

Graph Machine Learning Tools

- We use [PyG \(PyTorch Geometric\)](#):



- The ultimate library for Graph Neural Networks
- **We further recommend:**
 - [GraphGym](#): Platform for designing Graph Neural Networks.
 - Modularized GNN implementation, simple hyperparameter tuning, flexible user customization
 - Both platforms are very helpful for the course project (save your time & provide advanced GNN functionalities)
- **Other network analytics tools:** SNAP.PY, NetworkX

Stanford CS224W: Machine Learning with Graphs

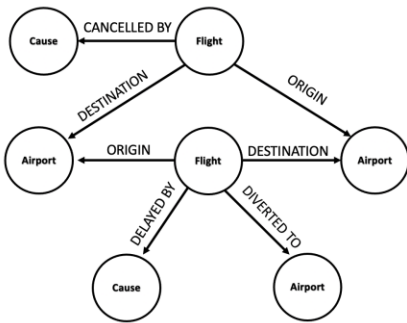
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
<http://cs224w.stanford.edu>



Why Graphs?

Graphs are a general language for describing and analyzing entities with relations/interactions

Many Types of Data are Graphs (1)

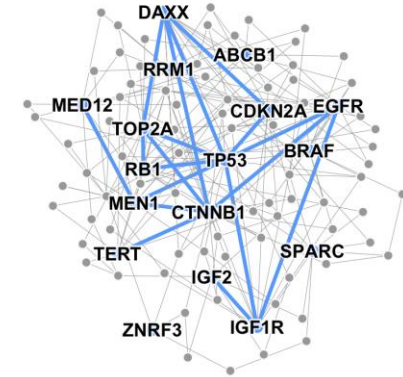


Event Graphs



Image credit: [SalientNetworks](#)

Computer Networks



Disease Pathways

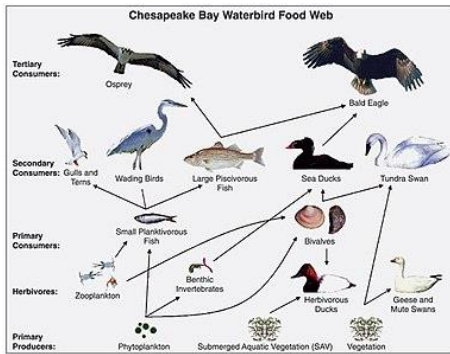


Image credit: [Wikipedia](#)

Food Webs

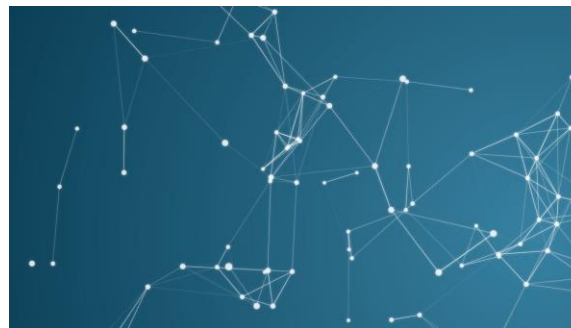


Image credit: [Pinterest](#)

Particle Networks



Image credit: [visitlondon.com](#)

Underground Networks

Many Types of Data are Graphs (2)



Image credit: [Medium](#)

Social Networks

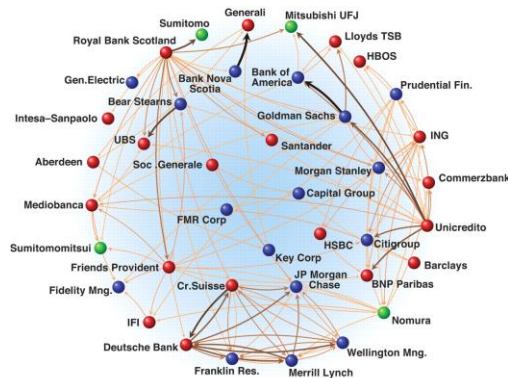


Image credit: [Science](#)

Economic Networks



Image credit: [Lumen Learning](#)

Communication Networks



Citation Networks



Image credit: [Missoula Current News](#)

Internet

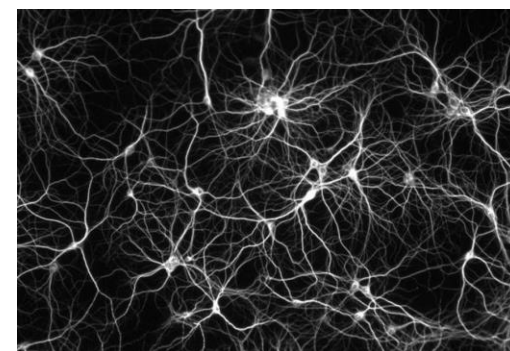


Image credit: [The Conversation](#)

Networks of Neurons

Many Types of Data are Graphs (3)

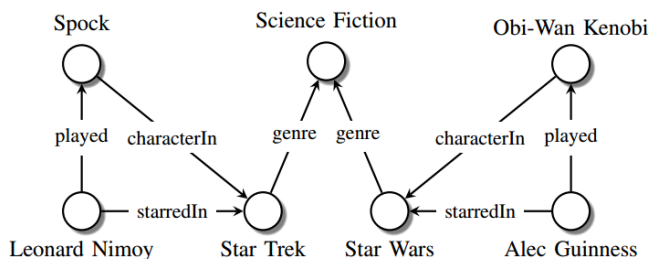


Image credit: [Maximilian Nickel et al](#)

Knowledge Graphs

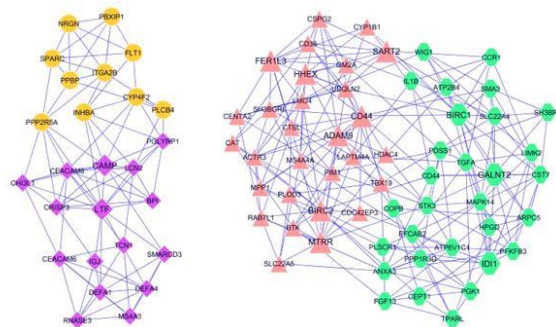


Image credit: [ese.wustl.edu](#)

Regulatory Networks

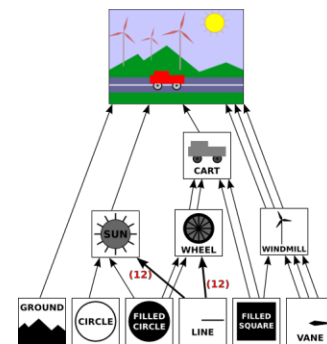


Image credit: [math.hws.edu](#)

Scene Graphs

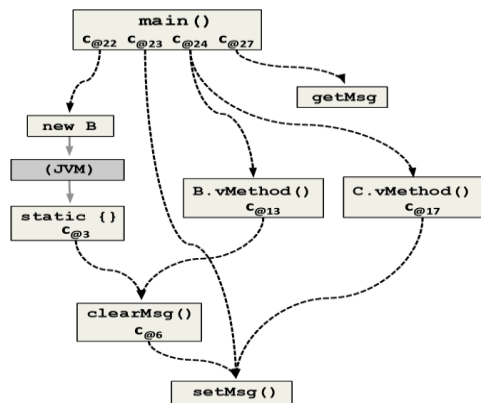


Image credit: [ResearchGate](#)

Code Graphs

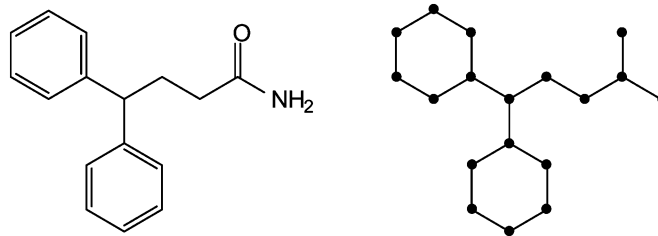


Image credit: [MDPI](#)

Molecules

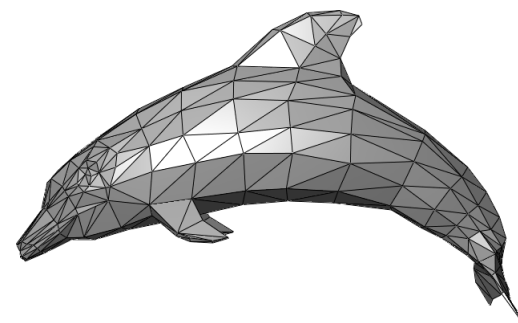
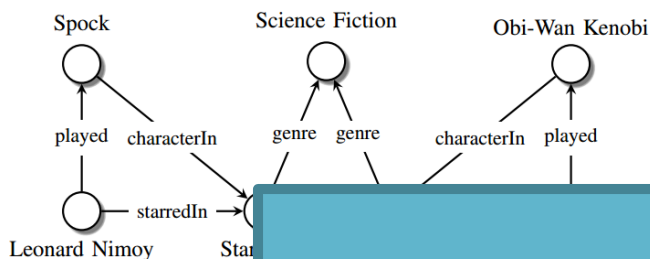
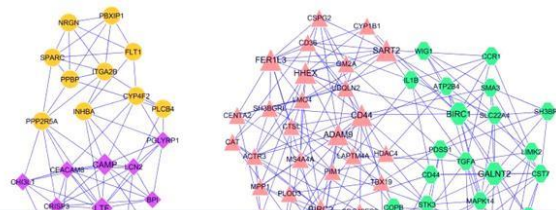
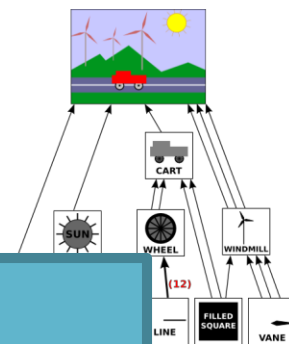


Image credit: [Wikipedia](#)

3D Shapes

Graphs and Relational Data



Main question:

How do we take advantage of relational structure for better prediction?

Know

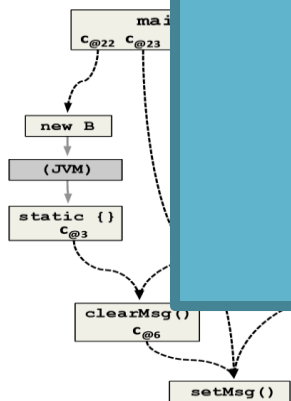


Image credit: [ResearchGate](#)

Code Graphs

Image credit: [MDPI](#)

Molecules

[math.hws.edu](#)

Graphs

Image credit: [Wikipedia](#)

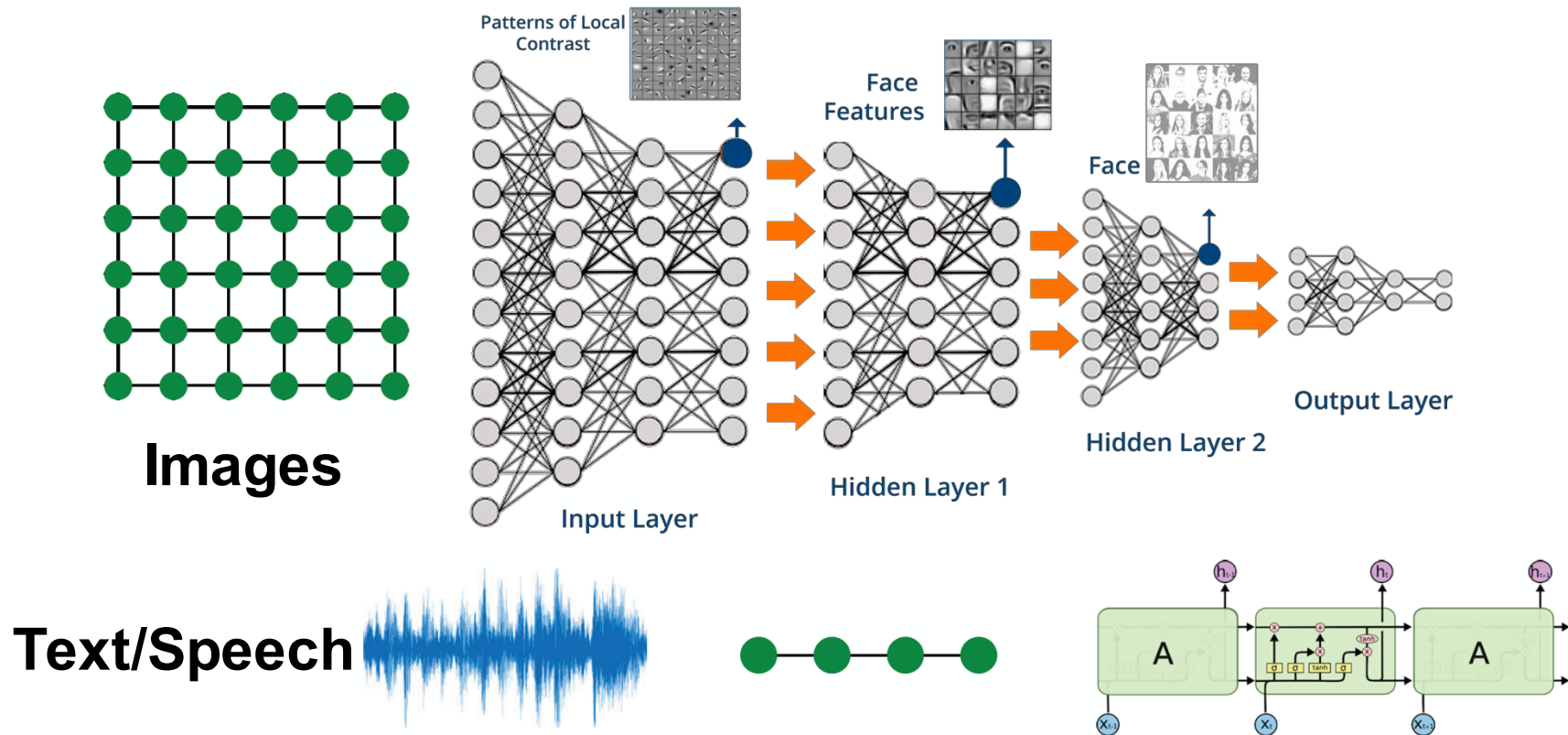
3D Shapes

Graphs: Machine Learning

Complex domains have a rich relational structure, which can be represented as a **relational graph**

By explicitly modeling relationships we achieve better performance!

Today: Modern ML Toolbox

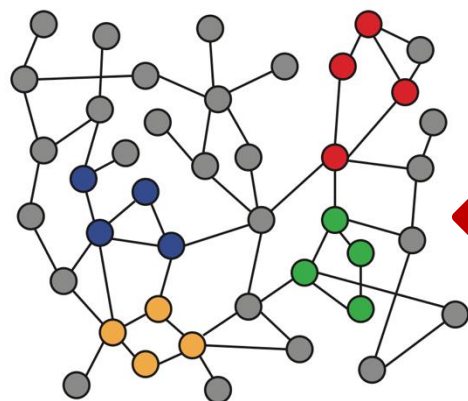


Modern deep learning toolbox is designed for simple sequences & grids

Why is Graph Deep Learning Hard?

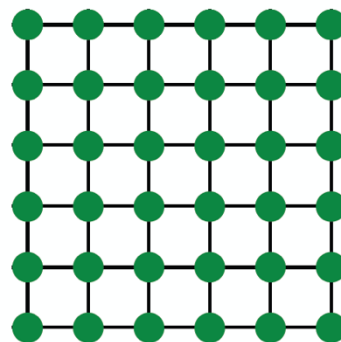
Networks are complex.

- Arbitrary size and complex topological structure (*i.e.*, no spatial locality like grids)



Networks

VS.



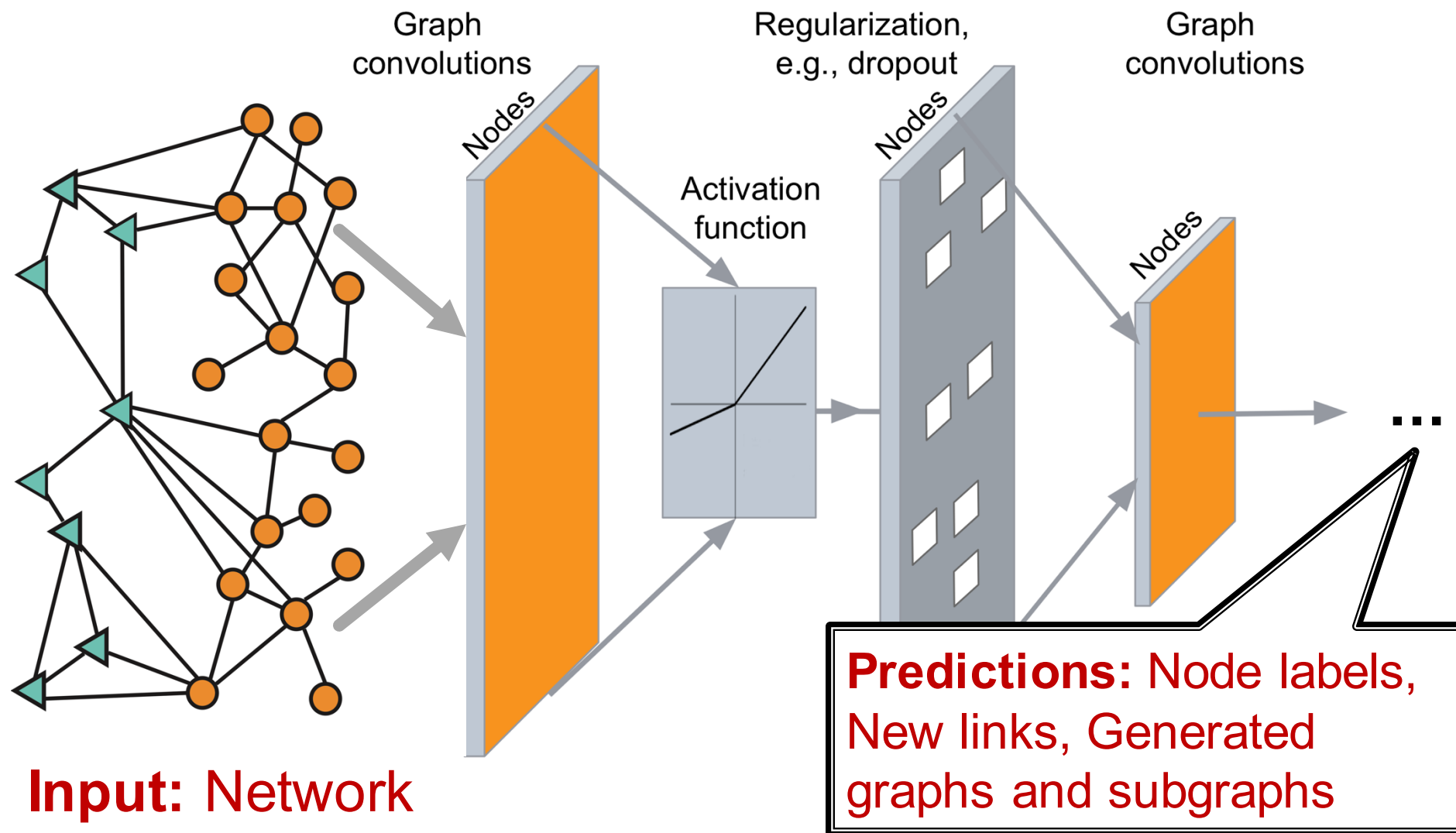
Images



Text

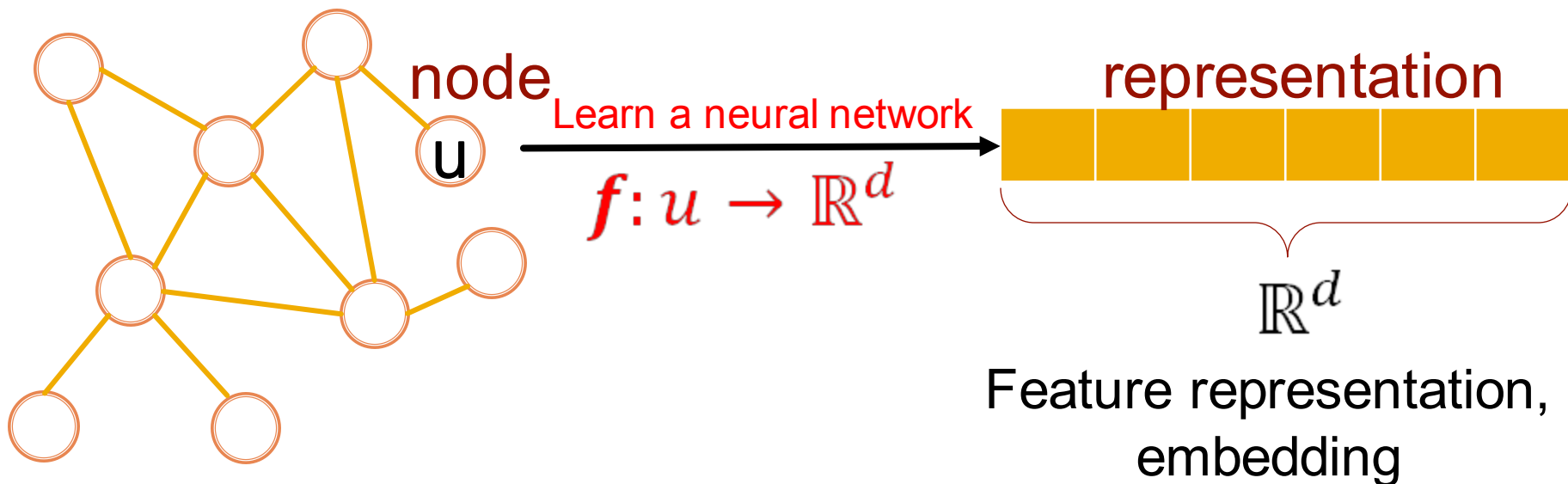
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

CS224W: Deep Learning in Graphs



CS224W & Representation Learning

Map nodes to d-dimensional **embeddings** such that **similar nodes in the network** are **embedded close together**

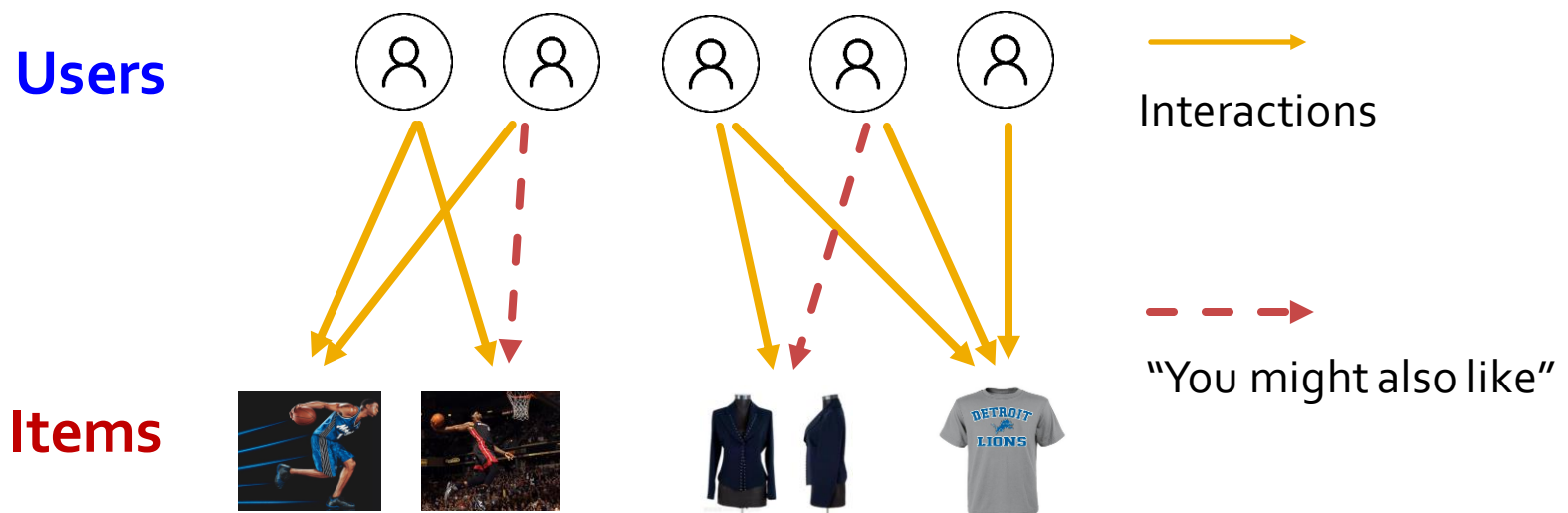


Classic Graph ML Tasks

- **Node classification**: Predict a property of a node
 - **Example**: Categorize online users / items
- **Link prediction**: Predict whether there are missing links between two nodes
 - **Example**: Knowledge graph completion
- **Graph classification**: Categorize different graphs
 - **Example**: Molecule property prediction
- **Clustering**: Detect if nodes form a community
 - **Example**: Social circle detection
- **Other tasks**:
 - **Graph generation**: Drug discovery
 - **Graph evolution**: Physical simulation

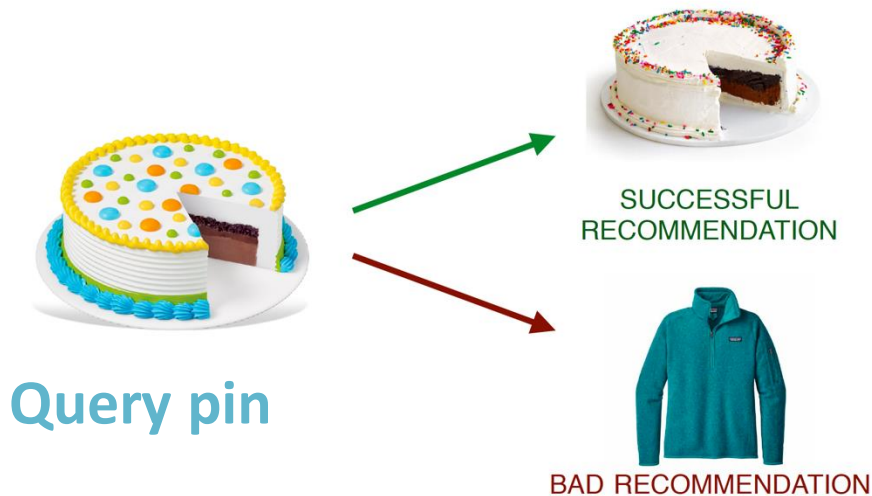
Example (2): Recommender Systems

- **Users interacts with items**
 - Watch movies, buy merchandise, listen to music
 - **Nodes:** Users and items
 - **Edges:** User-item interactions
- **Goal: Recommend items users might like**



PinSage: Graph-based Recommender

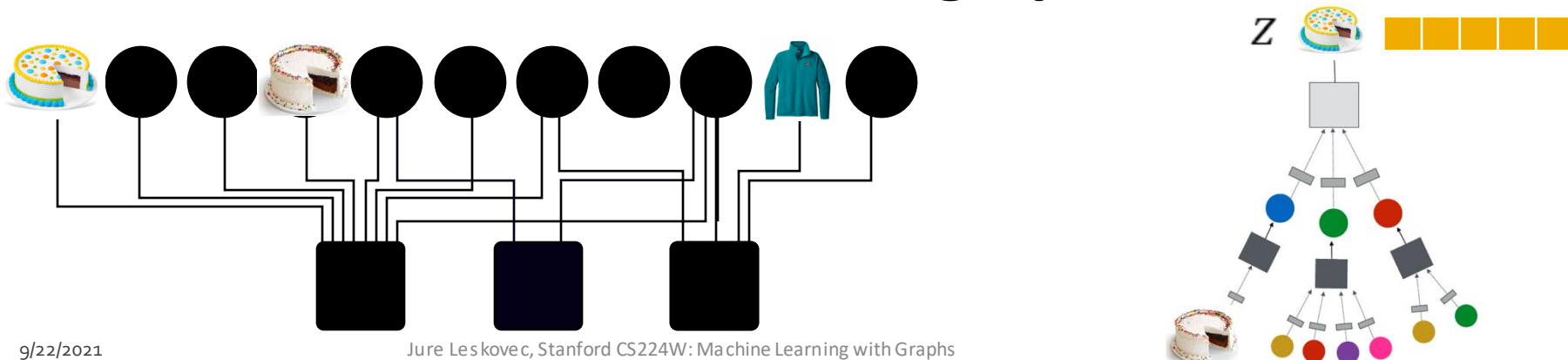
Task: Recommend related pins to users



Task: Learn node embeddings z_i such that

$$d(z_{cake1}, z_{cake2}) < d(z_{cake1}, z_{sweater})$$

Predict whether two nodes in a graph are related

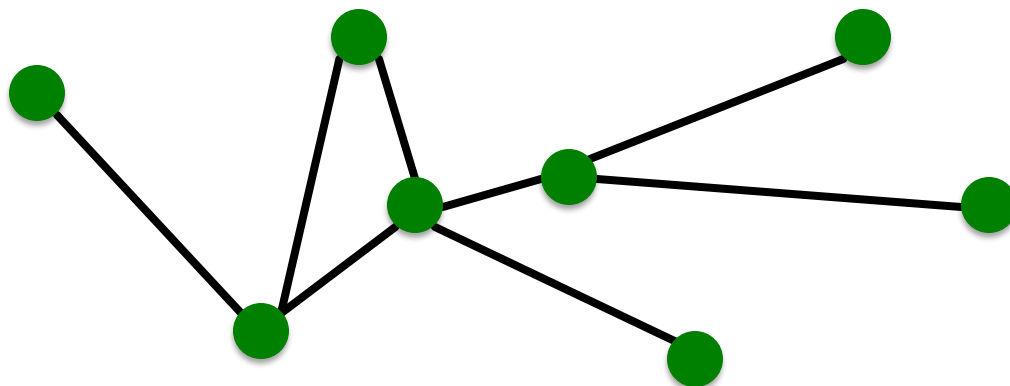


Stanford CS224W: Choice of Graph Representation

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



Components of a Network



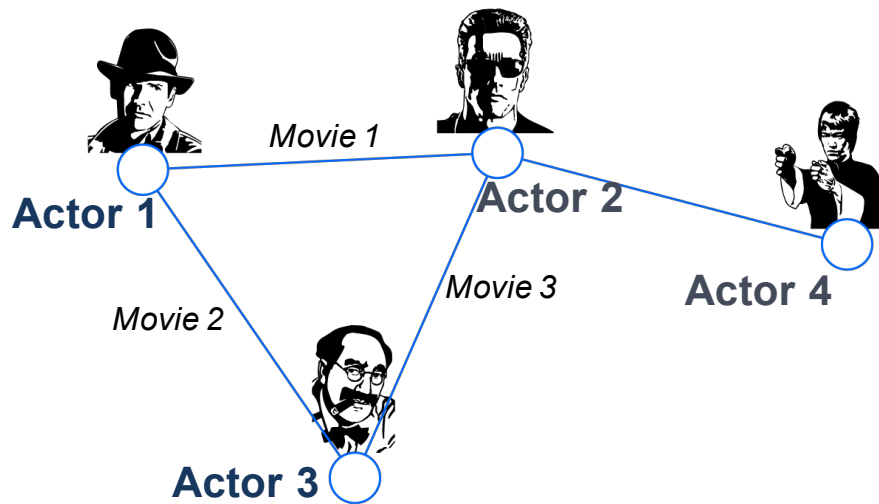
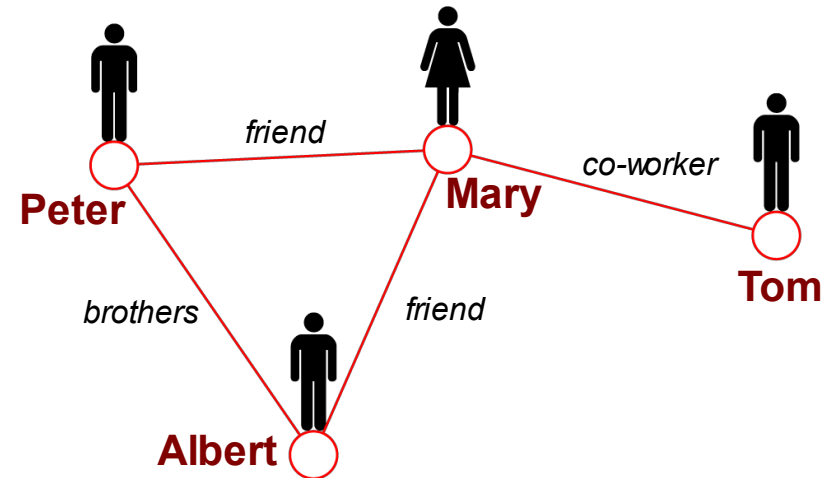
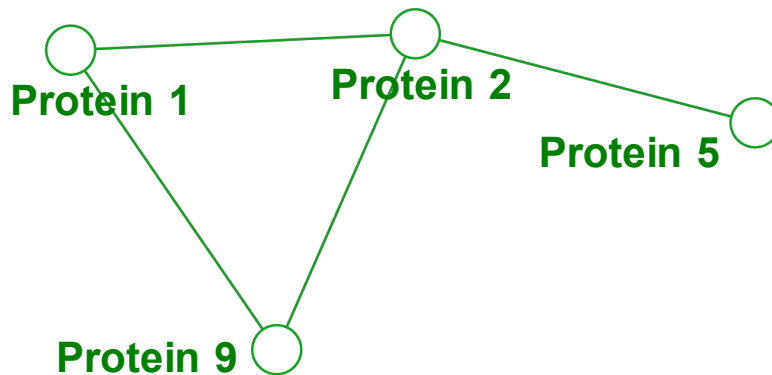
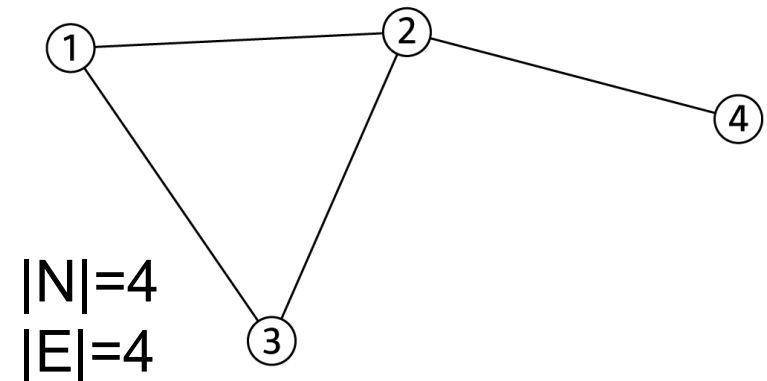
- **Objects:** nodes, vertices
- **Interactions:** links, edges
- **System:** network, graph

N

E

$G(N, E)$

Graphs: A Common Language



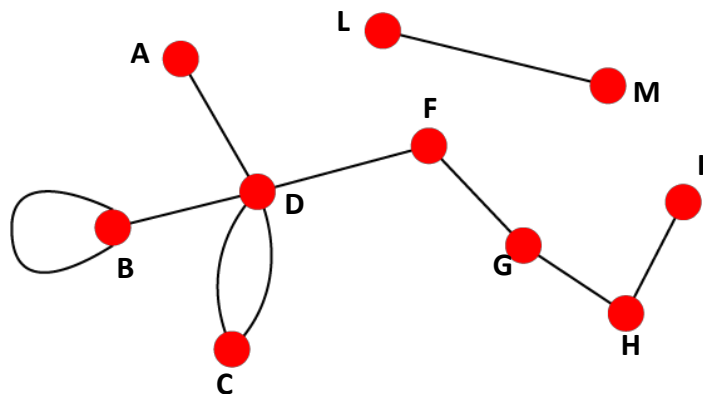
How do you define a graph?

- **How to build a graph:**
 - What are nodes?
 - What are edges?
- **Choice of the proper network representation of a given domain/problem determines our ability to use networks successfully:**
 - In some cases, there is a unique, unambiguous representation
 - In other cases, the representation is by no means unique
 - The way you assign links will determine the nature of the question you can study

Directed vs. Undirected Graphs

Undirected

- **Links:** undirected (symmetrical, reciprocal)

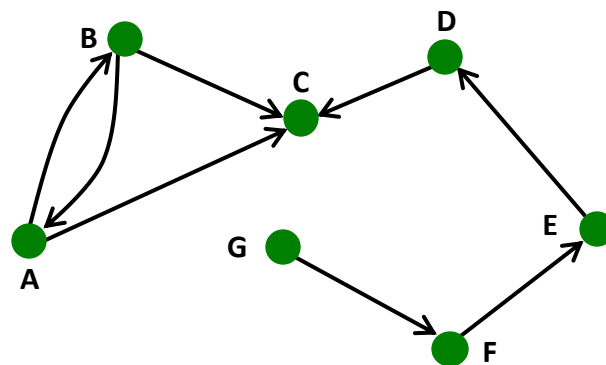


- **Examples:**

- Collaborations
- Friendship on Facebook

Directed

- **Links:** directed (arcs)



- **Examples:**

- Phone calls
- Following on Twitter

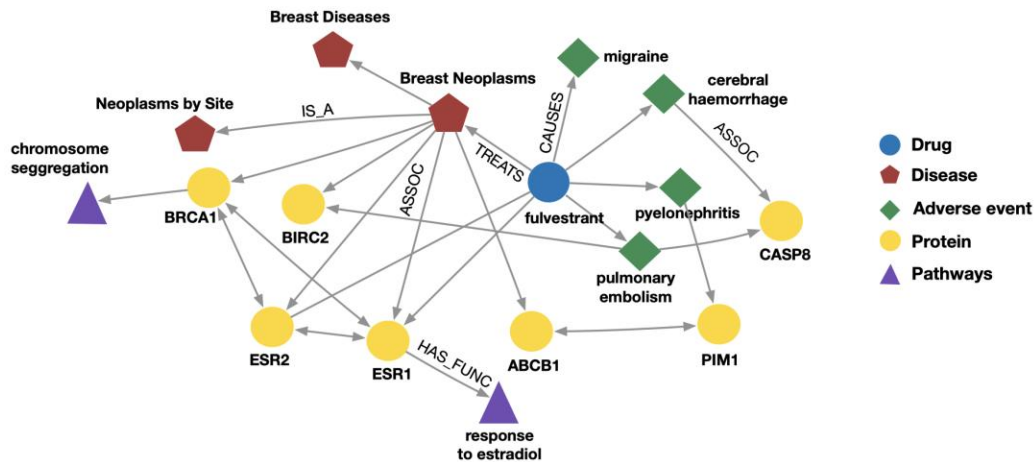
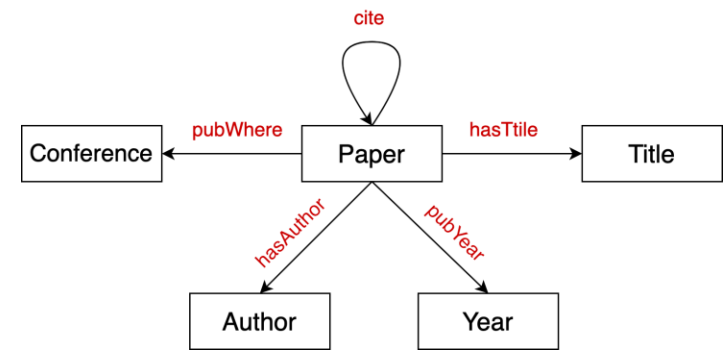
Heterogeneous Graphs

- A heterogeneous graph is defined as

$$G = (V, E, R, T)$$

- Nodes with node types $v_i \in V$
- Edges with relation types $(v_i, r, v_j) \in E$
- Node type $T(v_i)$
- Relation type $r \in R$

Many Graphs are Heterogeneous Graphs



Biomedical Knowledge Graphs

Example node: Migraine

Example edge: (fulvestrant, Treats, Breast Neoplasms)

Example node type: Protein

Example edge type (relation): Causes

Academic Graphs

Example node: ICML

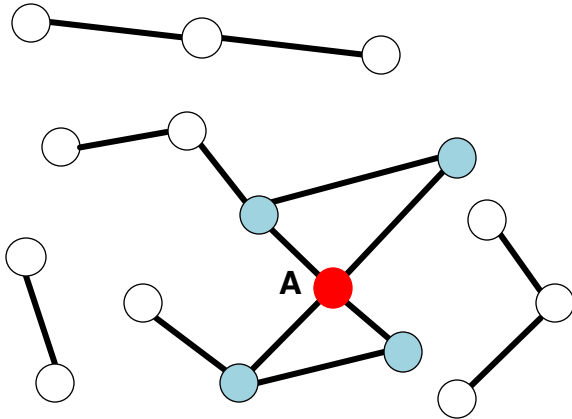
Example edge: (GraphSAGE, NeurIPS)

Example node type: Author

Example edge type (relation): pubYear

Node Degrees

Undirected

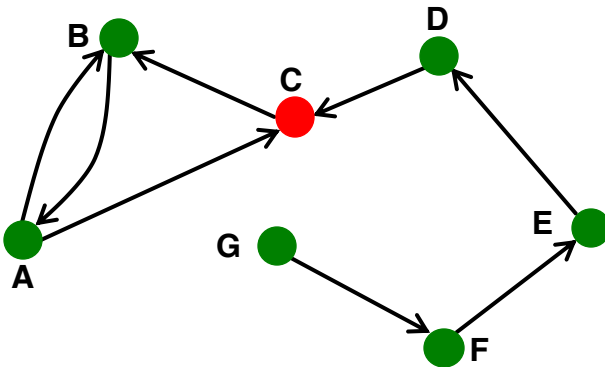


Node degree, k_i : the number of edges adjacent to node i

$$k_A = 4$$

Avg. degree: $\bar{k} = \langle k \rangle = \frac{1}{N} \sum_{i=1}^N k_i = \frac{2E}{N}$

Directed



In directed networks we define an **in-degree** and **out-degree**.

The (total) degree of a node is the sum of in- and out-degrees.

$$k_C^{in} = 2 \quad k_C^{out} = 1 \quad k_C = 3$$

$$\bar{k} = \frac{E}{N}$$

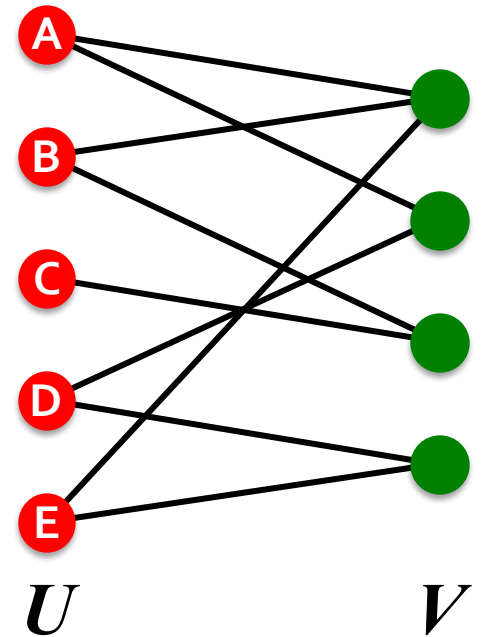
$$\overline{k^{in}} = \overline{k^{out}}$$

Source: Node with $k^{in} = 0$

Sink: Node with $k^{out} = 0$

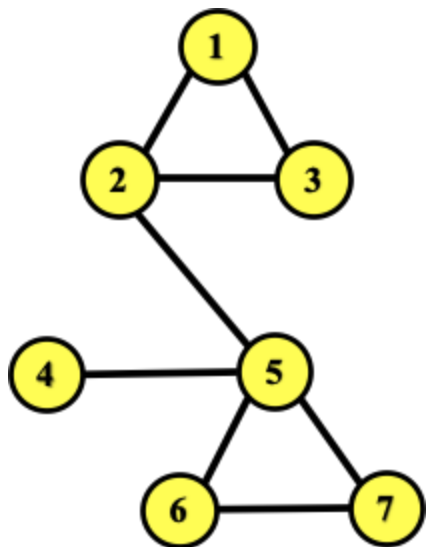
Bipartite Graph

- **Bipartite graph** is a graph whose nodes can be divided into two disjoint sets U and V such that every link connects a node in U to one in V ; that is, U and V are **independent sets**
- **Examples:**
 - Authors-to-Papers (they authored)
 - Actors-to-Movies (they appeared in)
 - Users-to-Movies (they rated)
 - Recipes-to-Ingredients (they contain)
- **“Folded” networks:**
 - Author collaboration networks
 - Movie co-rating networks



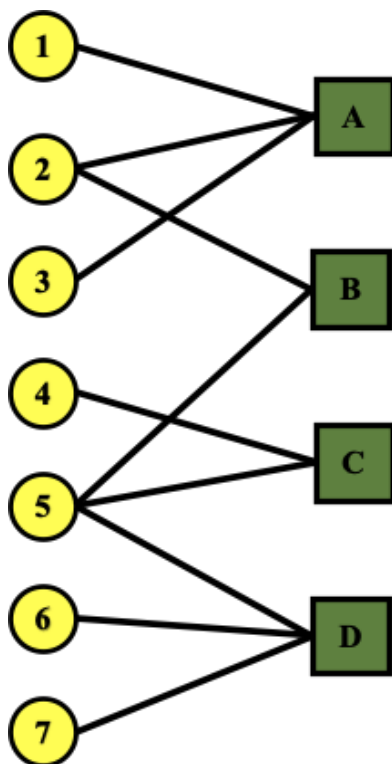
Folded/Projected Bipartite Graphs

Projection U

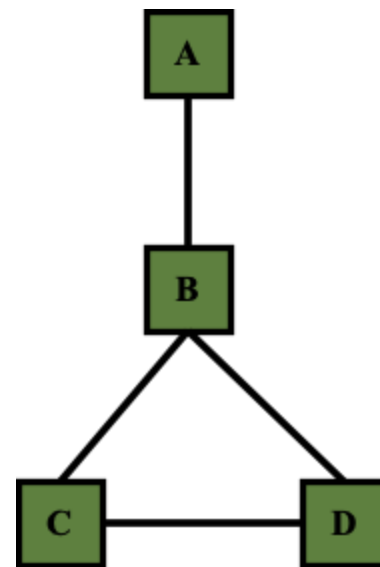


U

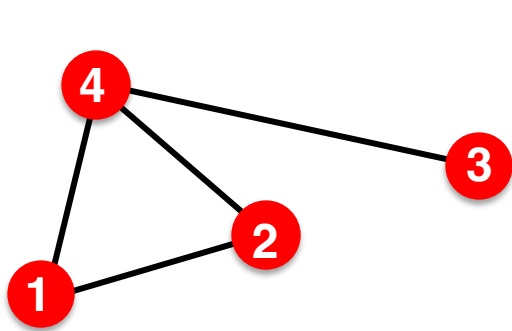
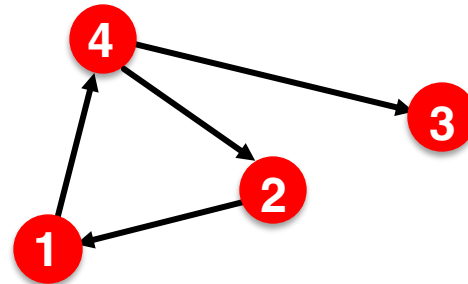
V



Projection V



Representing Graphs: Adjacency Matrix



$A_{ij} = 1$ if there is a link from node i to node j

$A_{ij} = 0$ otherwise

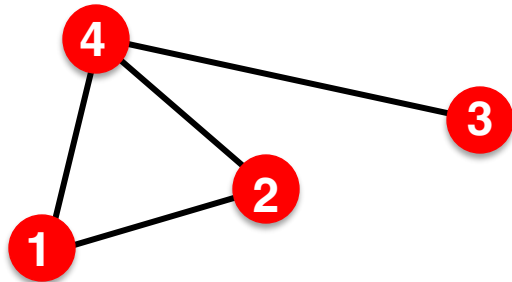
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

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

Note that for a directed graph (right) the matrix is not symmetric.

Adjacency Matrix

Undirected



$$A_{ij} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$A_{ij} = A_{ji}$$

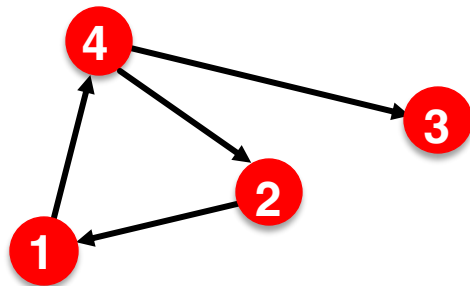
$$A_{ii} = 0$$

$$k_i = \sum_{j=1}^N A_{ij}$$

$$k_j = \sum_{i=1}^N A_{ij}$$

$$L = \frac{1}{2} \sum_{i=1}^N k_i = \frac{1}{2} \sum_{ij} A_{ij}$$

Directed



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

$$A_{ij} \neq A_{ji}$$

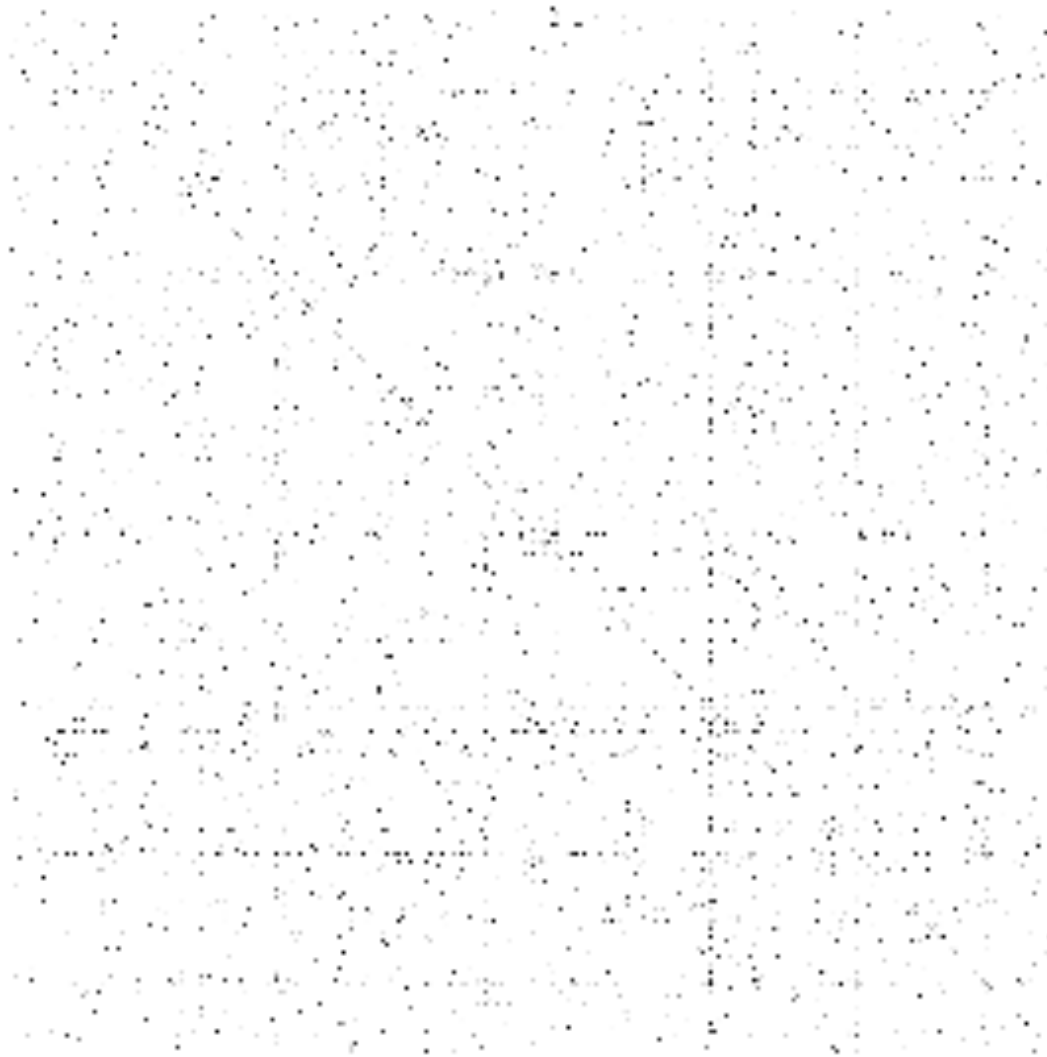
$$A_{ii} = 0$$

$$k_i^{out} = \sum_{j=1}^N A_{ij}$$

$$k_j^{in} = \sum_{i=1}^N A_{ij}$$

$$L = \sum_{i=1}^N k_i^{in} = \sum_{j=1}^N k_j^{out} = \sum_{i,j} A_{ij}$$

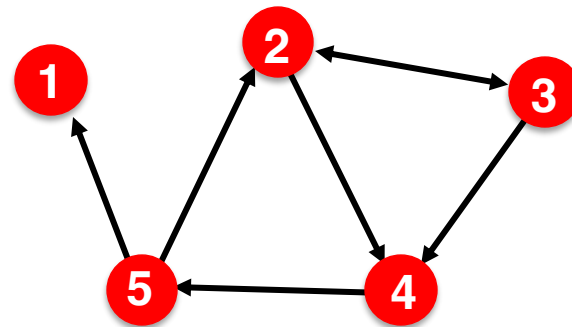
Adjacency Matrices are Sparse



Representing Graphs: Edge list

- Represent graph as a **list of edges**:

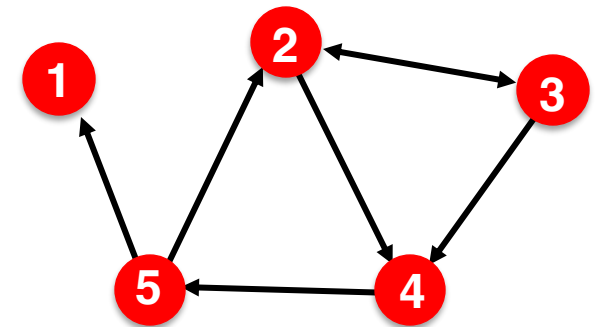
- (2, 3)
- (2, 4)
- (3, 2)
- (3, 4)
- (4, 5)
- (5, 2)
- (5, 1)



Representing Graphs: Adjacency list

■ Adjacency list:

- Easier to work with if network is
 - Large
 - Sparse
- Allows us to quickly retrieve all neighbors of a given node
 - 1:
 - 2: 3, 4
 - 3: 2, 4
 - 4: 5
 - 5: 1, 2



Node and Edge Attributes

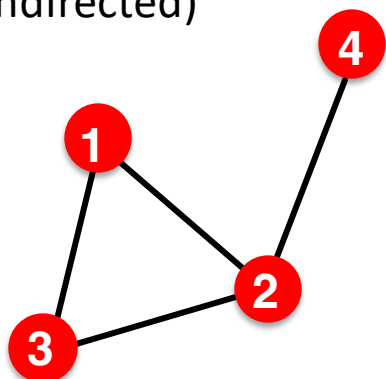
Possible options:

- Weight (*e.g.*, frequency of communication)
- Ranking (best friend, second best friend...)
- Type (friend, relative, co-worker)
- Sign: Friend vs. Foe, Trust vs. Distrust
- Properties depending on the structure of the rest of the graph: Number of common friends

More Types of Graphs

■ Unweighted

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

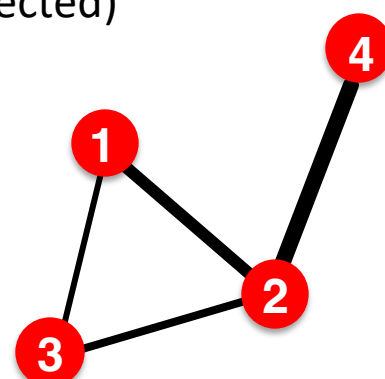
$$A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1}^N A_{ij} \quad \bar{k} = \frac{2E}{N}$$

Examples: Friendship, Hyperlink

■ Weighted

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 0.5 & 0 \\ 2 & 0 & 1 & 4 \\ 0.5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

$$A_{ij} = A_{ji}$$

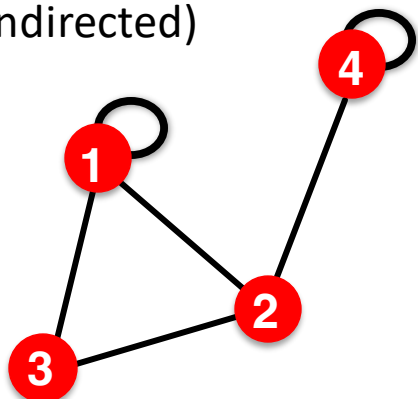
$$E = \frac{1}{2} \sum_{i,j=1}^N \text{nonzero}(A_{ij}) \quad \bar{k} = \frac{2E}{N}$$

Examples: Collaboration, Internet, Roads

More Types of Graphs

■ Self-edges (self-loops)

(undirected)



$$A_{ij} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$A_{ii} \neq 0$$

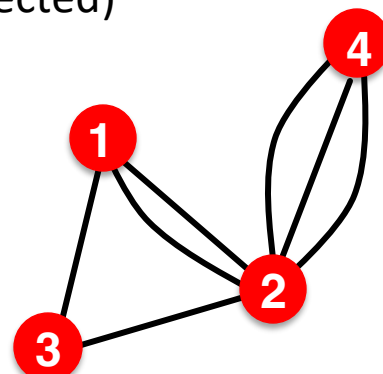
$$A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1, i \neq j}^N A_{ij} + \sum_{i=1}^N A_{ii}$$

Examples: Proteins, Hyperlinks

■ Multigraph

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 3 \\ 1 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

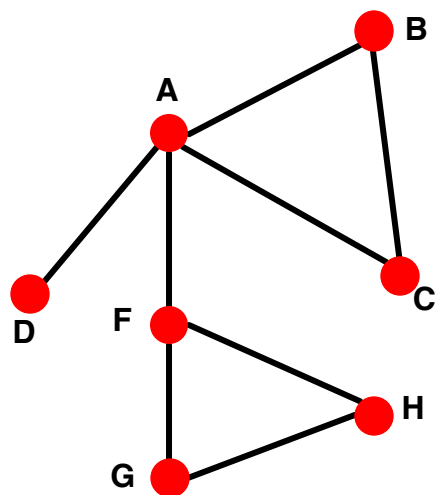
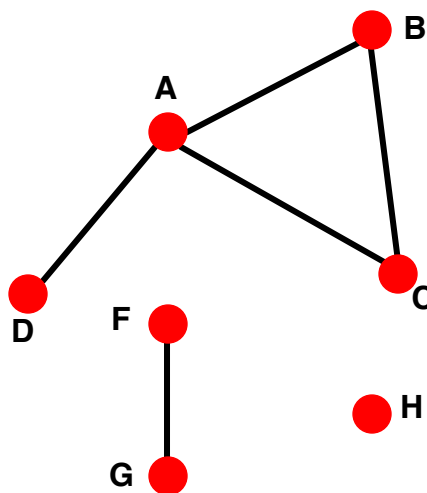
$$A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1}^N \text{nonzero}(A_{ij}) \quad \bar{k} = \frac{2E}{N}$$

Examples: Communication, Collaboration

Connectivity of Undirected Graphs

- **Connected (undirected) graph:**
 - Any two vertices can be joined by a path
- A disconnected graph is made up by two or more connected components

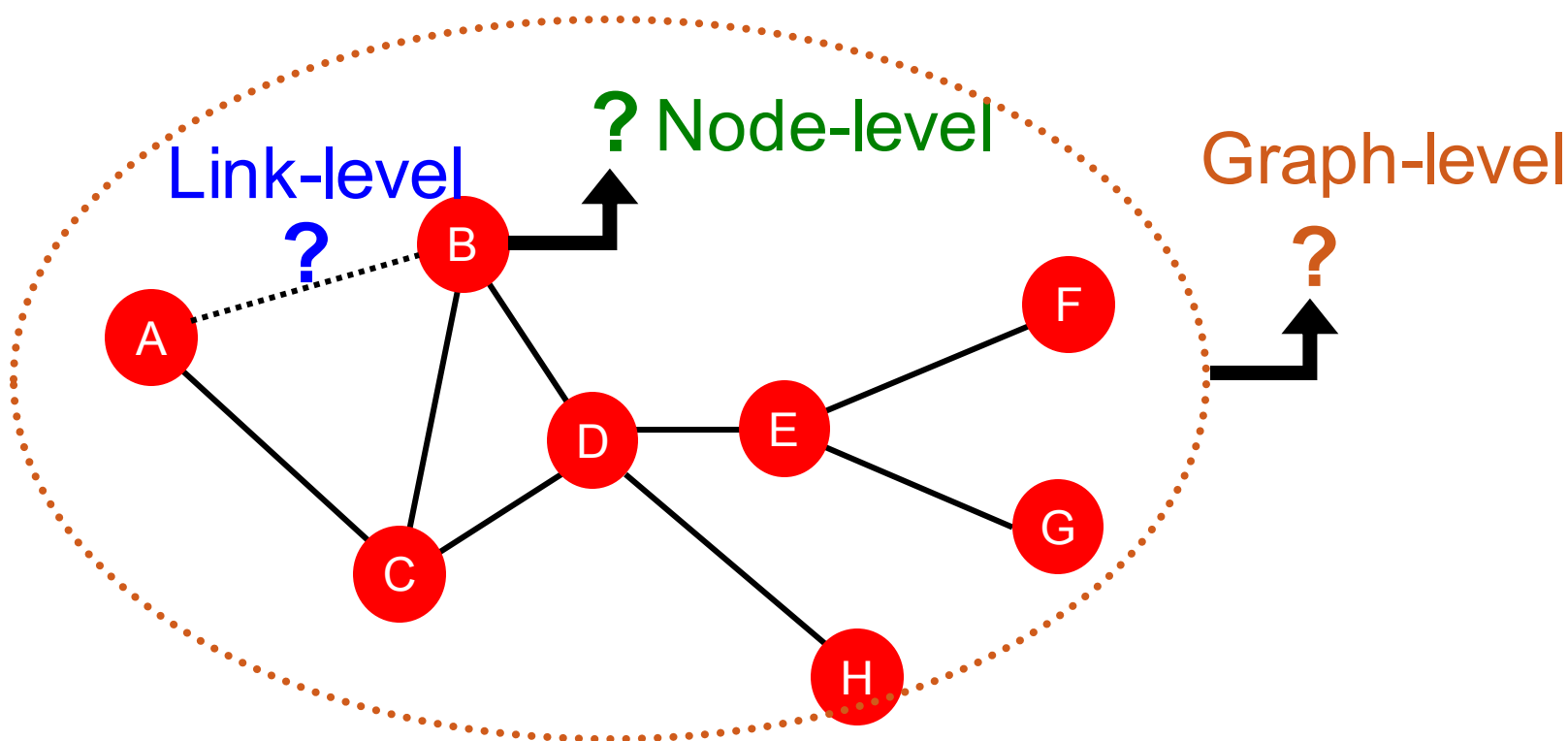


Largest Component:
Giant Component

Isolated node (node H)

Machine Learning Tasks: Review

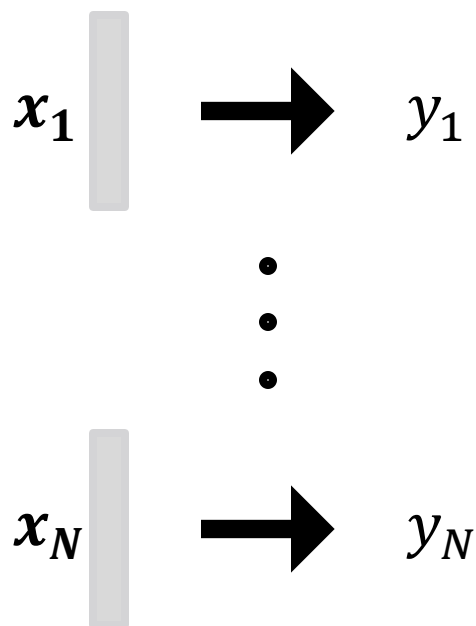
- Node-level prediction
- Link-level prediction
- Graph-level prediction



Traditional ML Pipeline

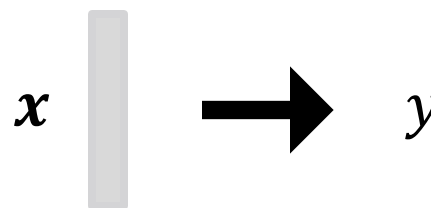
■ Train an ML model:

- Random forest
- SVM
- Neural network, etc.



■ Apply the model:

- Given a new node/link/graph, obtain its features and make a prediction



This Lecture: Feature Design

- Using effective features over graphs is the key to achieving good model performance.
- Traditional ML pipeline uses hand-designed features.
- In this lecture, we overview the traditional features for:
 - Node-level prediction
 - Link-level prediction
 - Graph-level prediction
- For simplicity, we focus on undirected graphs.

Machine Learning in Graphs

Goal: Make predictions for a set of objects

Design choices:

- **Features:** d -dimensional vectors
- **Objects:** Nodes, edges, sets of nodes, entire graphs
- **Objective function:**
 - What task are we aiming to solve?

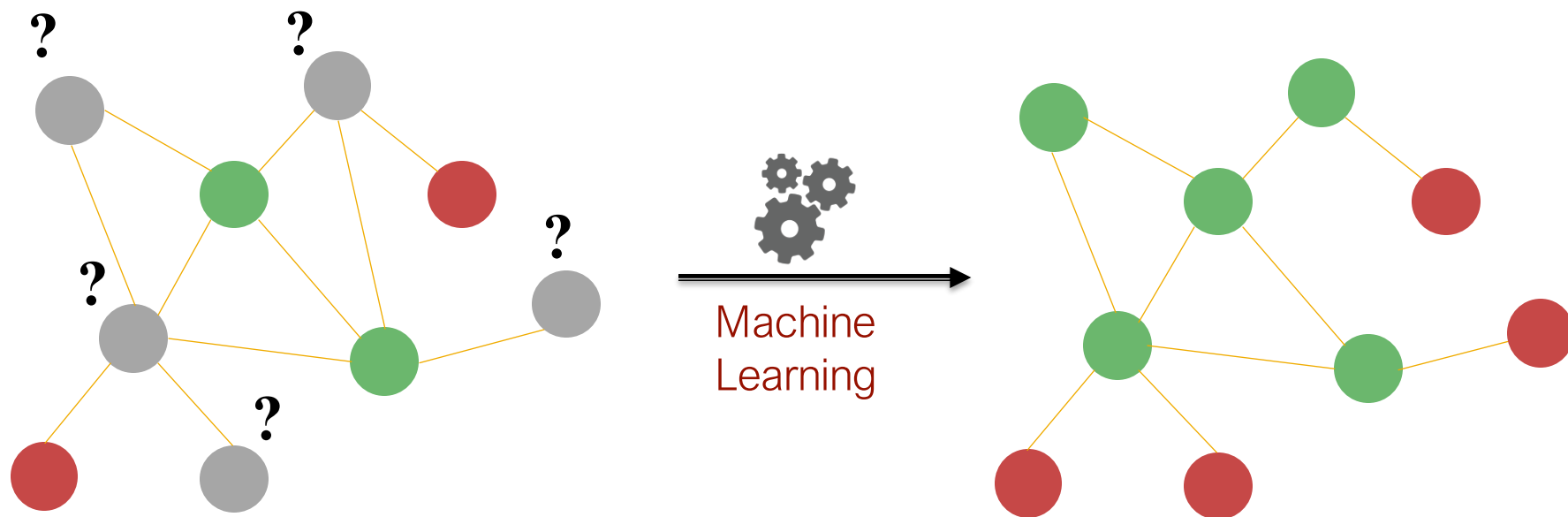
Machine Learning in Graphs

Example: Node-level prediction

- Given: $G = (V, E)$
- Learn a function: $f : V \rightarrow \mathbb{R}$

How do we learn the function?

Node-Level Tasks



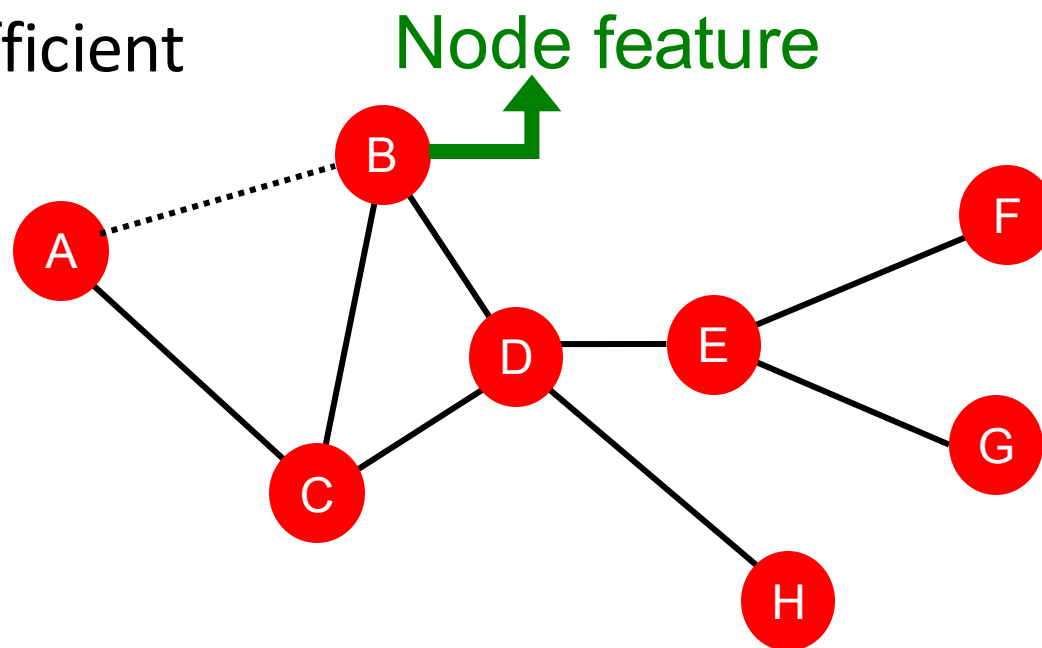
Node classification

ML needs features.

Node-Level Features: Overview

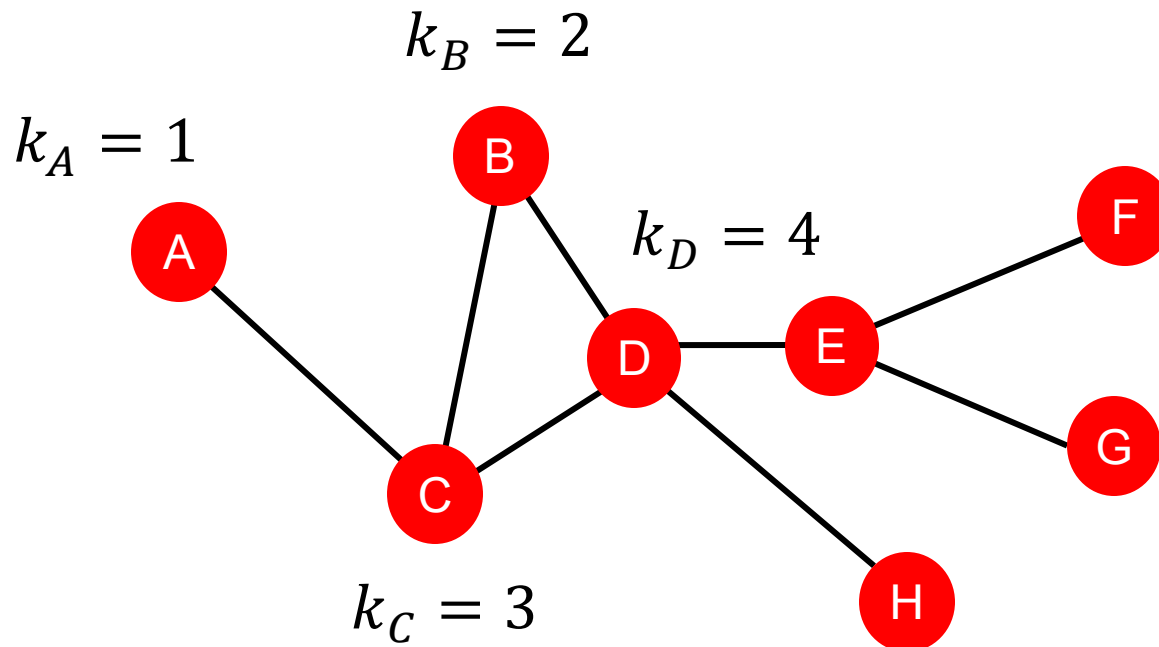
Goal: Characterize the structure and position of a node in the network:

- Node degree
- Node centrality
- Clustering coefficient
- Graphlets



Node Features: Node Degree

- The degree k_v of node v is the number of edges (neighboring nodes) the node has.
- Treats all neighboring nodes equally.



Node Features: Node Centrality

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality c_v takes the node importance in a graph into account
- **Different ways to model importance:**
 - Eigenvector centrality
 - Betweenness centrality
 - Closeness centrality
 - and many others...

Node Centrality (1)

■ Eigenvector centrality:

- A node v is important if **surrounded by important neighboring nodes** $u \in N(v)$.
- We model the centrality of node v as **the sum of the centrality of neighboring nodes**:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

λ is normalization constant (it will turn out to be the largest eigenvalue of A)

- Notice that the above equation models centrality in a **recursive manner**. **How do we solve it?**

Node Centrality (1)

■ Eigenvector centrality:

- Rewrite the recursive equation in the matrix form.

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \quad \longleftrightarrow \quad \lambda \mathbf{c} = \mathbf{A} \mathbf{c}$$

λ is normalization const
(largest eigenvalue of A)

- A : Adjacency matrix
 $A_{uv} = 1$ if $u \in N(v)$
- $\mathbf{c}$: Centrality vector
- λ : Eigenvalue

- We see that centrality $\mathbf{c}$ is the **eigenvector of A !**
- The largest eigenvalue λ_{max} is always positive and unique (by Perron-Frobenius Theorem).
- The eigenvector $\mathbf{c}_{max}$ corresponding to λ_{max} is used for centrality.

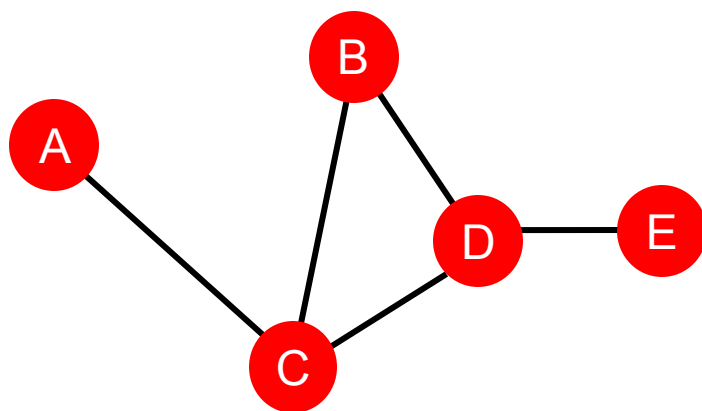
Node Centrality (2)

■ Betweenness centrality:

- A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths between } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

■ Example:



$$c_A = c_B = c_E = 0$$
$$c_C = 3$$
$$(A-\underline{C}-B, A-\underline{C}-D, A-\underline{C}-D-E)$$

$$c_D = 3$$
$$(A-C-\underline{D}-E, B-\underline{D}-E, C-\underline{D}-E)$$

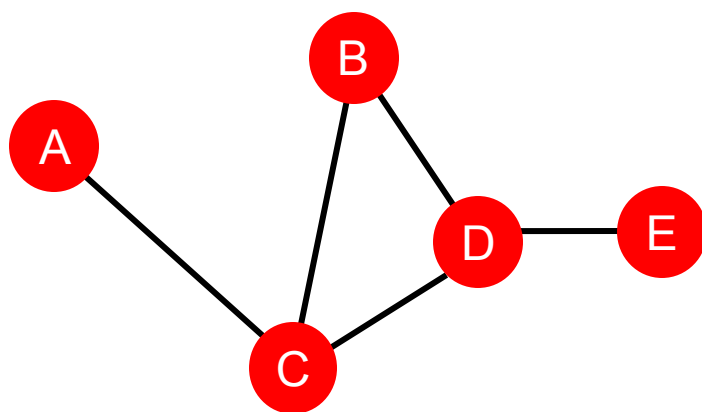
Node Centrality (3)

■ Closeness centrality:

- A node is important if it has small shortest path lengths to all other nodes.

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

■ Example:



$$c_A = 1/(2 + 1 + 2 + 3) = 1/8$$

(A-C-B, A-C, A-C-D, A-C-D-E)

$$c_D = 1/(2 + 1 + 1 + 1) = 1/5$$

(D-C-A, D-B, D-C, D-E)

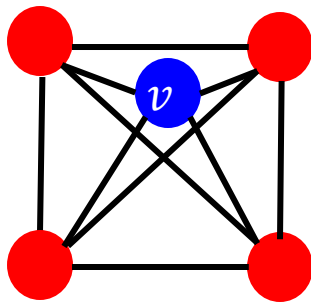
Node Features: Clustering Coefficient

- Measures how connected v 's neighboring nodes are:

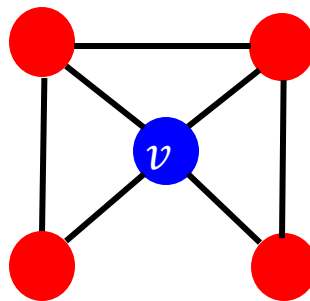
$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\binom{k_v}{2}} \in [0,1]$$

#(node pairs among k_v neighboring nodes)
In our examples below the denominator is 6 (4 choose 2).

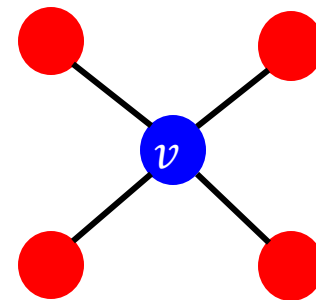
- Examples:**



$$e_v = 1$$



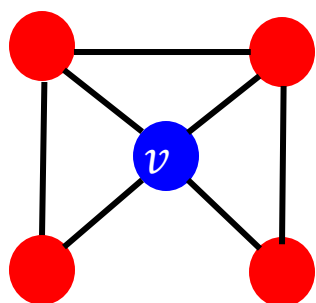
$$e_v = 0.5$$



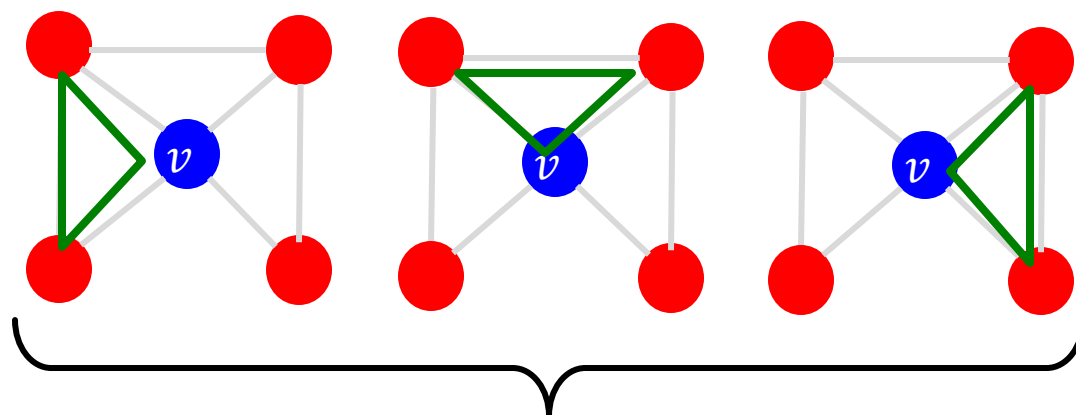
$$e_v = 0$$

Node Features: Graphlets

- **Observation:** Clustering coefficient counts the #(triangles) in the ego-network



$$e_v = 0.5$$

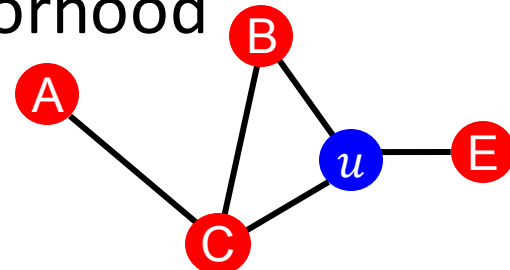


3 triangles (out of 6 node triplets)

- We can generalize the above by counting #(pre-specified subgraphs, i.e., **graphlets**).

Node Features: Graphlets

- **Goal:** Describe network structure around node u
 - **Graphlets** are small subgraphs that describe the structure of node u 's network neighborhood

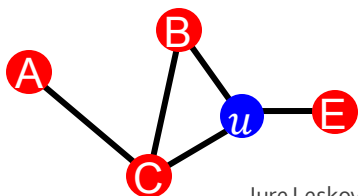
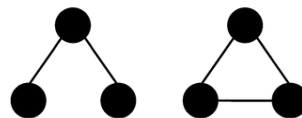


Analogy:

- **Degree** counts **#(edges)** that a node touches
- **Clustering coefficient** counts **#(triangles)** that a node touches.
- **Graphlet Degree Vector (GDV):** Graphlet-base features for nodes
 - **GDV** counts **#(graphlets)** that a node touches

Node Features: Graphlets

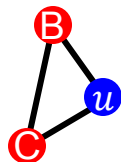
- Considering graphlets of size 2-5 nodes we get:
 - **Vector of 73 coordinates** is a signature of a node that describes the topology of node's neighborhood
- Graphlet degree vector provides a measure of a **node's local network topology**:
 - Comparing vectors of two nodes provides a more detailed measure of local topological similarity than node degrees or clustering coefficient.



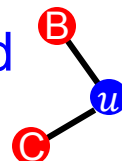
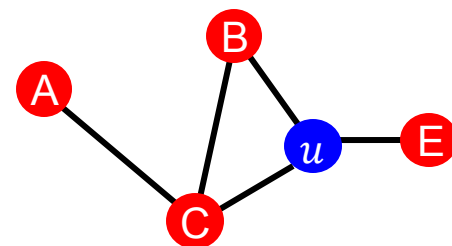
Induced Subgraph & Isomorphism

- **Def: Induced subgraph** is another graph, formed from a subset of vertices and *all* of the edges connecting the vertices in that subset.

Induced subgraph:

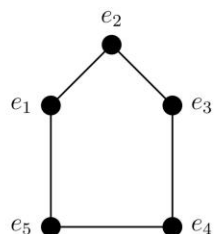
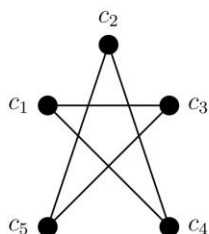


Not induced subgraph:



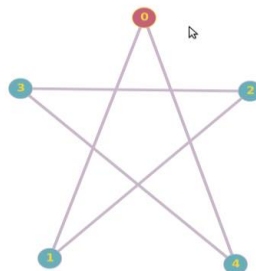
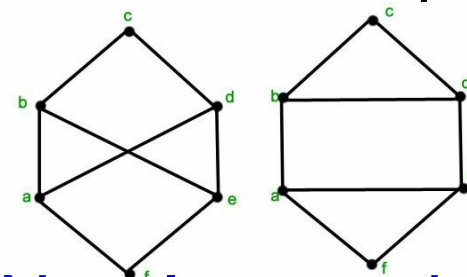
- **Def: Graph Isomorphism**

- Two graphs which contain the same number of nodes connected in the same way are said to be isomorphic.



Isomorphic

Node mapping: (e2,c2), (e1, c5), (e3,c4), (e5,c3), (e4,c1)



Non-Isomorphic

The right graph has cycles of length 3 but the left graph does not, so the graphs cannot be isomorphic.

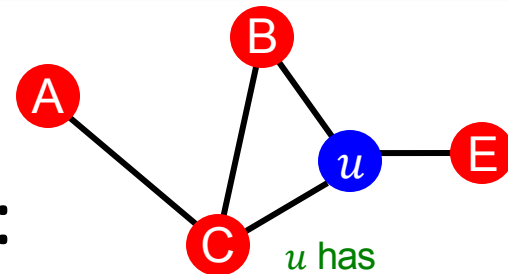
Source: Mathoverflow

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

Node Features: Graphlets

Graphlets: Rooted connected induced non-isomorphic subgraphs:

Take some nodes and all the edges between them.

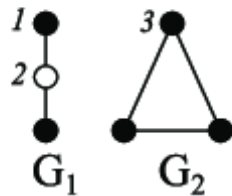


u has graphlets: 0, 1, 2, 3, 5, 10, 11, ...

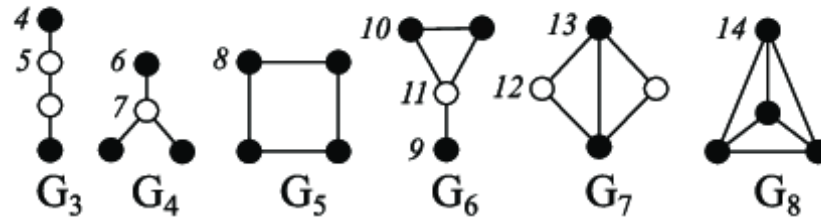
2-node graphlet



3-node graphlets

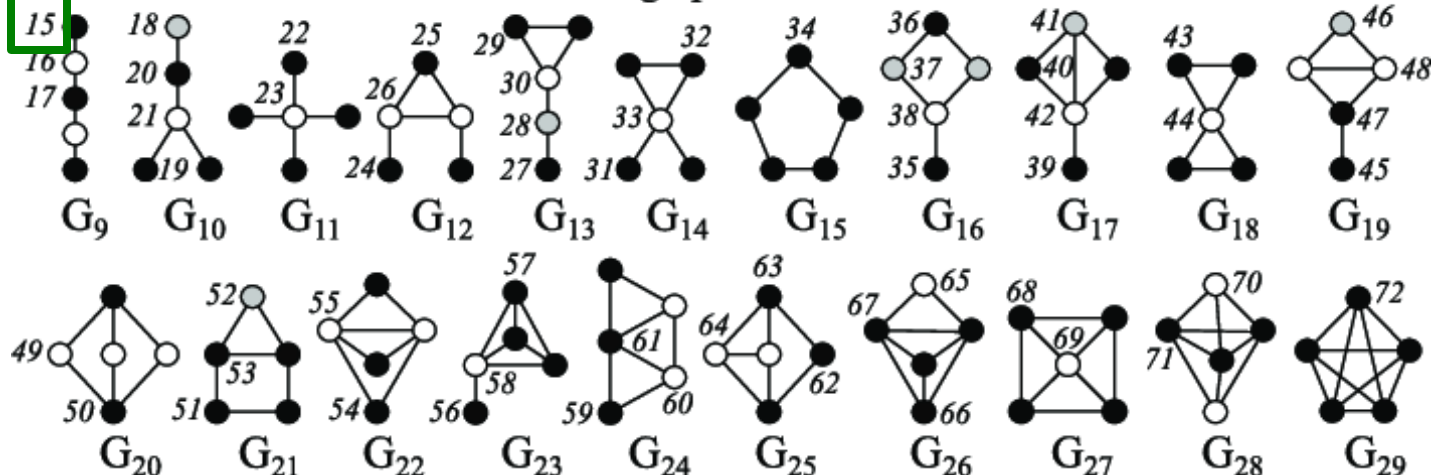


4-node graphlets



Graphlet id (Root/ "position" of node u)

5-node graphlets



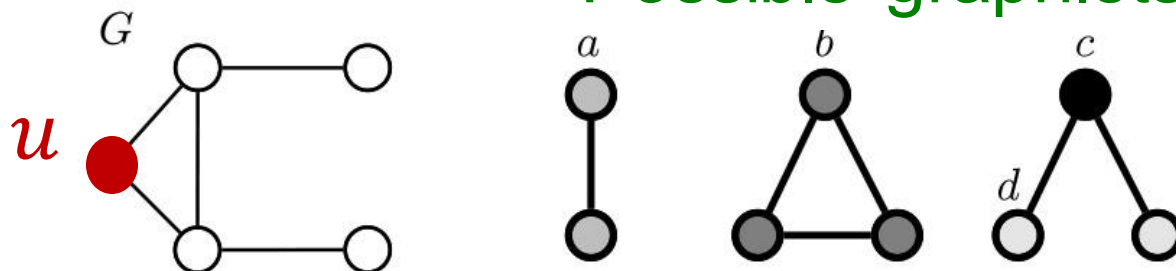
There are 73 different graphlets on up to 5 nodes

Node Features: Graphlets

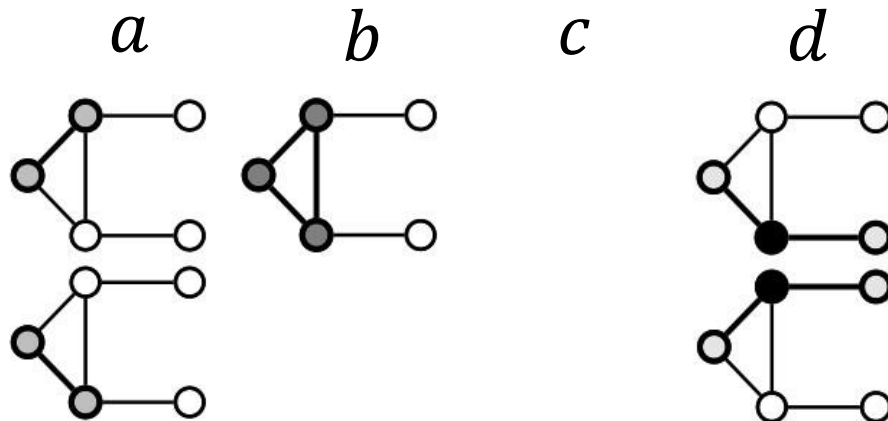
- **Graphlet Degree Vector (GDV):** A count vector of graphlets rooted at a given node.

- **Example:**

Possible graphlets up to size 3



Graphlet instances of node u :



GDV of node u :
 a, b, c, d
 $[2, 1, 0, 2]$

Node-Level Feature: Summary

- We have introduced different ways to obtain node features.
- They can be categorized as:
 - Importance-based features:
 - Node degree
 - Different node centrality measures
 - Structure-based features:
 - Node degree
 - Clustering coefficient
 - Graphlet count vector

Node-Level Feature: Summary

- **Importance-based features:** capture the importance of a node in a graph
 - Node degree:
 - Simply counts the number of neighboring nodes
 - Node centrality:
 - Models **importance of neighboring nodes** in a graph
 - Different modeling choices: eigenvector centrality, betweenness centrality, closeness centrality
- Useful for predicting influential nodes in a graph
 - **Example:** predicting celebrity users in a social network

Node-Level Feature: Summary

- **Structure-based features:** Capture topological properties of local neighborhood around a node.
 - **Node degree:**
 - Counts the number of neighboring nodes
 - **Clustering coefficient:**
 - Measures how connected neighboring nodes are
 - **Graphlet degree vector:**
 - Counts the occurrences of different graphlets
- **Useful for predicting a particular role a node plays in a graph:**
 - **Example:** Predicting protein functionality in a protein-protein interaction network.

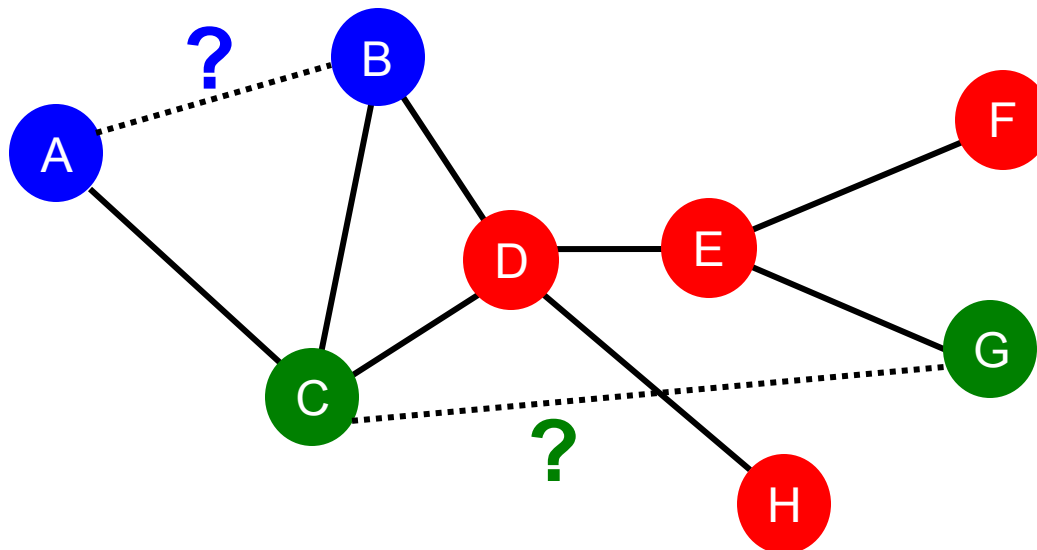
Stanford CS224W: Link Prediction Task and Features

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



Link-Level Prediction Task: Recap

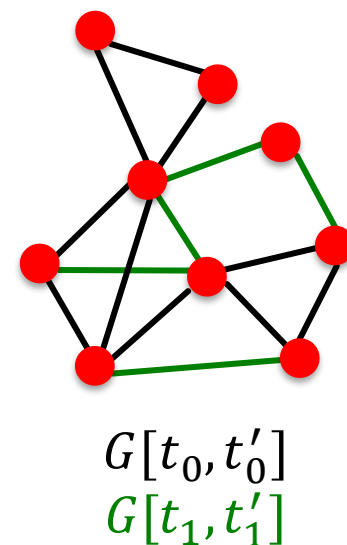
- The task is to predict **new links** based on the existing links.
- At test time, node pairs (with no existing links) are ranked, and top K node pairs are predicted.
- The key is to design features for a **pair of nodes**.



Link Prediction as a Task

Two formulations of the link prediction task:

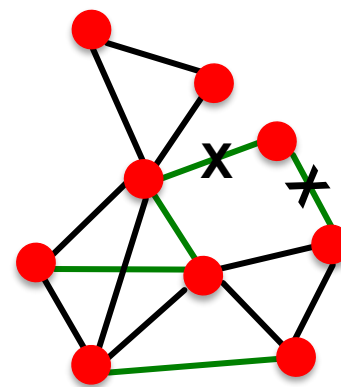
- **1) Links missing at random:**
 - Remove a random set of links and then aim to predict them
- **2) Links over time:**
 - Given $G[t_0, t'_0]$ a graph defined by edges up to time t'_0 , **output a ranked list L** of edges (not in $G[t_0, t'_0]$) that are predicted to appear in time $G[t_1, t'_1]$
 - **Evaluation:**
 - $n = |E_{new}|$: # new edges that appear during the test period $[t_1, t'_1]$
 - Take top n elements of L and count correct edges



Link Prediction via Proximity

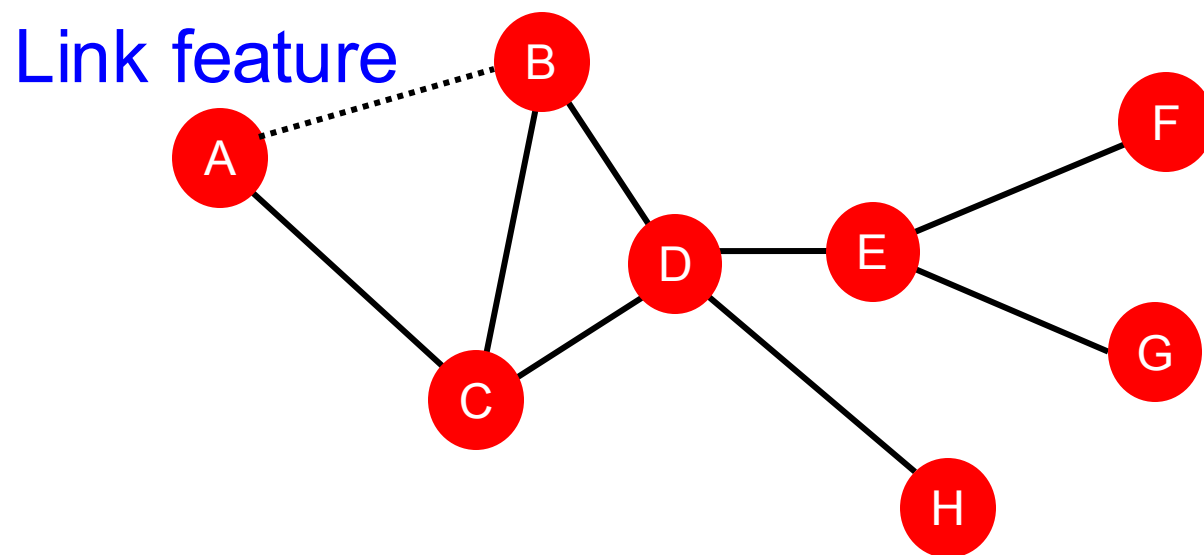
■ Methodology:

- For each pair of nodes (x,y) compute score $c(x,y)$
 - For example, $c(x,y)$ could be the # of common neighbors of x and y
- Sort pairs (x,y) by the decreasing score $c(x,y)$
- **Predict top n pairs as new links**
- **See which of these links actually appear in $G[t_1, t'_1]$**



Link-Level Features: Overview

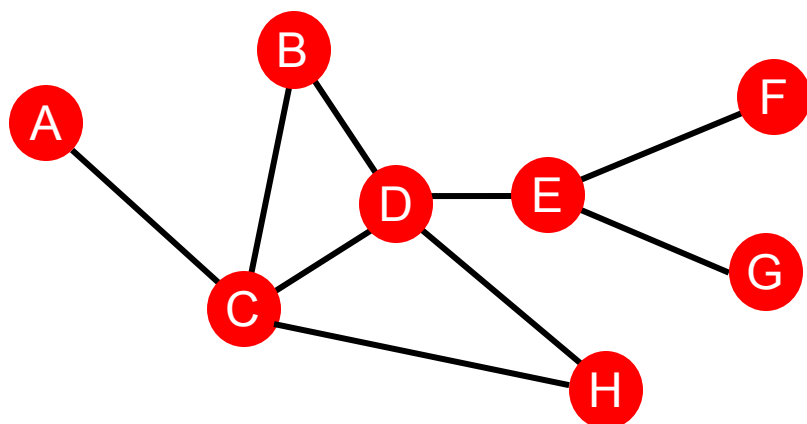
- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap



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.

Local Neighborhood Overlap

Captures # neighboring nodes shared between two nodes v_1 and v_2 :

- **Common neighbors:** $|N(v_1) \cap N(v_2)|$

- Example: $|N(A) \cap N(B)| = |\{C\}| = 1$

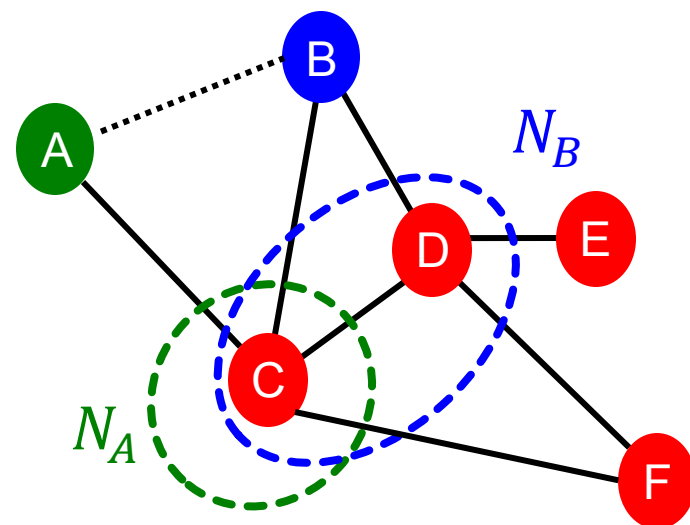
- **Jaccard's coefficient:** $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$

- Example: $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C, D\}|} = \frac{1}{2}$

- **Adamic-Adar index:**

$$\sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)}$$

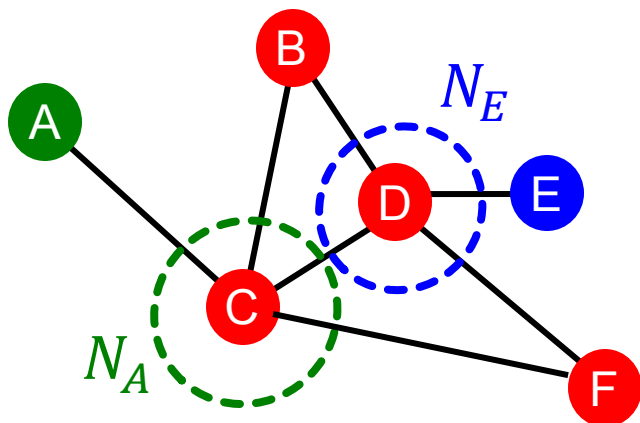
- Example: $\frac{1}{\log(k_C)} = \frac{1}{\log 4}$



Global Neighborhood Overlap

- **Limitation of local neighborhood features:**

- Metric is always zero if the two nodes do not have any neighbors in common.



$$N_A \cap N_E = \phi$$
$$|N_A \cap N_E| = 0$$

- However, the two nodes may still potentially be connected in the future.
- **Global neighborhood overlap** metrics resolve the limitation by considering the entire graph.

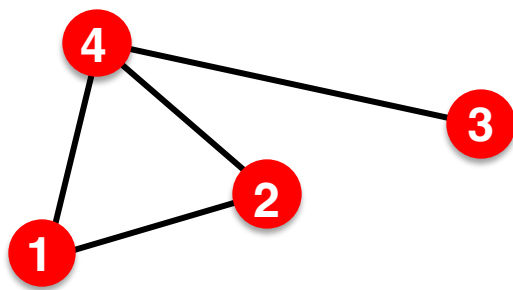
Global Neighborhood Overlap

- **Katz index:** count the number of walks of all lengths between a given pair of nodes.
- **Q: How to compute #walks between two nodes?**
- Use **powers of the graph adjacency matrix!**

Intuition: Powers of Adj Matrices

■ Computing #walks between two nodes

- Recall: $A_{uv} = 1$ if $u \in N(v)$
- Let $P_{uv}^{(K)} = \text{\#walks of length } K \text{ between } u \text{ and } v$
- We will show $P^{(K)} = A^k$
- $P_{uv}^{(1)} = \text{\#walks of length 1 (direct neighborhood) between } u \text{ and } v = A_{uv}$



$$P_{12}^{(1)} = A_{12}$$
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Intuition: Powers of Adj Matrices

- How to compute $P_{uv}^{(2)}$?
 - Step 1: Compute **#walks** of length 1 **between each of u 's neighbor and v**
 - Step 2: **Sum up** these #walks across u 's neighbors
 - $P_{uv}^{(2)} = \sum_i A_{ui} * P_{iv}^{(1)} = \sum_i A_{ui} * A_{iv} = A_{uv}^2$

Node 1's neighbors

#walks of length 1 between

Node 1's neighbors and Node 2

$$P_{12}^{(2)} = A_{12}^2$$

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

Power of adjacency

Global Neighborhood Overlap

- **Katz index:** count the number of walks of all lengths between a pair of nodes.
- How to compute #walks between two nodes?
- Use **adjacency matrix powers!**
 - A_{uv} specifies #walks of length 1 (direct neighborhood) between u and v .
 - A_{uv}^2 specifies #walks of **length 2** (neighbor of neighbor) between u and v .
 - And, A_{uv}^l specifies #walks of **length l** .

Global Neighborhood Overlap

- **Katz index** between v_1 and v_2 is calculated as

Sum over *all walk lengths*

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \boxed{\beta^l} \boxed{A_{v_1 v_2}^l}$$

#walks of length l between v_1 and v_2

$0 < \beta < 1$: discount factor

- Katz index matrix is computed in closed-form:

$$S = \sum_{i=1}^{\infty} \beta^i A^i = \underbrace{(I - \beta A)^{-1} - I}_{= \sum_{i=0}^{\infty} \beta^i A^i}$$

by geometric series of matrices

Link-Level Features: Summary

- **Distance-based features:**
 - Uses the shortest path length between two nodes but does not capture how neighborhood overlaps.
- **Local neighborhood overlap:**
 - Captures how many neighboring nodes are shared by two nodes.
 - Becomes zero when no neighbor nodes are shared.
- **Global neighborhood overlap:**
 - Uses global graph structure to score two nodes.
 - Katz index counts #walks of all lengths between two nodes.

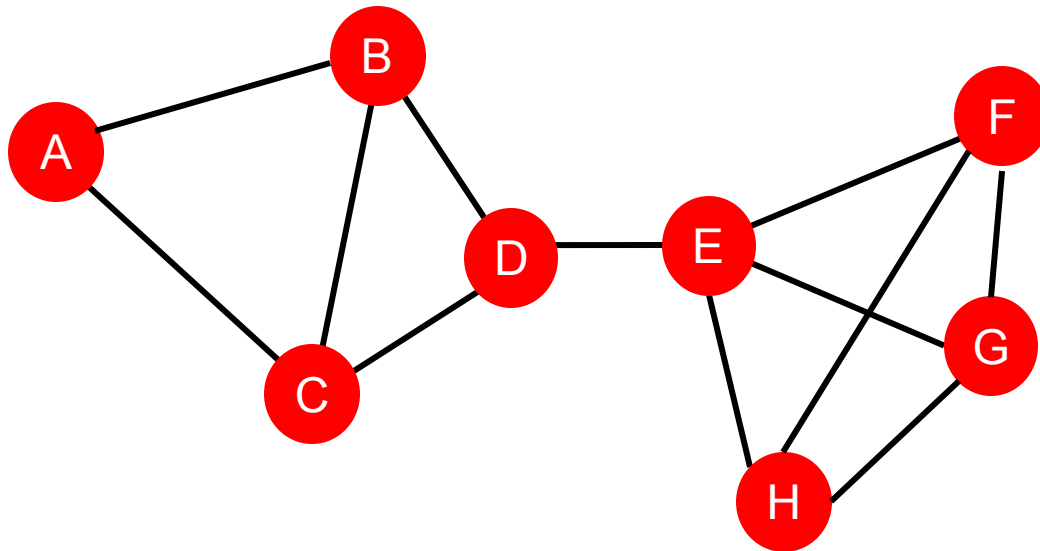
Stanford CS224W: Graph-Level Features and Graph Kernels

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



Graph-Level Features

- **Goal:** We want features that characterize the structure of an entire graph.
- **For example:**



Background: Kernel Methods

- **Kernel methods** are widely-used for traditional ML for graph-level prediction.
- **Idea: Design kernels instead of feature vectors.**
- **A quick introduction to Kernels:**
 - Kernel $K(G, G') \in \mathbb{R}$ measures similarity b/w data
 - Kernel matrix $\mathbf{K} = (K(G, G'))_{G, G'}$ must always be positive semidefinite (i.e., has positive eigenvalues)
 - There exists a feature representation $\phi(\cdot)$ such that $K(G, G') = \phi(G)^T \phi(G')$
 - Once the kernel is defined, off-the-shelf ML model, such as **kernel SVM**, can be used to make predictions.

Graph-Level Features: Overview

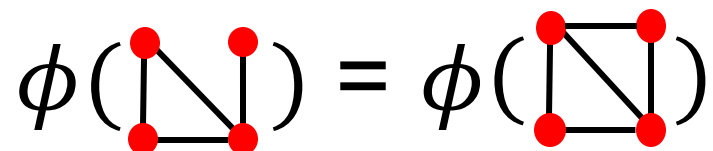
- **Graph Kernels:** Measure similarity between two graphs:
 - Graphlet Kernel [1]
 - Weisfeiler-Lehman Kernel [2]
 - Other kernels are also proposed in the literature (beyond the scope of this lecture)
 - Random-walk kernel
 - Shortest-path graph kernel
 - And many more...

[1] Shervashidze, Nino, et al. "Efficient graphlet kernels for large graph comparison." Artificial Intelligence and Statistics. 2009.

[2] Shervashidze, Nino, et al. "Weisfeiler-lehman graph kernels." Journal of Machine Learning Research 12.9 (2011).

Graph Kernel: Key Idea

- **Goal:** Design graph feature vector $\phi(G)$
- **Key idea:** **Bag-of-Words (BoW)** for a graph
 - **Recall:** BoW simply uses the word counts as features for documents (no ordering considered).
 - Naïve extension to a graph: **Regard nodes as words.**
 - Since both graphs have **4 red nodes**, we get the same feature vector for two different graphs...

$$\phi(\text{triangle}) = \phi(\text{square})$$


Graph Kernel: Key Idea

What if we use Bag of node degrees?

Deg1: ● Deg2: ● Deg3: ●

$$\phi(\text{triangle}) = \text{count}(\text{triangle with colored nodes}) = [1, 2, 1]$$



Obtains different features for different graphs!

$$\phi(\text{square}) = \text{count}(\text{square with colored nodes}) = [0, 2, 2]$$

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

Weisfeiler-Lehman Kernel

- **Goal:** Design an efficient graph feature descriptor $\phi(G)$
- **Idea:** Use neighborhood structure to iteratively enrich node vocabulary.
 - Generalized version of **Bag of node degrees** since node degrees are one-hop neighborhood information.
- **Algorithm to achieve this:**

Color refinement

Color Refinement

- **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

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

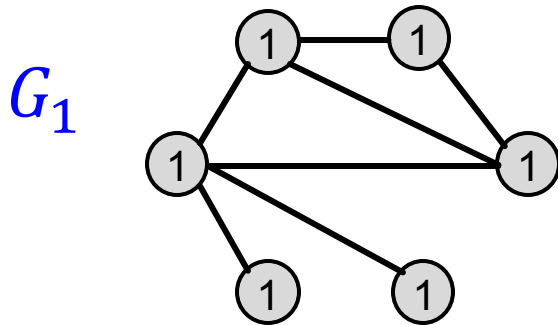
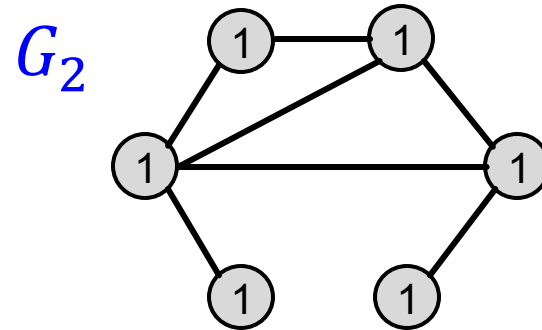
where **HASH** maps different inputs to different colors.

- After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K -hop neighborhood

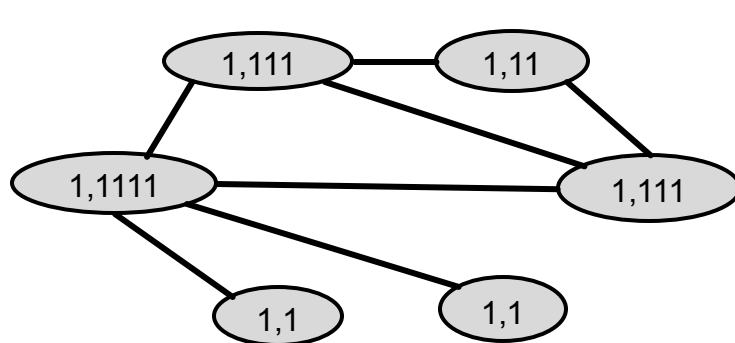
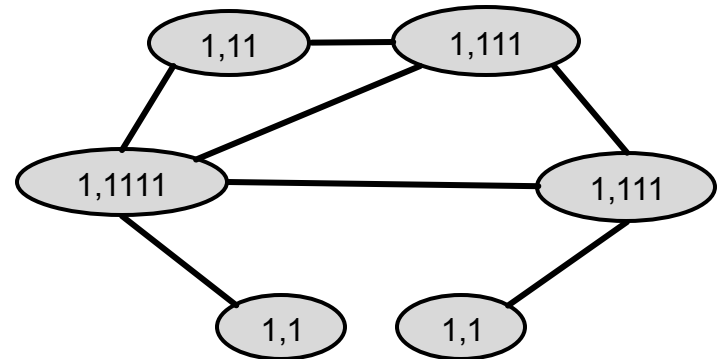
Color Refinement (1)

Example of color refinement given two graphs

- Assign initial colors



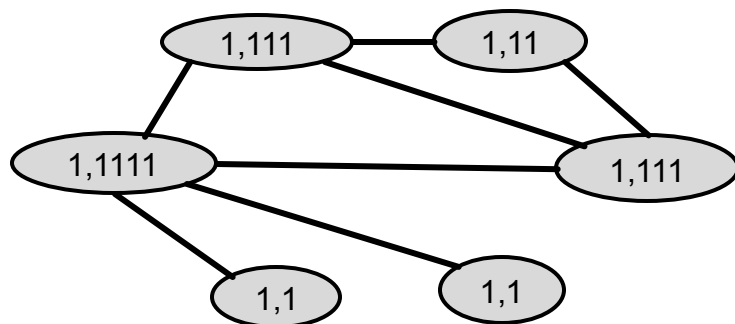
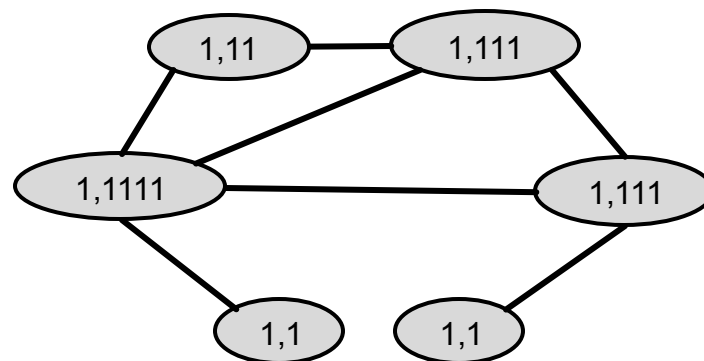
- Aggregate neighboring colors



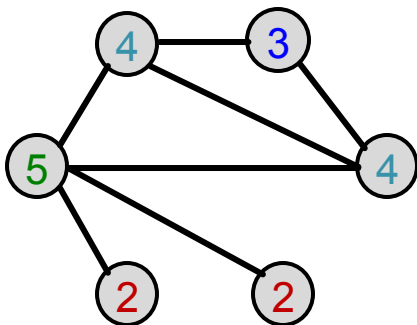
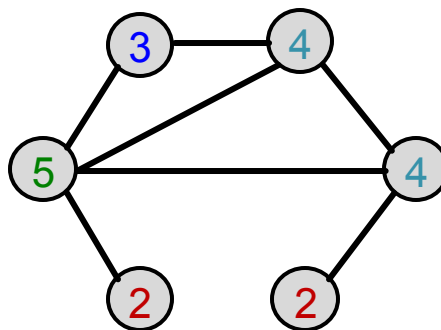
Color Refinement (2)

Example of color refinement given two graphs

- Aggregated colors



- Hash aggregated colors



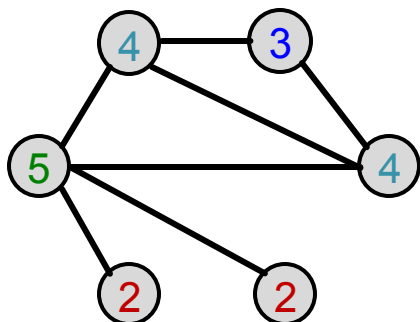
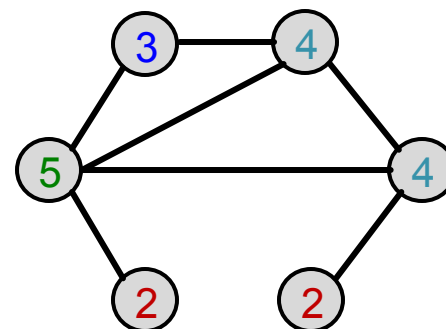
Hash table

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

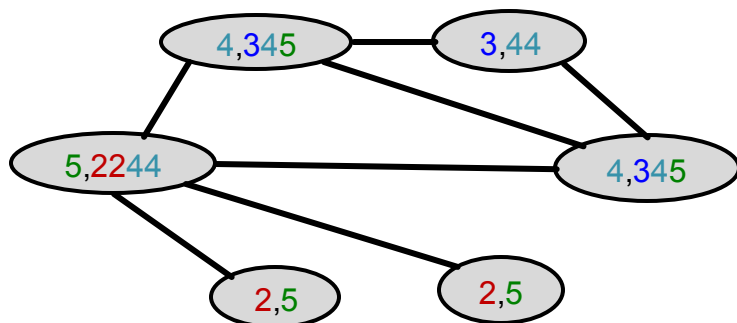
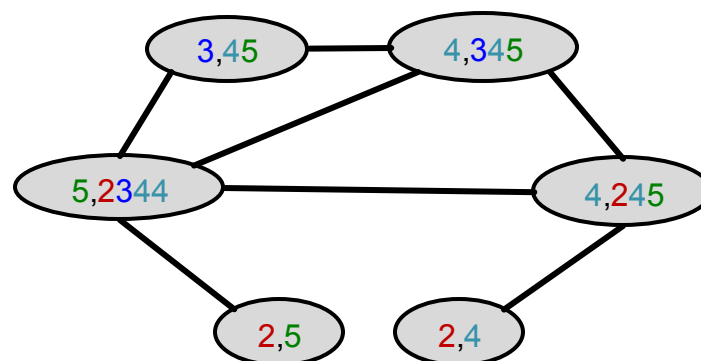
Color Refinement (3)

Example of color refinement given two graphs

- Aggregated colors



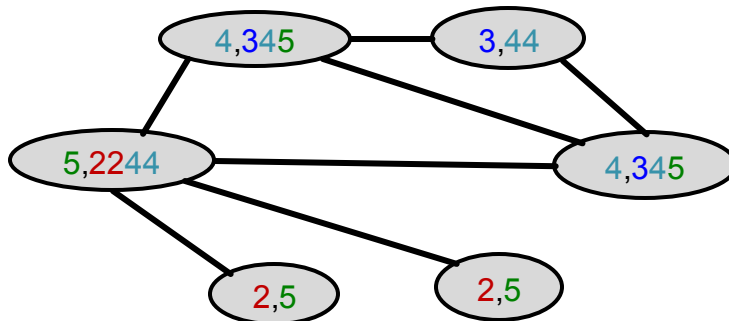
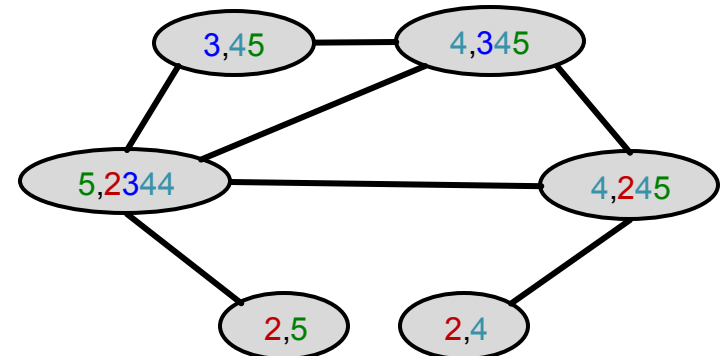
- Hash aggregated colors



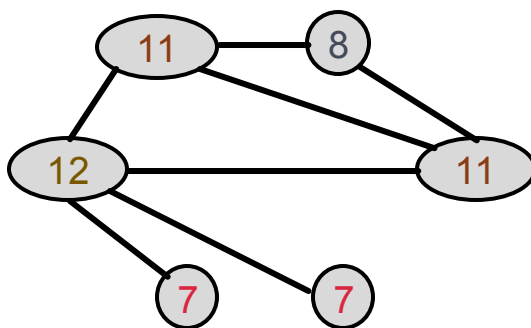
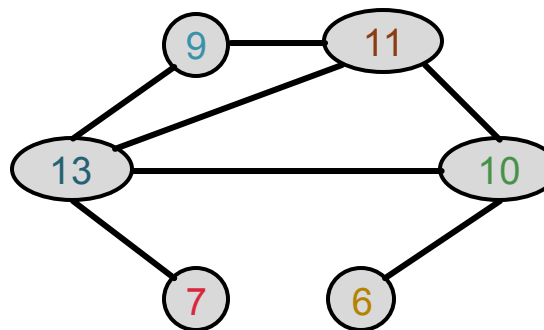
Color Refinement (4)

Example of color refinement given two graphs

■ Aggregated colors



■ Hash aggregated colors

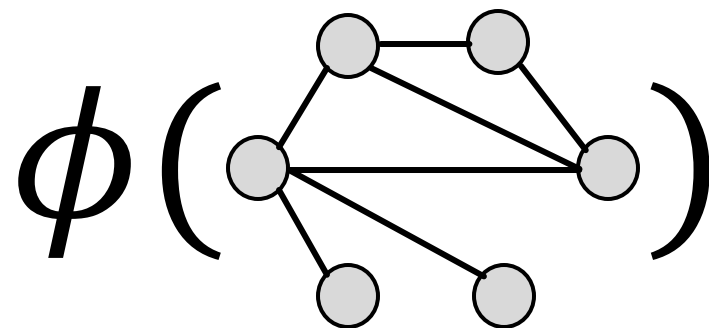


Hash table

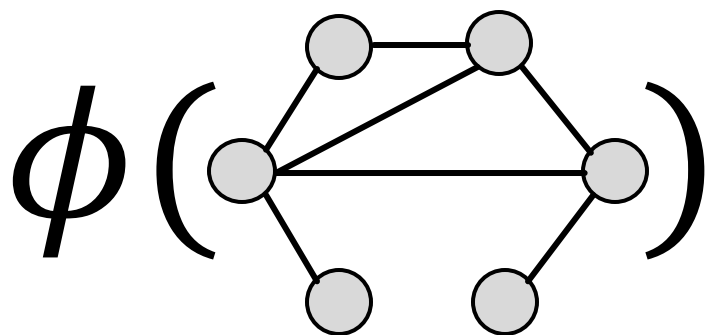
2,4	-->	6
2,5	-->	7
3,44	-->	8
3,45	-->	9
4,245	-->	10
4,345	-->	11
5,2244	-->	12
5,2344	-->	13

Weisfeiler-Lehman Graph Features

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]

Weisfeiler-Lehman Kernel

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

$$\begin{aligned} K(\text{graph}_1, \text{graph}_2) &= \phi(\text{graph}_1)^T \phi(\text{graph}_2) \\ &= 49 \end{aligned}$$

Weisfeiler-Lehman Kernel

- WL kernel is **computationally efficient**
 - The time complexity for color refinement at each step is linear in $\#(\text{edges})$, since it involves aggregating neighboring colors.
- When computing a kernel value, only colors appeared in the two graphs need to be tracked.
 - Thus, $\#(\text{colors})$ is at most the total number of nodes.
- Counting colors takes linear-time w.r.t. $\#(\text{nodes})$.
- In total, time complexity is **linear in $\#(\text{edges})$** .

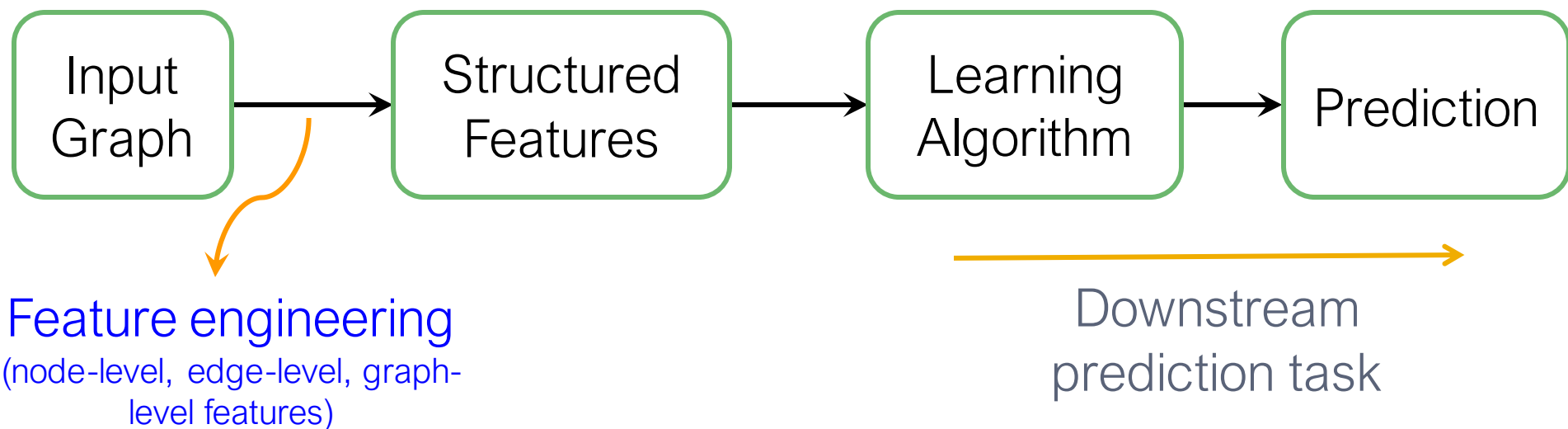
Stanford CS224W: Node Embeddings

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



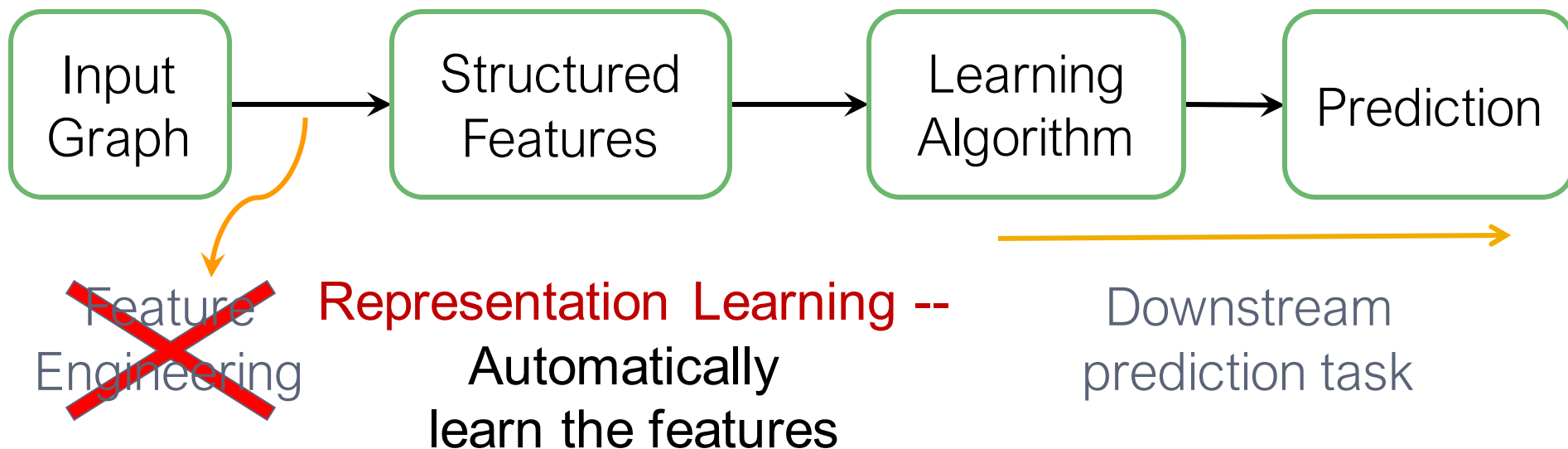
Recap: Traditional ML for Graphs

Given an input graph, extract node, link and graph-level features, learn a model (SVM, neural network, etc.) that maps features to labels.



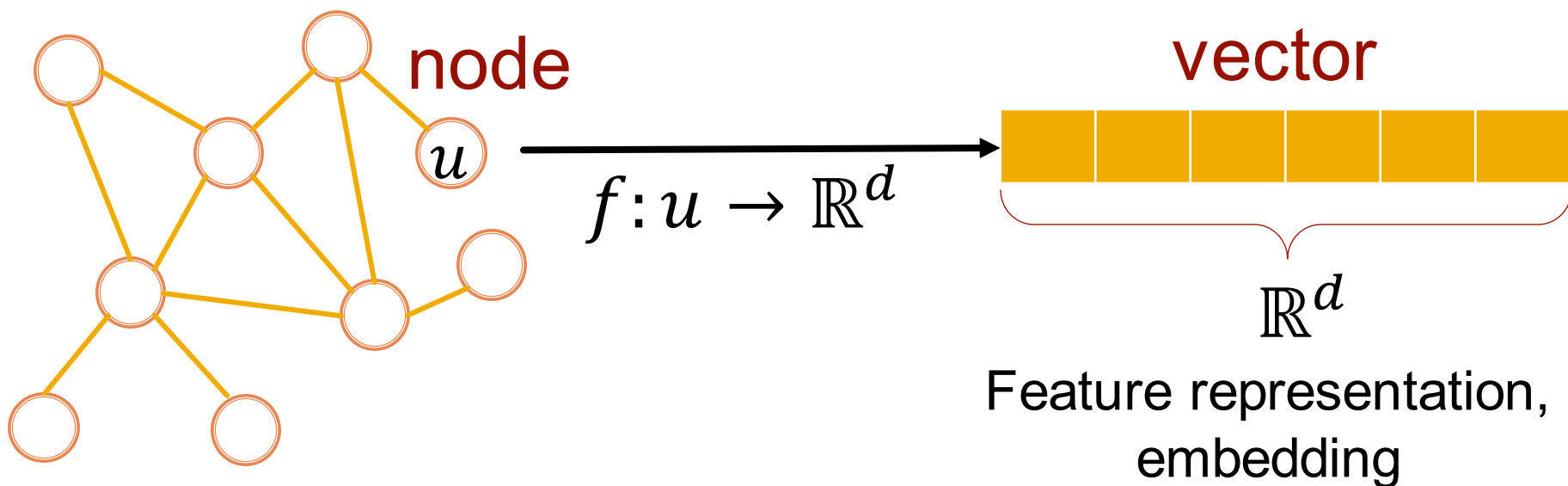
Graph Representation Learning

Graph Representation Learning alleviates the need to do feature engineering **every single time**.



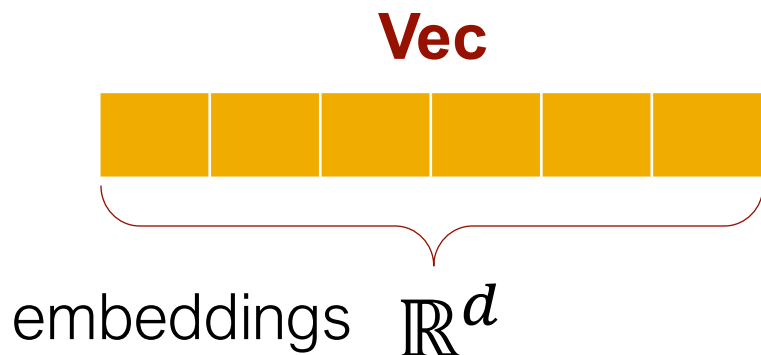
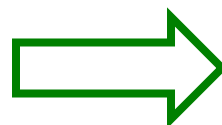
Graph Representation Learning

Goal: Efficient task-independent feature learning for machine learning with graphs!



Why Embedding?

- **Task: Map nodes into an embedding space**
 - Similarity of embeddings between nodes indicates their similarity in the network. For example:
 - Both nodes are close to each other (connected by an edge)
 - Encode network information
 - Potentially used for many downstream predictions

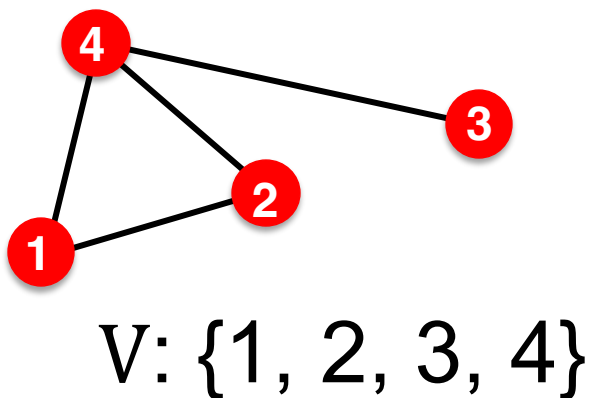


Tasks

- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
-

Setup

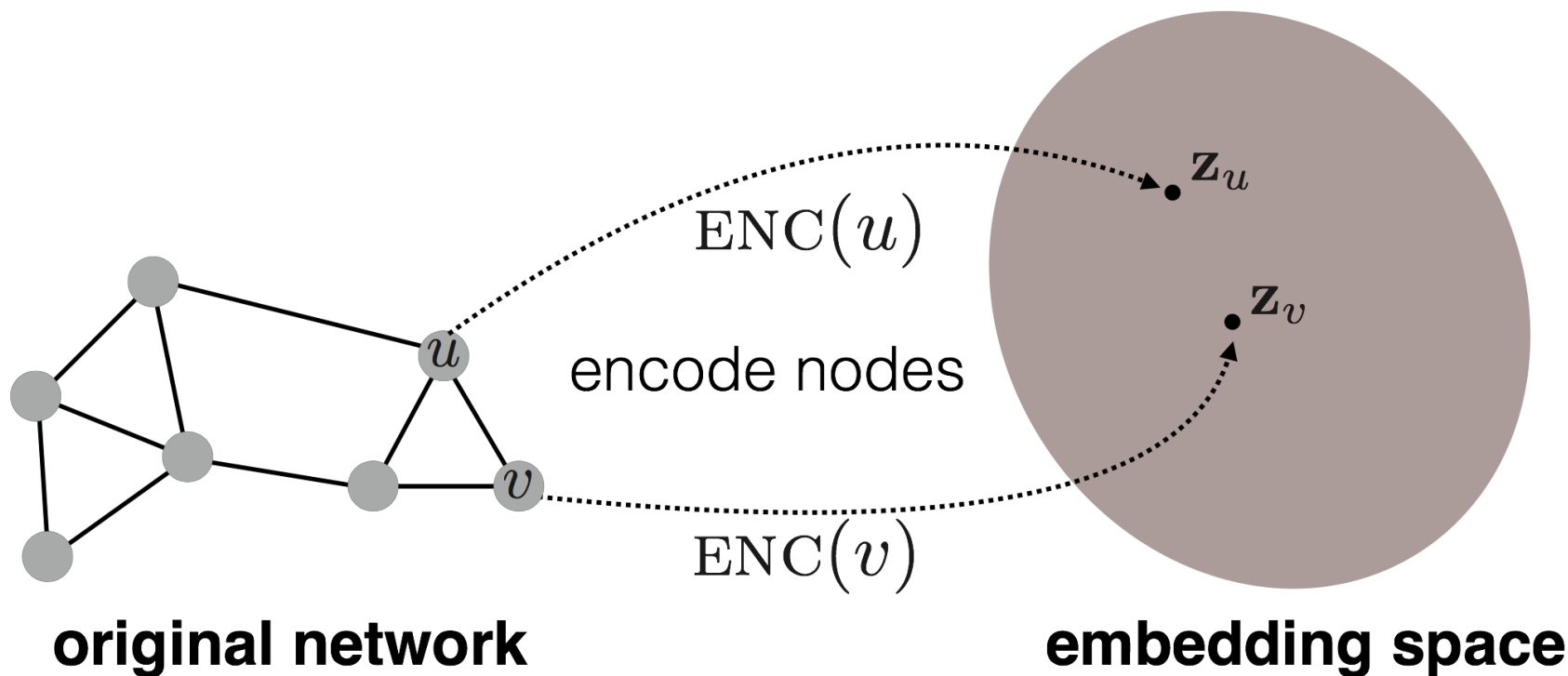
- Assume we have a graph G :
 - V is the vertex set.
 - A is the adjacency matrix (assume binary).
 - **For simplicity: No node features or extra information is used**



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

Embedding Nodes

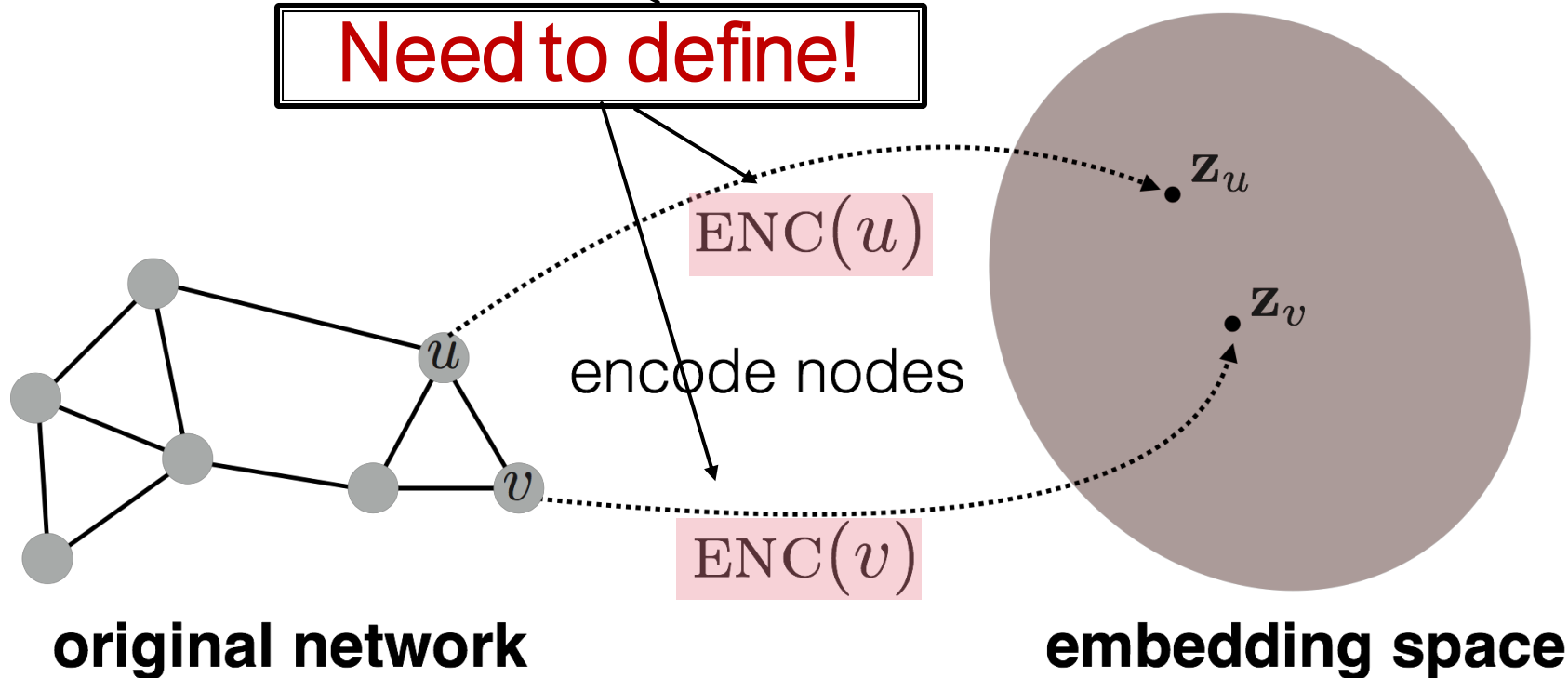
- Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the graph



Embedding Nodes

Goal: $\text{similarity}(u, v)$ $\approx \mathbf{z}_v^T \mathbf{z}_u$
in the original network Similarity of the embedding

Need to define!



Learning Node Embeddings

1. **Encoder** maps from nodes to embeddings
2. **Define a node similarity function** (i.e., a measure of similarity in the original network)
3. **Decoder DEC** maps from embeddings to the similarity score
4. **Optimize the parameters of the encoder so that:**

$$\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$$

in the original network

Similarity of the embedding

$$\text{DEC}(\mathbf{z}_v^T \mathbf{z}_u)$$

Two Key Components

- **Encoder:** maps each node to a low-dimensional vector

$$\text{ENC}(v) = \mathbf{z}_v$$

node in the input graph

d -dimensional embedding

- **Similarity function:** specifies how the relationships in vector space map to the relationships in the original network

$$\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$$

Decoder

Similarity of u and v in the original network

dot product between node embeddings

“Shallow” Encoding

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

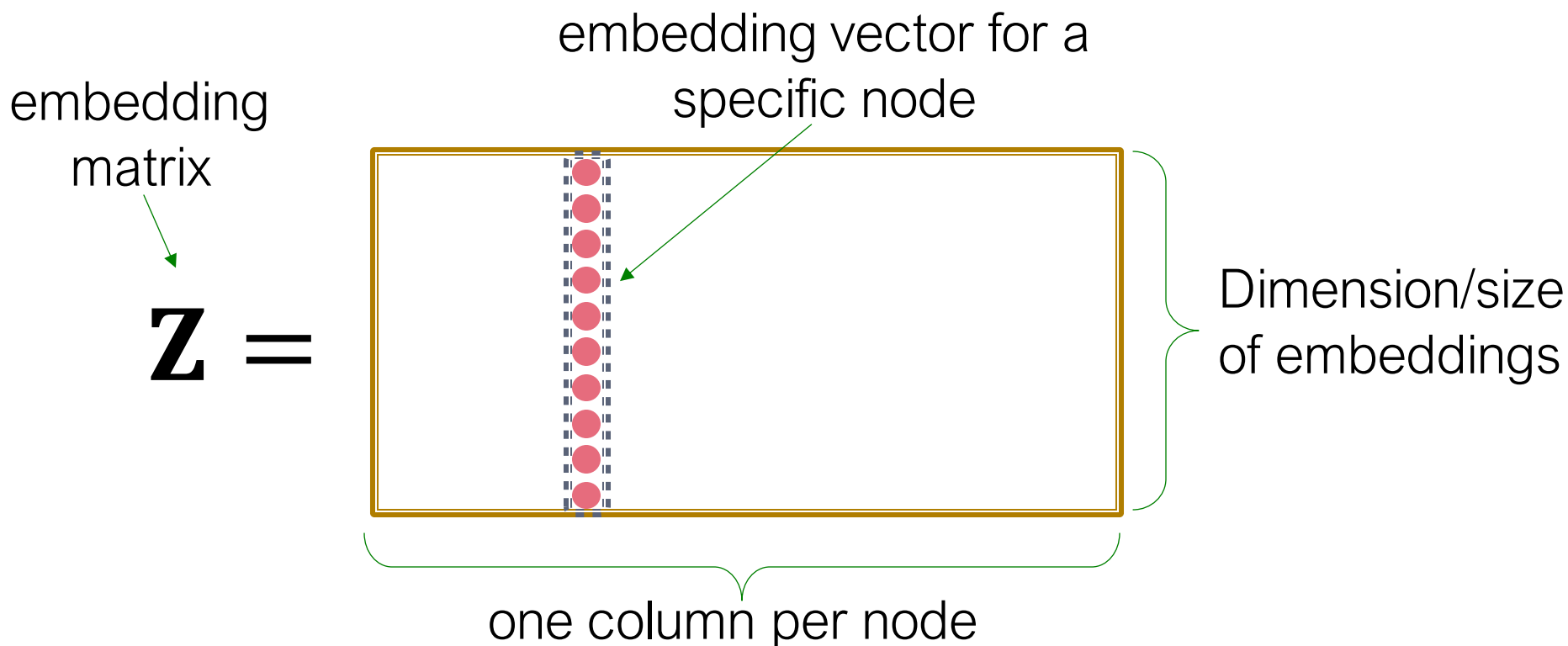
$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

$\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ matrix, each column is a node embedding [what we learn / optimize]

$v \in \mathbb{I}^{|\mathcal{V}|}$ indicator vector, all zeroes except a one in column indicating node v

“Shallow” Encoding

Simplest encoding approach: **encoder is just an embedding-lookup**



“Shallow” Encoding

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

**Each node is assigned a unique
embedding vector**

(i.e., we directly optimize
the embedding of each node)

Many methods: DeepWalk, node2vec

How to Define Node Similarity?

- Key choice of methods is **how they define node similarity**.
- Should two nodes have a similar embedding if they...
 - are linked?
 - share neighbors?
 - have similar “structural roles”?
- We will now learn node similarity definition that uses **random walks**, and how to optimize embeddings for such a similarity measure.

Note on Node Embeddings

- This is **unsupervised/self-supervised** way of learning node embeddings.
 - We are **not** utilizing node labels
 - We are **not** utilizing node features
 - The goal is to directly estimate a set of coordinates (i.e., the embedding) of a node so that some aspect of the network structure (captured by DEC) is preserved.
- These embeddings are **task independent**
 - They are not trained for a specific task but can be used for any task.

Stanford CS224W: Random Walk Approaches for Node Embeddings

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



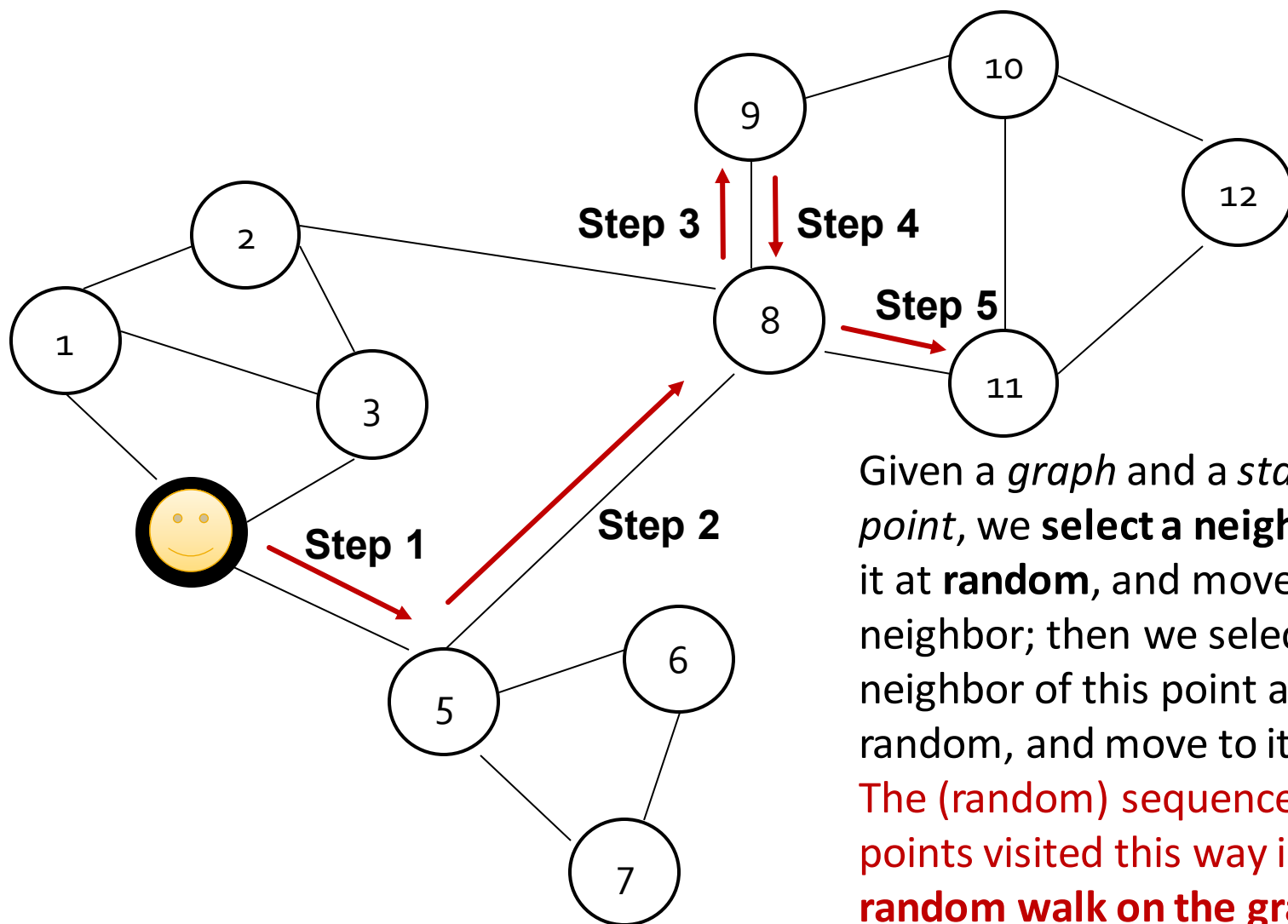
Notation

- **Vector** $\mathbf{z}_u$:
 - The embedding of node u (what we aim to find).
 - **Probability** $P(v | \mathbf{z}_u)$:  Our model prediction based on $\mathbf{z}_u$
 - The **(predicted) probability** of visiting node v on random walks starting from node u .
-

Non-linear functions used to produce predicted **probabilities**

- **Softmax** function:
 - Turns vector of K real values (model predictions) into K probabilities that sum to 1: $\sigma(\mathbf{z})[i] = \frac{e^{\mathbf{z}[i]}}{\sum_{j=1}^K e^{\mathbf{z}[j]}}$
- **Sigmoid** function:
 - S-shaped function that turns real values into the range of $(0, 1)$.
Written as $S(x) = \frac{1}{1+e^{-x}}$.

Random Walk



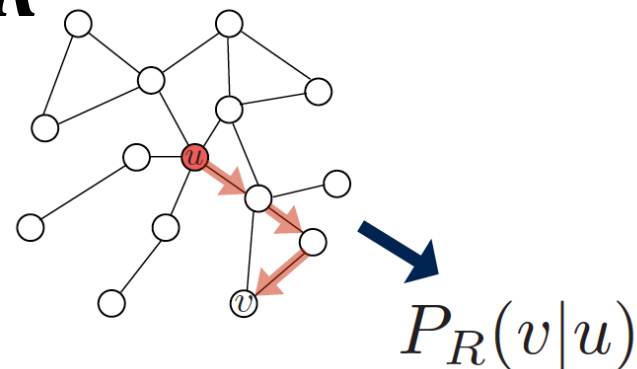
Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The (random) sequence of points visited this way is a **random walk on the graph**.

Random-Walk Embeddings

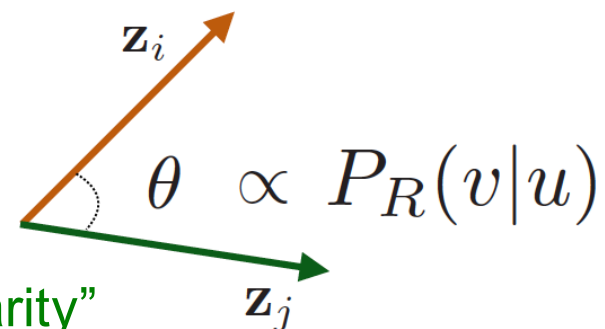
$\mathbf{z}_u^T \mathbf{z}_v \approx$ probability that u
and v co-occur on a
random walk over
the graph

Random-Walk Embeddings

1. Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R



2. Optimize embeddings to encode these random walk statistics:



Similarity in embedding space (Here: dot product = $\cos(\theta)$) encodes random walk “similarity”

Why Random Walks?

1. **Expressivity:** Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information
Idea: if random walk starting from node u visits v with high probability, u and v are similar (high-order multi-hop information)
2. **Efficiency:** Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks

Unsupervised Feature Learning

- **Intuition:** Find embedding of nodes in d -dimensional space that preserves similarity
- **Idea:** Learn node embedding such that **nearby** nodes are close together in the network
- **Given a node u , how do we define nearby nodes?**
 - $N_R(u)$... neighbourhood of u obtained by some **random walk strategy R**

Feature Learning as Optimization

- Given $G = (V, E)$,
- Our goal is to learn a mapping $f: u \rightarrow \mathbb{R}^d$:
 $f(u) = \mathbf{z}_u$

- Log-likelihood objective:

$$\max_f \sum_{u \in V} \log P(N_R(u) | \mathbf{z}_u)$$

- $N_R(u)$ is the neighborhood of node u by strategy R
- Given node u , we want to learn feature representations that are predictive of the nodes in its random walk neighborhood $N_R(u)$.

Random Walk Optimization

1. Run **short fixed-length random walks** starting from each node u in the graph using some random walk strategy R .
2. For each node u collect $N_R(u)$, the multiset* of nodes visited on random walks starting from u .
3. Optimize embeddings according to: **Given node u , predict its neighbors $N_R(u)$.**

$$\max_f \sum_{u \in V} \log P(N_R(u) | \mathbf{z}_u) \quad \Rightarrow \quad \text{Maximum likelihood objective}$$

* $N_R(u)$ can have repeat elements since nodes can be visited multiple times on random walks

Random Walk Optimization

Equivalently,

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

- **Intuition:** Optimize embeddings $\mathbf{z}_u$ to maximize the likelihood of random walk co-occurrences.
- **Parameterize $P(v|\mathbf{z}_u)$ using softmax:**

$$P(v|\mathbf{z}_u) = \frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}$$

Why softmax?

We want node v to be most similar to node u (out of all nodes n).

Intuition: $\sum_i \exp(x_i) \approx \max_i \exp(x_i)$

Random Walk Optimization

Putting it all together:

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} - \log \left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)} \right)$$

sum over all nodes u

sum over nodes v seen on random walks starting from u

predicted probability of u and v co-occurring on random walk

Optimizing random walk embeddings =
Finding embeddings $\mathbf{z}_u$ that minimize $\mathcal{L}$

Random Walk Optimization

But doing this naively is too expensive!

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}\right)$$


Nested sum over nodes gives
 $O(|V|^2)$ complexity!

Random Walk Optimization

But doing this naively is too expensive!

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}\right)$$

The normalization term from the softmax is the culprit... can we approximate it?

Negative Sampling

■ Solution: Negative sampling

$$\log\left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}\right)$$

$$\approx \log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_v)\right) - \sum_{i=1}^k \log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_{n_i})\right), n_i \sim P_V$$

sigmoid function

(makes each term a “probability”
between 0 and 1)

random distribution
over nodes

Why is the approximation valid?

Technically, this is a different objective. But Negative Sampling is a form of Noise Contrastive Estimation (NCE) which approx. maximizes the log probability of softmax.

New formulation corresponds to using a logistic regression (sigmoid func.) to distinguish the target node v from nodes n_i sampled from background distribution P_v .

More at <https://arxiv.org/pdf/1402.3722.pdf>

Instead of normalizing w.r.t. all nodes, just
normalize against k random “**negative samples**” n_i

- Negative sampling allows for quick likelihood calculation.

Negative Sampling

$$\log\left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}\right)$$

random distribution
over nodes



$$\approx \log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_v)\right) - \sum_{i=1}^k \log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_{n_i})\right), n_i \sim P_V$$

- Sample k negative nodes each with prob. proportional to its degree
- Two considerations for k (# negative samples):
 1. Higher k gives more robust estimates
 2. Higher k corresponds to higher bias on negative events

In practice $k = 5-20$.

Can negative sample be any node or only the nodes not on the walk? People often use any nodes (for efficiency). However, the most “correct” way is to use nodes not on the walk.

Stochastic Gradient Descent

- After we obtained the objective function, how do we optimize (minimize) it?

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

- Gradient Descent**: a simple way to minimize $\mathcal{L}$:
 - Initialize $\mathbf{z}_u$ at some randomized value for all nodes u .
 - Iterate until convergence:
 - For all u , compute the derivative $\frac{\partial \mathcal{L}}{\partial \mathbf{z}_u}$.
 - For all u , make a step in reverse direction of derivative: $\mathbf{z}_u \leftarrow \mathbf{z}_u - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{z}_u}$.

η : learning rate



Stochastic Gradient Descent

- **Stochastic Gradient Descent:** Instead of evaluating gradients over all examples, evaluate it for each **individual** training example.
 - Initialize z_u at some randomized value for all nodes u .
 - Iterate until convergence: $\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$
 - Sample a node u , for all v calculate the derivative $\frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$.
 - For all v , update: $z_v \leftarrow z_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$.

Random Walks: Summary

1. Run **short fixed-length** random walks starting from each node on the graph
2. For each node u collect $N_R(u)$, the multiset of nodes visited on random walks starting from u .
3. Optimize embeddings using Stochastic Gradient Descent:

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

We can efficiently approximate this using
negative sampling!

How should we randomly walk?

- So far we have described how to optimize embeddings given a random walk strategy R
- **What strategies should we use to run these random walks?**
 - Simplest idea: **Just run fixed-length, unbiased random walks starting from each node** (i.e., [DeepWalk from Perozzi et al., 2013](#))
 - The issue is that such notion of similarity is too constrained
- **How can we generalize this?**

Reference: Perozzi et al. 2014. [DeepWalk: Online Learning of Social Representations](#). *KDD*.

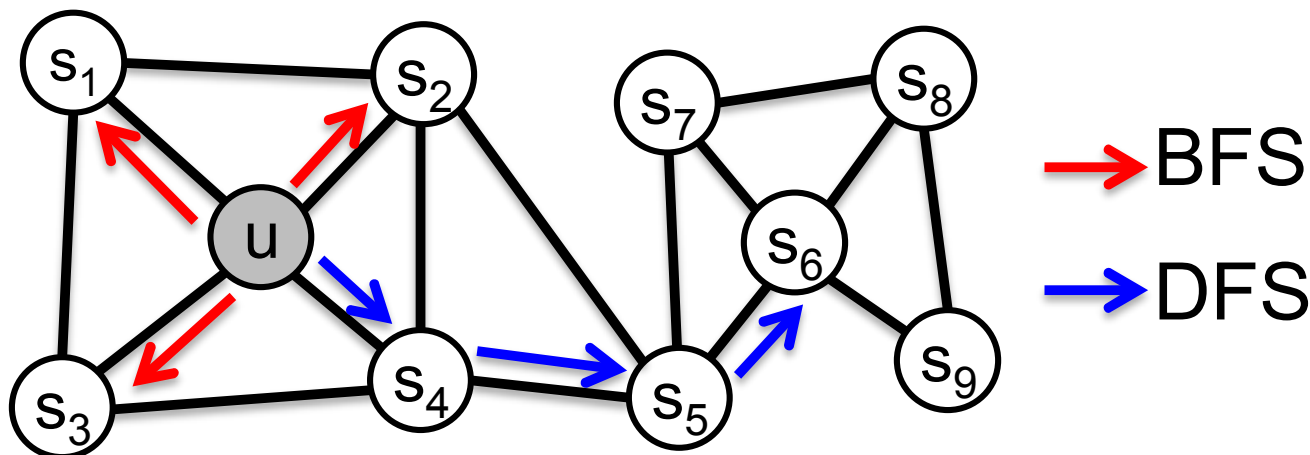
Overview of node2vec

- **Goal:** Embed nodes with similar network neighborhoods close in the feature space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- **Key observation:** Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 2nd order random walk R to generate network neighborhood $N_R(u)$ of node u

Reference: Grover et al. 2016. [node2vec: Scalable Feature Learning for Networks](#). *KDD*.

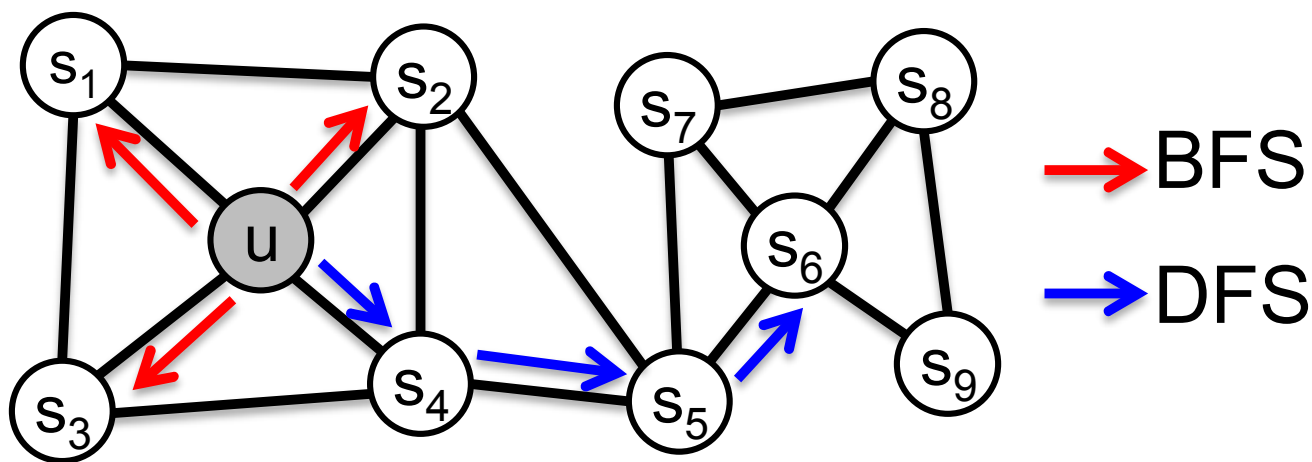
node2vec: Biased Walks

Idea: use flexible, biased random walks that can trade off between **local** and **global** views of the network ([Grover and Leskovec, 2016](#)).



node2vec: Biased Walks

Two classic strategies to define a neighborhood $N_R(u)$ of a given node u :

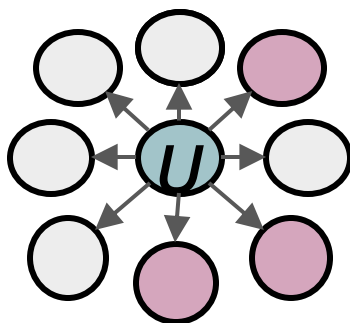


Walk of length 3 ($N_R(u)$ of size 3):

$N_{BFS}(u) = \{s_1, s_2, s_3\}$ Local microscopic view

$N_{DFS}(u) = \{s_4, s_5, s_6\}$ Global macroscopic view

BFS vs. DFS



BFS:

Micro-view of
neighbourhood



DFS:

Macro-view of
neighbourhood

Interpolating BFS and DFS

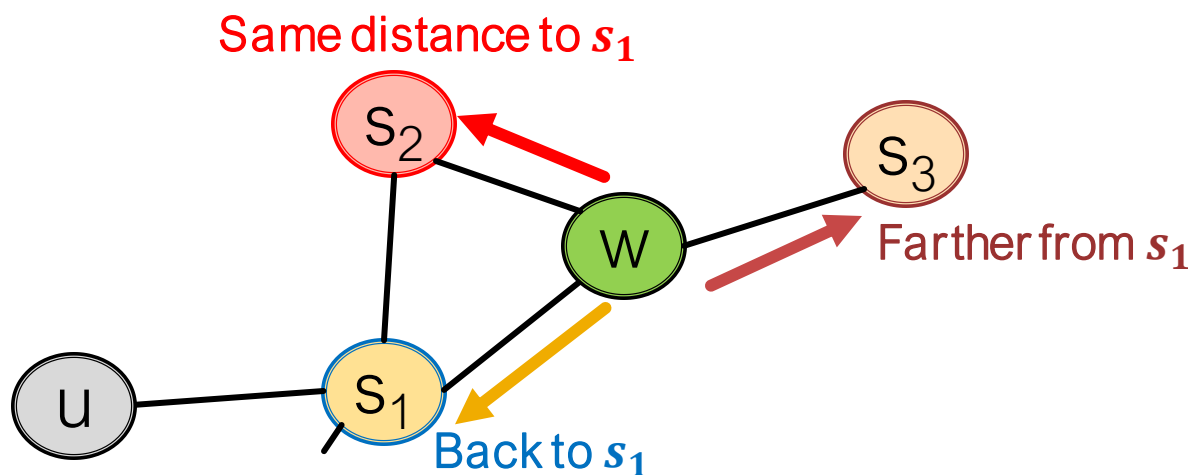
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- Two parameters:
 - **Return parameter p :**
 - Return back to the previous node
 - **In-out parameter q :**
 - Moving outwards (DFS) vs. inwards (BFS)
 - Intuitively, q is the “ratio” of BFS vs. DFS

Biased Random Walks

Biased 2nd-order random walks explore network neighborhoods:

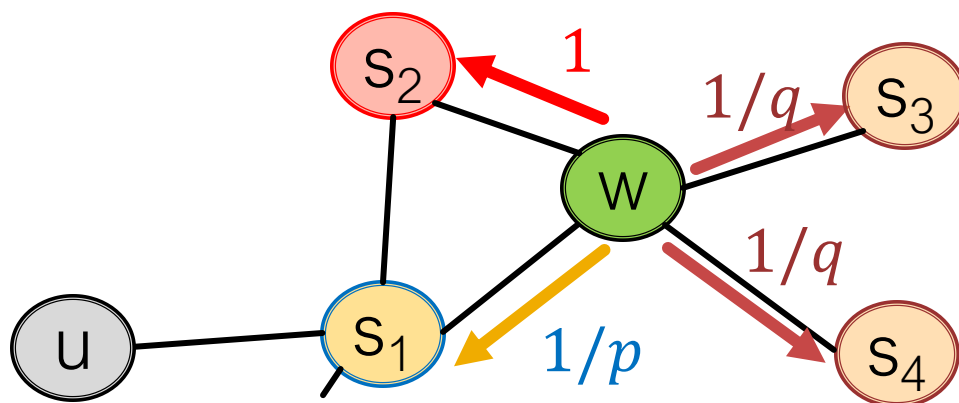
- Rnd. walk just traversed edge (s_1, w) and is now at w
- **Insight:** Neighbors of w can only be:



Idea: Remember where the walk came from

Biased Random Walks

- Walker came over edge (s_1, w) and is at **w**.
Where to go next?



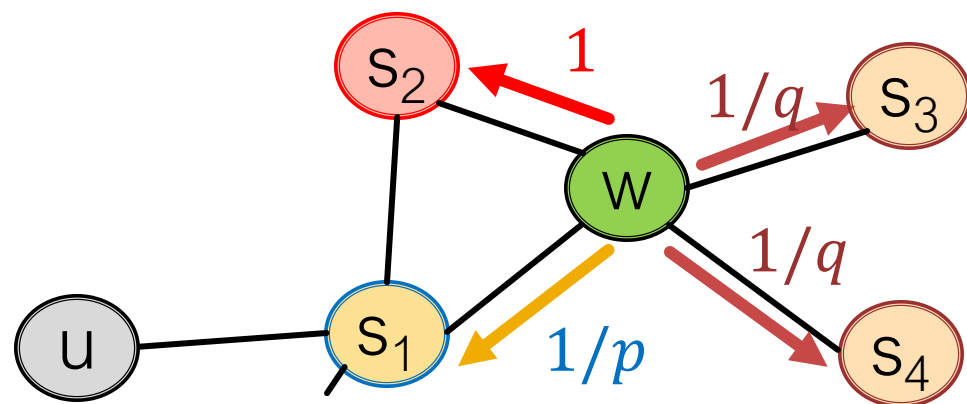
$1/p, 1/q, 1$ are
unnormalized
probabilities

- p, q model transition probabilities
 - p ... return parameter
 - q ... "walk away" parameter

Biased Random Walks

- Walker came over edge (s_1, w) and is at **w**.

Where to go next?



w →

Target t	Prob.	Dist. (s_1, t)
s_1	$1/p$	0
s_2	1	1
s_3	$1/q$	2
s_4	$1/q$	2

Unnormalized
transition prob.
segmented based
on distance from s_1

- BFS-like walk: Low value of p
- DFS-like walk: Low value of q

$N_R(u)$ are the nodes visited by the biased walk

node2vec algorithm

- 1) Compute random walk probabilities
- 2) Simulate r random walks of length l starting from each node u
- 3) Optimize the node2vec objective using Stochastic Gradient Descent
- **Linear-time complexity**
- All 3 steps are **individually parallelizable**

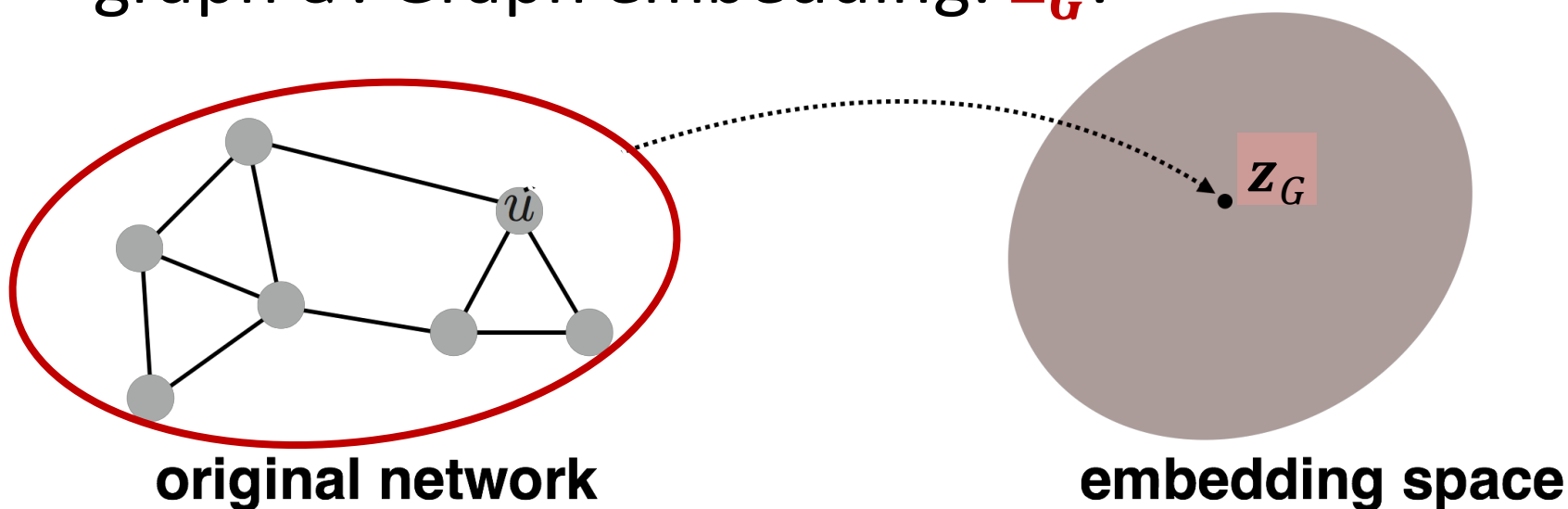
Stanford CS224W: Embedding Entire Graphs

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



Embedding Entire Graphs

- **Goal:** Want to embed a subgraph or an entire graph G . Graph embedding: $\mathbf{z}_G$.



- **Tasks:**
 - Classifying toxic vs. non-toxic molecules
 - Identifying anomalous graphs

Approach 1

Simple (but effective) approach 1:

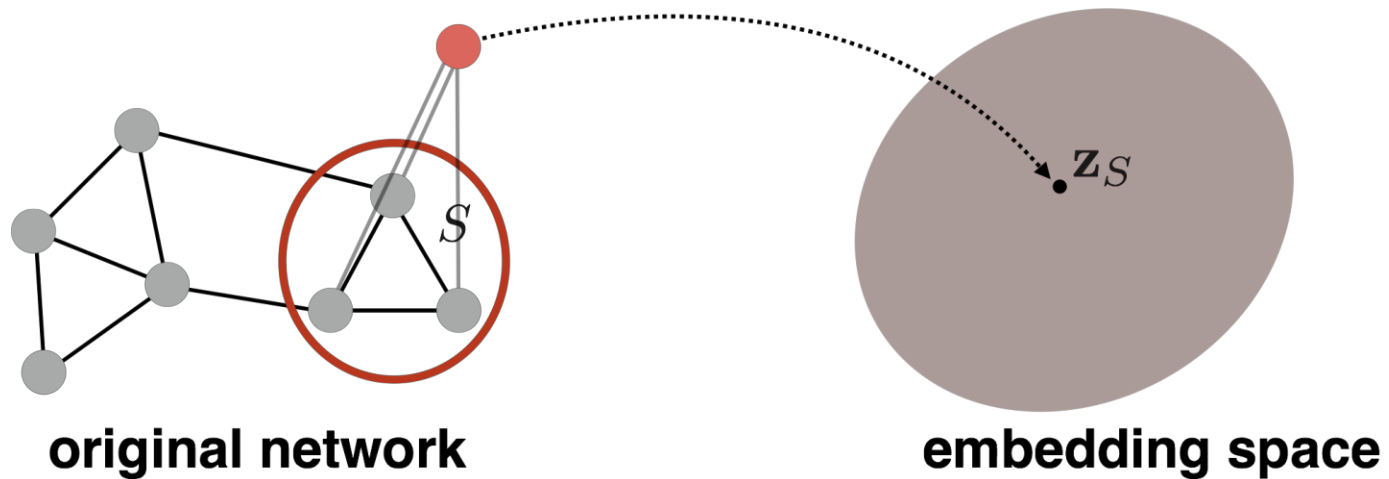
- Run a standard graph embedding technique *on* the (sub)graph G .
- Then just sum (or average) the node embeddings in the (sub)graph G .

$$\mathbf{z}_G = \sum_{v \in G} \mathbf{z}_v$$

- Used by [Duvenaud et al., 2016](#) to classify molecules based on their graph structure

Approach 2

- **Approach 2:** Introduce a “**virtual node**” to represent the (sub)graph and run a standard graph embedding technique



- Proposed by [Li et al., 2016](#) as a general technique for subgraph embedding

Stanford CS224W: Graph as Matrix: PageRank, Random Walks and Embeddings

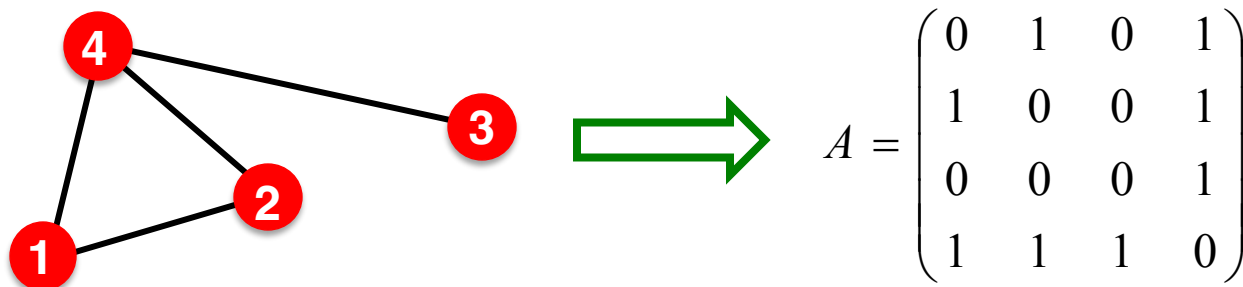
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
<http://cs224w.stanford.edu>



Graph as Matrix

In this lecture, we investigate graph analysis and learning from a matrix perspective.

- Treating a graph as a matrix allows us to:
 - Determine node importance via **random walk** (PageRank)
 - Obtain node embeddings via **matrix factorization (MF)**
 - View other **node embeddings** (e.g. Node2Vec) as MF
- **Random walk, matrix factorization and node embeddings are closely related!**



Example: The Web as a Graph

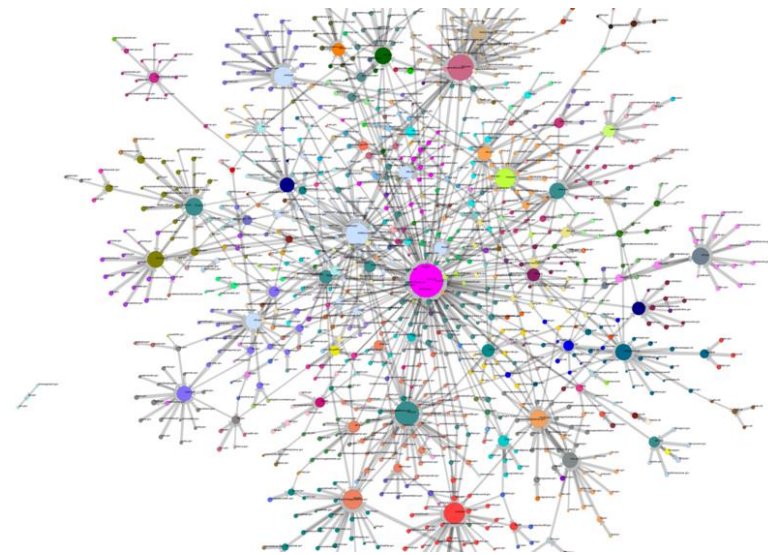
Q: What does the Web “look like” at a global level?

- **Web as a graph:**

- Nodes = web pages
- Edges = hyperlinks

- **Side issue: What is a node?**

- Dynamic pages created on the fly
- “dark matter” – inaccessible database generated pages



The Web as a Graph

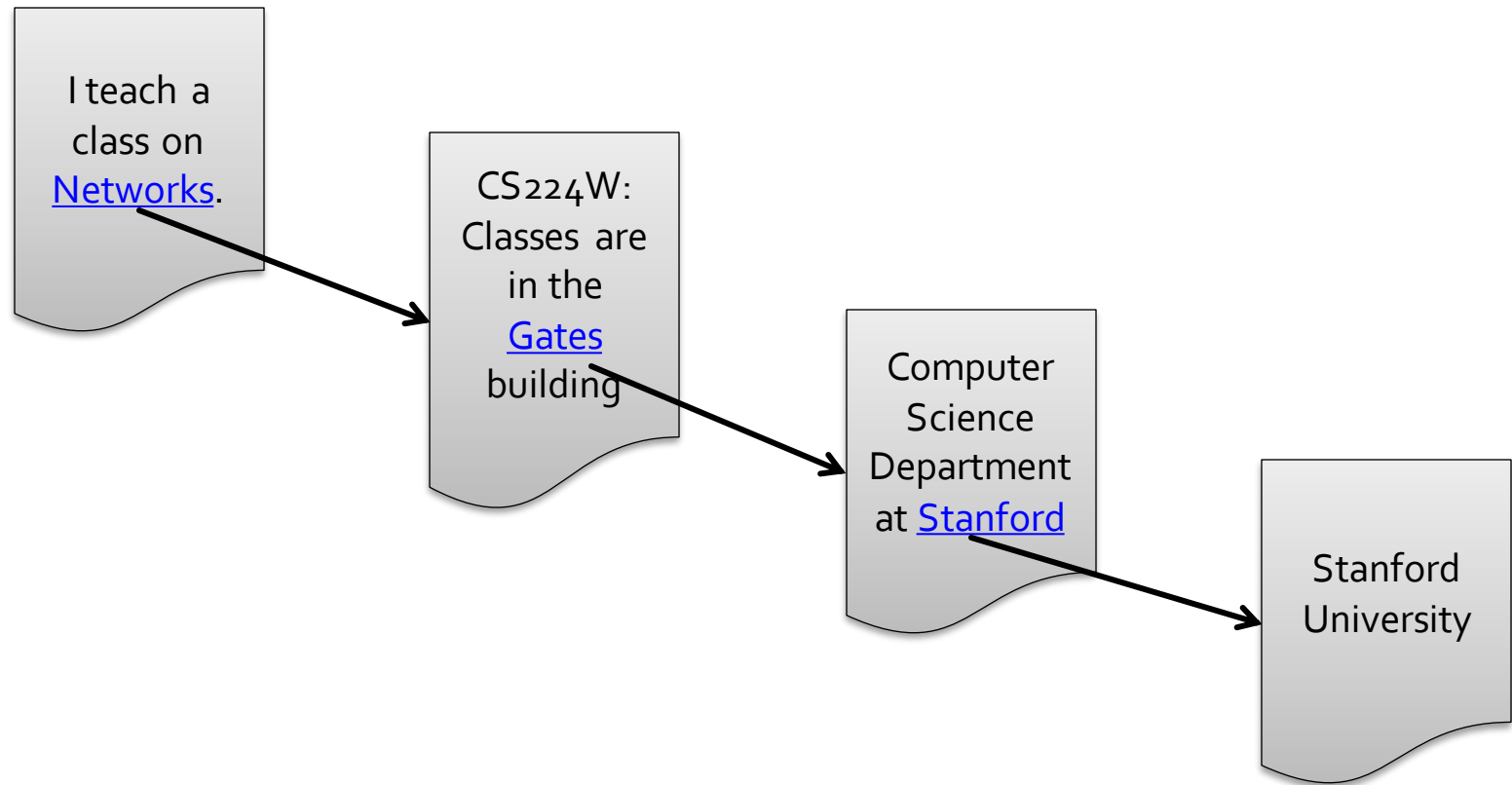
I teach a
class on
Networks.

CS224W:
Classes are
in the
Gates
building

Computer
Science
Department
at Stanford

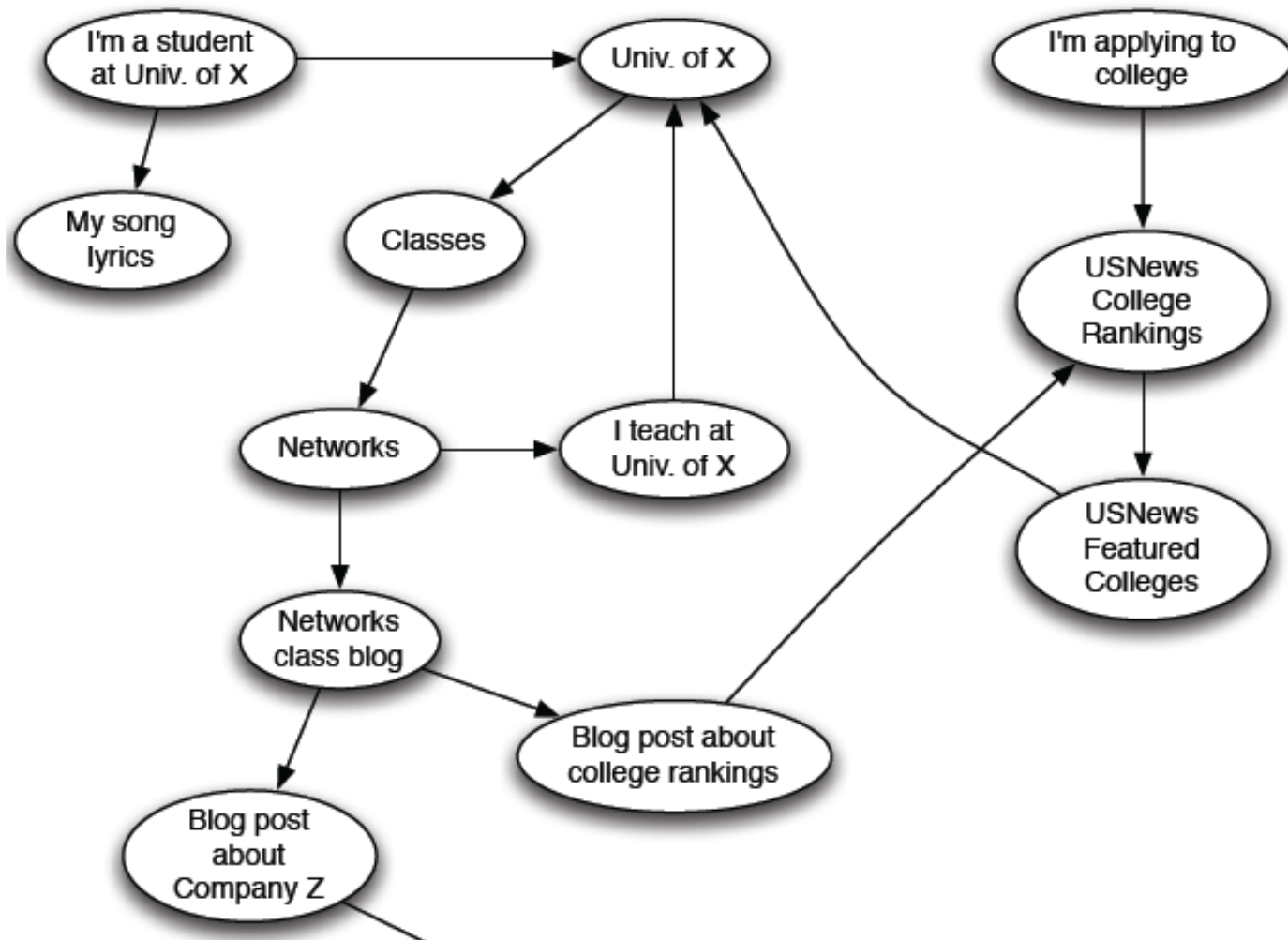
Stanford
University

The Web as a Graph



- In early days of the Web links were **navigational**
- Today many links are **transactional** (used not to navigate from page to page, but to post, comment, like, buy, ...)

The Web as a Directed Graph

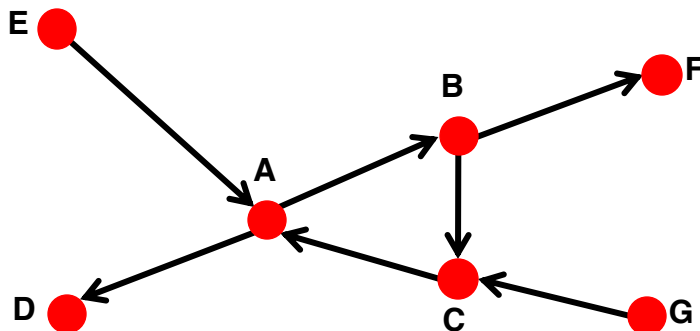


What Does the Web Look Like?

- How is the Web linked?
- What is the “map” of the Web?

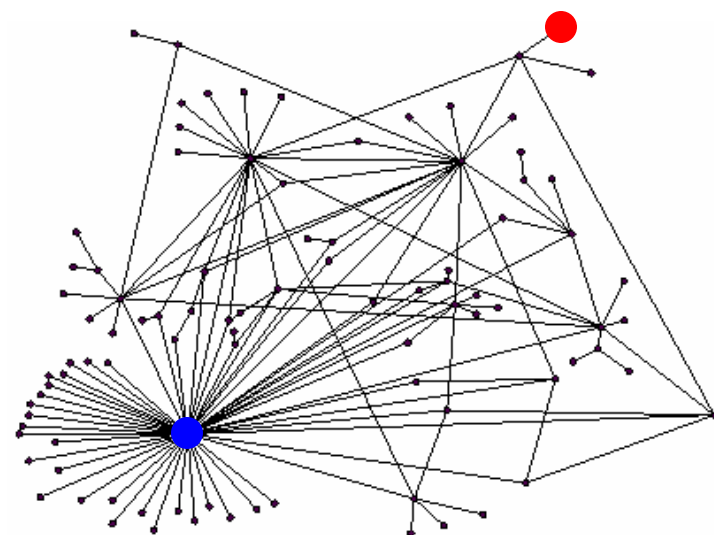
Web as a directed graph [Broder et al. 2000]:

- Given node v , what nodes can v reach?
- What other nodes can reach v ?



Ranking Nodes on the Graph

- All web pages are not equally “important”
thispersondoesnotexist.com vs. www.stanford.edu
- There is large diversity in the web-graph node connectivity.
- So, let's rank the pages using the web graph link structure!



Link Analysis Algorithms

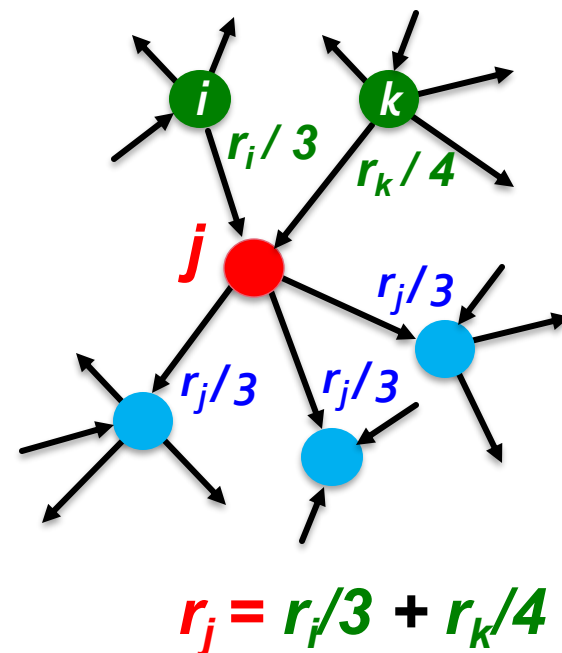
- We will cover the following **Link Analysis approaches** to compute the **importance** of nodes in a graph:
 - PageRank
 - Personalized PageRank (PPR)
 - Random Walk with Restarts

Links as Votes

- **Idea: Links as votes**
 - Page is more important if it has more links
 - In-coming links? Out-going links?
- **Think of in-links as votes:**
 - www.stanford.edu has 23,400 in-links
 - thispersondoesnotexist.com has 1 in-link
- **Are all in-links equal?**
 - Links from important pages count more
 - Recursive question!

PageRank: The “Flow” Model

- A “vote” from an important page is worth more:
 - Each link’s vote is proportional to the **importance** of its source page
 - If page i with importance r_i has d_i out-links, each link gets r_i / d_i votes
 - Page j ’s own importance r_j is the sum of the votes on its in-links



PageRank: The “Flow” Model

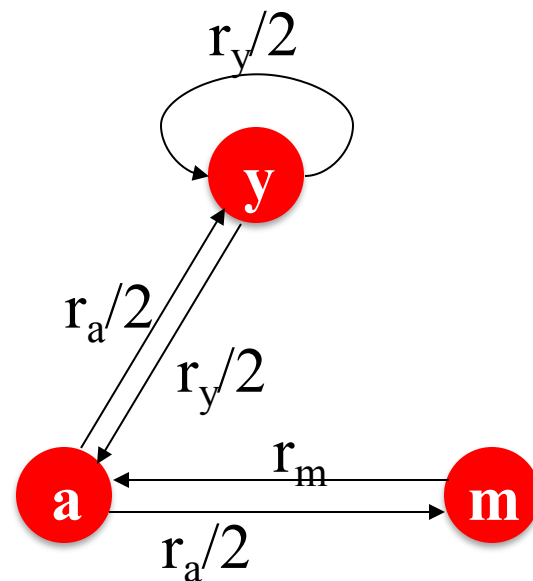
- A page is important if it is pointed to by other important pages
- Define “rank” r_j for node j

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

d_i ... out-degree of node i

You might wonder: Let’s just use Gaussian elimination to solve this system of linear equations. Bad idea!

The web in 1839



“Flow” equations:

$$r_y = r_y/2 + r_a/2$$

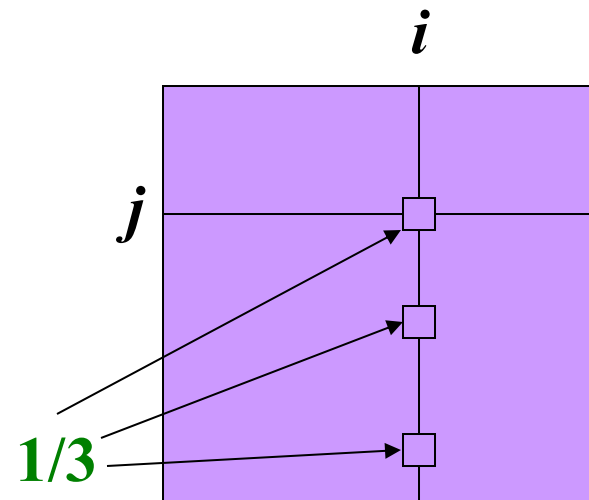
$$r_a = r_y/2 + r_m$$

$$r_m = r_a/2$$

PageRank: Matrix Formulation

- **Stochastic adjacency matrix M**

- d_i is the outdegree of node i
- If $i \rightarrow j$, then $M_{ji} = \frac{1}{d_i}$
 - M is a **column stochastic matrix**
 - Columns sum to 1



- **Rank vector r :** An entry per page

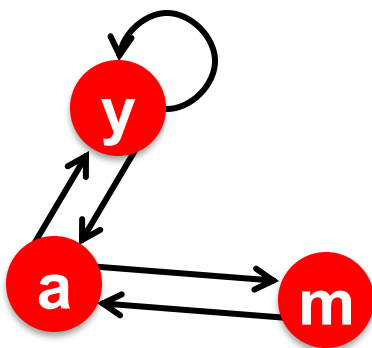
- r_i is the importance score of page i
- $\sum_i r_i = 1$

- **The flow equations can be written**

$$r = M \cdot r$$

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

Example: Flow Equations & M



	r_y	r_a	r_m
r_y	$1/2$	$1/2$	0
r_a	$1/2$	0	1
r_m	0	$1/2$	0

$$r_y = r_y / 2 + r_a / 2$$

$$r_a = r_y / 2 + r_m$$

$$r_m = r_a / 2$$

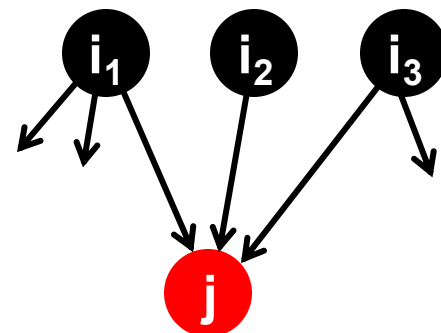
$$\begin{matrix} r_y \\ r_a \\ r_m \end{matrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{matrix} r_y \\ r_a \\ r_m \end{matrix}$$

$\mathbf{r} \qquad \qquad \mathbf{M} \qquad \qquad \mathbf{r}$

Connection to Random Walk

- **Imagine a random web surfer:**

- At any time t , surfer is on some page i
- At time $t + 1$, the surfer follows an out-link from i uniformly at random
- Ends up on some page j linked from i
- Process repeats indefinitely



$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

- **Let:**

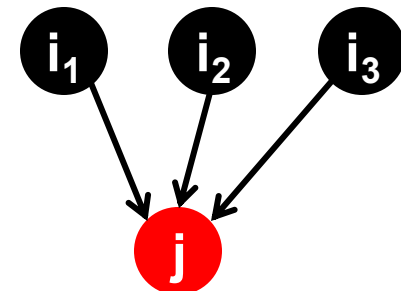
- $\mathbf{p}(t)$... vector whose i^{th} coordinate is the prob. that the surfer is at page i at time t
- So, $\mathbf{p}(t)$ is a probability distribution over pages

The Stationary Distribution

- Where is the surfer at time $t+1$?

- Follow a link uniformly at random

$$\mathbf{p}(t+1) = \mathbf{M} \cdot \mathbf{p}(t)$$



$$\mathbf{p}(t+1) = \mathbf{M} \cdot \mathbf{p}(t)$$

- Suppose the random walk reaches a state

$$\mathbf{p}(t+1) = \mathbf{M} \cdot \mathbf{p}(t) = \mathbf{p}(t)$$

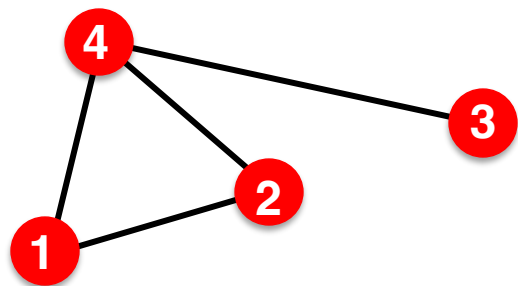
then $\mathbf{p}(t)$ is **stationary distribution** of a random walk

- Our original rank vector $\mathbf{r}$ satisfies $\mathbf{r} = \mathbf{M} \cdot \mathbf{r}$

- So, $\mathbf{r}$ is a stationary distribution for the random walk

Recall Eigenvector of A Matrix

- Recall from lecture 2 (eigenvector centrality), let $A \in \{0, 1\}^{n \times n}$ be an adj. matrix of undir. graph:



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

- Eigenvector of adjacency matrix:
vectors satisfying $\lambda \mathbf{c} = A \mathbf{c}$
- $\mathbf{c}$: eigenvector; λ : eigenvalue
- Note:
 - This is the definition of eigenvector centrality (for undirected graphs).
 - PageRank is defined for directed graphs

Eigenvector Formulation

- The flow equation:

$$1 \cdot \mathbf{r} = \mathbf{M} \cdot \mathbf{r}$$

$$\begin{array}{|c|} \hline \mathbf{r}_y \\ \hline \mathbf{r}_a \\ \hline \mathbf{r}_m \\ \hline \end{array} = \begin{array}{|ccc|} \hline \frac{1}{2} & \frac{1}{2} & 0 \\ \hline \frac{1}{2} & 0 & 1 \\ \hline 0 & \frac{1}{2} & 0 \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{r}_y \\ \hline \mathbf{r}_a \\ \hline \mathbf{r}_m \\ \hline \end{array}$$

$\mathbf{r} \qquad \mathbf{M} \qquad \mathbf{r}$

- So the rank vector $\mathbf{r}$ is an **eigenvector** of the stochastic adj. matrix $\mathbf{M}$ (with eigenvalue 1)
 - Starting from any vector $\mathbf{u}$, the limit $\mathbf{M}(\mathbf{M}(\dots \mathbf{M}(\mathbf{M} \mathbf{u})))$ is the **long-term distribution** of the surfers.
 - **PageRank** = Limiting distribution = **principal eigenvector** of $\mathbf{M}$
 - **Note**: If $\mathbf{r}$ is the limit of the product $\mathbf{M}\mathbf{M} \dots \mathbf{M}\mathbf{u}$, then $\mathbf{r}$ satisfies the **flow equation** $1 \cdot \mathbf{r} = \mathbf{M}\mathbf{r}$
 - So $\mathbf{r}$ is the **principal eigenvector** of $\mathbf{M}$ with eigenvalue 1
- **We can now efficiently solve for $\mathbf{r}$!**
 - The method is called **Power iteration**

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 ($\sum_i |r_i^{t+1} - r_i^t| < \epsilon$)
 - Calculate the page rank of each node

$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

d_i out-degree of node i

Power Iteration Method

- Given a web graph with N nodes, where the nodes are pages and edges are hyperlinks
- **Power iteration:** a simple iterative scheme

- Initialize: $\mathbf{r}^{(0)} = [1/N, \dots, 1/N]^T$

- Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$

- Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

d_i out-degree of node i

$|\mathbf{x}|_1 = \sum_1^N |x_i|$ is the **L1** norm

Can use any other vector norm, e.g., Euclidean

About 50 iterations is sufficient to estimate the limiting solution.

PageRank: How to solve?

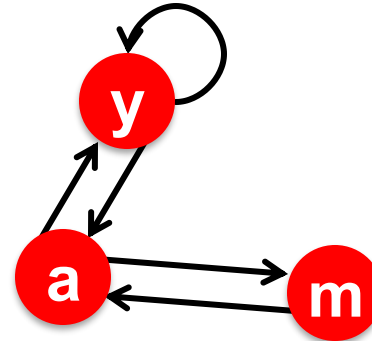
■ Power Iteration:

- Set $r_j \leftarrow 1/N$
- 1: $r'_j \leftarrow \sum_{i \rightarrow j} \frac{r_i}{d_i}$
- 2: If $|r - r'| > \varepsilon$:
 - $r \leftarrow r'$
- 3: go to 1

■ Example:

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

Iteration 0, 1, 2, ...



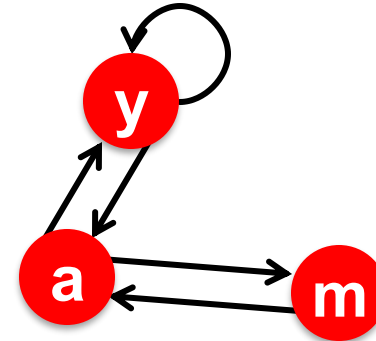
	y	a	m
y	1/2	1/2	0
a	1/2	0	1
m	0	1/2	0

$$\begin{aligned} r_y &= r_y/2 + r_a/2 \\ r_a &= r_y/2 + r_m \\ r_m &= r_a/2 \end{aligned}$$

PageRank: How to solve?

■ Power Iteration:

- Set $r_j \leftarrow 1/N$
- 1: $r'_j \leftarrow \sum_{i \rightarrow j} \frac{r_i}{d_i}$
- 2: If $|r - r'| > \varepsilon$:
 - $r \leftarrow r'$
- 3: go to 1



	y	a	m
y	1/2	1/2	0
a	1/2	0	1
m	0	1/2	0

$$\begin{aligned}
 r_y &= r_y/2 + r_a/2 \\
 r_a &= r_y/2 + r_m \\
 r_m &= r_a/2
 \end{aligned}$$

■ Example:

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \quad \begin{bmatrix} 1/3 \\ 3/6 \\ 1/6 \end{bmatrix} \quad \begin{bmatrix} 5/12 \\ 1/3 \\ 3/12 \end{bmatrix} \quad \begin{bmatrix} 9/24 \\ 11/24 \\ 1/6 \end{bmatrix} \quad \dots \quad \begin{bmatrix} 6/15 \\ 6/15 \\ 3/15 \end{bmatrix}$$

Iteration 0, 1, 2, ...

PageRank: Three Questions

$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i} \quad \text{or equivalently} \quad r = Mr$$

- Does this converge?
- Does it converge to what we want?
- Are results reasonable?

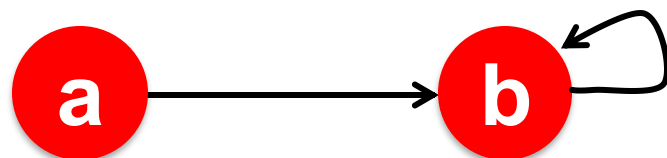
PageRank: Problems

Two problems:

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

Does this converge?

- The “Spider trap” problem:



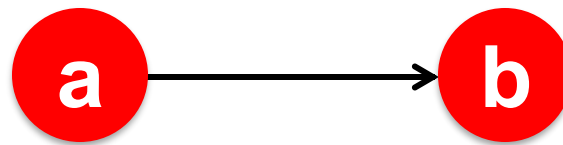
$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

- Example:

Iteration:		0,	1,	2,	3...
r_a	=	1	0	0	0
r_b		0	1	1	1

Does it converge to what we want?

- The “Dead end” problem:



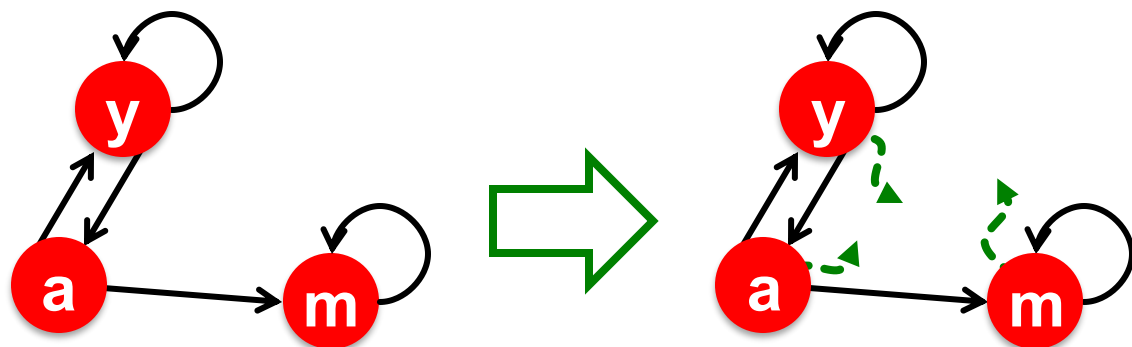
$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

- Example:

Iteration:		0,	1,	2,	3...
r_a	=	1	0	0	0
r_b		0	1	0	0

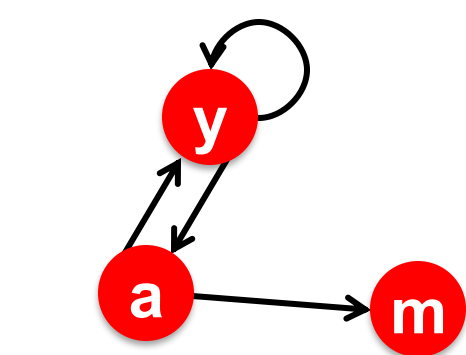
Solution to Spider Traps

- **Solution for spider traps:** At each time step, the random surfer has two options
 - With prob. β , follow a link at random
 - With prob. $1-\beta$, jump to a random page
 - Common values for β are in the range 0.8 to 0.9
- **Surfer will teleport out of spider trap within a few time steps**

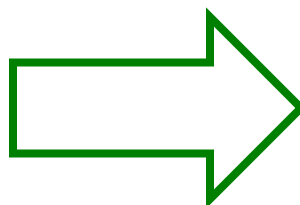
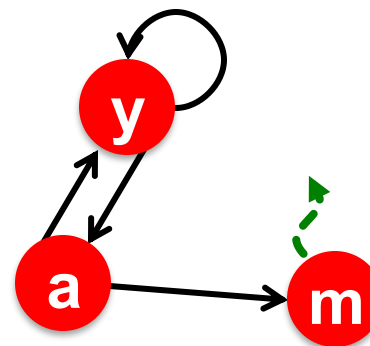


Solution to Dead Ends

- **Teleports:** Follow random teleport links with total probability **1.0** from dead-ends
 - Adjust matrix accordingly



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	0
a	$\frac{1}{2}$	0	0
m	0	$\frac{1}{2}$	0



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$
a	$\frac{1}{2}$	0	$\frac{1}{3}$
m	0	$\frac{1}{2}$	$\frac{1}{3}$

Why Teleports Solve the Problem?

Why are dead-ends and spider traps a problem and **why do teleports solve the problem?**

- **Spider-traps** are not a problem, but with traps PageRank scores are **not** what we want
 - **Solution:** Never get stuck in a spider trap by teleporting out of it in a finite number of steps
- **Dead-ends** are a problem
 - The matrix is not column stochastic so our initial assumptions are not met
 - **Solution:** Make matrix column stochastic by always teleporting when there is nowhere else to go

Solution: Random Teleports

- Google's solution that does it all:

At each step, random surfer has two options:

- With probability β , follow a link at random
- With probability $1-\beta$, jump to some random page

- **PageRank equation** [Brin-Page, 98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

d_i ... out-degree of node i

This formulation assumes that M has no dead ends. We can either preprocess matrix M to remove all dead ends or explicitly follow random teleport links with probability 1.0 from dead-ends.

The Google Matrix

- **PageRank equation** [Brin-Page, '98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

- **The Google Matrix G :**

$[1/N]_{N \times N} \dots N$ by N matrix
where all entries are $1/N$

$$G = \beta M + (1 - \beta) \left[\frac{1}{N} \right]_{N \times N}$$

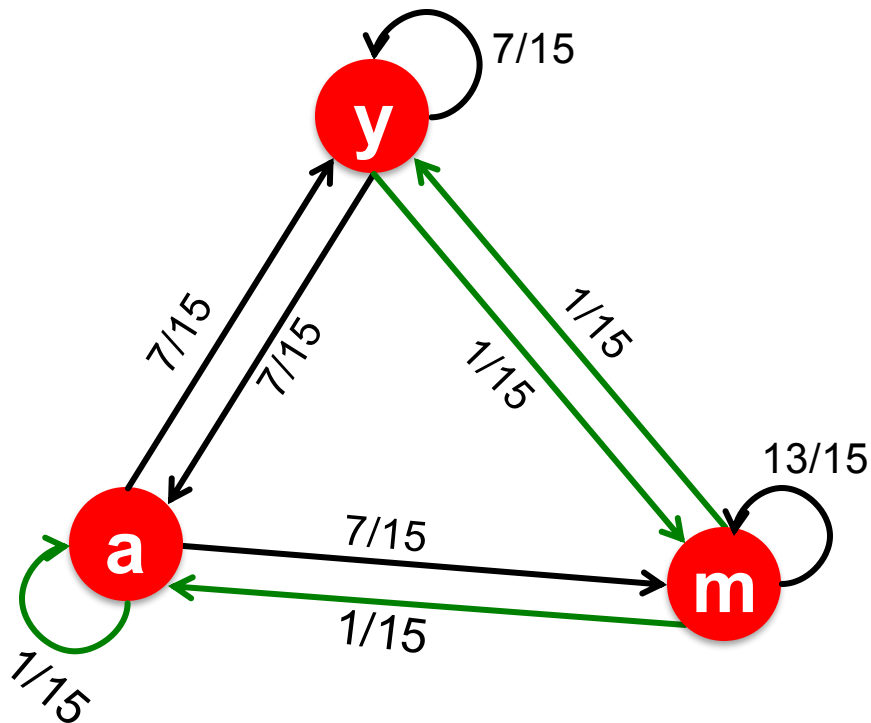
- **We have a recursive problem: $\mathbf{r} = G \cdot \mathbf{r}$**

And the Power method still works!

- **What is β ?**

- In practice $\beta = 0.8, 0.9$ (make 5 steps on avg., jump)

Random Teleports ($\beta = 0.8$)



$$0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} + 0.2 \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

G

y	7/15	7/15	1/15
a	7/15	1/15	1/15
m	1/15	7/15	13/15

y	=	1/3	0.33	0.24	0.26	7/33
a		1/3	0.20	0.20	0.18	5/33
m		1/3	0.46	0.52	0.56	21/33

PageRank Example

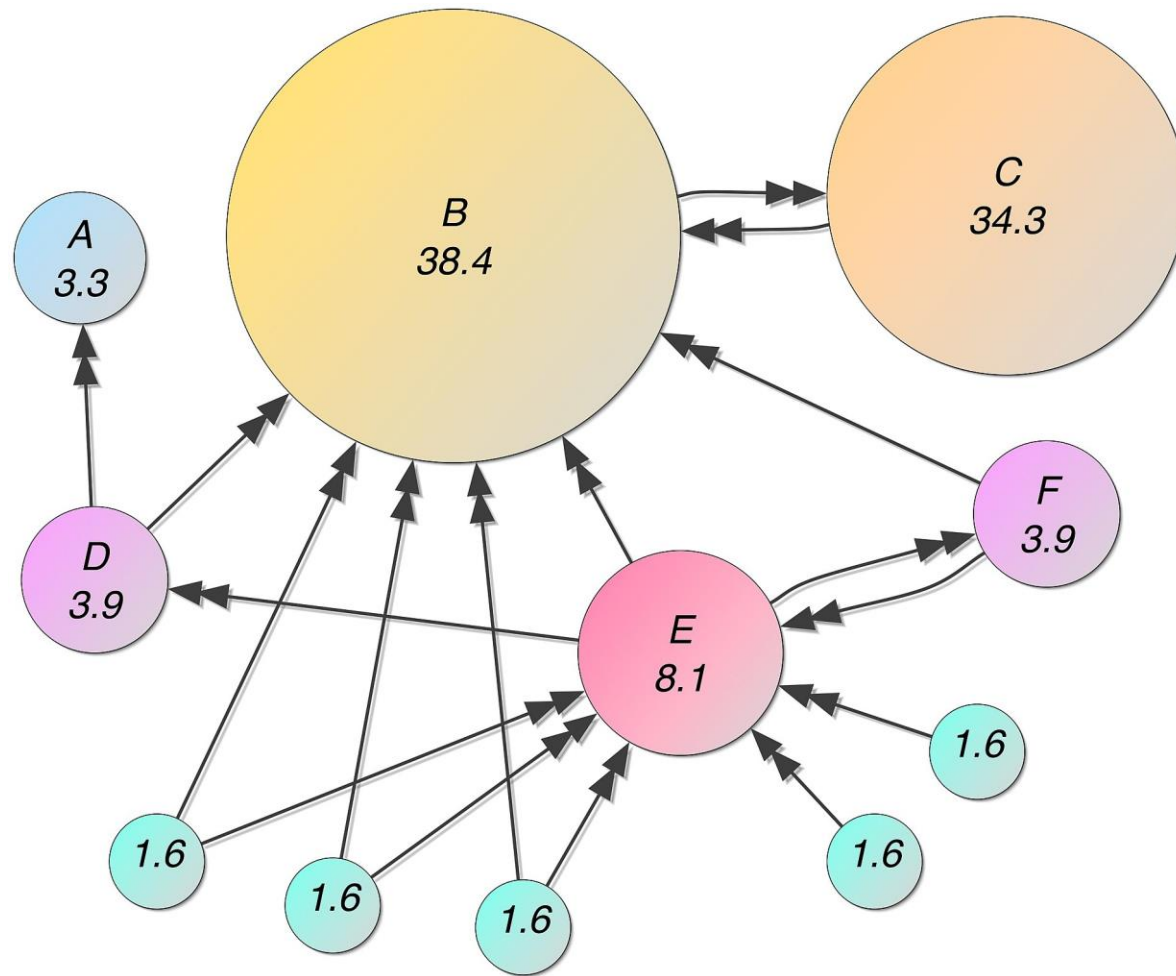


Image credit: [Wikipedia](https://en.wikipedia.org/wiki/File:PageRank_algorithm_diagram.svg)

Solving PageRank: Summary

- PageRank solves for $\mathbf{r} = \mathbf{G}\mathbf{r}$ and can be efficiently computed by power iteration of the stochastic adjacency matrix ($\mathbf{G}$)
- Adding random uniform teleportation solves issues of dead-ends and spider-traps

Stanford CS224W: Random Walk with Restarts and Personalized PageRank

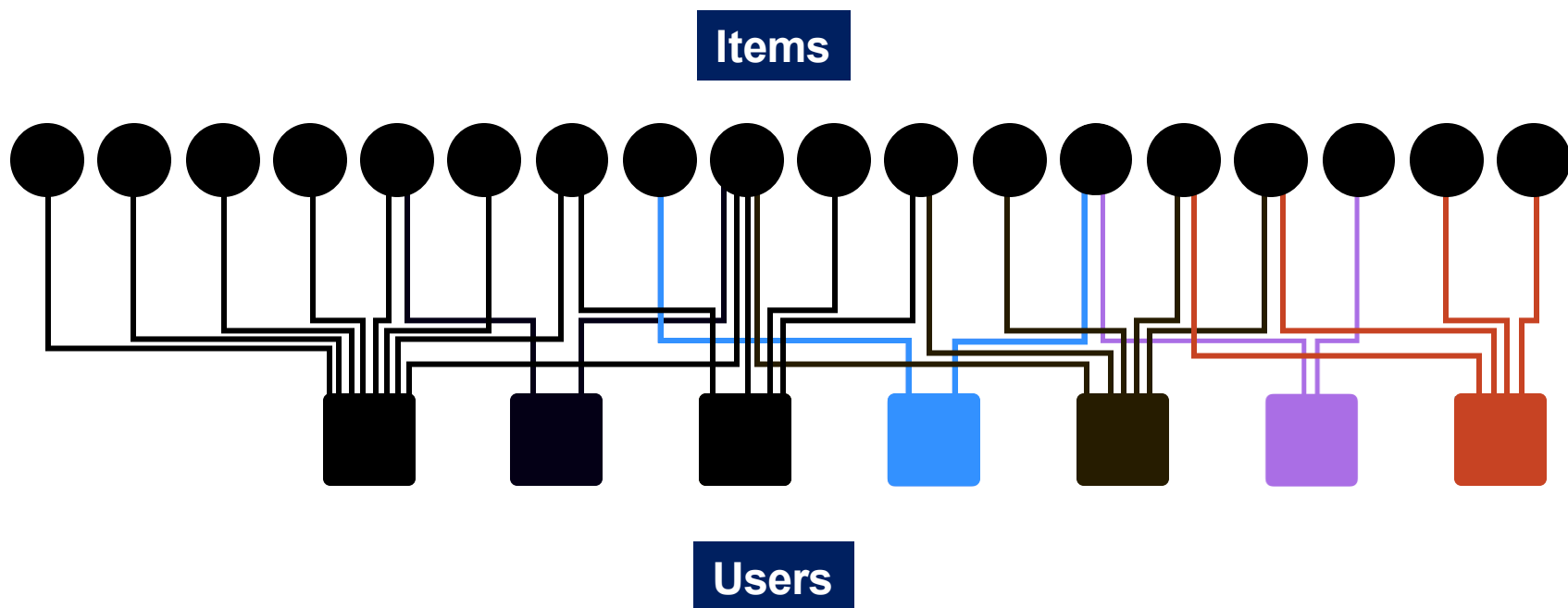
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
<http://cs224w.stanford.edu>



Example: Recommendation

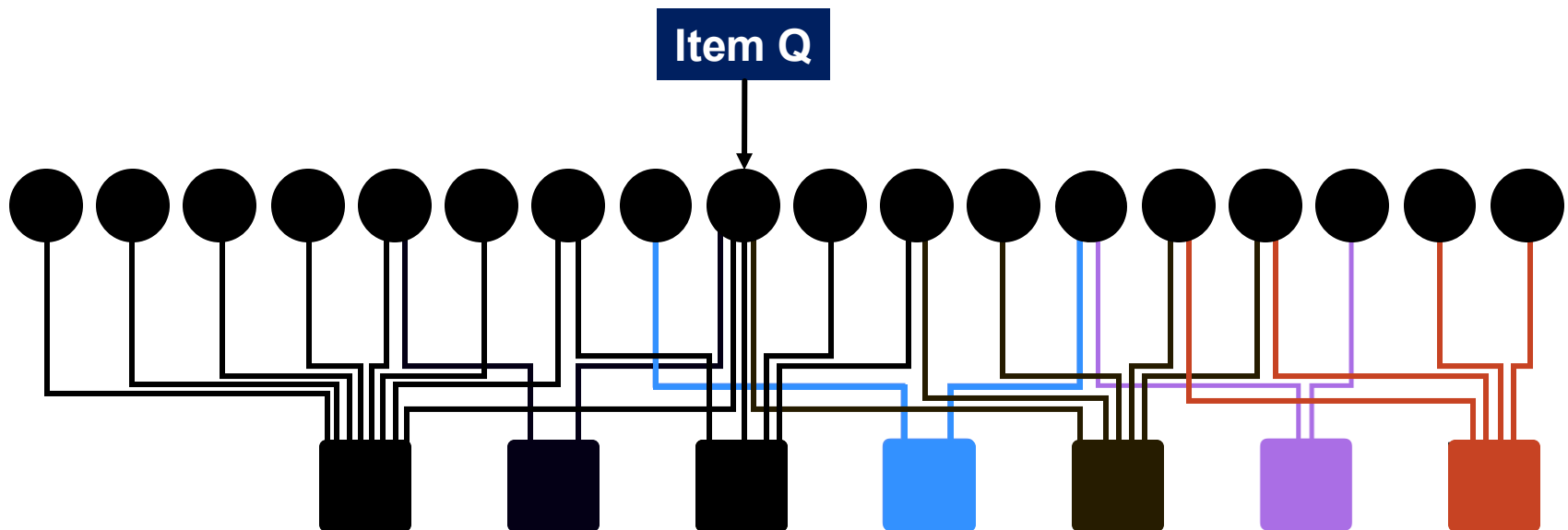
- **Given:**

A bipartite graph representing user and item interactions (e.g. purchase)



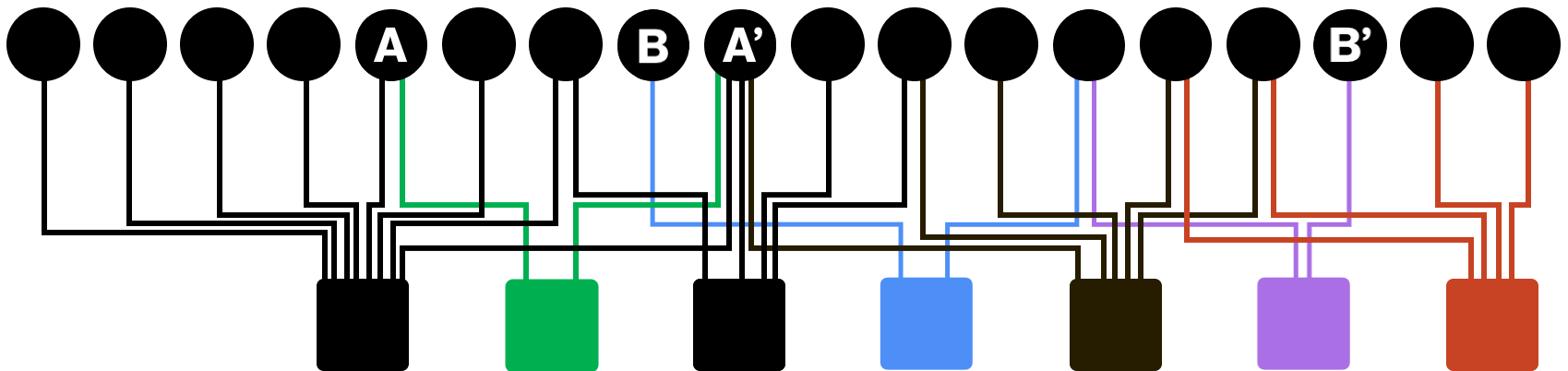
Bipartite User-Item Graph

- **Goal:** Proximity on graphs
 - What items should we recommend to a user who interacts with item Q?
 - **Intuition:** if items Q and P are interacted by similar users, recommend P when user interacts with Q



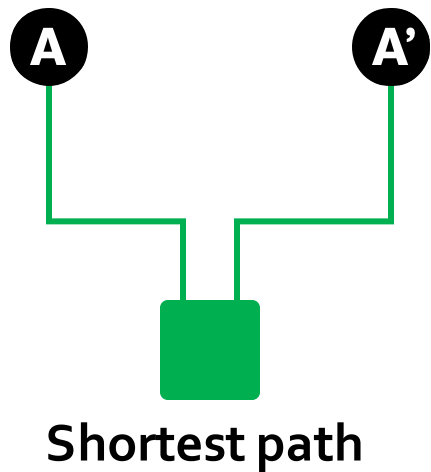
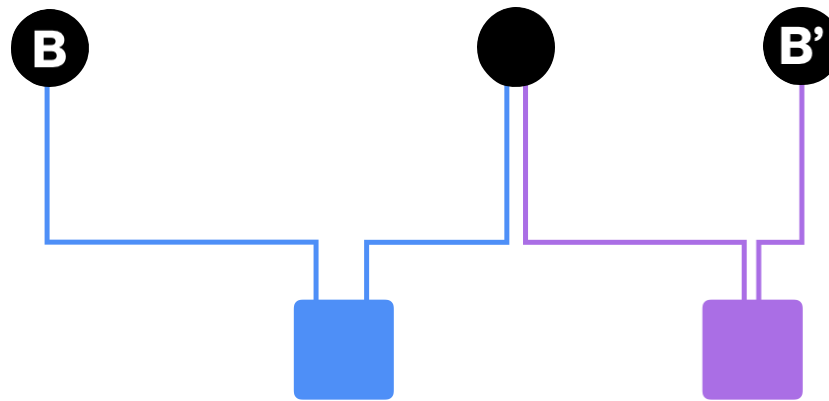
Bipartite User-to-Item Graph

- Which is more related A,A' or B,B'?



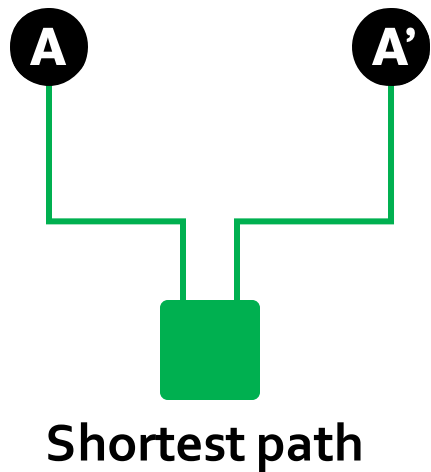
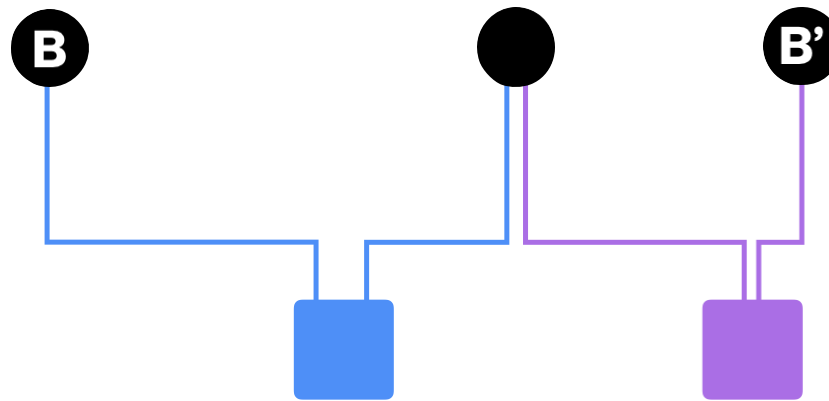
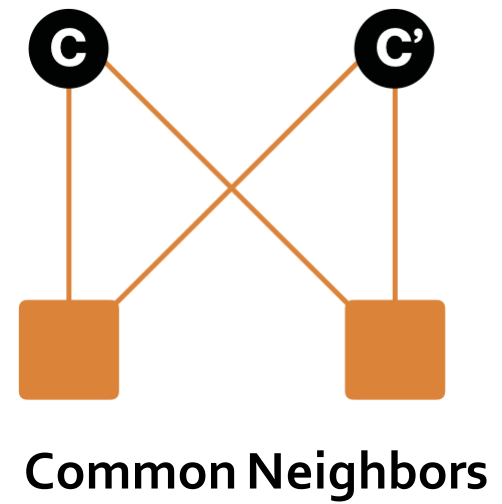
Node proximity Measurements

- Which is more related A,A', B,B' or C,C'?



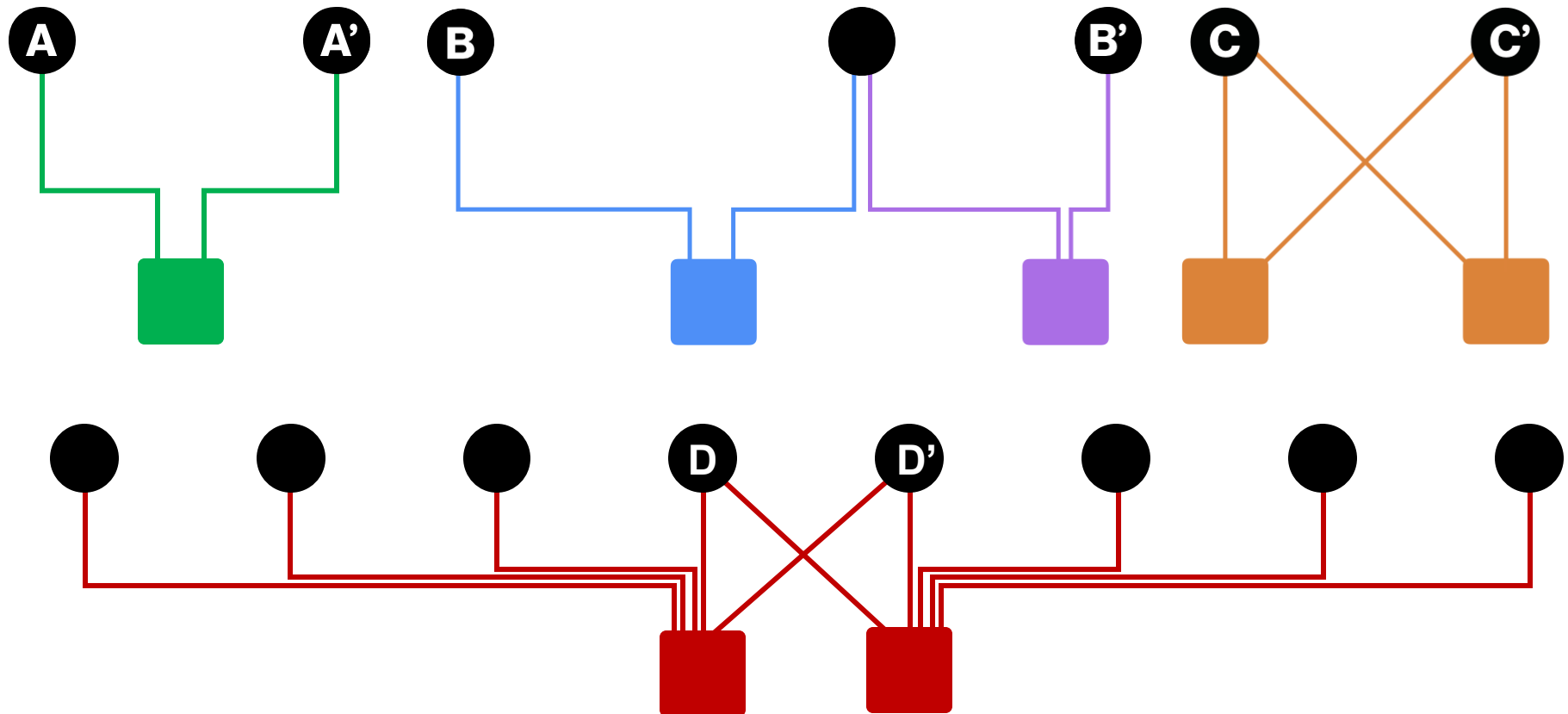
Node proximity Measurements

- Which is more related A,A', B,B' or C,C'?



Node proximity Measurements

- Which is more related A,A', B,B' or C,C'?



Personalized Page Rank/Random Walk with Restarts

Proximity on Graphs

- **PageRank:**
 - Ranks nodes by “importance”
 - Teleports with uniform probability to any node in the network
- **Personalized PageRank:**
 - Ranks proximity of nodes to the teleport nodes S
- **Proximity on graphs:**
 - **Q:** What is most related item to **Item Q**?
 - **Random Walks with Restarts**
 - Teleport back to the starting node: $S = \{Q\}$

Idea: Random Walks

■ Idea

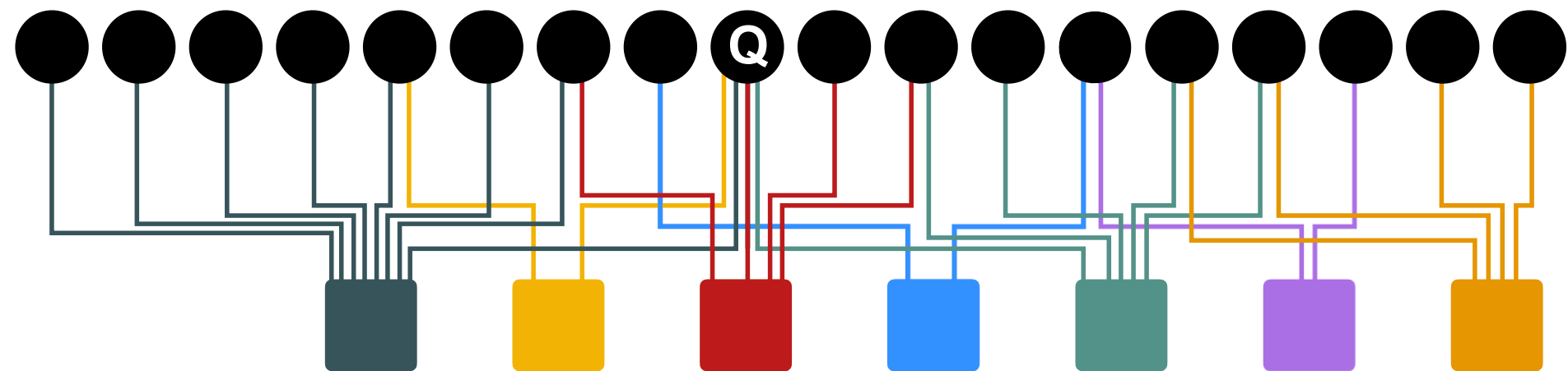
- Every node has some importance
- Importance gets evenly split among all edges and pushed to the neighbors:

■ Given a set of QUERY_NODES, we simulate a random walk:

- Make a step to a random neighbor and record the visit (visit count)
- With probability ALPHA, restart the walk at one of the QUERY_NODES
- The nodes with the highest visit count have highest proximity to the QUERY_NODES

Random Walks

- **Idea:**
 - Every node has some importance
 - Importance gets evenly split among all edges and pushed to the neighbors
- Given a set of **QUERY NODES Q**, simulate a random walk:



Random Walk Algorithm

■ Proximity to query node(s) Q :

$\text{ALPHA} = 0.5$

$\text{QUERY_NODES} = \{ \text{Q} \}$

```
item = QUERY_NODES.sample_by_weight()
```

```
for i in range( N_STEPS ):
```

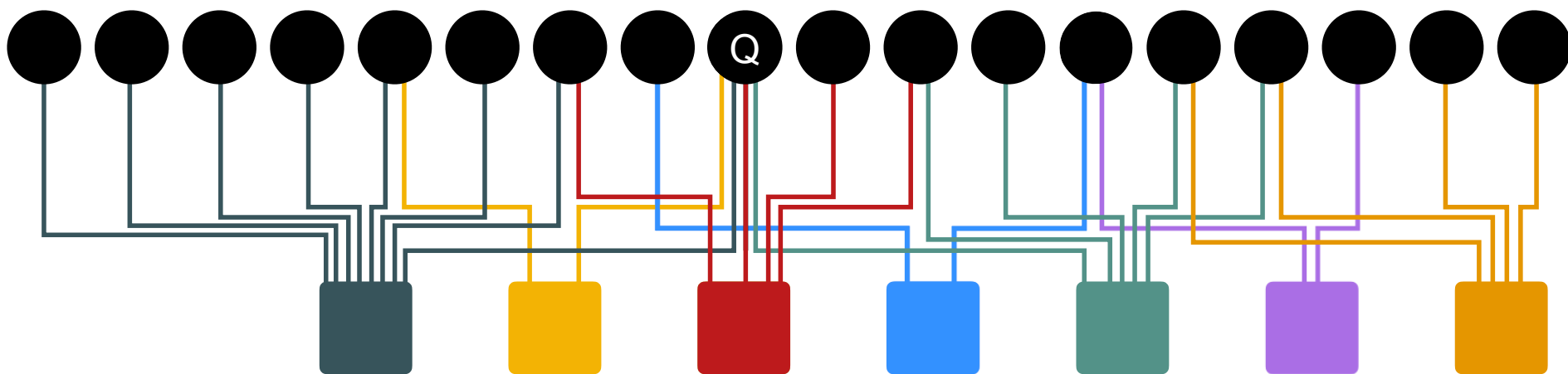
```
    user = item.get_random_neighbor()
```

```
    item = user.get_random_neighbor()
```

```
    item.visit_count += 1
```

```
    if random() < ALPHA:
```

```
        item = QUERY_NODES.sample.by_weight()
```



Random Walk Algorithm

■ Proximity to query node(s) Q :

ALPHA = 0.5

QUERY_NODES =



```
item = QUERY_NODES.sample_by_weight()
```

```
for i in range( N_STEPS ):
```

```
    user = item.get_random_neighbor()
```

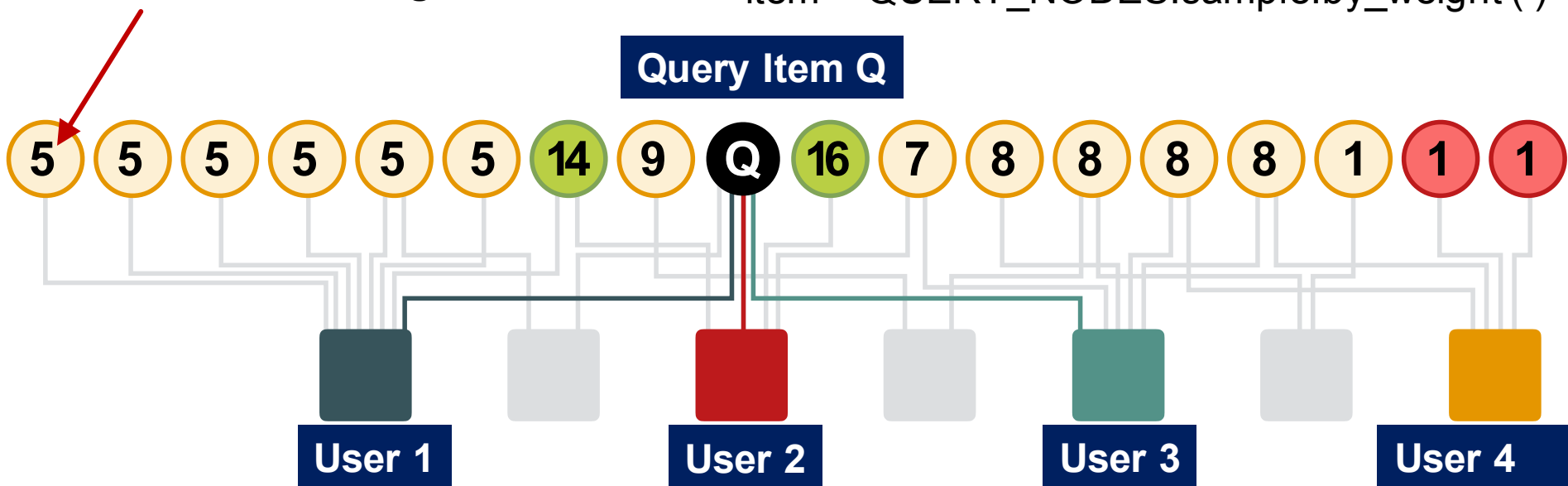
```
    item = user.get_random_neighbor()
```

```
    item.visit_count += 1
```

```
    if random( ) < ALPHA:
```

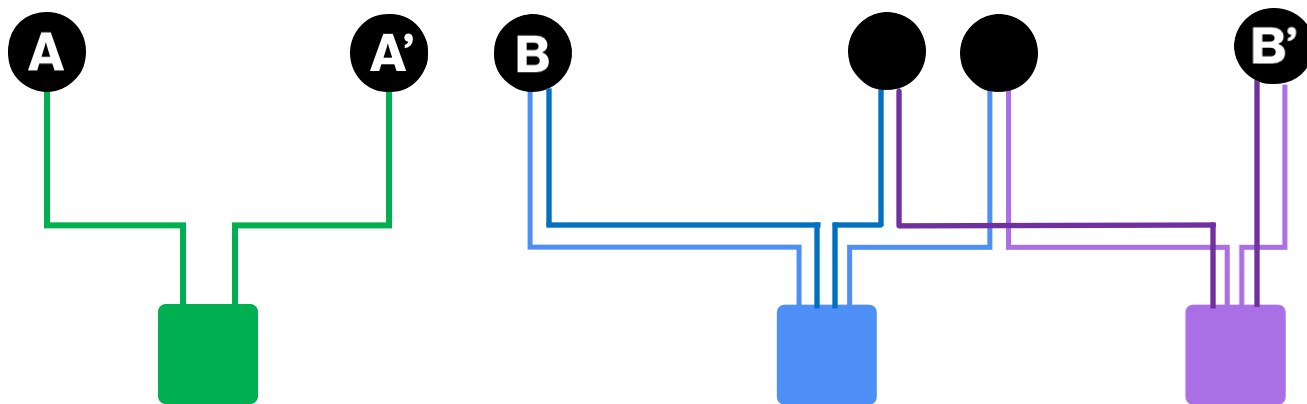
```
        item = QUERY_NODES.sample.by_weight( )
```

Number of visits by
random walks starting at Q



Benefits

- Why is this a good solution?
- Because the “similarity” considers:
 - Multiple connections
 - Multiple paths
 - Direct and indirect connections
 - Degree of the node



Summary: Page Rank Variants

■ PageRank:

- Teleports to any node
- Nodes can have the same probability of the surfer landing:
 $S = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$

■ Topic-Specific PageRank aka Personalized PageRank:

- Teleports to a specific set of nodes
- Nodes can have different probabilities of the surfer landing there:

$$S = [0.1, 0, 0, 0.2, 0, 0, 0.5, 0, 0, 0.2]$$

■ Random Walk with Restarts:

- Topic-Specific PageRank where teleport is always to the same node:

$$S = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$$

Summary

- A graph is naturally represented as a matrix
- We defined a random walk process over the graph
 - Random surfer moving across the links and with random teleportation
 - Stochastic adjacency matrix M
- PageRank = Limiting distribution of the surfer location represented node importance
 - Corresponds to the leading eigenvector of transformed adjacency matrix M .

Summary

- **PageRank**
 - Measures importance of nodes in graph
 - Can be efficiently computed by **power iteration of adjacency matrix**
- **Personalized PageRank (PPR)**
 - Measures importance of nodes with respect to a particular node or set of nodes
 - Can be efficiently computed by **random walk**
- **Node embeddings** based on random walks can be expressed as **matrix factorization**
- **Viewing graphs as matrices plays a key role in all above algorithms!**

Stanford CS224W: Graph Neural Networks

CS224W: Machine Learning with Graphs

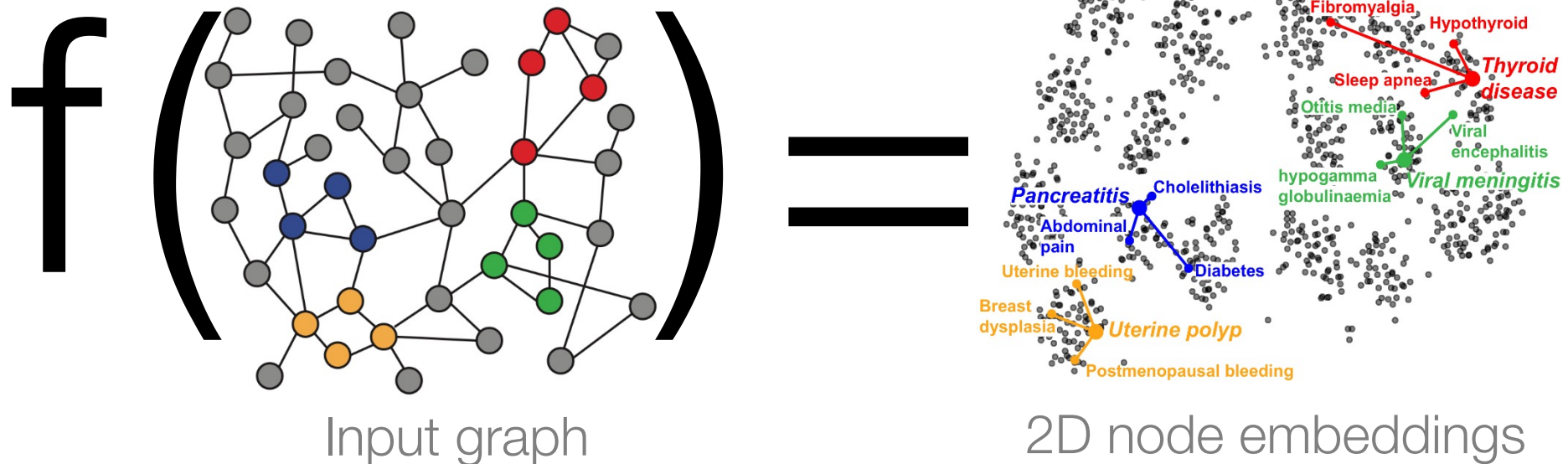
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



Recap: Node Embeddings

- **Intuition:** Map nodes to d -dimensional embeddings such that similar nodes in the graph are embedded close together

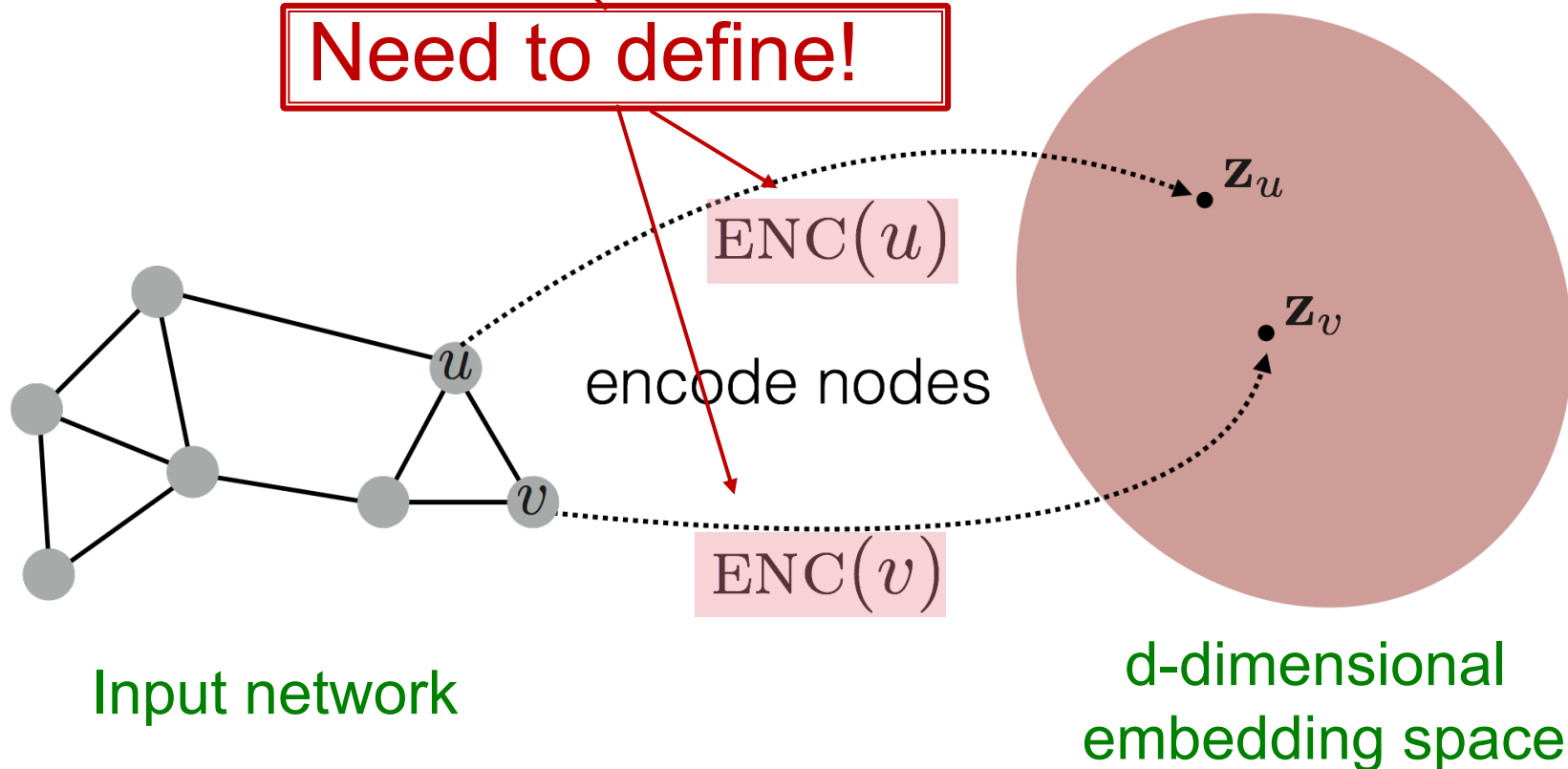


How to learn mapping function f ?

Recap: Node Embeddings

Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$

Need to define!



Recap: Two Key Components

- **Encoder:** Maps each node to a low-dimensional vector

$$\text{ENC}(v) = \mathbf{z}_v$$

d -dimensional embedding

node in the input graph

- **Similarity function:** Specifies how the relationships in vector space map to the relationships in the original network

$$\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$$

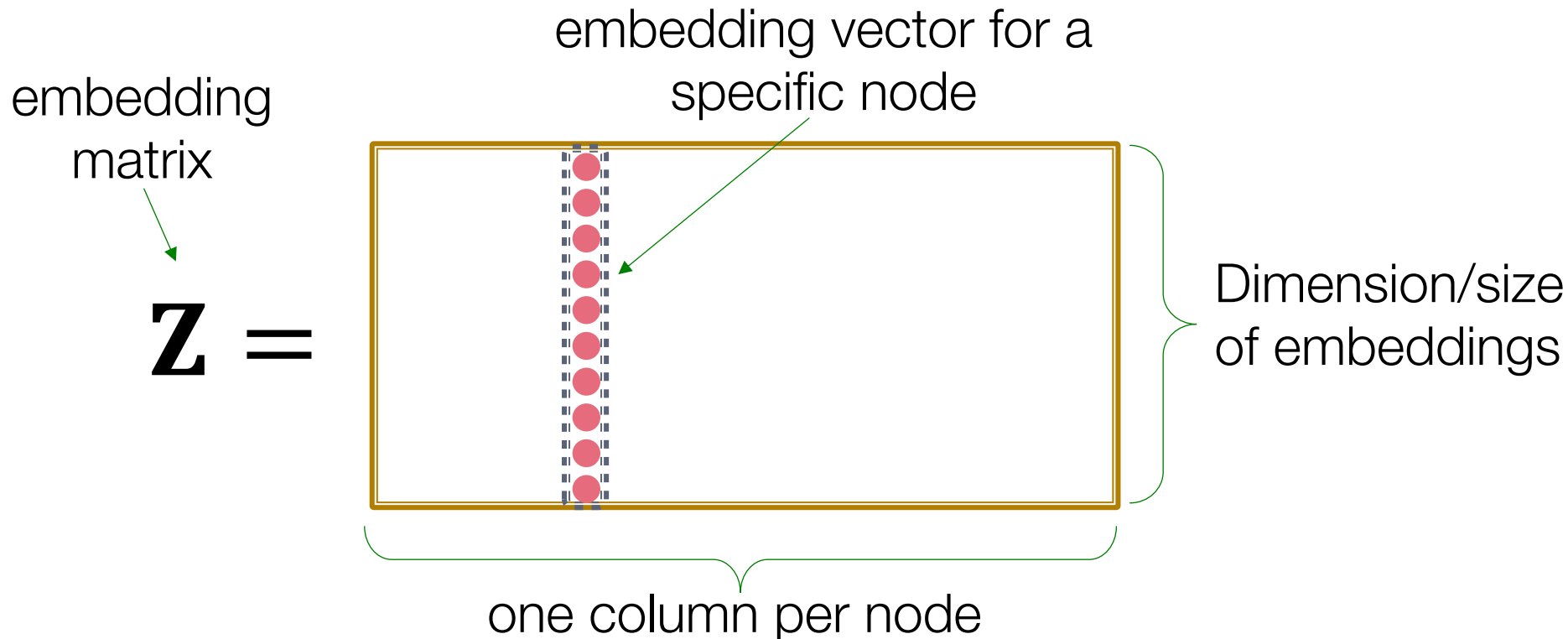
Similarity of u and v in the original network

dot product between node embeddings

Decoder

Recap: “Shallow” Encoding

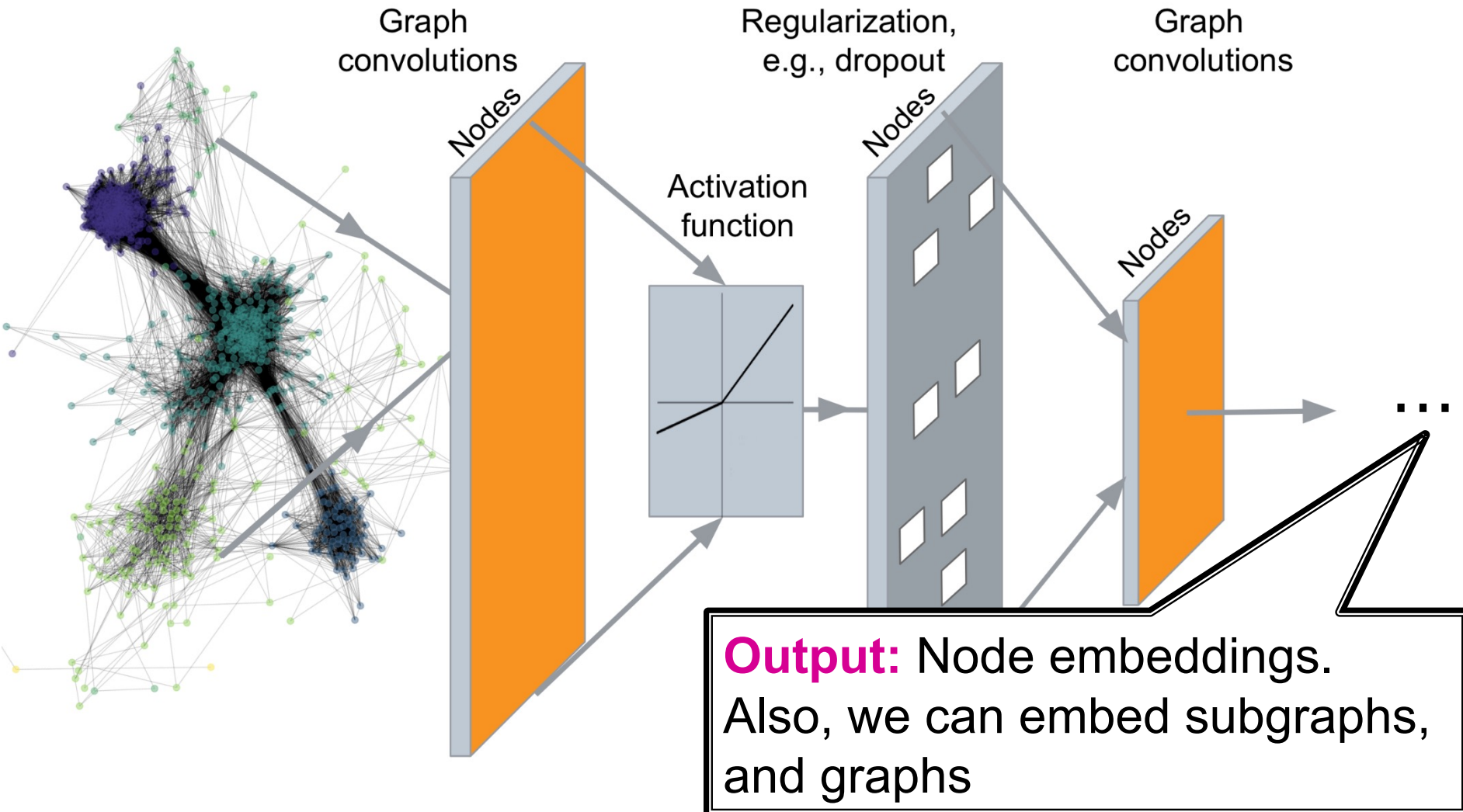
Simplest encoding approach: **Encoder is just an embedding-lookup**



Recap: Shallow Encoders

- **Limitations of shallow embedding methods:**
 - **$O(|V|)$ parameters are needed:**
 - No sharing of parameters between nodes
 - Every node has its own unique embedding
 - **Inherently “transductive”:**
 - Cannot generate embeddings for nodes that are not seen during training
 - **Do not incorporate node features:**
 - Nodes in many graphs have features that we can and should leverage

Deep Graph Encoders

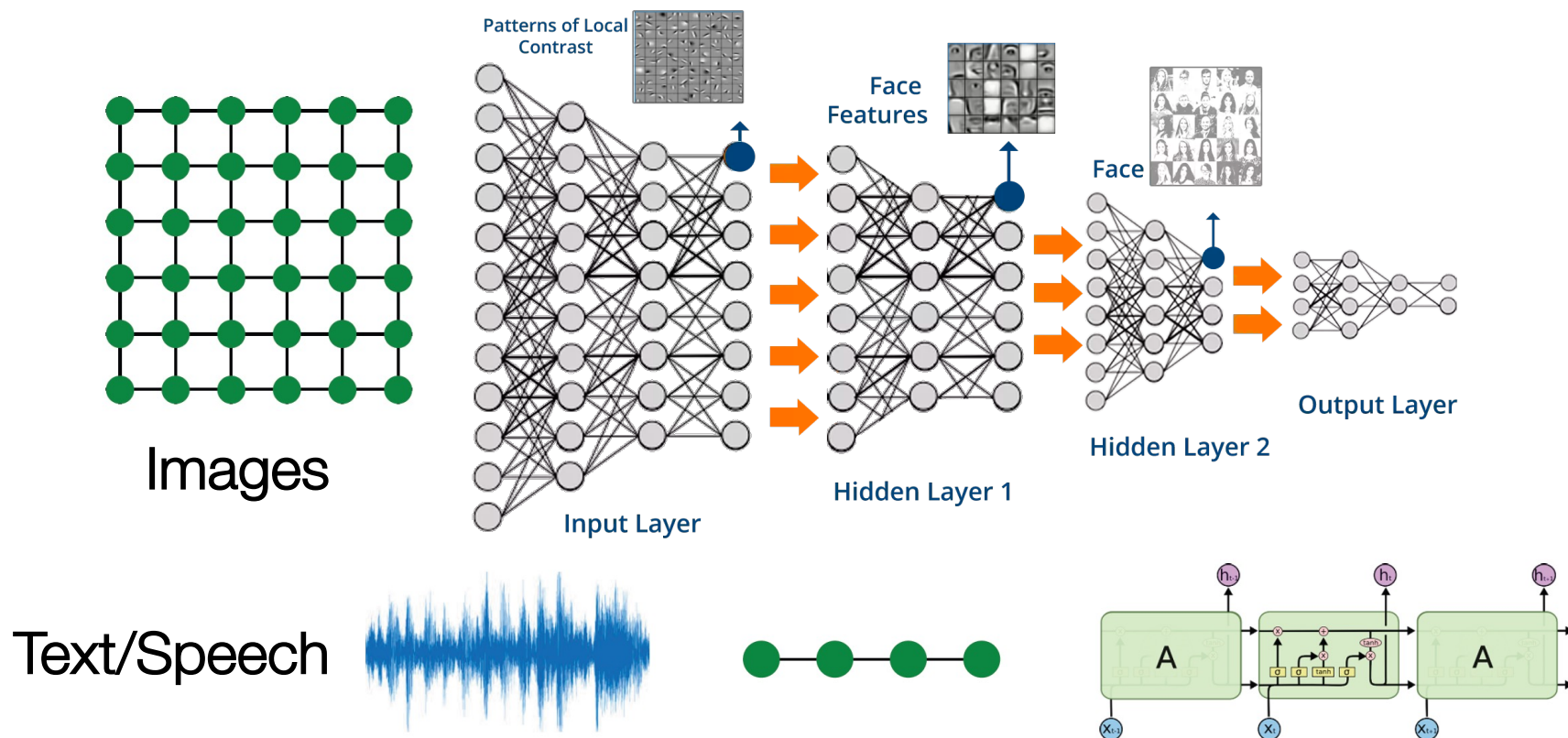


Tasks on Networks

Tasks we will be able to solve:

- **Node classification**
 - Predict a type of a given node
- **Link prediction**
 - Predict whether two nodes are linked
- **Community detection**
 - Identify densely linked clusters of nodes
- **Network similarity**
 - How similar are two (sub)networks

Modern ML Toolbox

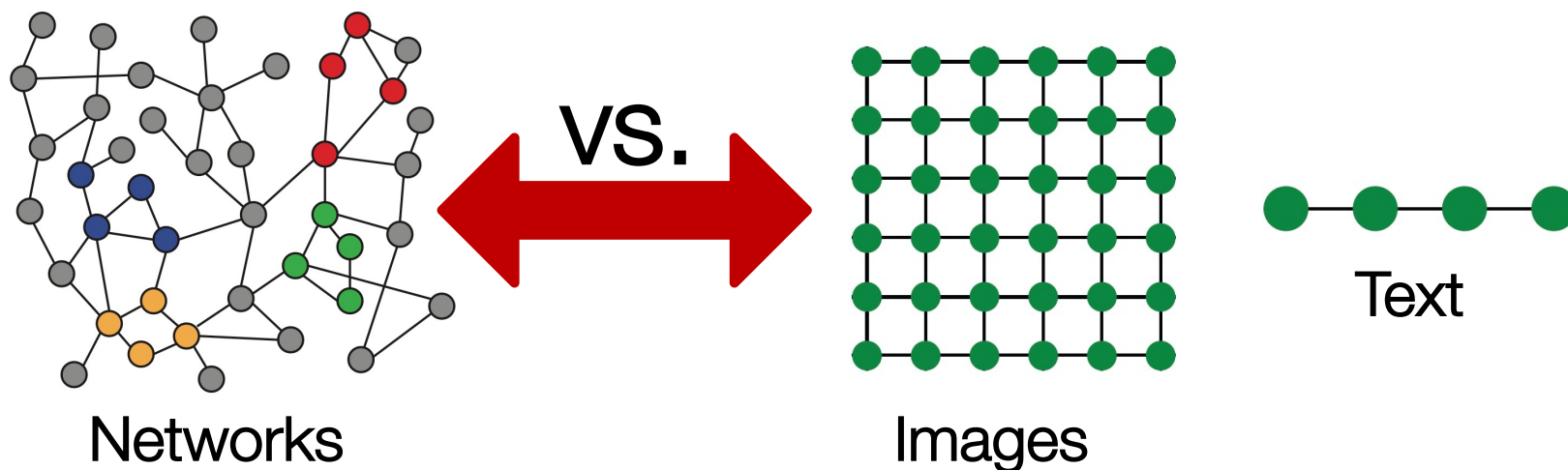


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

But networks are far more complex!

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)



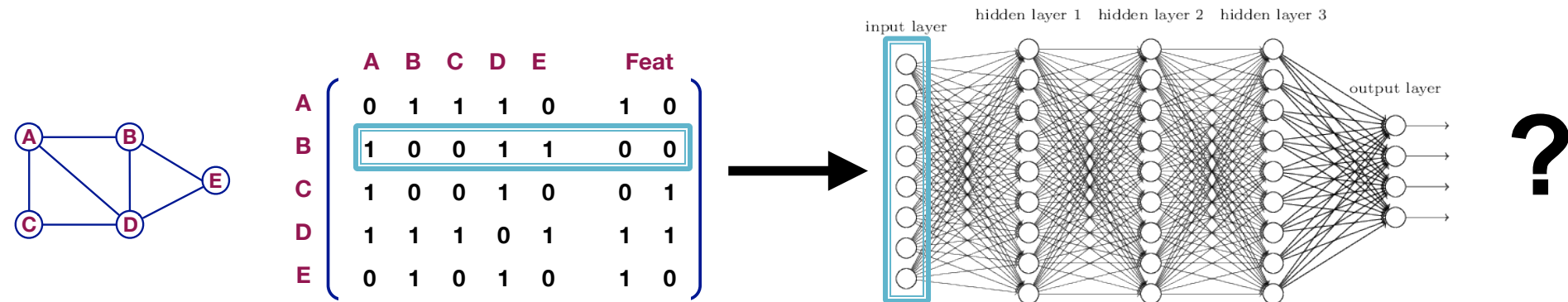
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Setup

- Assume we have a graph G :
 - V is the **vertex set**
 - A is the **adjacency matrix** (assume binary)
 - $X \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
 - v : a node in V ; $N(v)$: the set of neighbors of v .
 - **Node features:**
 - Social networks: User profile, User image
 - Biological networks: Gene expression profiles, gene functional information
 - When there is no node feature in the graph dataset:
 - Indicator vectors (one-hot encoding of a node)
 - Vector of constant 1: $[1, 1, \dots, 1]$

A Naïve Approach

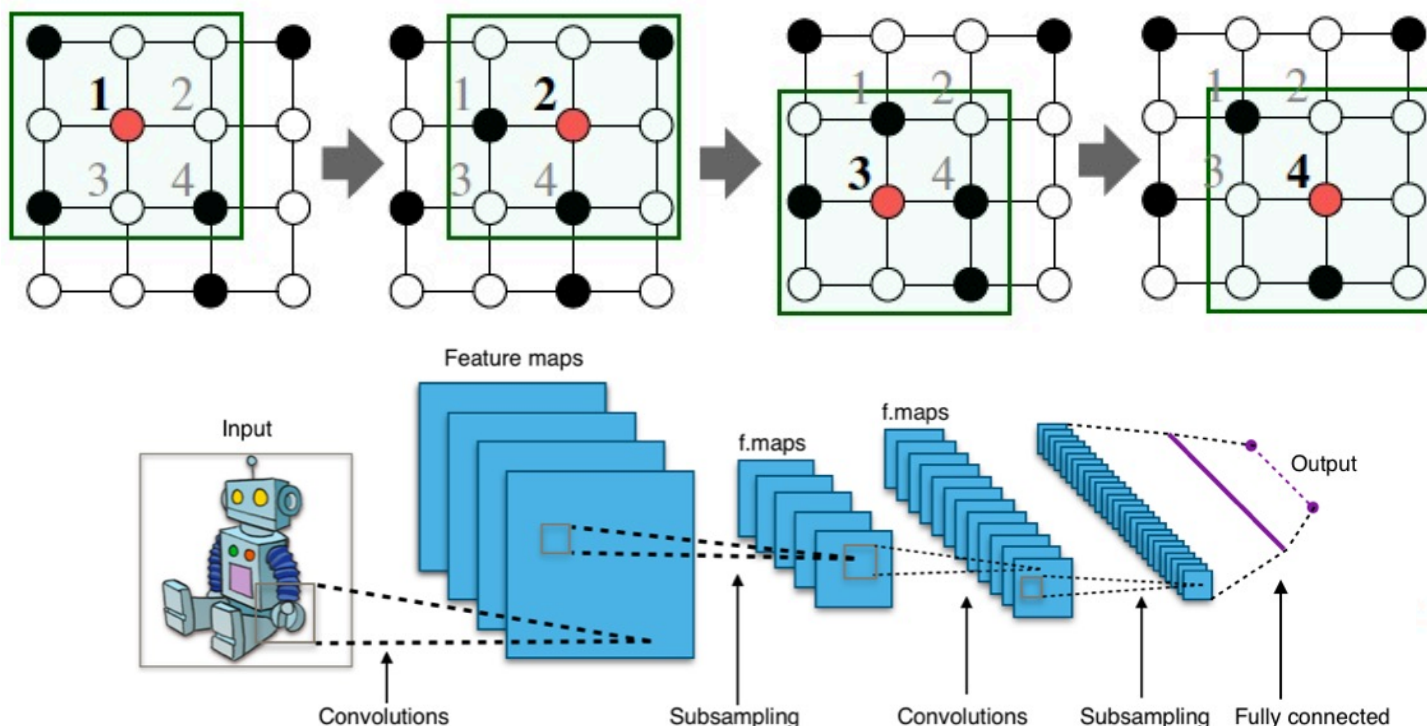
- Join adjacency matrix and features
- Feed them into a deep neural net:



- Issues with this idea:
 - $O(|V|)$ parameters
 - Not applicable to graphs of different sizes
 - Sensitive to node ordering

Idea: Convolutional Networks

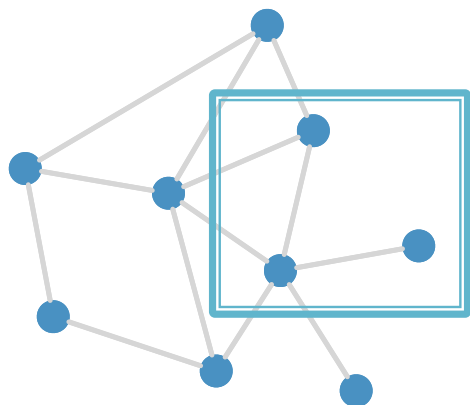
CNN on an image:



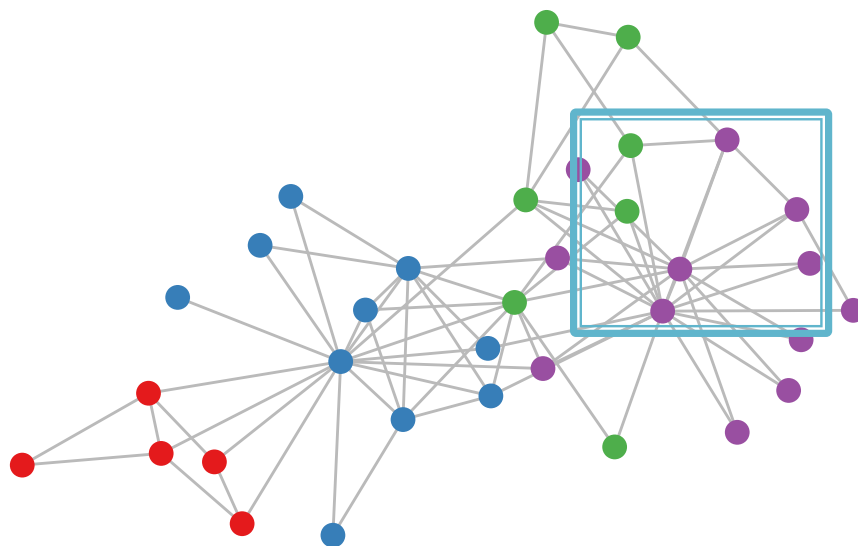
Goal is to generalize convolutions beyond simple lattices
Leverage node features/attributes (e.g., text, images)

Real-World Graphs

But our graphs look like this:



or this:



- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

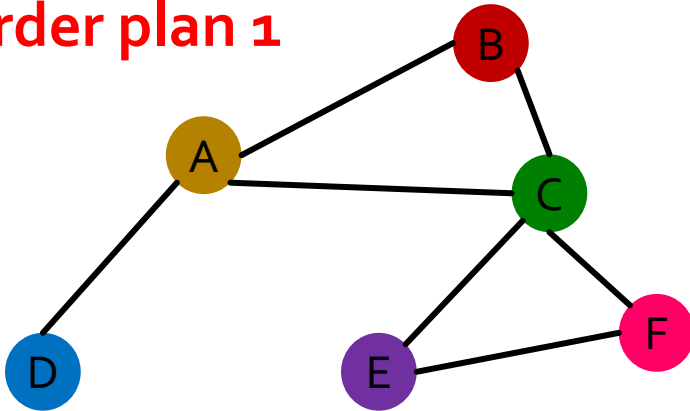
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**
- We can have many different order plans.

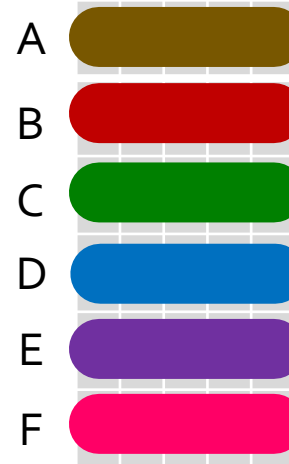
Permutation Invariance

- Graph does not have a canonical order of the nodes!

Order plan 1



Node features X_1



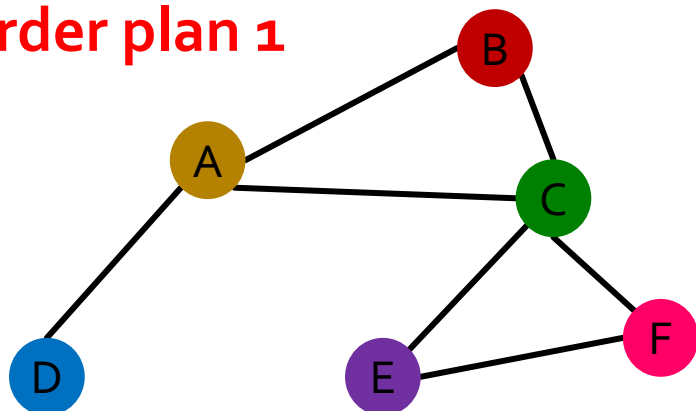
Adjacency matrix A_1

	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

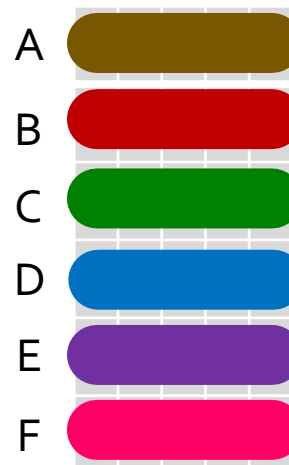
Permutation Invariance

- Graph does not have a canonical order of the nodes!

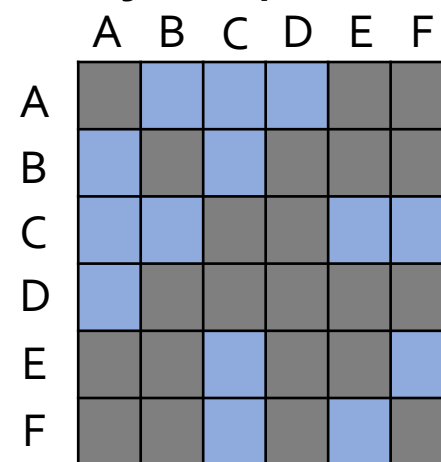
Order plan 1



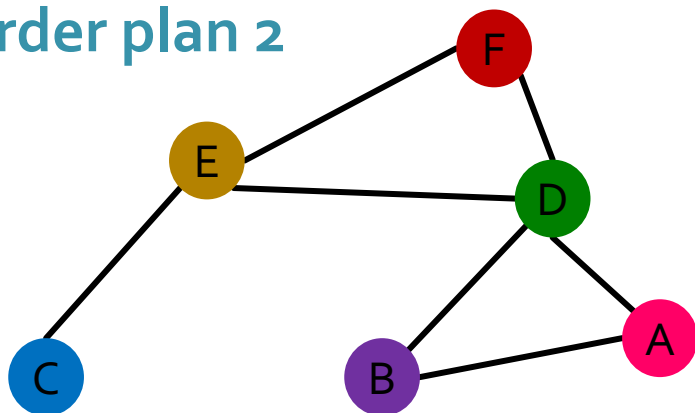
Node features X_1



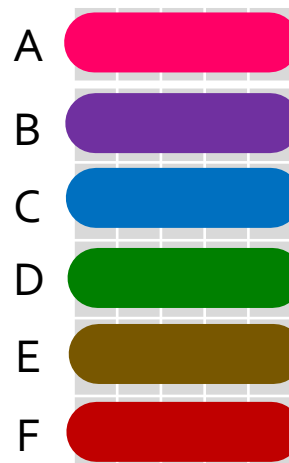
Adjacency matrix A_1



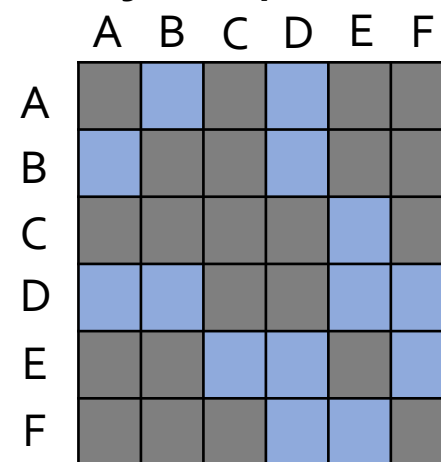
Order plan 2



Node features X_2



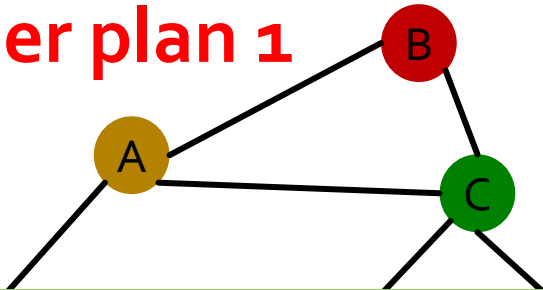
Adjacency matrix A_2



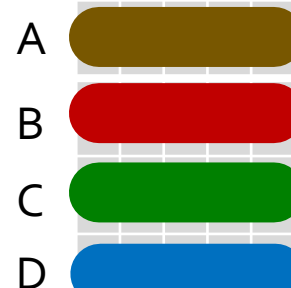
Permutation Invariance

- Graph does not have a canonical order of the nodes!

Order plan 1



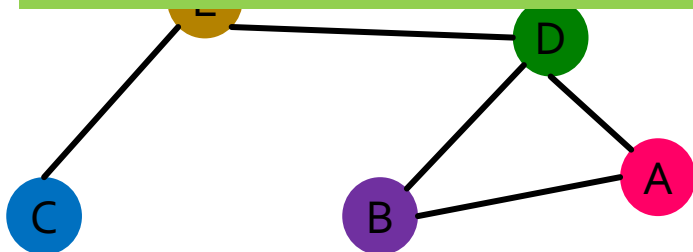
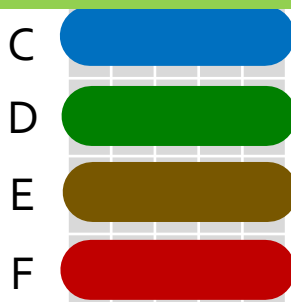
Node feature X_1



Adjacency matrix A_1

	A	B	C	D	E	F
A						
B						
C						
D						

Graph and node representations
should be the same for **Order plan 1**
and **Order plan 2**



	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

Permutation Invariance

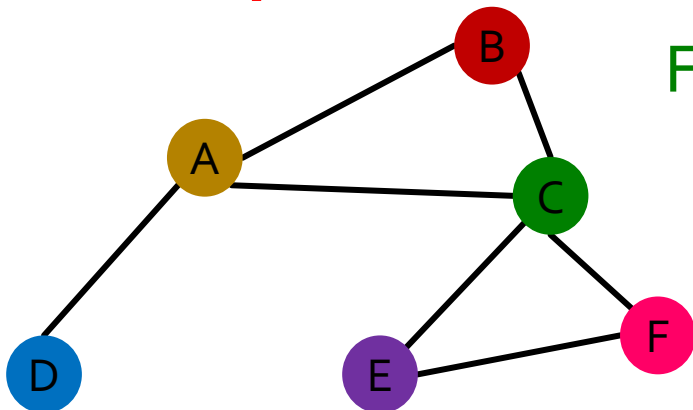
What does it mean by “graph representation is same for two order plans”?

- Consider we learn a function f that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$ then

$$f(A_1, X_1) = f(A_2, X_2)$$

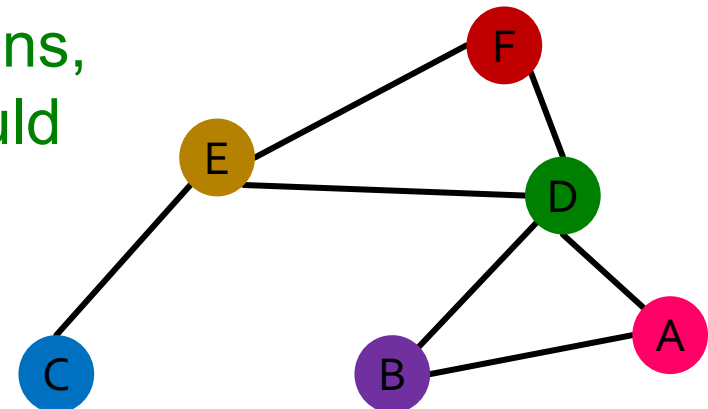
A is the adjacency matrix
 X is the node feature matrix

Order plan 1: A_1, X_1



For two order plans,
output of f should
be the same!

Order plan 2: A_2, X_2



Permutation Invariance

What does it mean by “graph representation is same for two order plans”?

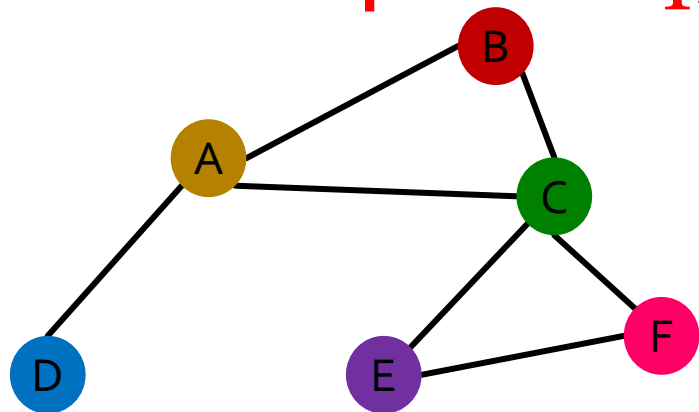
- Consider we learn a function f that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$.
 A is the adjacency matrix
 X is the node feature matrix
- Then, if $f(A_i, X_i) = f(A_j, X_j)$ for any order plan i and j , we formally say f is a **permutation invariant function**.

For a graph with m nodes, there are $m!$ different order plans.

Permutation Equivariance

Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

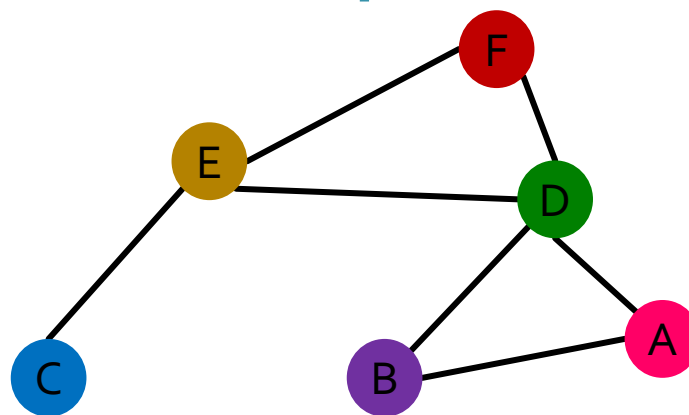
Order plan 1: A_1, X_1



A		
B		
C		
D		
E		
F		

$$f(A_1, X_1) =$$

Order plan 2: A_2, X_2



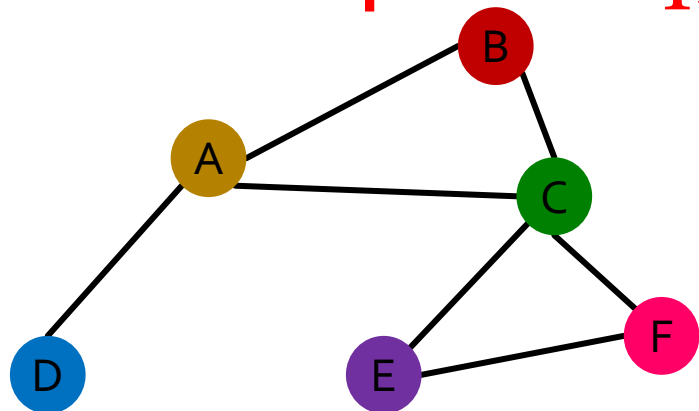
A		
B		
C		
D		
E		
F		

$$f(A_2, X_2) =$$

Permutation Equivariance

Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

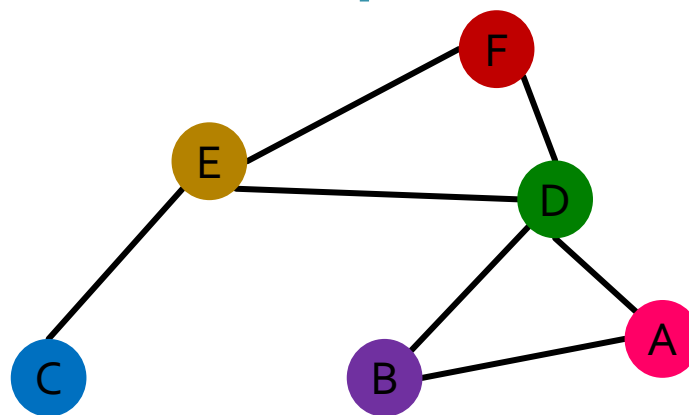


Representation vector of the brown node A

A		
B		
C		
D		
E		
F		

$$f(A_1, X_1) =$$

Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$

A		
B		
C		
D		
E		
F		

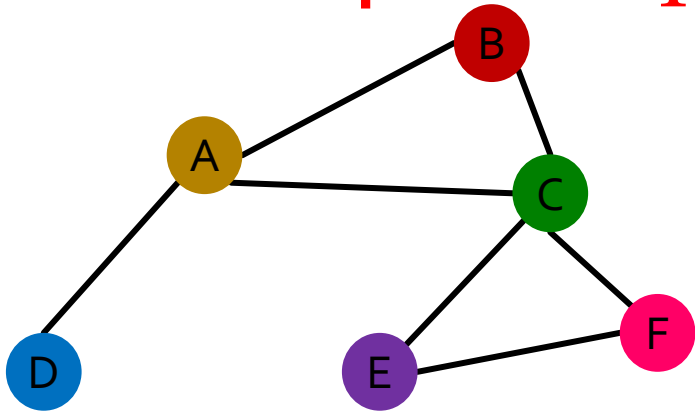
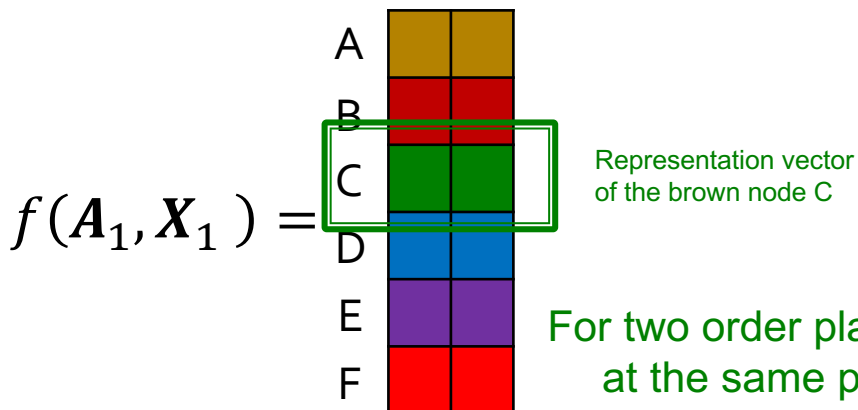
Representation vector of the brown node E

For two order plans, the vector of node at the same position is the same!

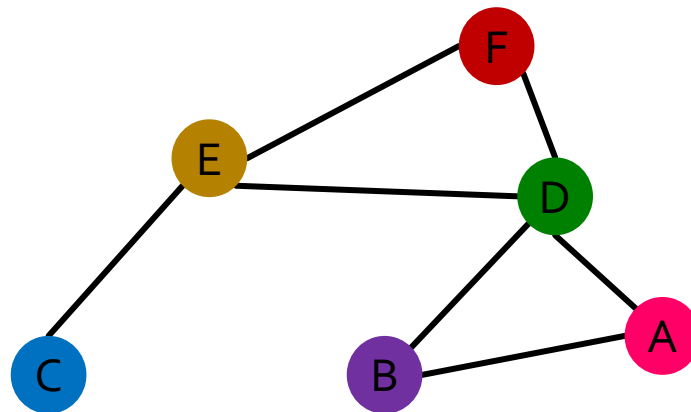
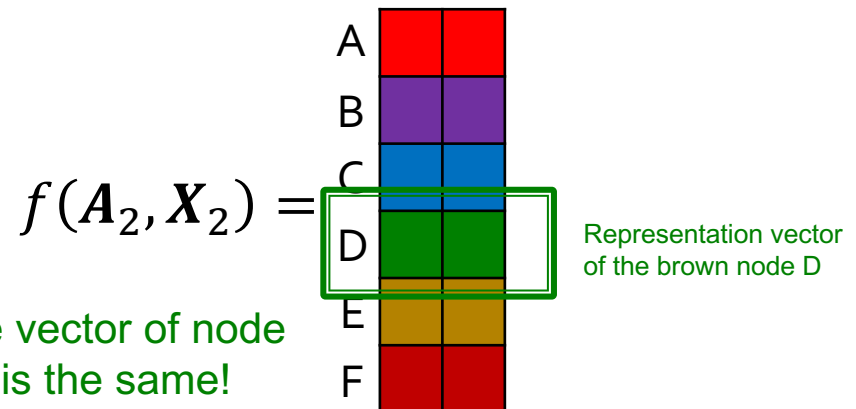
Permutation Equivariance

Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1



Order plan 2: A_2, X_2



For two order plans, the vector of node at the same position is the same!

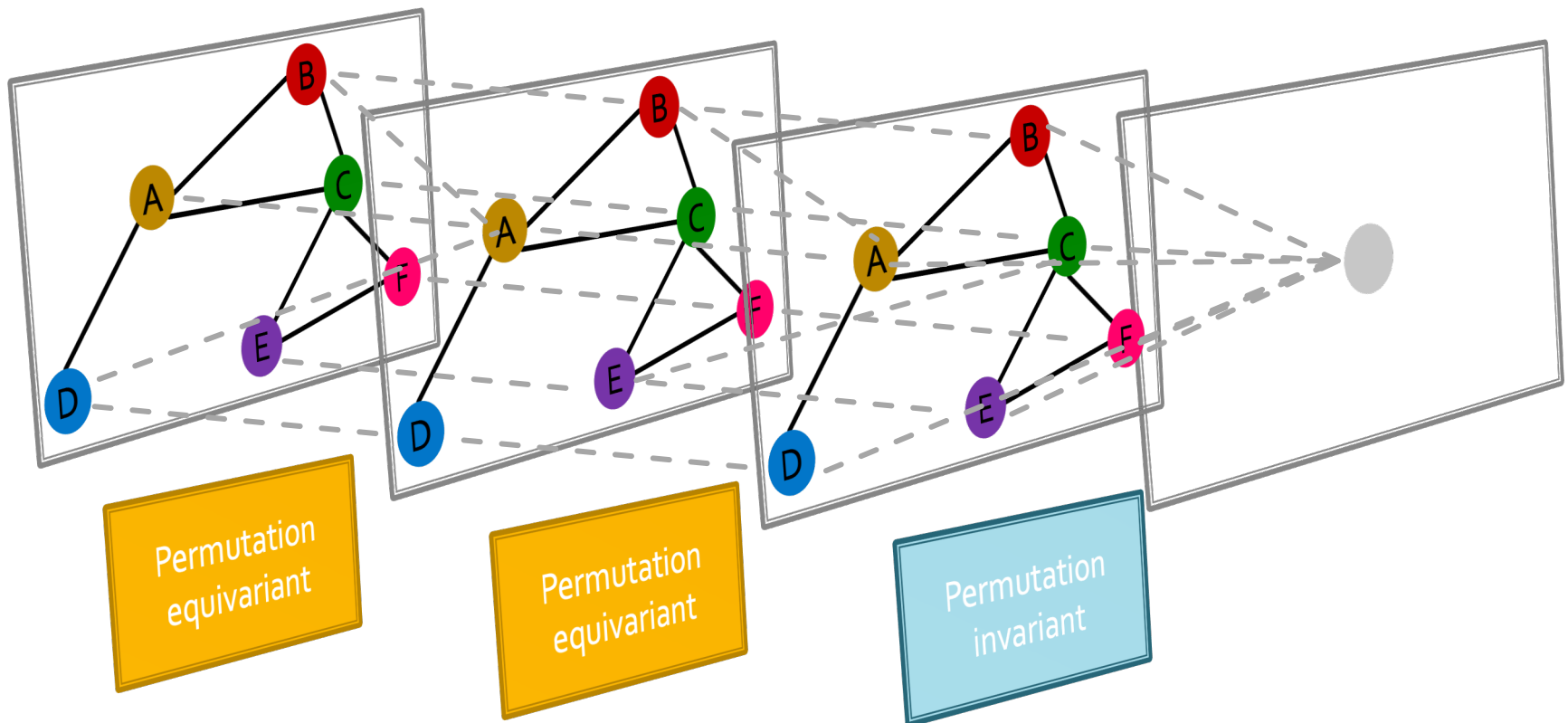
Permutation Equivariance

For node representation

- Consider we learn a function f that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{m \times d}$
 - graph has m nodes, each row is the embedding of a node.
- Similarly, if this property holds for any pair of order plan i and j , we say f is a **permutation equivariant function**.

Graph Neural Network Overview

- Graph neural networks consist of multiple permutation equivariant / invariant functions.

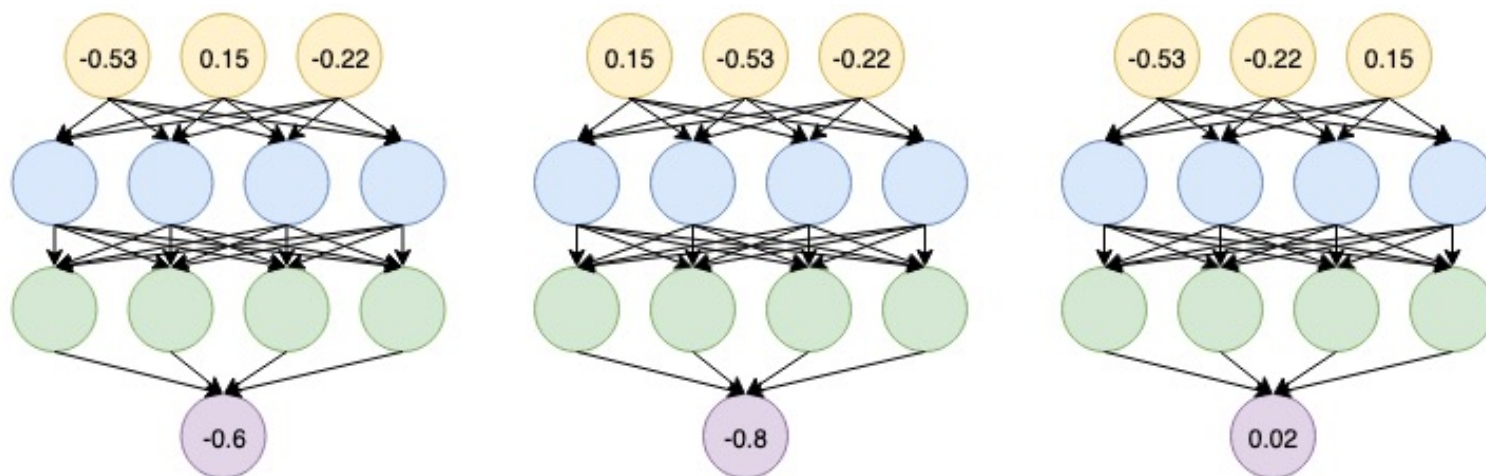


Graph Neural Network Overview

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

■ **No.**

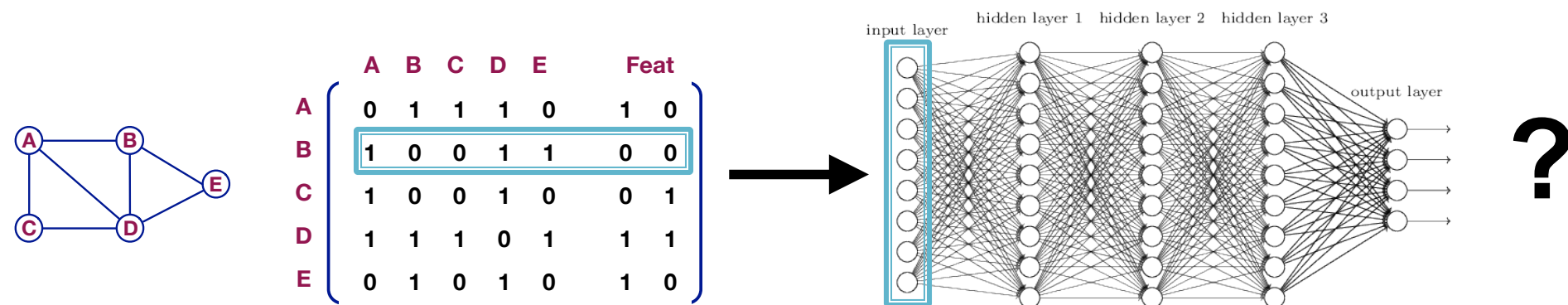
Switching the order of the input leads to different outputs!



Graph Neural Network Overview

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

■ **No.**



This explains why **the naïve MLP approach fails for graphs!**

Graph Neural Network Overview

- Are any neural network architecture, e.g.,

Next: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors!

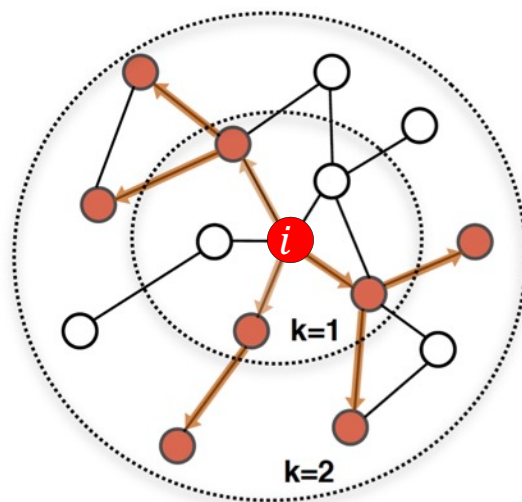
?

Outline of Today's Lecture

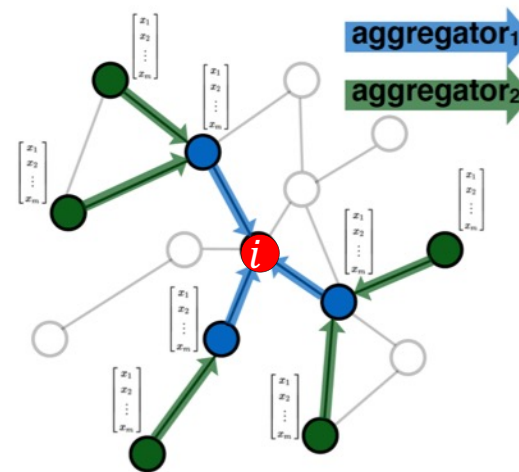
1. Basics of deep learning ✓
2. Deep learning for graphs ✓
3. Graph Convolutional Networks 
4. GNNs subsume CNNs and Transformers

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node
computation graph

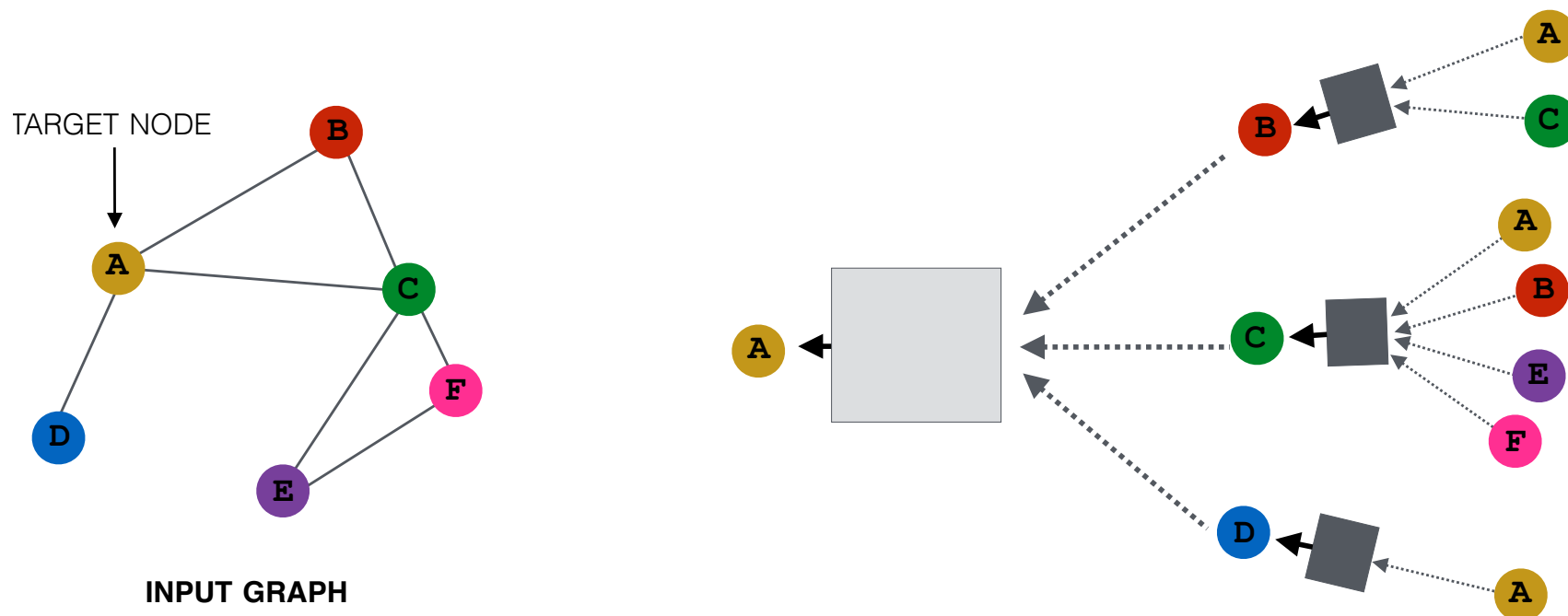


Propagate and
transform information

Learn how to propagate information across the graph to compute node features

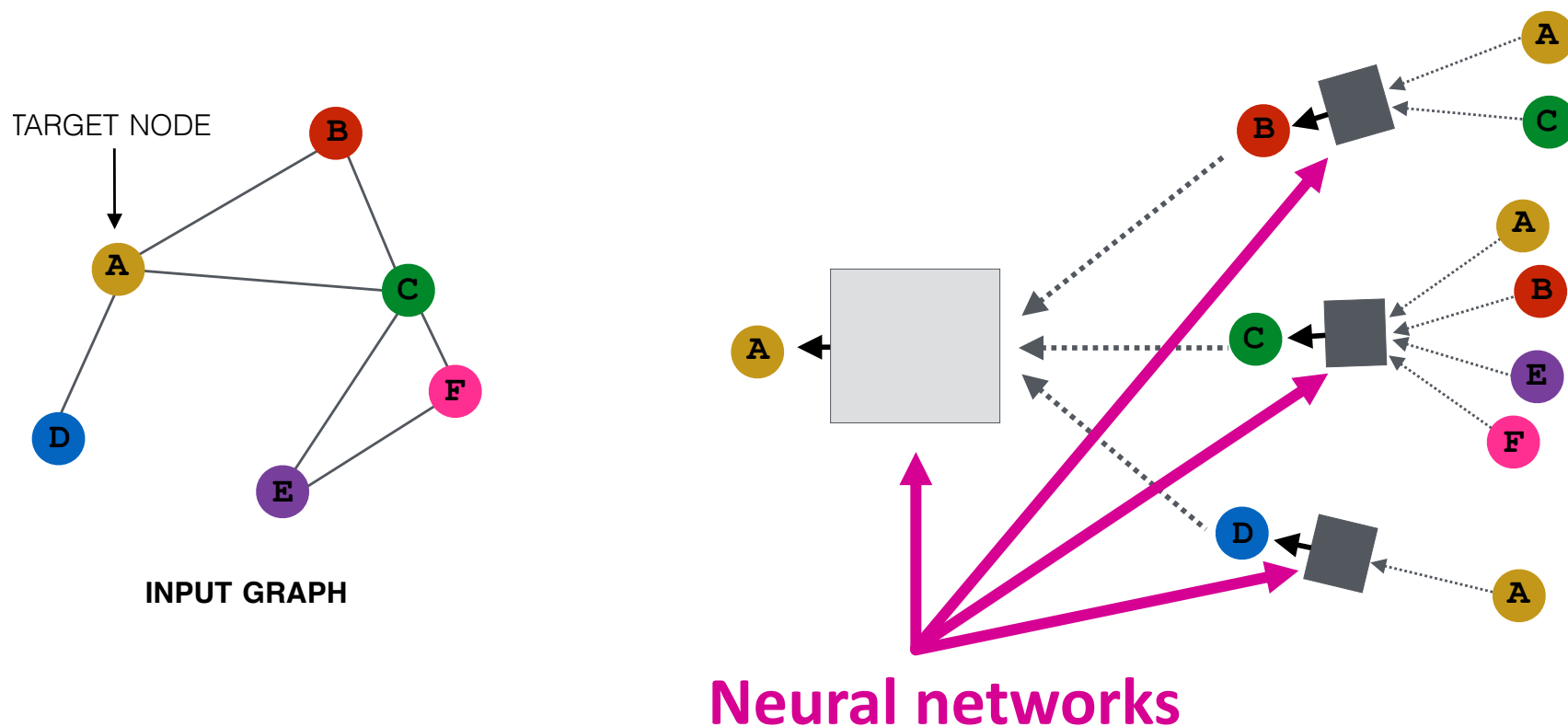
Idea: Aggregate Neighbors

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



Idea: Aggregate Neighbors

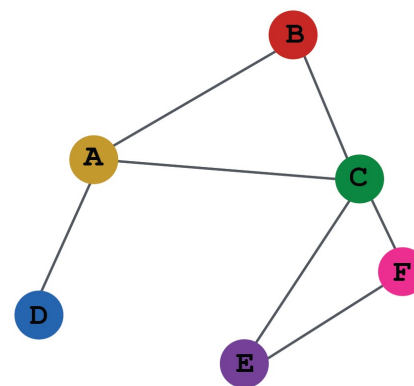
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



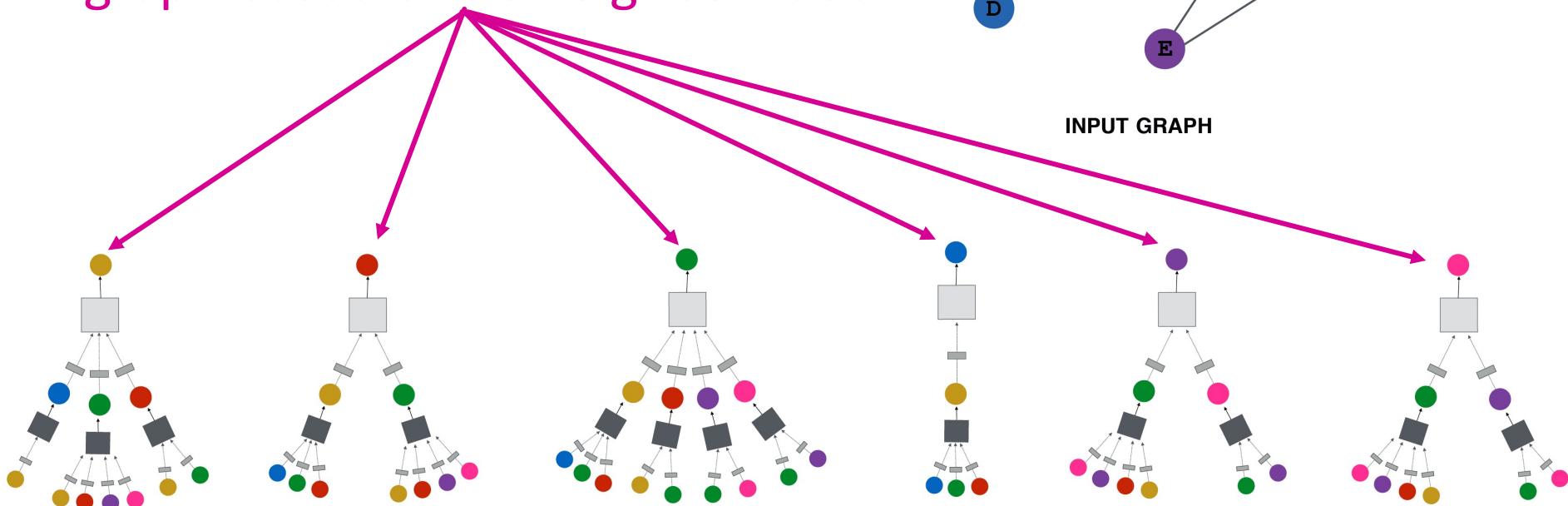
Idea: Aggregate Neighbors

- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!

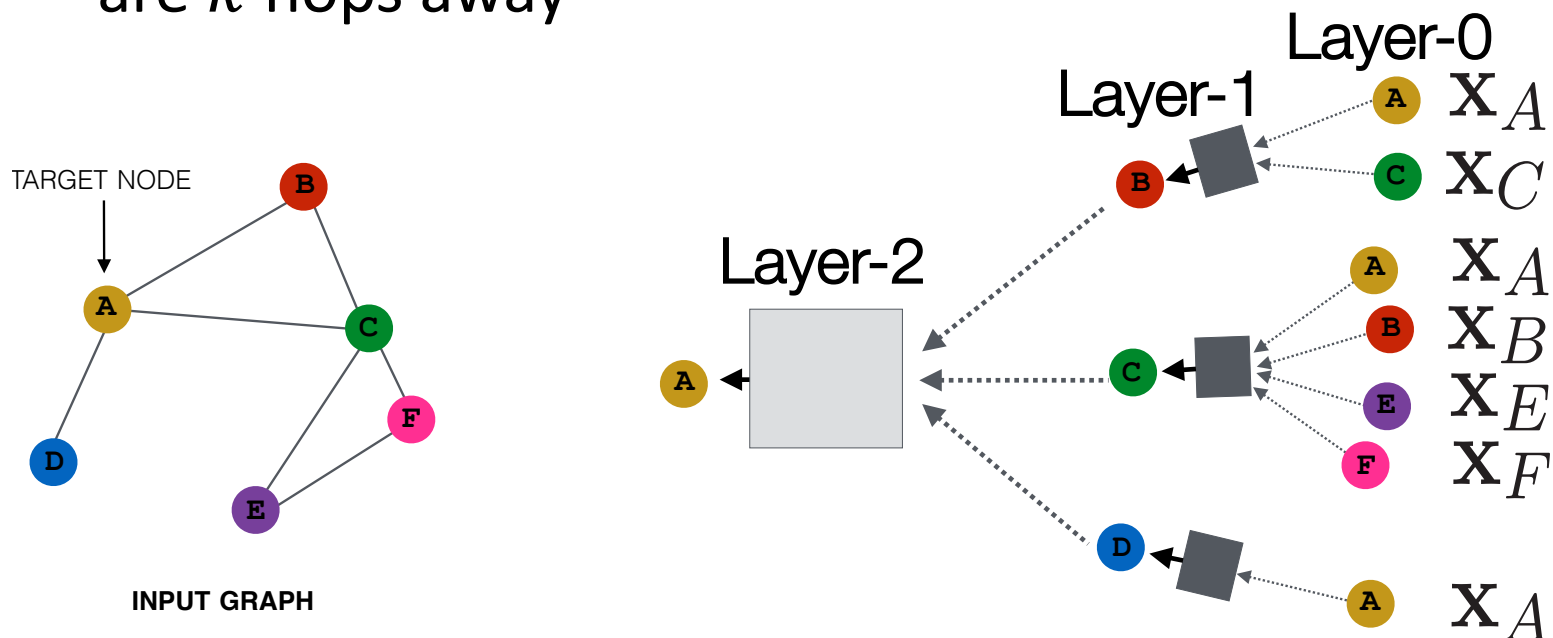


INPUT GRAPH



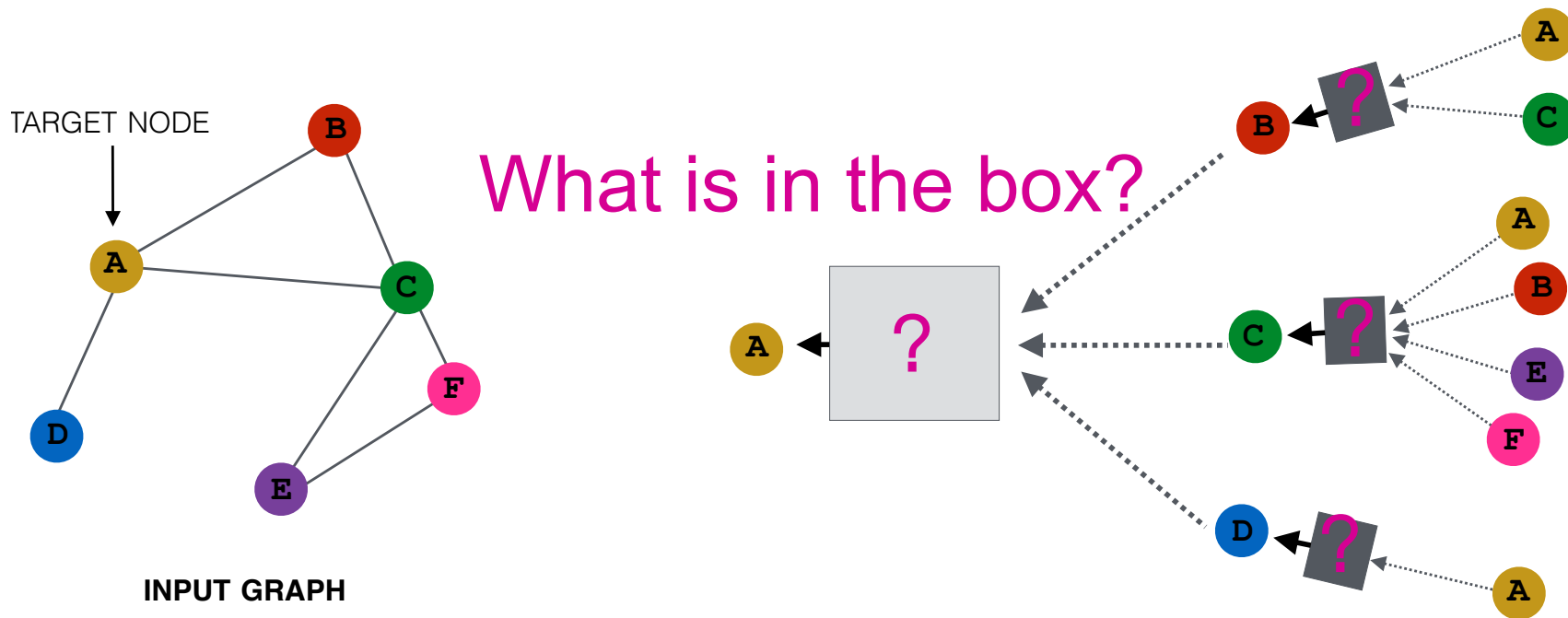
Deep Model: Many Layers

- Model can be of arbitrary depth:
 - Nodes have embeddings at each layer
 - Layer-0 embedding of node v is its input feature, x_v
 - Layer- k embedding gets information from nodes that are k hops away



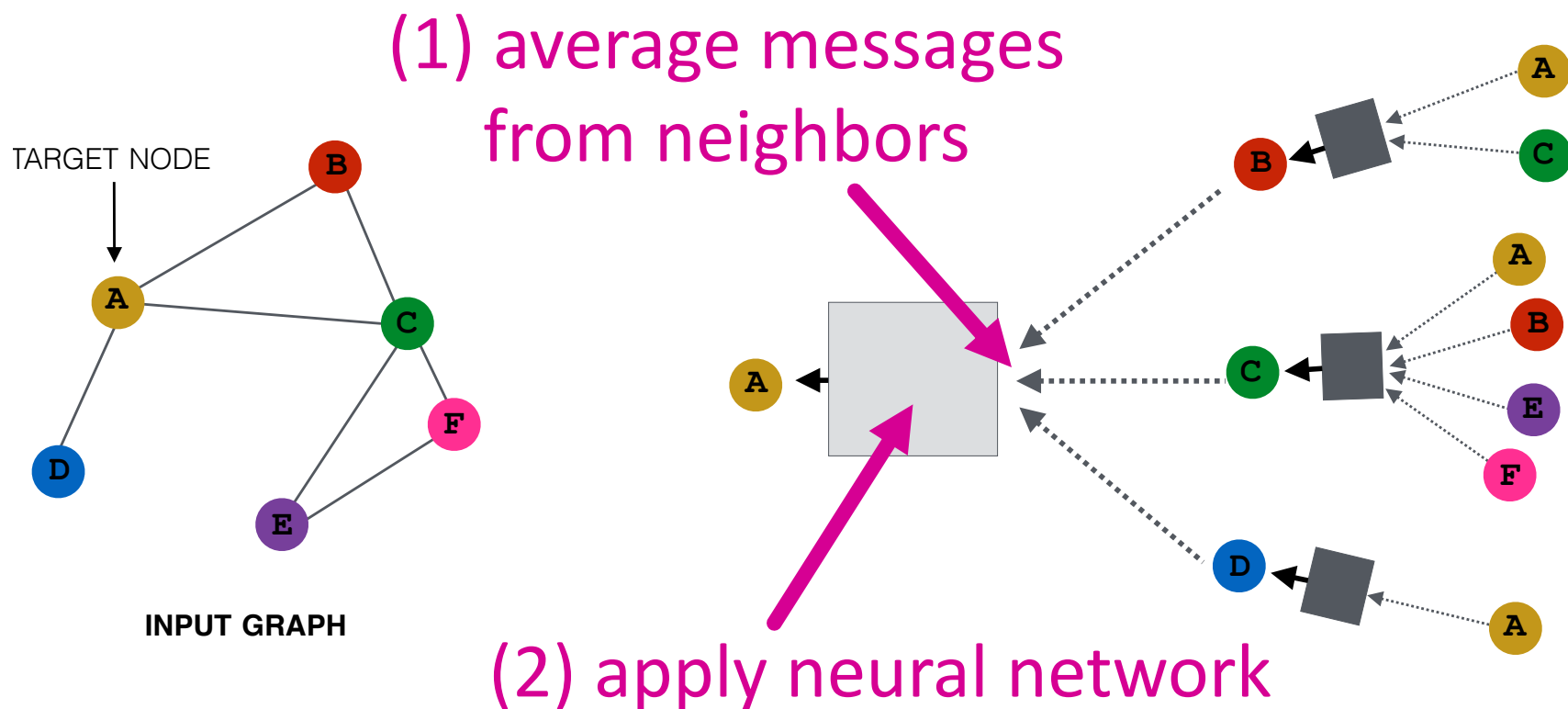
Neighborhood Aggregation

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



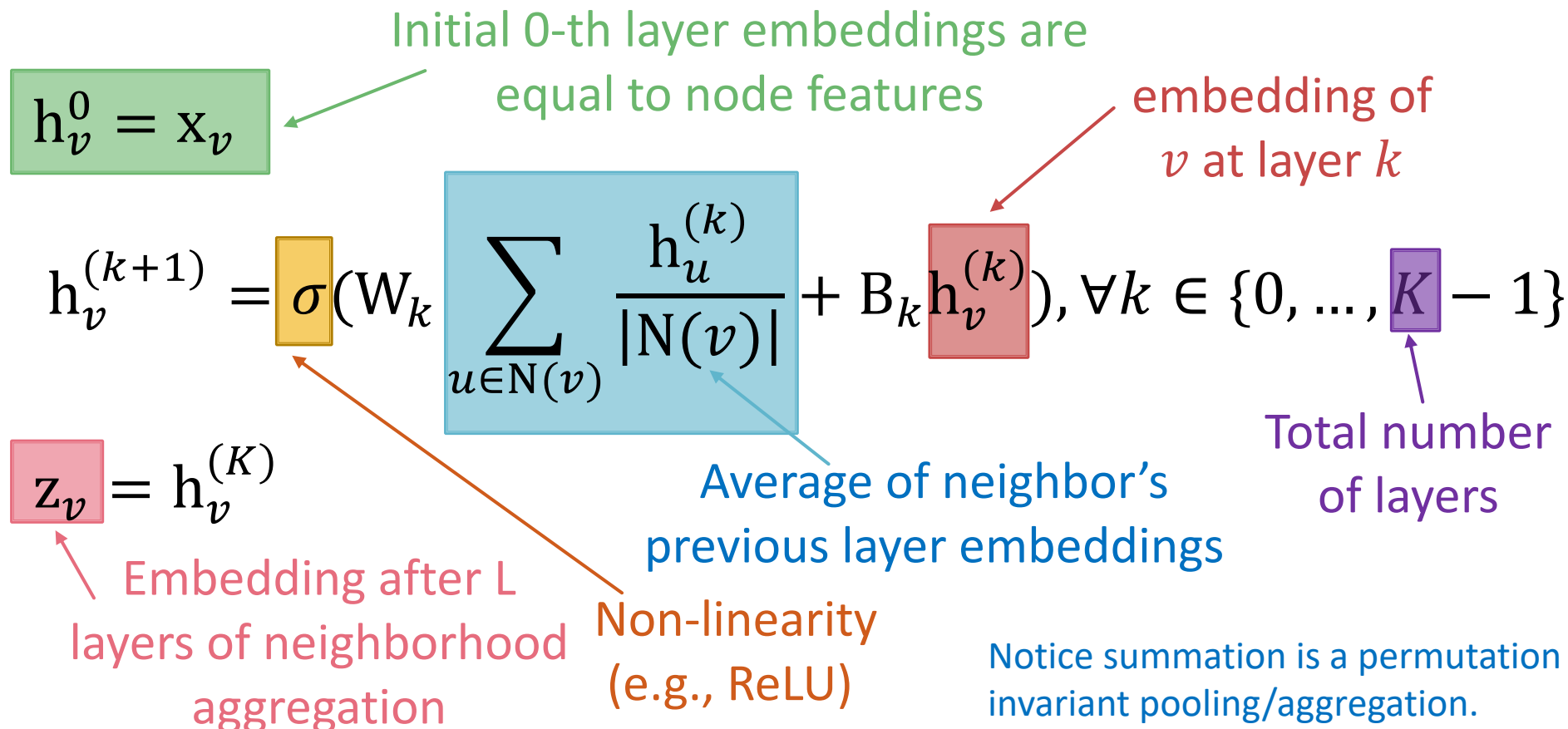
Neighborhood Aggregation

- **Basic approach:** Average information from neighbors and apply a neural network



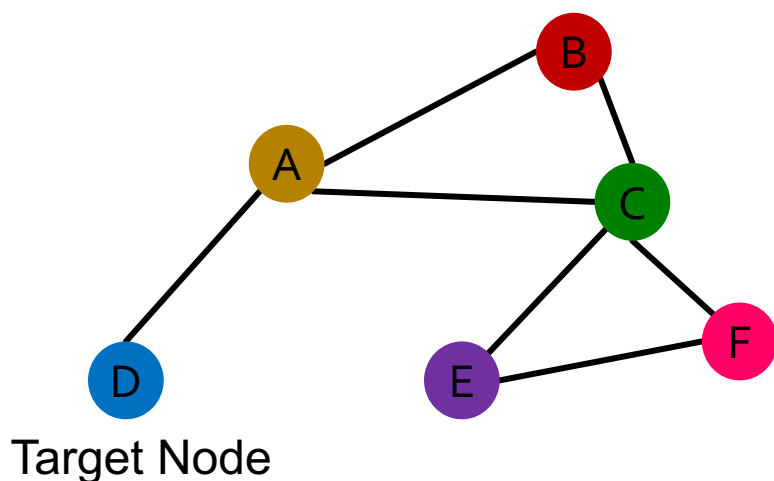
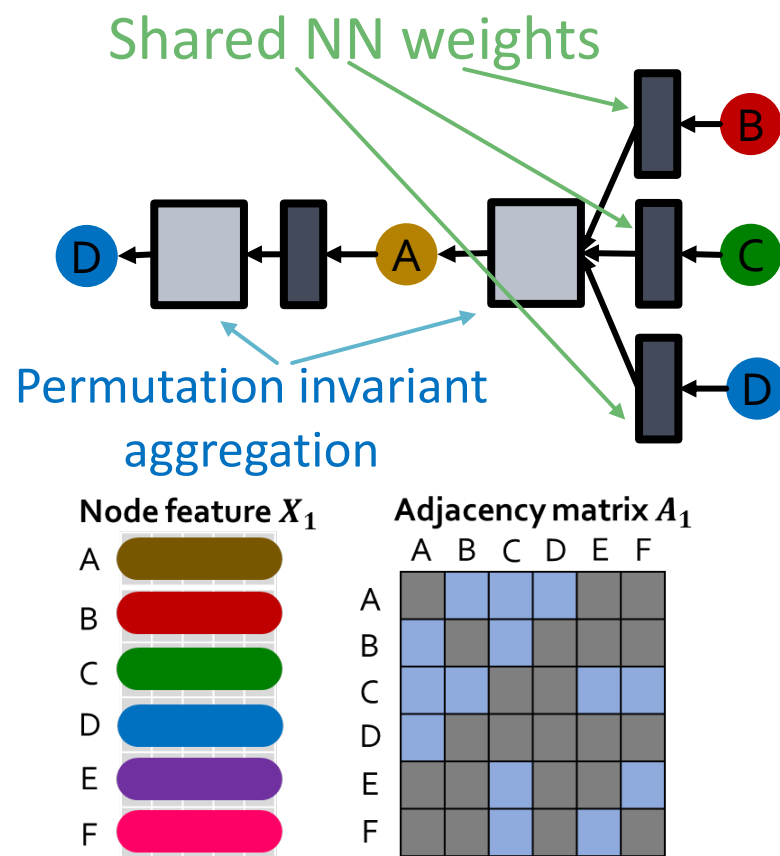
The Math: Deep Encoder

- **Basic approach:** Average neighbor messages and apply a neural network



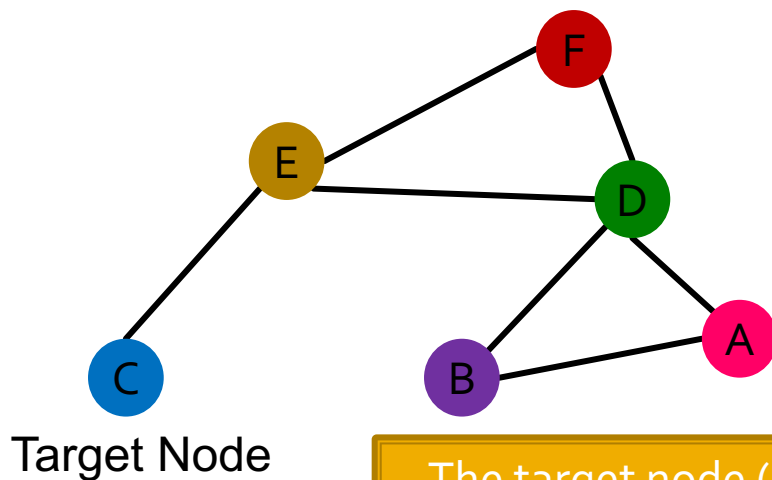
Equivariant Property

Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.

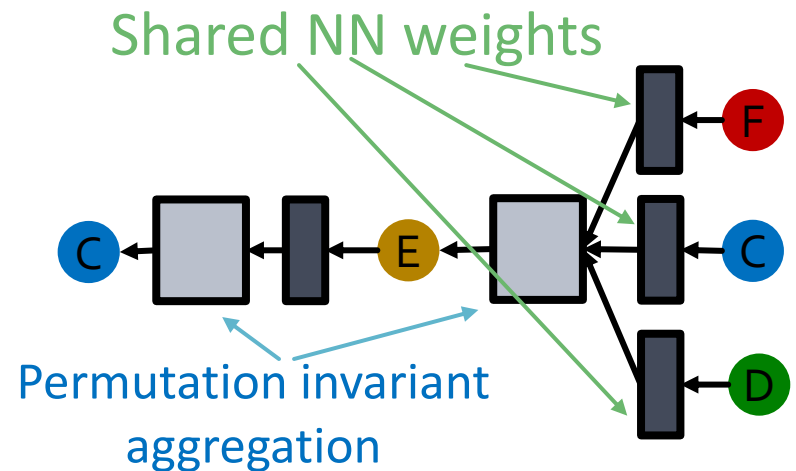


Equivariant Property

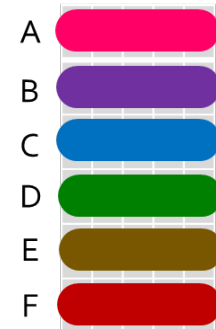
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.



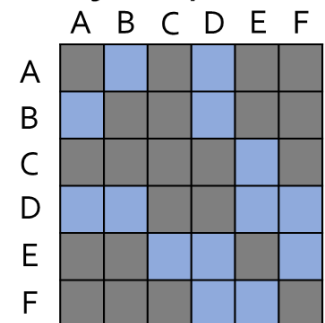
The target node (blue) has the same computation graph for different order plans



Node feature X_2

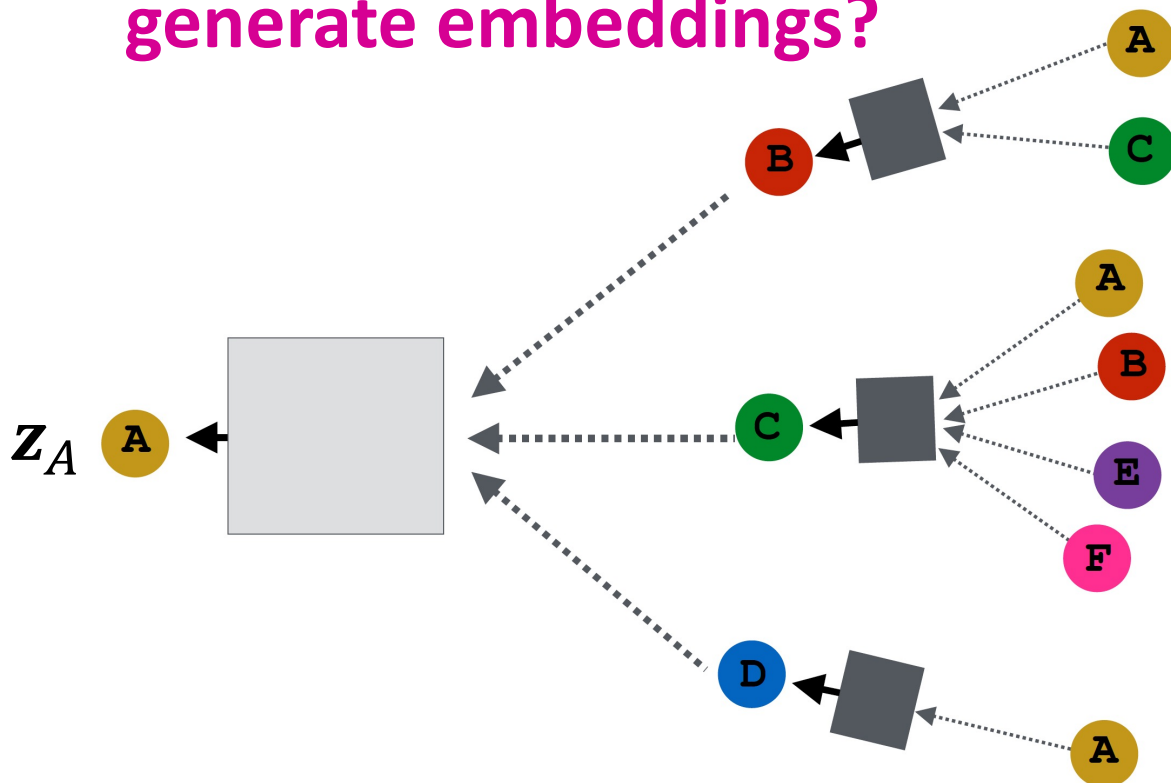


Adjacency matrix A_2



Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

Model Parameters

Trainable weight matrices
(i.e., what we learn)

$$\begin{aligned}h_v^{(0)} &= x_v \\h_v^{(k+1)} &= \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0..K-1\} \\z_v &= h_v^{(K)}\end{aligned}$$

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

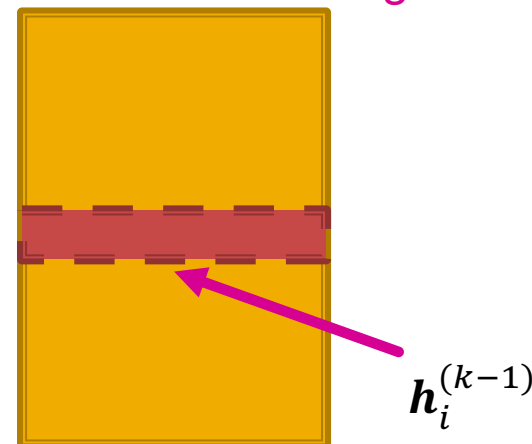
- h_v^k : the hidden representation of node v at layer k
- W_k : weight matrix for neighborhood aggregation
 - B_k : weight matrix for transforming hidden vector of self

Matrix Formulation (1)

- Many aggregations can be performed efficiently by (sparse) matrix operations

- Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$
- Then: $\sum_{u \in N_v} h_u^{(k)} = A_{v,:} H^{(k)}$
- Let D be diagonal matrix where $D_{v,v} = \text{Deg}(v) = |N(v)|$
 - The inverse of D : D^{-1} is also diagonal:
 $D_{v,v}^{-1} = 1/|N(v)|$
- Therefore,

Matrix of hidden embeddings $H^{(k-1)}$



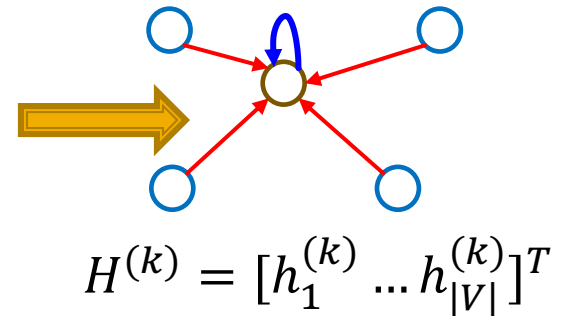
$$\boxed{\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|}} \Rightarrow H^{(k+1)} = D^{-1} A H^{(k)}$$

Matrix Formulation (2)

- Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)$$

where $\tilde{A} = D^{-1}A$



- Red: neighborhood aggregation
- Blue: self transformation
- In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)
- **Note:** not all GNNs can be expressed in matrix form, when aggregation function is complex

How to Train A GNN

- Node embedding $\mathbf{z}_v$ is a function of input graph
- **Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also Slide 15):

$$\min_{\theta} \mathcal{L}(\mathbf{y}, f(\mathbf{z}_v))$$

- $\mathbf{y}$: node label
- $\mathcal{L}$ could be L2 if $\mathbf{y}$ is real number, or cross entropy if $\mathbf{y}$ is categorical
- **Unsupervised setting**:
 - No node label available
 - **Use the graph structure as the supervision!**

Unsupervised Training

- “Similar” nodes have similar embeddings

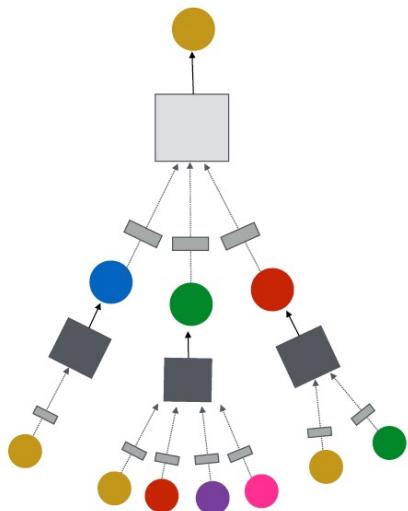
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

- Where $y_{u,v} = 1$ when node u and v are **similar**
- **CE** is the cross entropy (Slide 16)
- **DEC** is the decoder such as inner product (Lecture 4)
- **Node similarity** can be anything from Lecture 3, e.g., a loss based on:
 - **Random walks** (node2vec, DeepWalk, struc2vec)
 - **Matrix factorization**
 - **Node proximity in the graph**

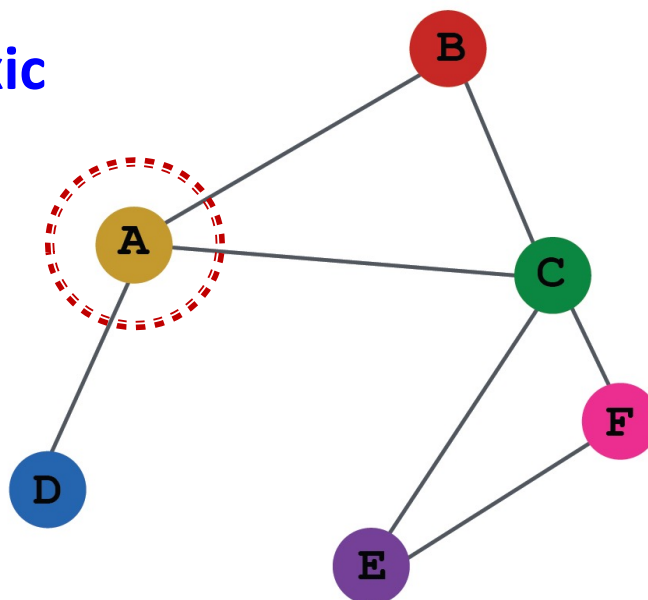
Supervised Training

Directly train the model for a supervised task (e.g., node classification)

Safe or toxic drug?



Safe or toxic drug?



E.g., a drug-drug interaction network

Supervised Training

Directly train the model for a supervised task
(e.g., **node classification**)

- Use cross entropy loss (Slide 16)

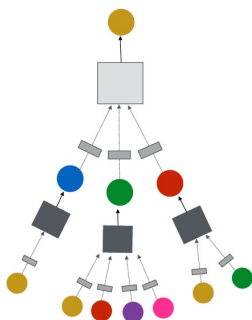
$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output:
node embedding

Classification
weights

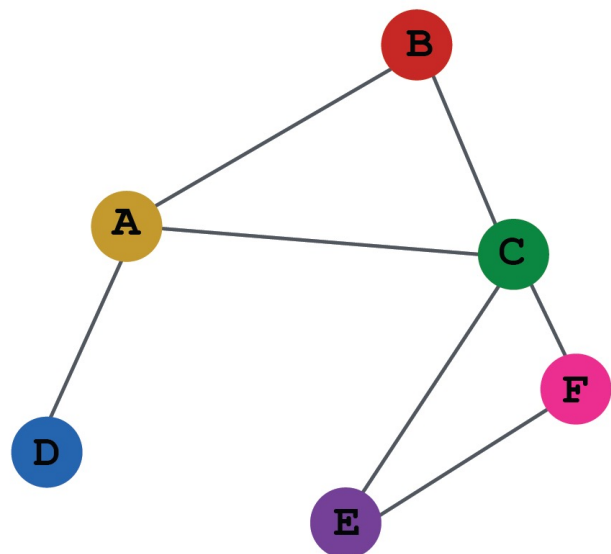
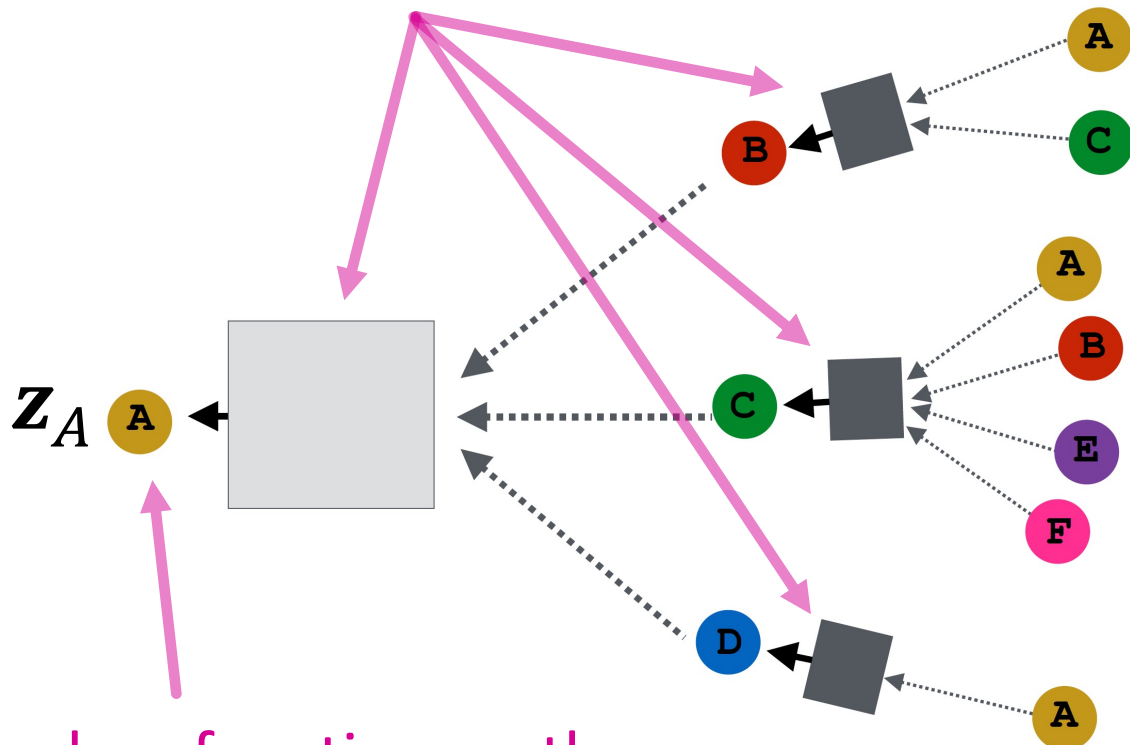
Node class
label

Safe or toxic drug?



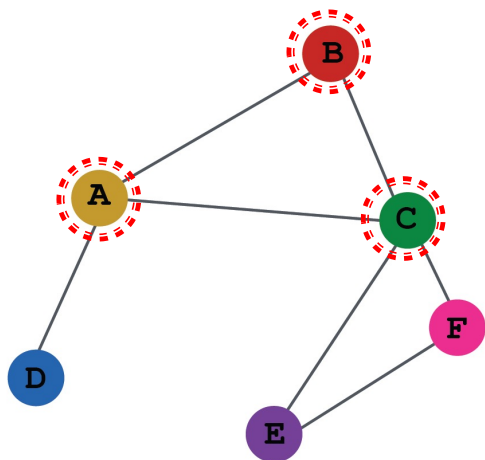
Model Design: Overview

(1) Define a neighborhood aggregation function



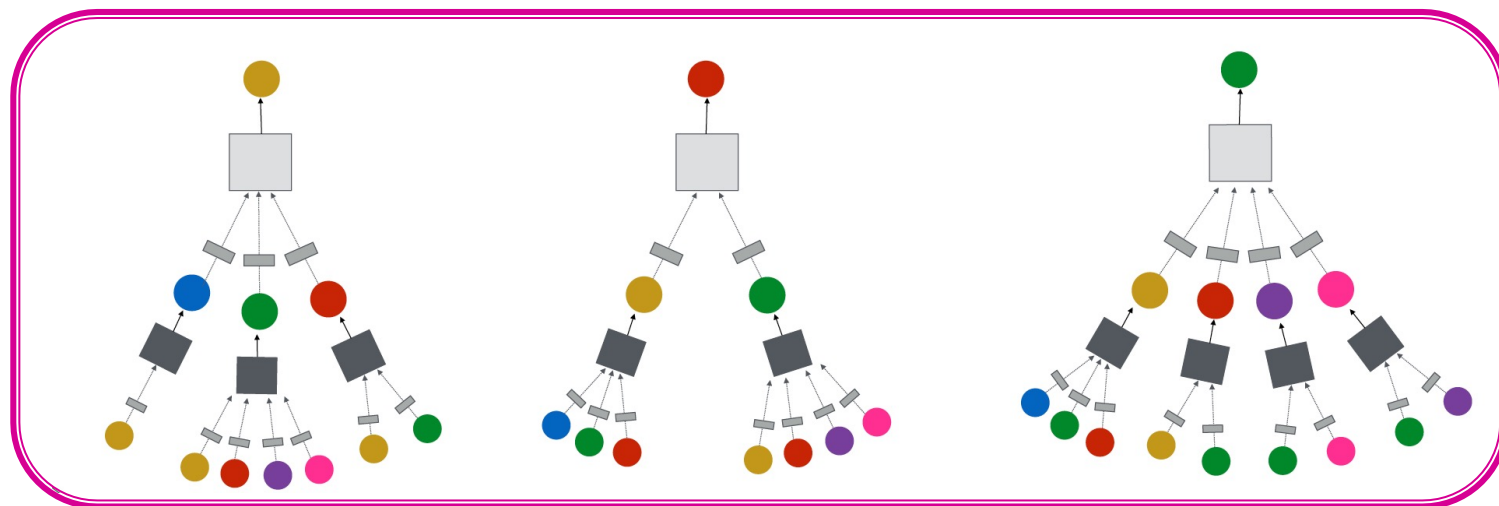
(2) Define a loss function on the embeddings

Model Design: Overview

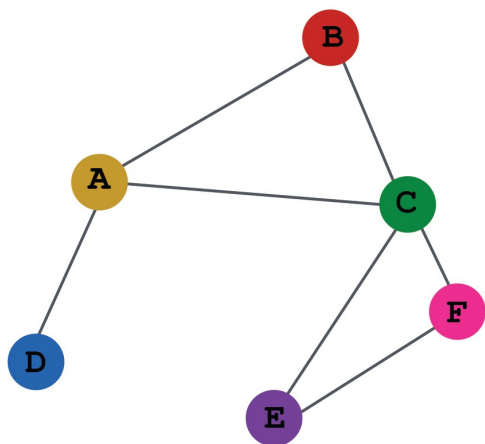


INPUT GRAPH

(3) Train on a set of nodes, i.e.,
a batch of compute graphs



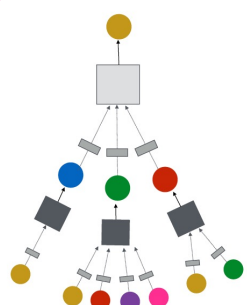
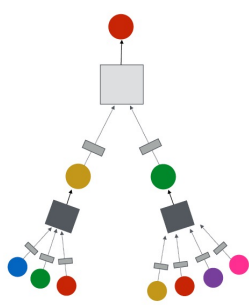
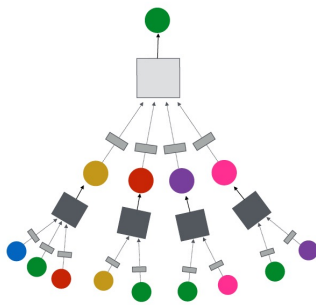
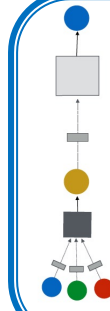
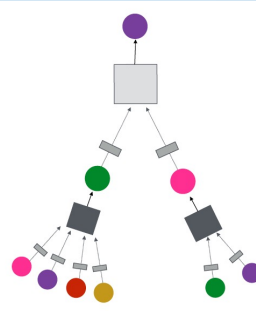
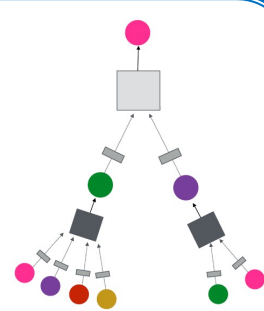
Model Design: Overview



INPUT GRAPH

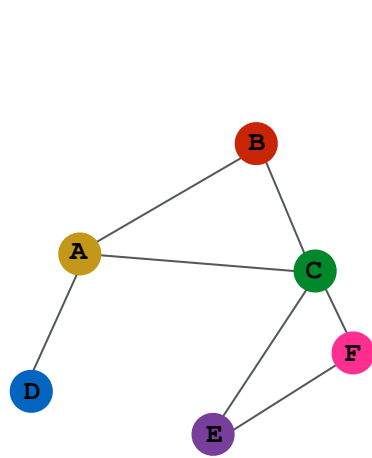
(4) Generate embeddings
for nodes as needed

Even for nodes we never
trained on!

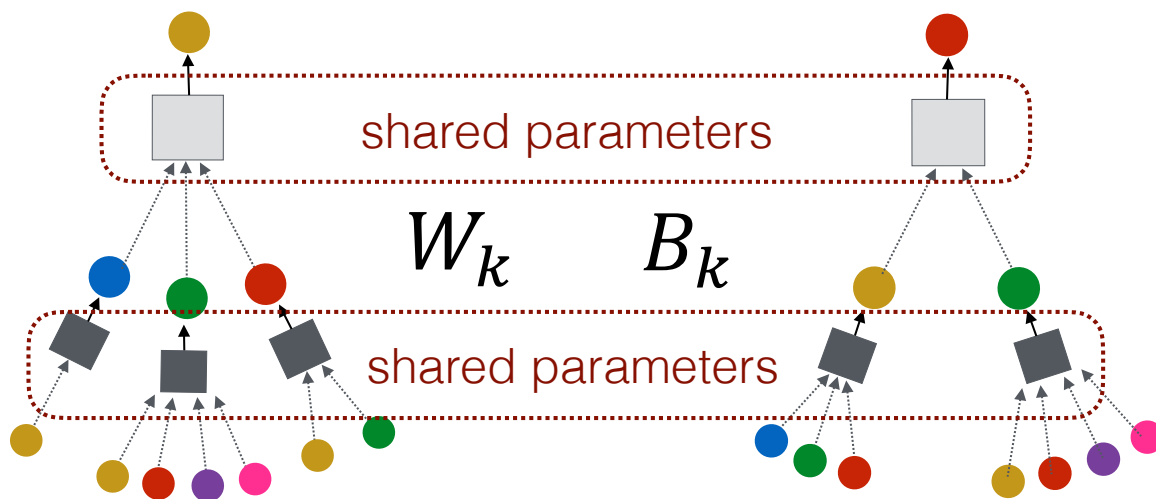


Inductive Capability

- The same aggregation parameters are shared for all nodes:
 - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes!**



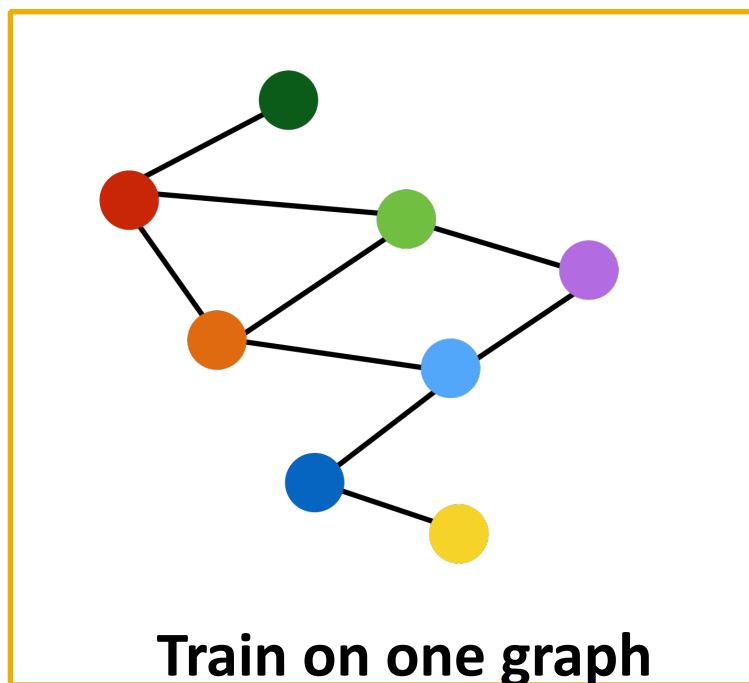
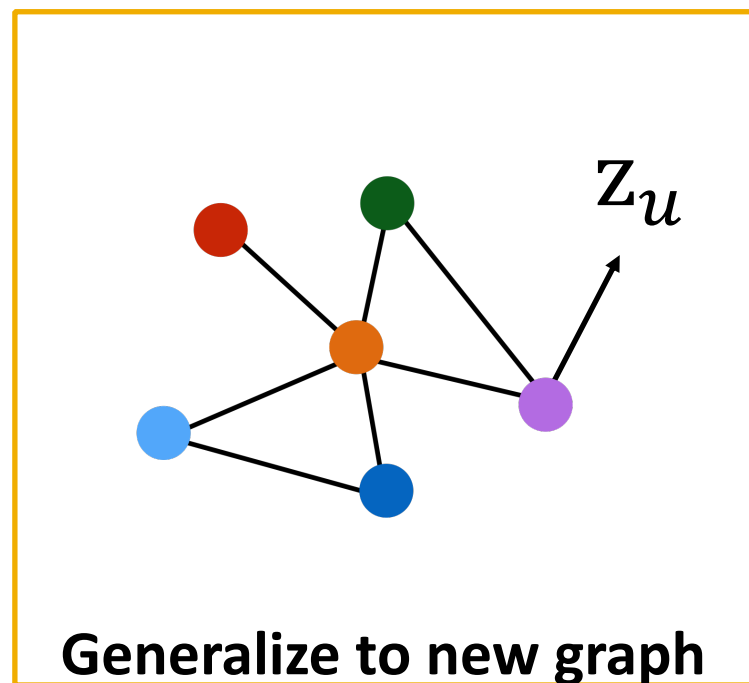
INPUT GRAPH



Compute graph for node A

Compute graph for node B

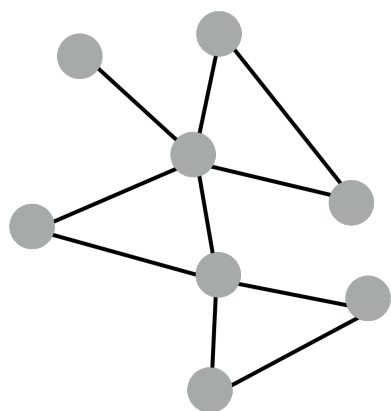
Inductive Capability: New Graphs



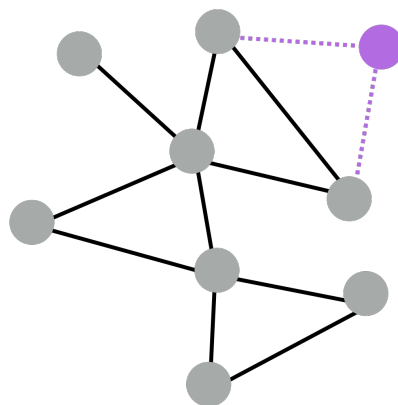
Inductive node embedding → Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

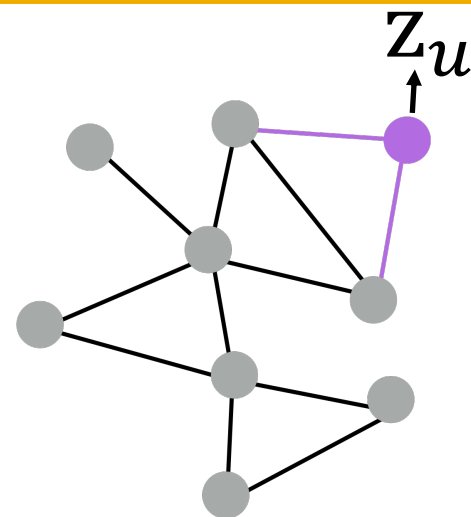
Inductive Capability: New Nodes



Train with snapshot



New node arrives



Generate embedding
for new node

- Many application settings constantly encounter previously unseen nodes:
 - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings “on the fly”

Outline of Today's Lecture

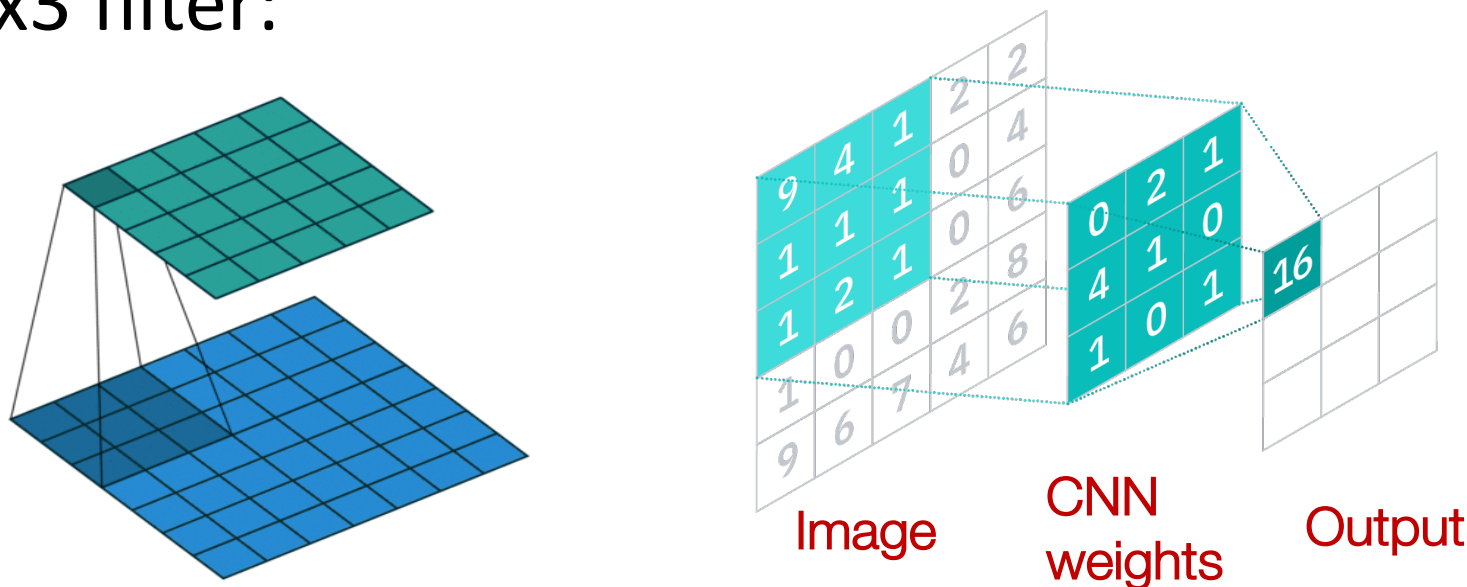
1. Basics of deep learning ✓
2. Deep learning for graphs ✓
3. Graph Convolutional Networks ✓
4. GNNs subsume CNNs and Transformers 

Architecture Comparison

- **How does GNNs compare to prominent architectures such as Convolutional Neural Nets, and Transformers?**

Convolutional Neural Network

Convolutional neural network (CNN) layer with 3x3 filter:

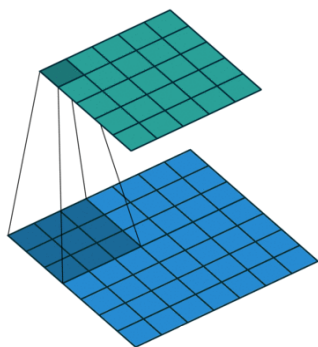


$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$$

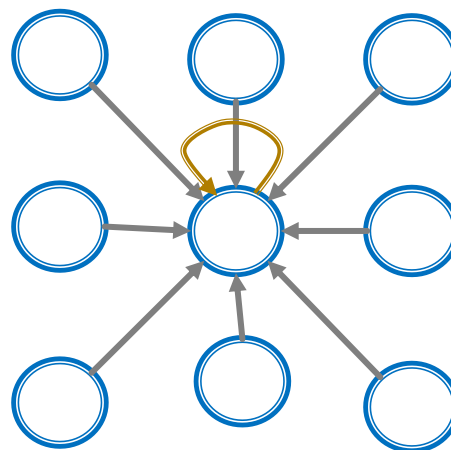
$N(v)$ represents the 8 neighbor pixels of v .

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

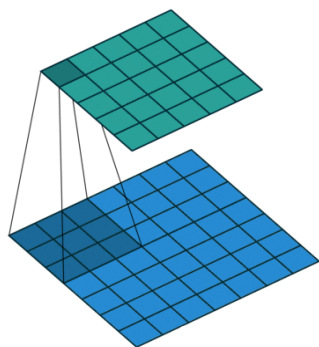
- GNN formulation (previous slide): $h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$
- CNN formulation:
if we rewrite:

$$h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \forall l \in \{0, \dots, L-1\}$$

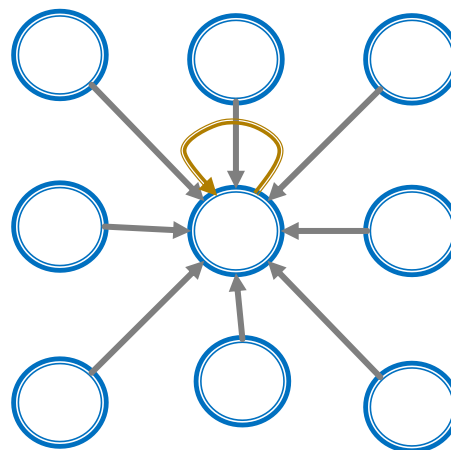
$$h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

$$\text{GNN formulation: } h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

Key difference: We can learn different $\mathbf{W}_l^u$ for different “neighbor” u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



- CNN can be seen as a special GNN with fixed neighbor size and ordering:
 - The size of the filter is pre-defined for a CNN.
 - The advantage of GNN is it processes arbitrary graphs with different degrees for each node.

Key difference: We can learn different W_l^u for different “neighbor” u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:

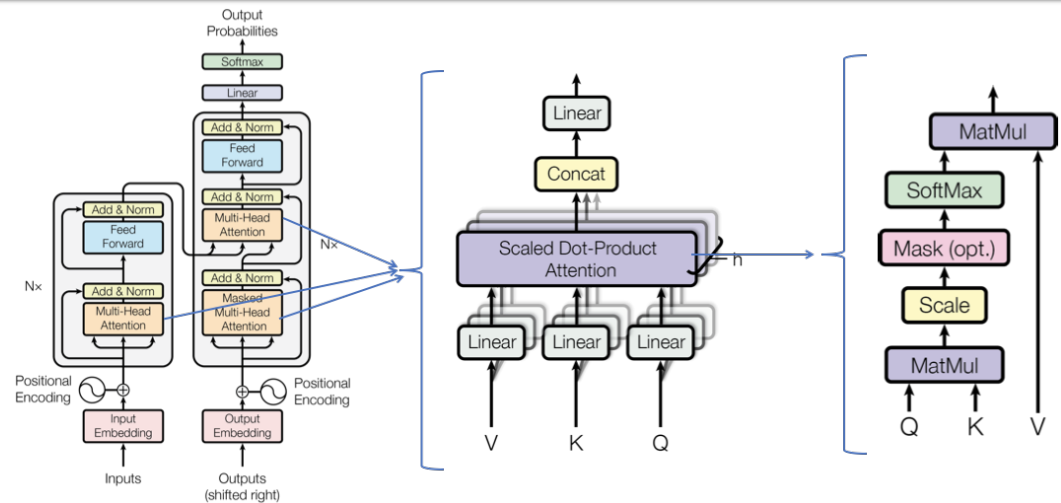


- CNN can be seen as a special GNN with fixed neighbor size and ordering.
- CNN is not permutation equivariant.
 - Switching the order of pixels will leads to different outputs.

Key difference: We can learn different W_l^u for different “neighbor” u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: $\{(-1,-1), (-1,0), (-1,1), \dots, (1,1)\}$

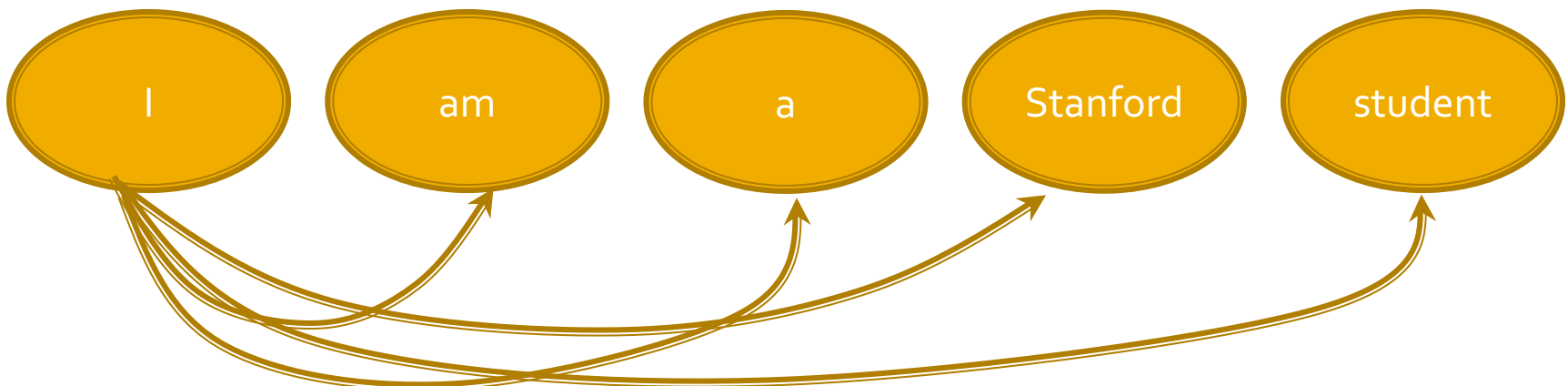
Transformer

Transformer is one of the most popular architectures that achieves great performance in many sequence modeling tasks.



Key component: self-attention

- Every token/word attends to all the other tokens/words via matrix calculation.



GNN vs. Transformer

Transformer layer can be seen as a special GNN that runs on a fully-connected “word” graph!

Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected “word” graph**.

