P(6) = 6$
- Iteration 1
- $P(1) = (1-0.8) + 0.8\left(\frac{2}{2}\right) = 1$
- $P(2) = (1-0.8) + 0.8\left(\frac{1}{2} + \frac{3}{1}\right) = 3$
- $P(3) = (1-0.8) + 0.8\left(\frac{1}{2} + \frac{3}{2} + \frac{4}{3}\right) = 2.86$
- $P(4) = (1-0.8) + 0.8\left(\frac{6}{2}\right) = 2.6$
- $P(5) = (1-0.8) + 0.8\left(\frac{2.6}{2} + \frac{6}{2}\right) = 3.29$
- $P(6) = (1-0.8) + 0.8\left(\frac{2.6}{3}\right) = 0.89$

Improved PageRank



- Iteration 2
- $P(1) = (1 - 0.8) + 0.8\left(\frac{3}{2}\right) = 1.4$
- $P(2) = (1 - 0.8) + 0.8\left(\frac{1.4}{2} + \frac{2.86}{1}\right) = 3.05$
- $P(3) = (1 - 0.8) + 0.8\left(\frac{1.4}{2} + \frac{3.05}{2} + \frac{2.6}{3}\right) = 2.64$
- $P(4) = (1 - 0.8) + 0.8\left(\frac{0.89}{2}\right) = 0.55$
- $P(5) = (1 - 0.8) + 0.8\left(\frac{0.55}{3} + \frac{0.89}{2}\right) = 0.71$
- $P(6) = (1 - 0.8) + 0.8\left(\frac{0.55}{3}\right) = 0.35$
- Ranking: [2, 3, 1, 5, 4, 6]

Next Improvement

$$P = (1-d)\frac{1}{n} + d(A^T P)$$

$$P = (1-d)\frac{1}{n} + d\left(\frac{PR(i)}{O(i)} + \frac{PR(j)}{O(j)} + \frac{PR(k)}{O(k)} + \cdots + \frac{PR(n)}{O(n)}\right)$$

#Iteration 1

P(1) = (1 – 0.8)/6 + 0.8(2/2) = 0.83

P(2) = (1 – 0.8)/6 + 0.8 (0.83/2 + 3/1) = 2.76

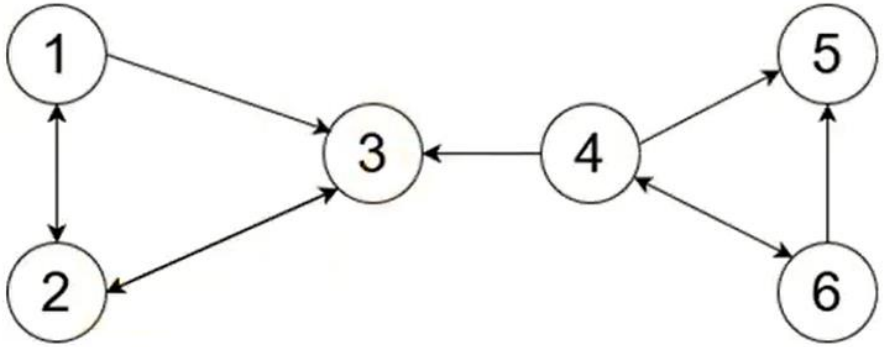
P(3) = (1 – 0.8)/6 + 0.8 (0.83/2 + 2.76/2 + 4/3) = 2.53

P(4) = (1 – 0.8)/6 + 0.8 (6/2) = 2.43

P(5) = (1 – 0.8)/6 + 0.8 (2.43/3 + 6/2) = 3.08

P(6) = (1 – 0.8)/6 + 0.8 (2.43/3) = 0.68

Ranking: [5, 2, 3, 4, 1, 6]



Thank You!