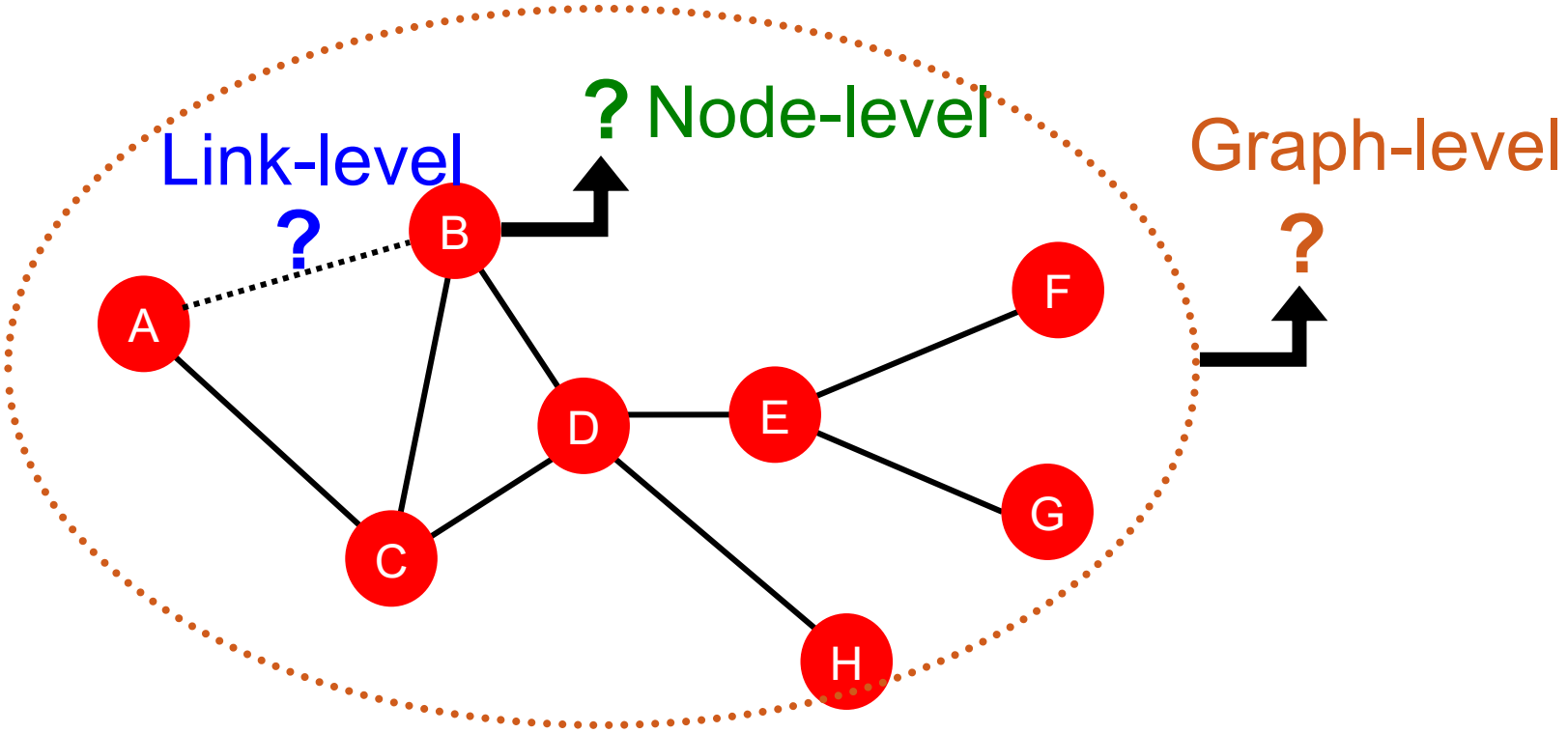
Stanford CS224W: Traditional Methods for Machine Learning in Graphs

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

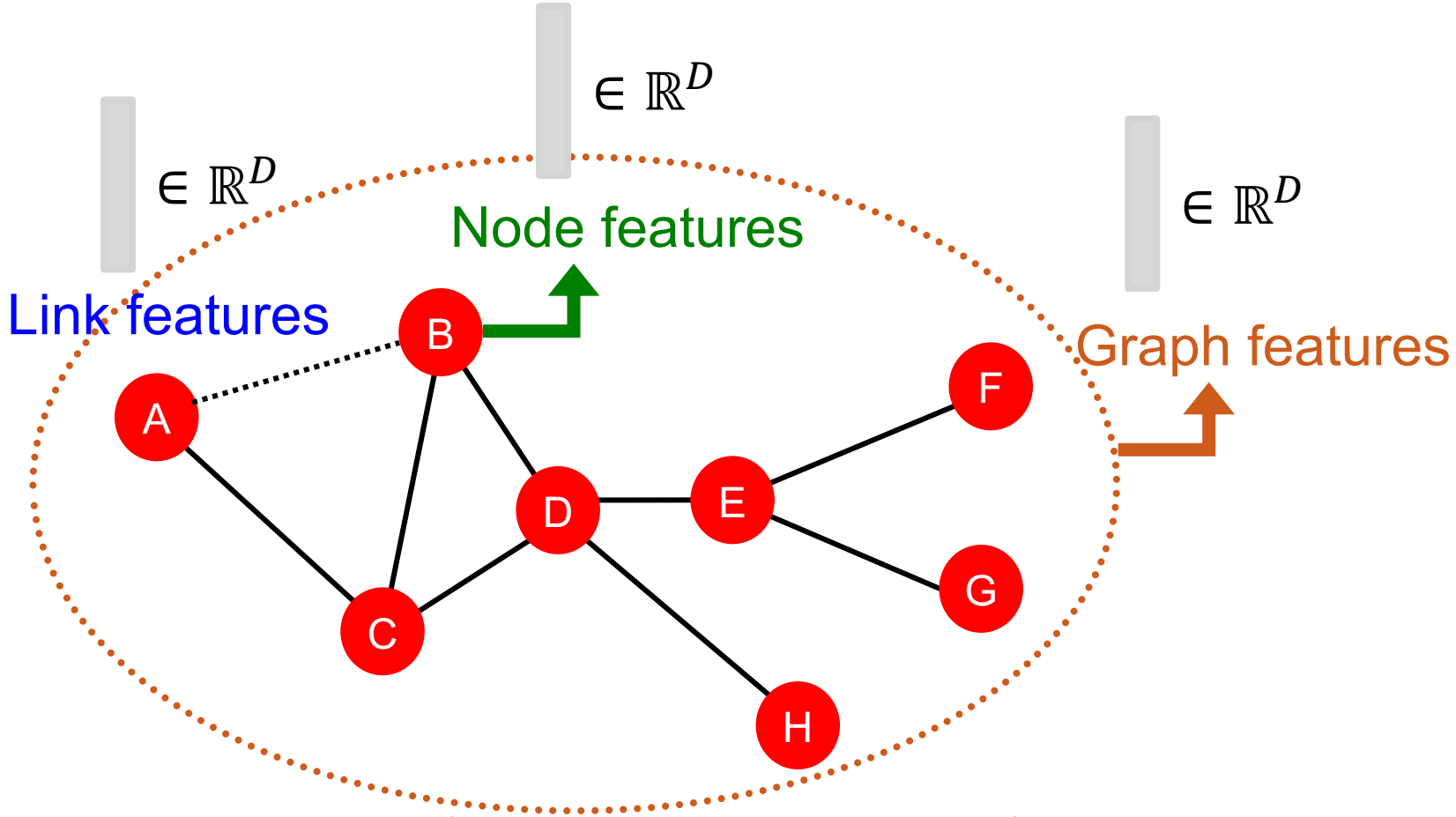
Machine Learning Tasks: Review

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



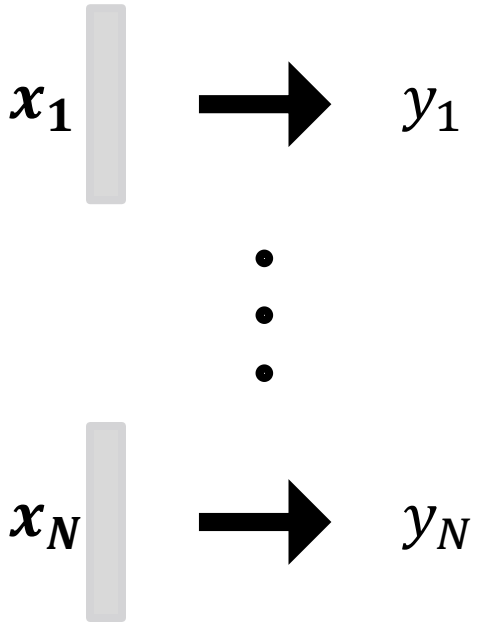
Traditional ML Pipeline

- Design features for nodes/links/graphs
- Obtain features for all training data

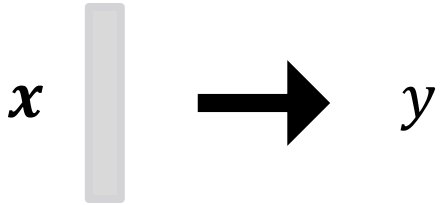


Traditional ML Pipeline

- **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 test 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?

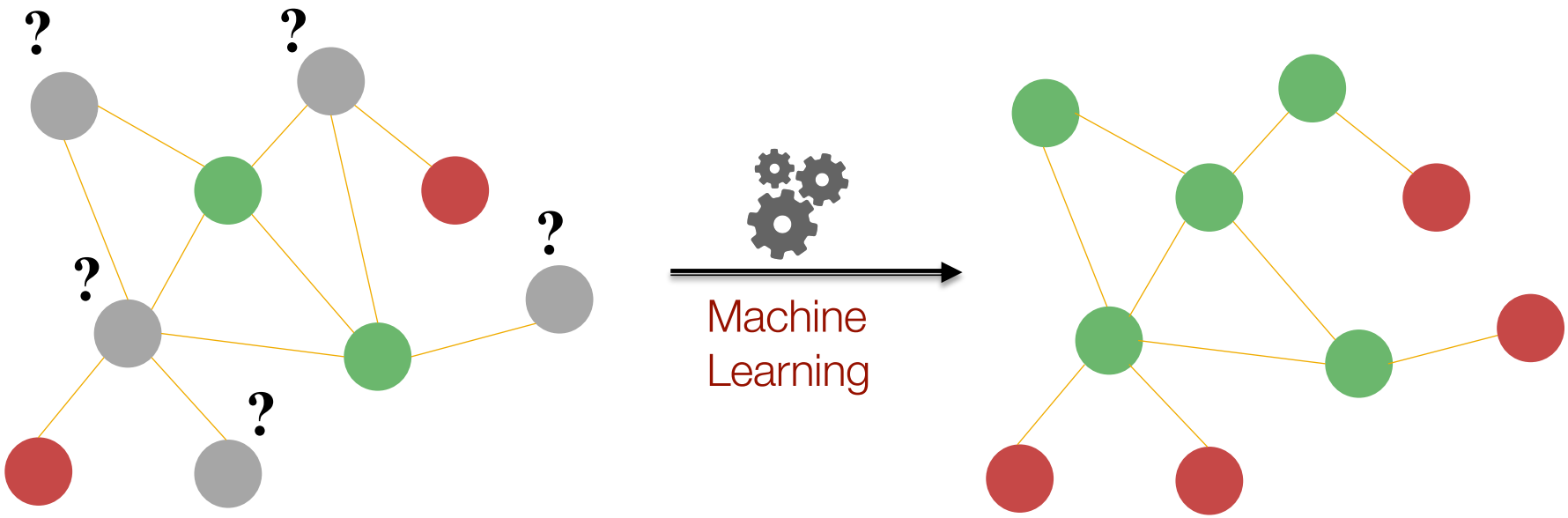
Stanford CS224W: Node-level Tasks and Features

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Node-level Tasks



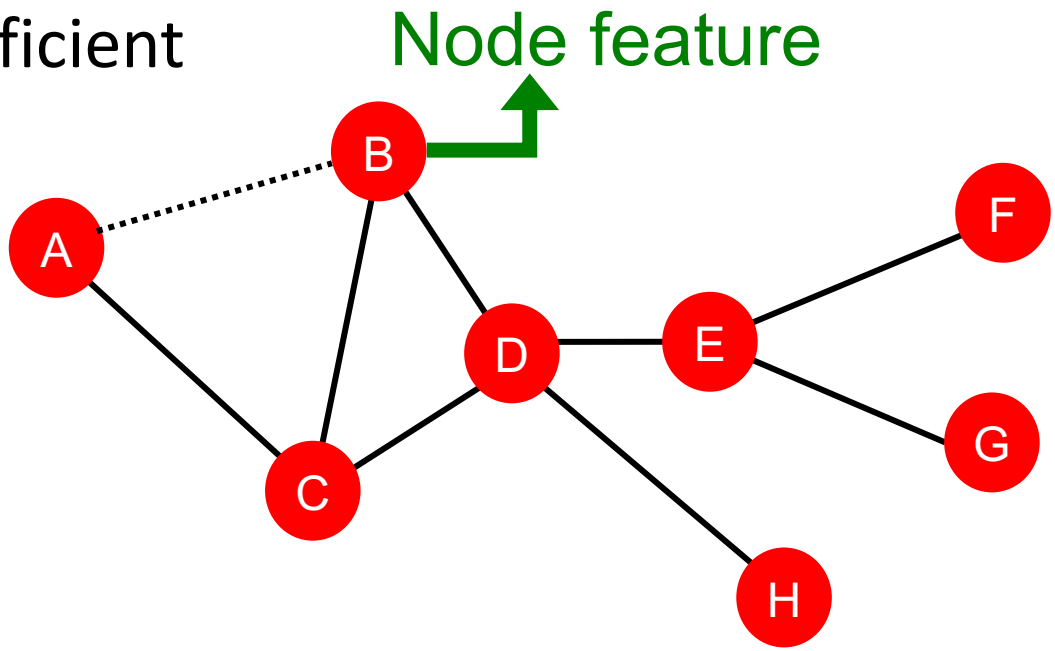
Node classification

ML needs features.

Node-Level Features: Overview

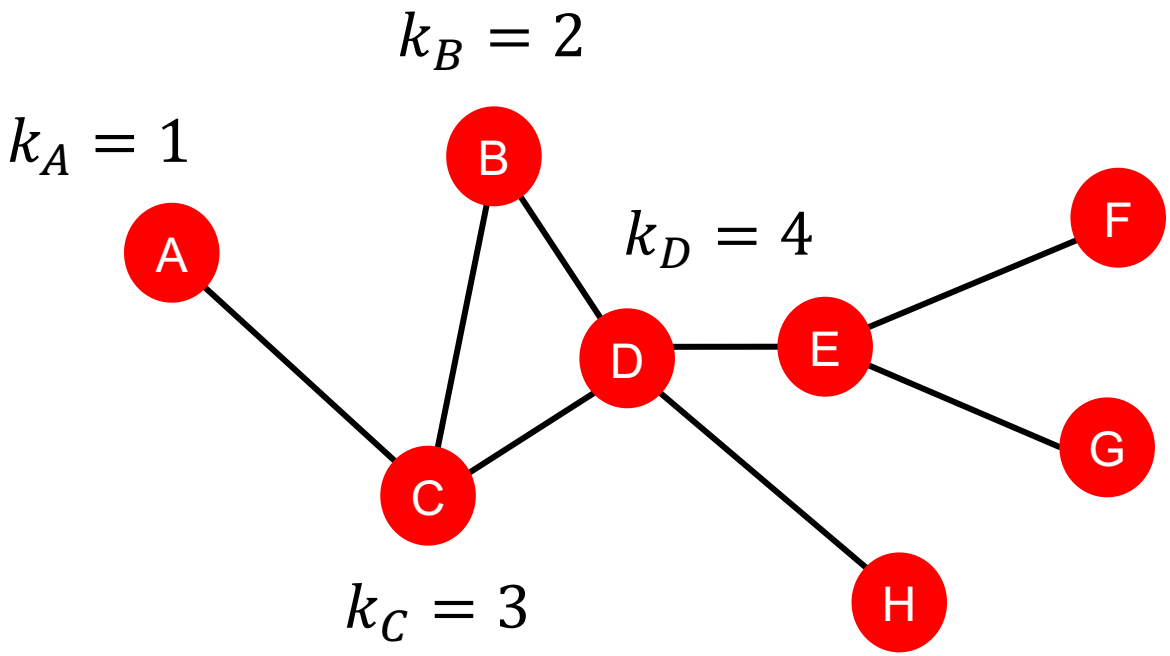
Goal: Characterize the structure and position of a node in the network:

- 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:**
  - Engienvector 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.

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

λ is normalization const (largest eigenvalue of A)

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

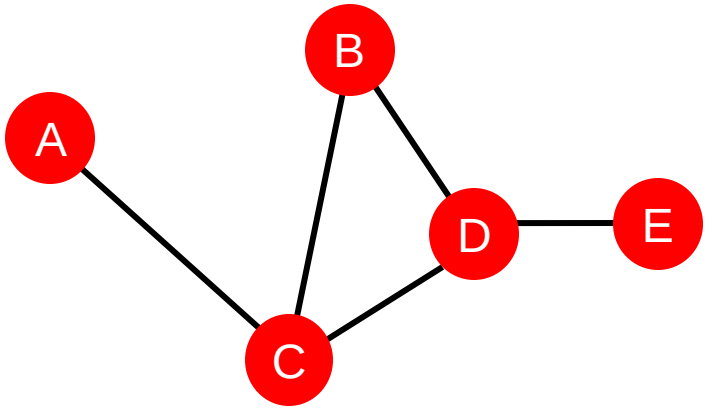
  - We see that centrality c is the **eigenvector of $\boldsymbol{A}$**!
  - The largest eigenvalue λ_{max} is always positive and unique (by Perron-Frobenius Theorem).
  - The eigenvector $\boldsymbol{c}_{max}$ corresponding to λ_{max} is used for centrality.

Node Centrality (2)

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

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

  - **Example:**



$c_A = c_B = c_E = 0$

$c_C = 3$

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

$c_D = 3$

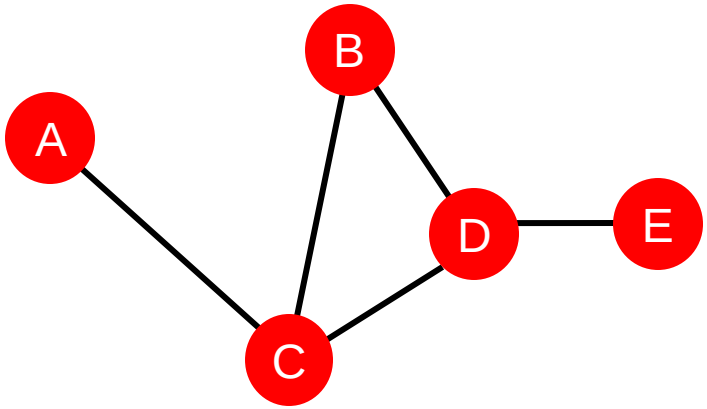
(A-C-D-E, B-D-E, C-D-E)

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

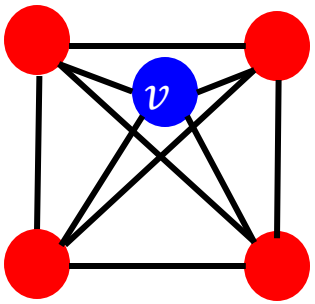
- Measures how connected $v's$ neighboring nodes are:

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

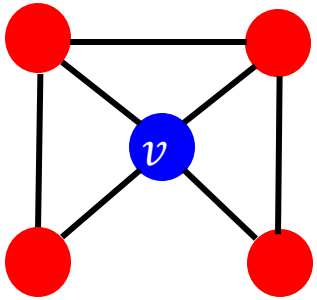
$\#(\text{node pairs among } k_v \text{ neighboring nodes})$

In our examples below the denominator is 6 (4 choose 2).

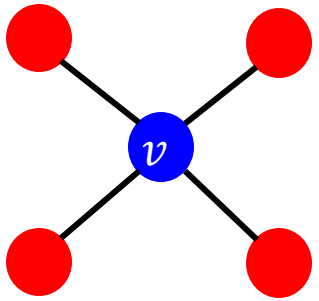
- **Examples:**



$e_v = 1$



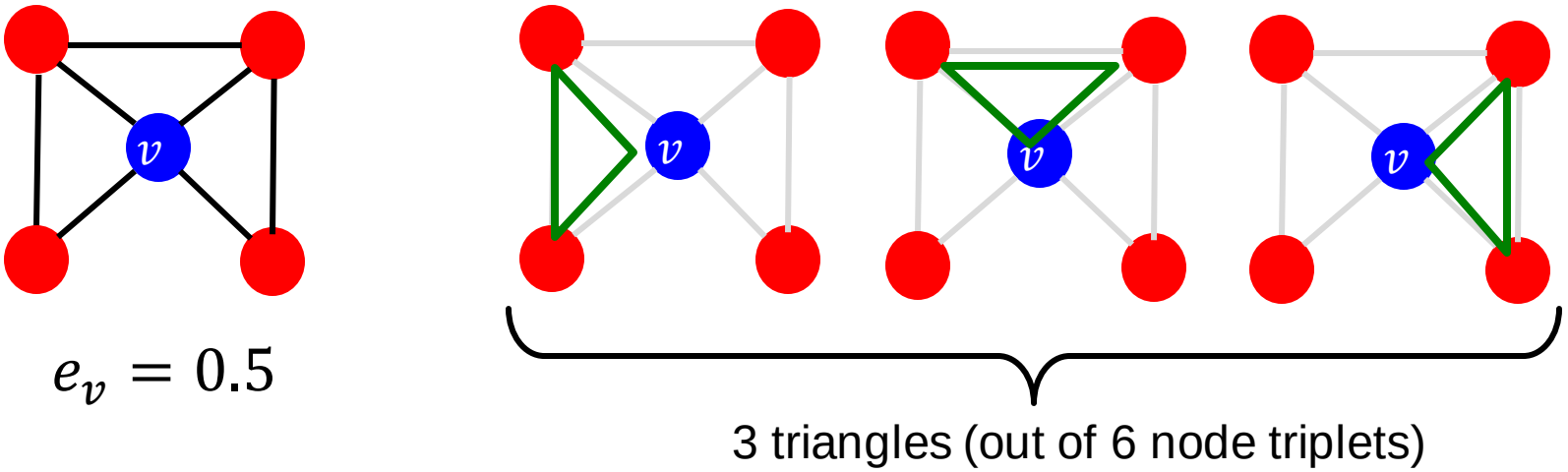
$e_v = 0.5$



$e_v = 0$

Node Features: Graphlets

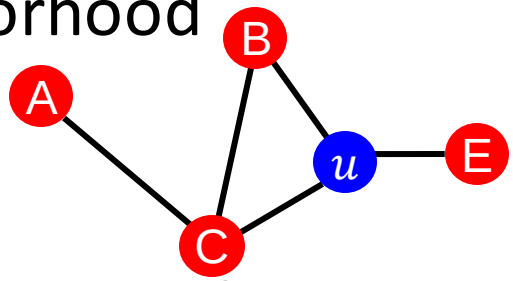
- **Observation:** Clustering coefficient counts the #(triangles) in the ego-network



- 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



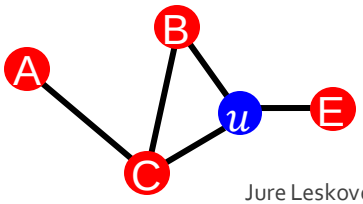
Analogy:

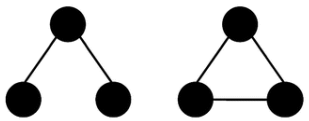
- **Degree** counts **#(edges)** that a node touches
- **Clustering coefficient** counts **#(triangles)** that a node touches.
- **Graphlet Degree Vector (GDV):** Graphlet-base features for nodes
  - **GDV** counts **#(graphlets)** that a node touches

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.

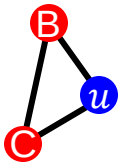




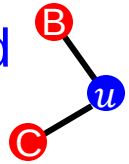
Induced Subgraph & Isomorphism

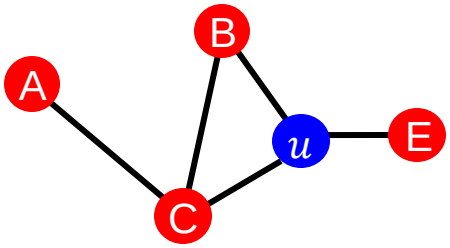
- **Def: Induced subgraph** is another graph, formed from a subset of vertices and *all* of the edges connecting the vertices in that subset.

Induced subgraph:

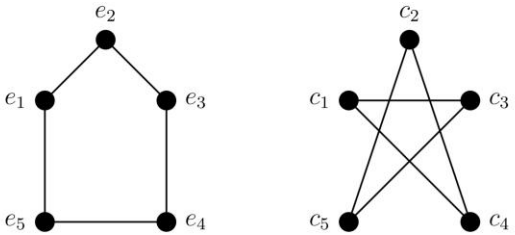


Not induced subgraph:

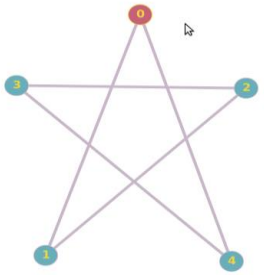


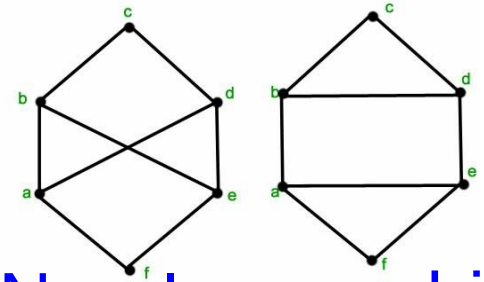


- **Def: Graph Isomorphism**
  - Two graphs which contain the same number of nodes connected in the same way are said to be isomorphic.



Isomorphic





Non-Isomorphic

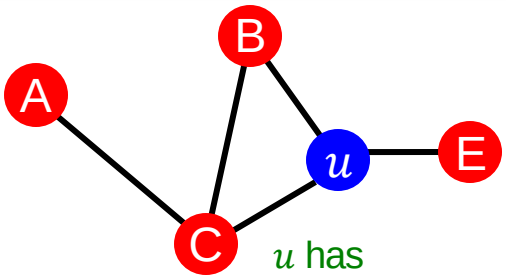
Node mapping: (e2,c2), (e1, c5), (e3,c4), (e5,c3), (e4,c1)

Source: Mathoverflow

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

Node Features: Graphlets

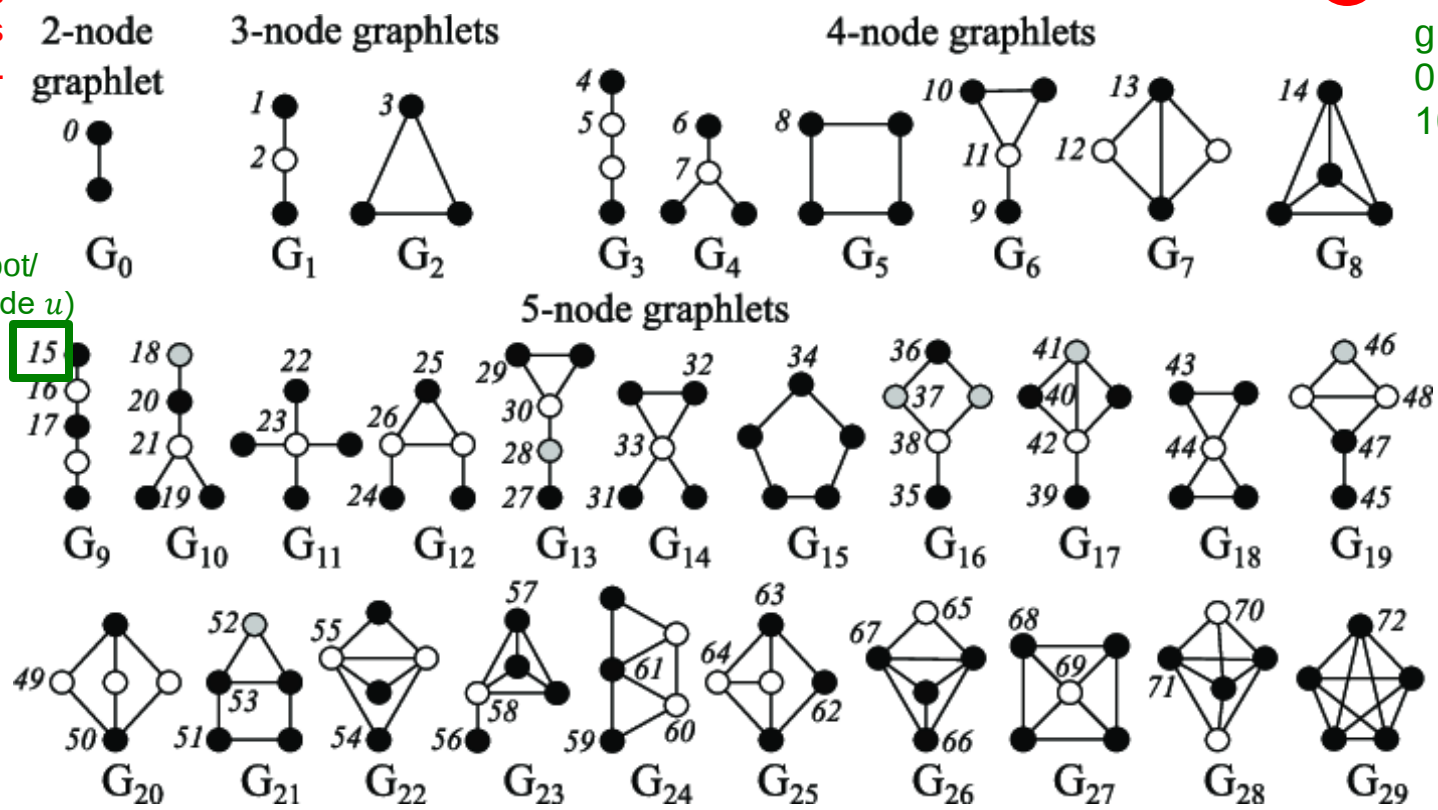
Graphlets: Rooted connected induced **non-isomorphic** subgraphs:



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

Take some nodes and all the edges between them.

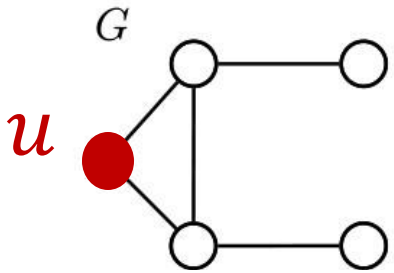
Graphlet id (Root/ "position" of node u)



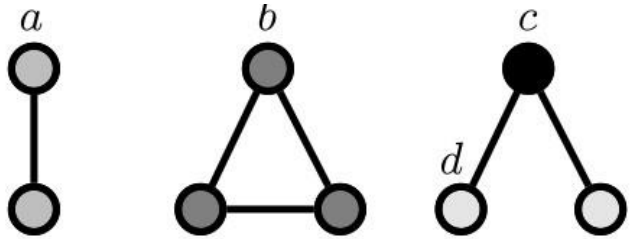
There are 73 different graphlets on up to 5 nodes

Node Features: Graphlets

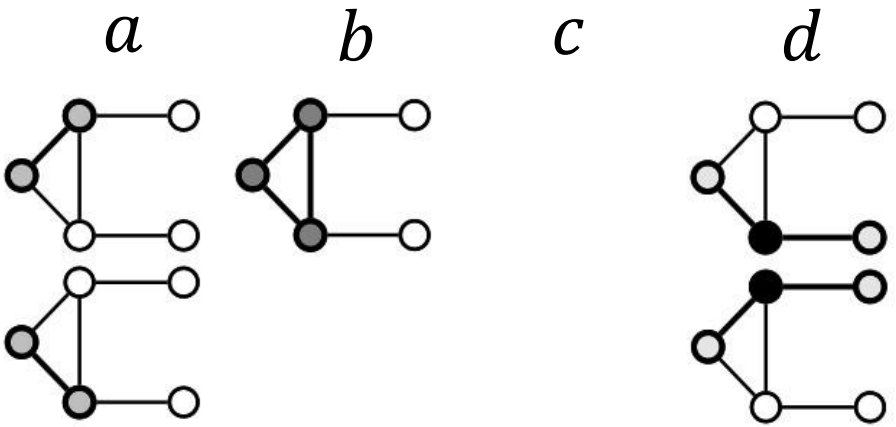
- **Graphlet Degree Vector (GDV):** A count vector of graphlets rooted at a given node.
- **Example:**



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

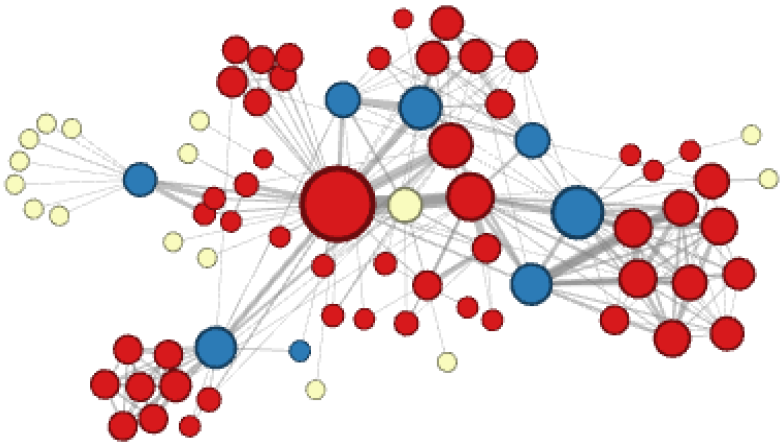
- **Importance-based features**: capture the importance of a node in a graph
  - Node degree:
    - Simply counts the number of neighboring nodes
  - Node centrality:
    - Models importance of neighboring nodes in a graph
    - Different modeling choices: eigenvector centrality, betweenness centrality,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

Node-Level Feature: Summary

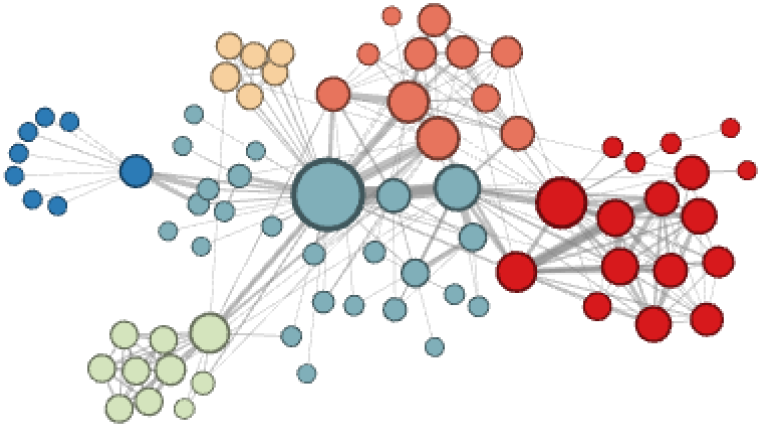
- **Structure-based features**: Capture topological properties of local neighborhood around a node.
  - **Node degree:**
    - Counts the number of neighboring nodes
  - **Clustering coefficient:**
    - Measures how connected neighboring nodes are
  - **Graphlet degree vector:**
    - Counts the occurrences 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.

Discussion

Different ways to label nodes of the network:



Node features defined so far would allow to distinguish nodes in the above example



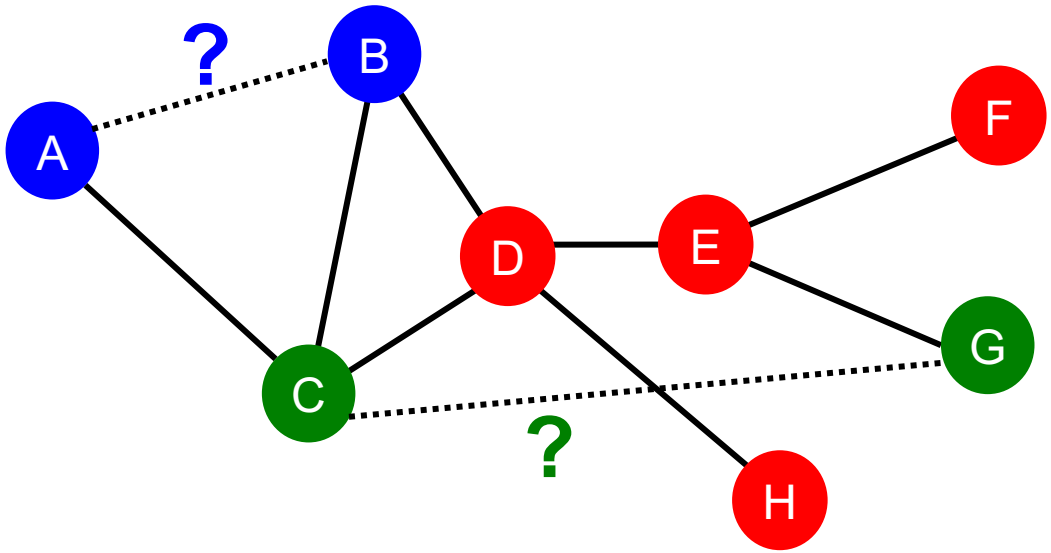
However, the features defines so far would not allow for distinguishing the above node labelling

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

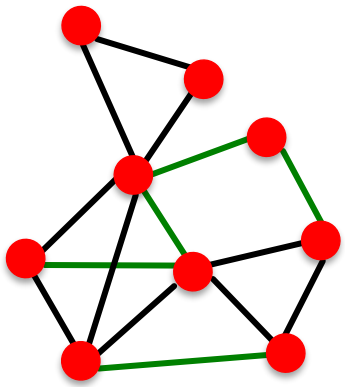
- The task is to predict **new links** based on existing links.
- At test time, all node pairs (no existing links) are ranked, and top K node pairs are predicted.
- The key is to design features for **a pair of nodes**.



Link Prediction as a Task

Two formulations of the link prediction task:

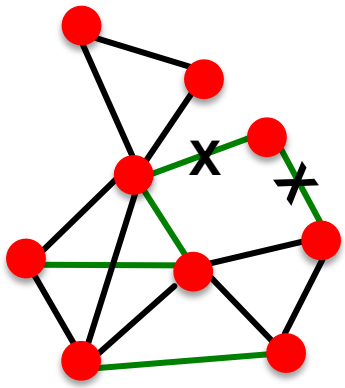
- **1) Links missing at random:**
  - Remove a random set of links and then aim to predict them
- **2) Links over time:**
  - Given $G[t_0, t_0']$ a graph defined by edges up to time t_0', **output a ranked list L** of edges (not in $G[t_0, t_0']$) that are predicted to appear in time $G[t_1, t_1']$
  - **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



$G[t_0, t_0']$
$G[t_1, t_1']$

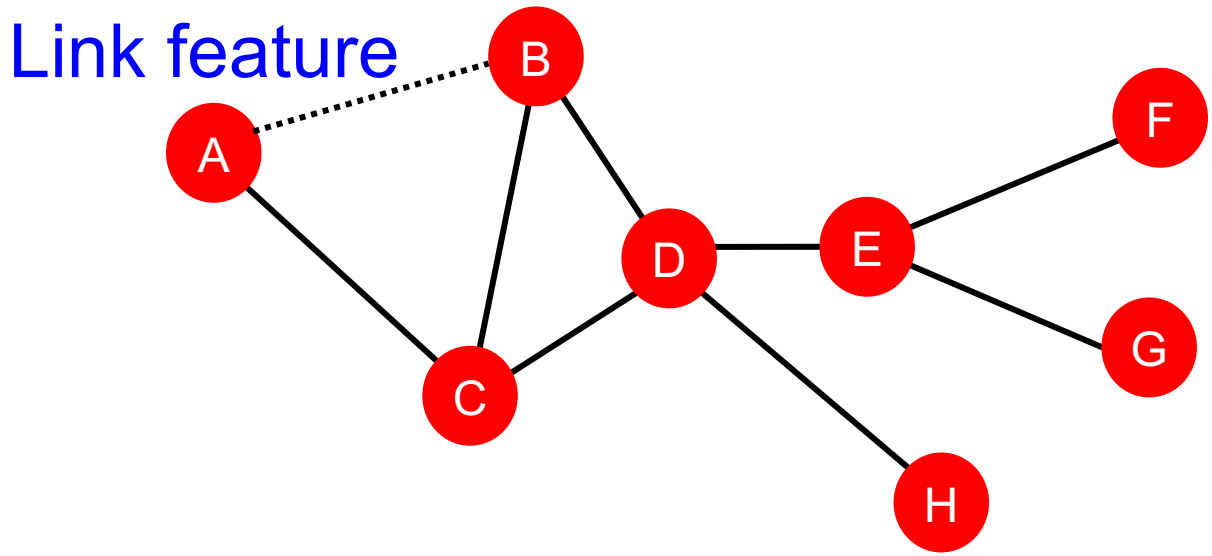
Link Prediction via Proximity

- **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

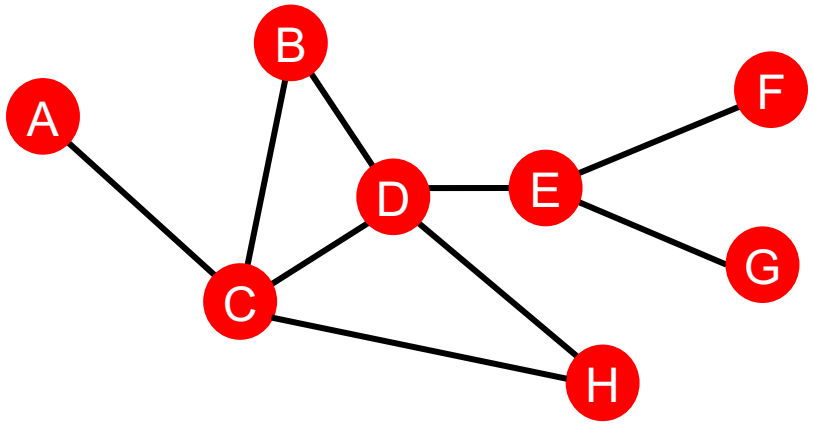
- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap



Distance-Based Features

Shortest-path distance between two nodes

- Example:



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

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

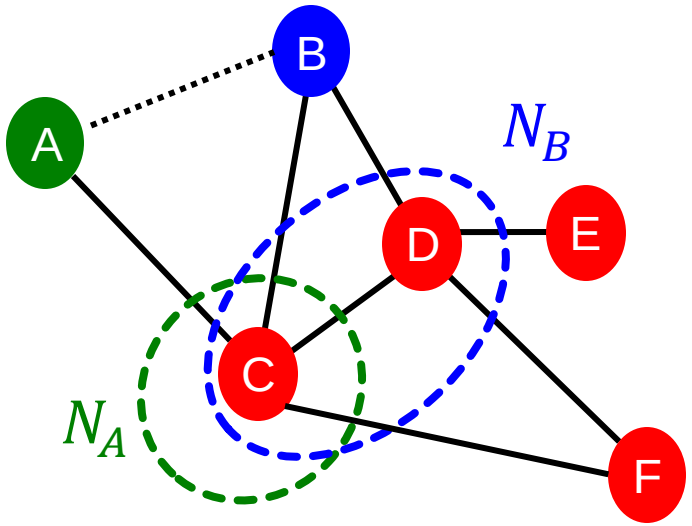
- However, this does not capture the degree of neighborhood overlap:
  - Node pair *(B, H)* has 2 shared neighboring nodes, while pairs *(B, E)* and *(A, B)* only have 1 such node.

Local Neighborhood Overlap

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

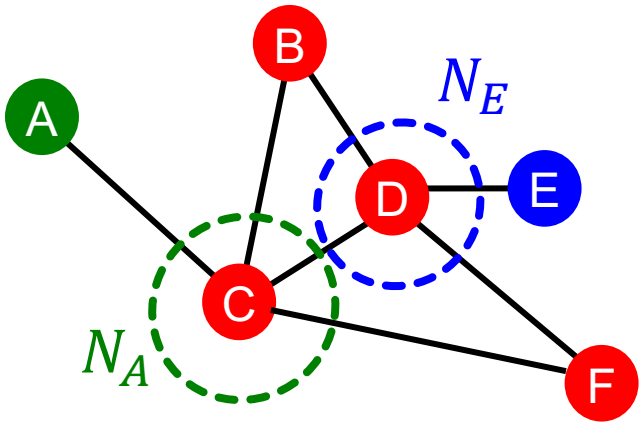
- **Common neighbors:** $|N(v_1) \cap N(v_2)|$
  - Example: $|N(A) \cap N(B)| = |\{C\}| = 1$
- **Jaccard's coefficient:** $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
  - Example: $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C,D\}|} = \frac{1}{2}$
- **Adamic-Adar index:**

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

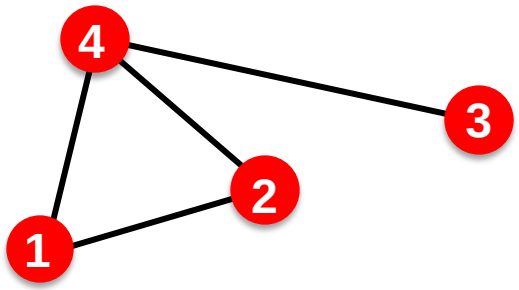
- **Katz index:** count the number of paths of all lengths between a given pair of nodes.

- **Q: How to compute #paths 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 $P_{uv}^{(K)}$ = #walks of length K between u and v
  - We will show $P^{(K)} = A^k$
  - $P_{uv}^{(1)}$ = #walks of length 1 (direct neighborhood) between u 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*

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1 v_2}$$

#walks of length l between v_1 and v_2

$0 < \beta < 1$: discount factor

- Katz index matrix is computed in closed-form:

$$\boldsymbol{S} = \sum_{i=1}^{\infty} \beta^i \boldsymbol{A}^i = \underbrace{(\boldsymbol{I} - \beta \boldsymbol{A})^{-1}}_{= \sum_{i=0}^{\infty} \beta^i \boldsymbol{A}^i \text{ by geometric series of matrices}} - \boldsymbol{I},$$

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

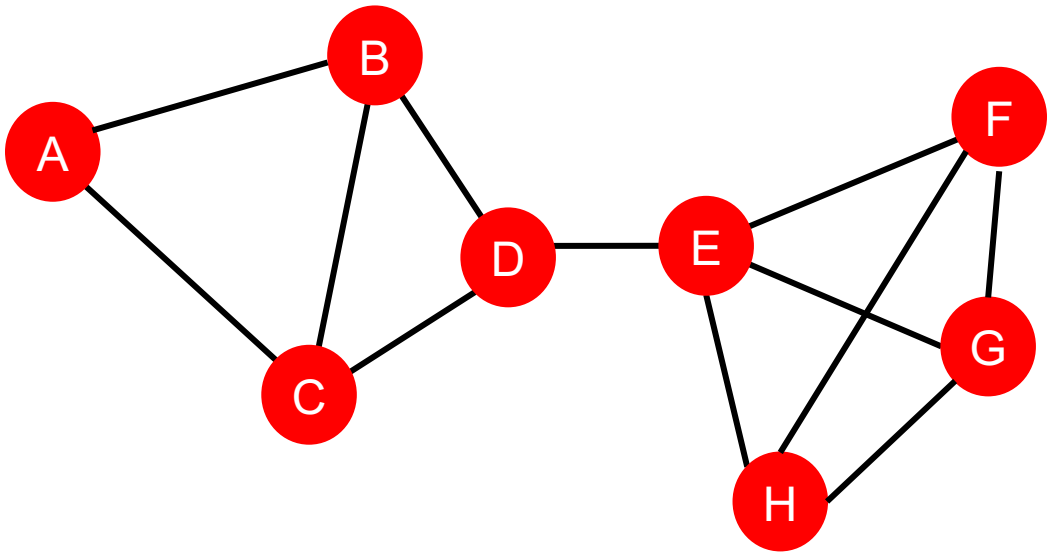
Jure Leskovec, Stanford University

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 $\boldsymbol{K} = \left(K(G, G')\right)_{G,G'}$ must always be positive semidefinite (i.e., has positive eigenvalues)
  - There exists a feature representation $\phi(\cdot)$ such that $K(G, G') = \phi(\mathrm{G})^{\mathrm{T}}\phi(G')$
  - Once the kernel is defined, off-the-shelf ML model, such as kernel SVM, can be used to make predictions.

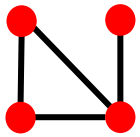
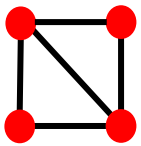
Graph-Level Features: Overview

- **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

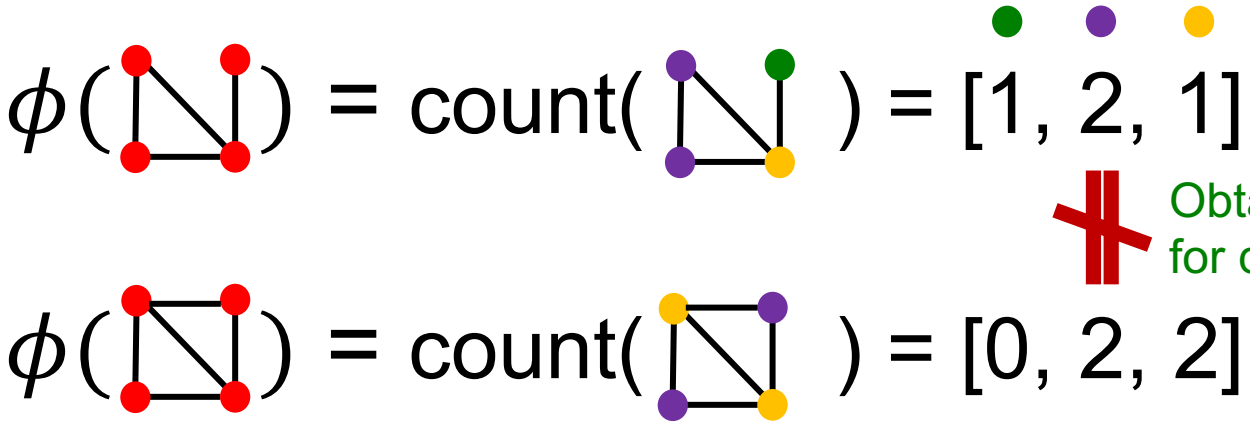
- **Goal**: Design graph feature vector $\phi(G)$
- **Key idea**: Bag-of-Words (BoW) for a graph
  - **Recall:** BoW simply uses the word counts as features for documents (no ordering considered).
  - Naïve extension to a graph: Regard nodes as words.
  - Since both graphs have 4 red nodes, we get the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

$\phi($  $) = \phi($  $)$

Graph Kernel: Key Idea

What if we use Bag of **node degrees**?

Deg1: ● Deg2: ● Deg3: ●



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

Graphlet Features

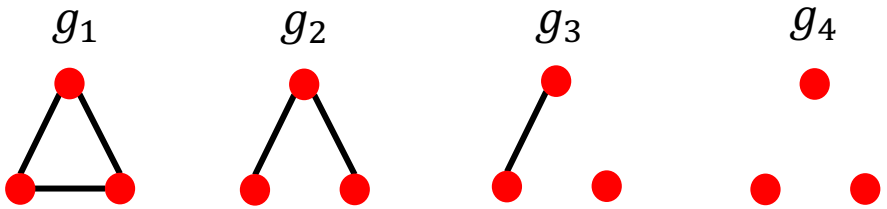
- **Key idea:** Count the number of different graphlets in a graph.

  - **Note:** Definition of graphlets here is slightly different from node-level features.
  - The two differences are:
    - Nodes in graphlets here do **not need to be connected** (allows for isolated nodes)
    - The graphlets here are not rooted.
    - Examples in the next slide illustrate this.

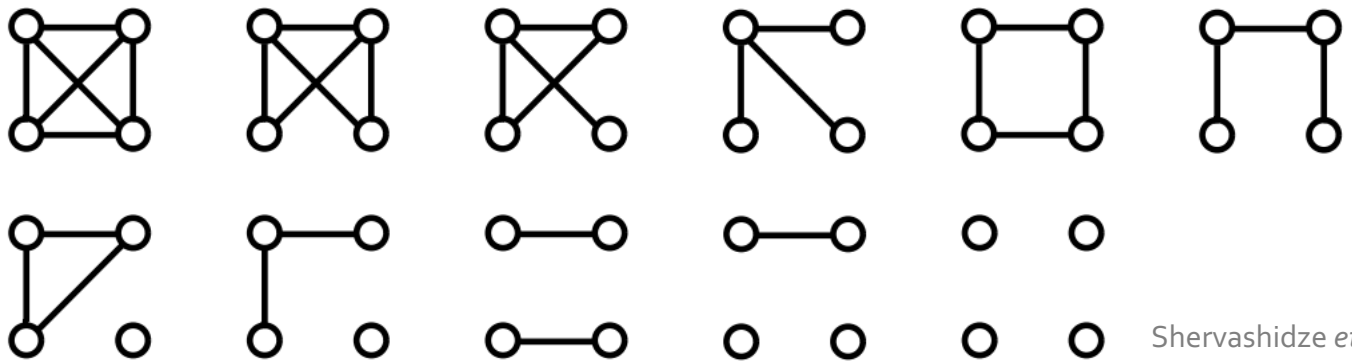
Graphlet Features

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

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



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



Shervashidze *et al.*, AISTATS 2011

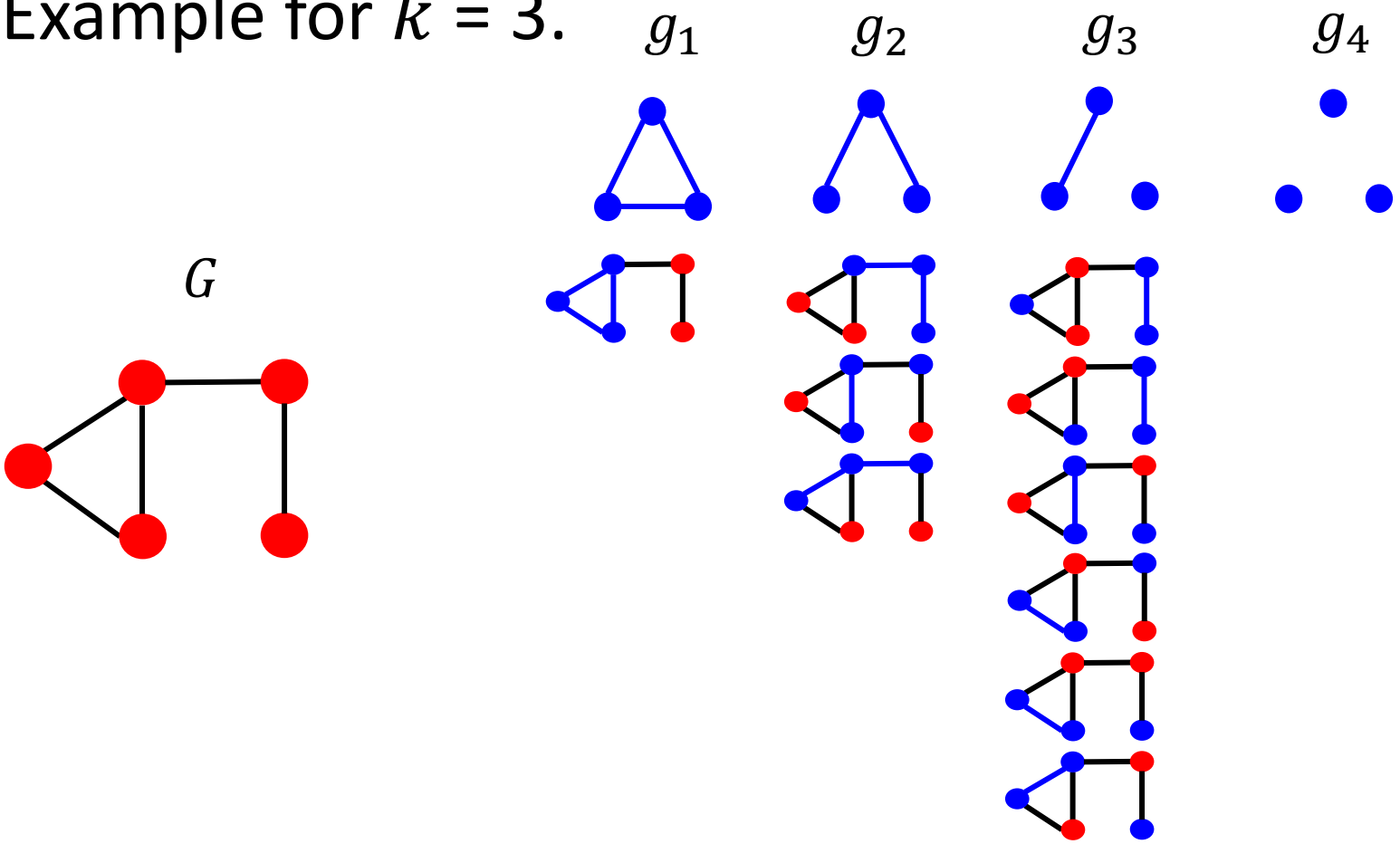
Graphlet Features

- Given graph G, and a graphlet list $\mathcal{G}_k = (g_1, g_2, \ldots, g_{n_k})$, define the graphlet count vector $\boldsymbol{f}_G \in \mathbb{R}^{n_k}$ as

$$(\boldsymbol{f}_G)_i = \#(g_i \subseteq G) \text{ for } i = 1,2, \ldots, n_k.$$

Graphlet Features

- Example for $k = 3$.



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

Graphlet Kernel

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

$$K(G, G') = \boldsymbol{f}_G{}^{\mathrm{T}} \boldsymbol{f}_{G'}$$

- **Problem:** if G and G' have different sizes, that will greatly skew the value.
- **Solution:** normalize each feature vector

$$\boldsymbol{h}_G = \frac{\boldsymbol{f}_G}{\mathrm{Sum}(\boldsymbol{f}_G)} \qquad K(G, G') = \boldsymbol{h}_G{}^{\mathrm{T}} \boldsymbol{h}_{G'}$$

Graphlet Kernel

Limitations: Counting graphlets is expensive!

- Counting size-k graphlets for a graph with size n by enumeration takes n^k.
- This is unavoidable in the worst-case since subgraph isomorphism test (judging whether a graph is a subgraph of another graph) is **NP-hard**.
- If a graph's node degree is bounded by d, an $O(nd^{k-1})$ algorithm exists to count all the graphlets of size k.

Can we design a more efficient graph kernel?

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

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

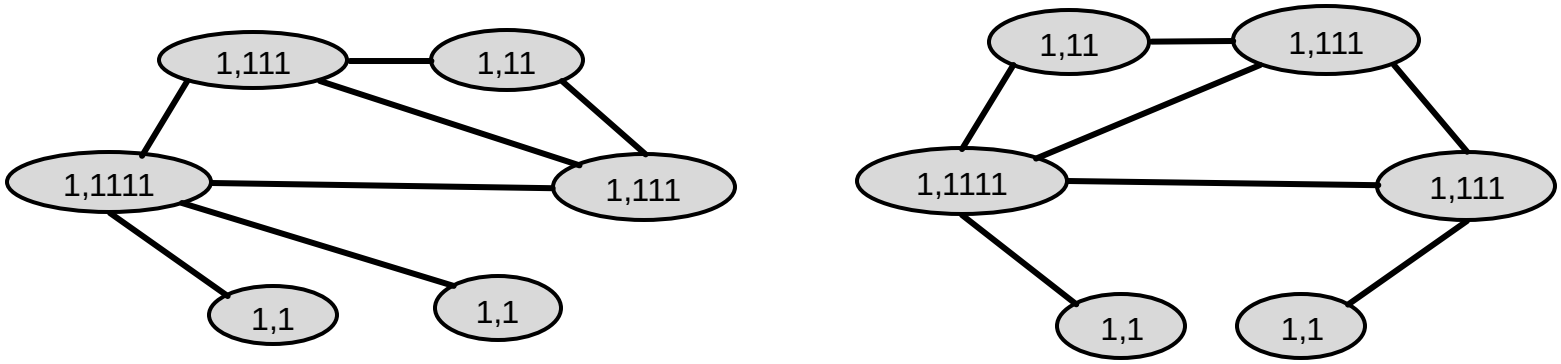
    where HASH maps different inputs to different colors.
  - After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood

Color Refinement (1)

Example of color refinement given two graphs

- Assign initial colors

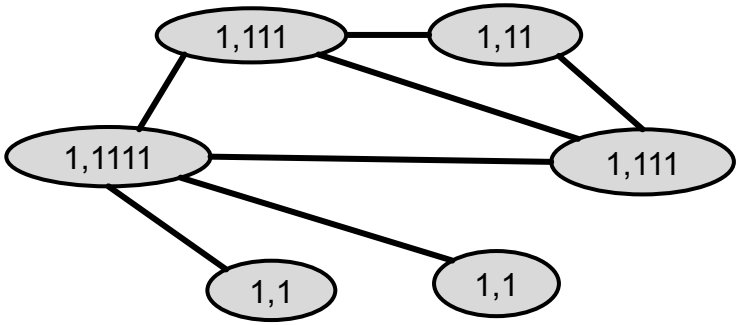
- Aggregate neighboring colors

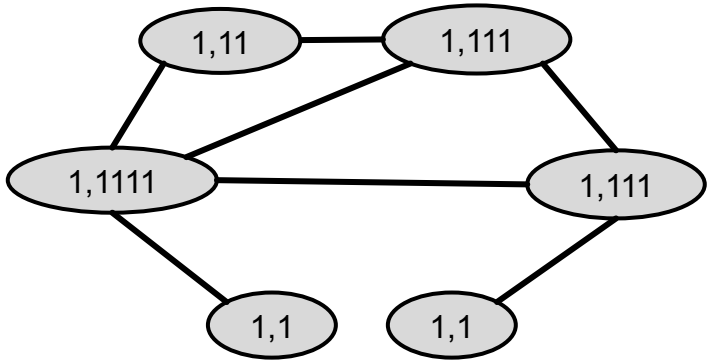


Color Refinement (2)

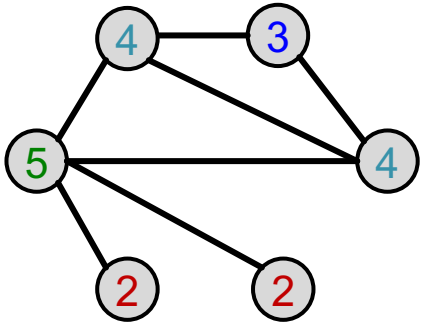
Example of color refinement given two graphs

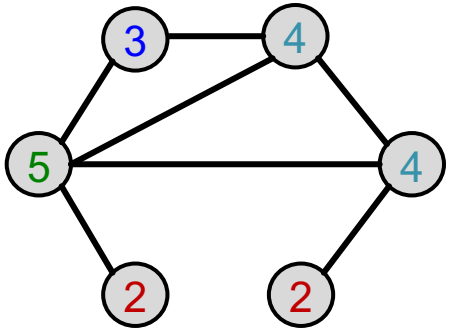
- Aggregated colors





- Hash aggregated colors





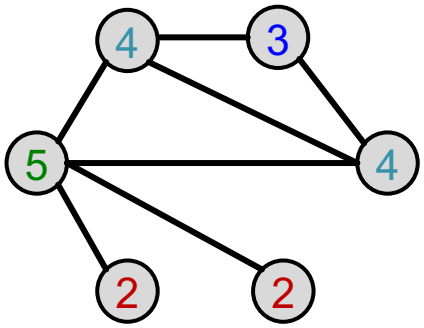
Hash table

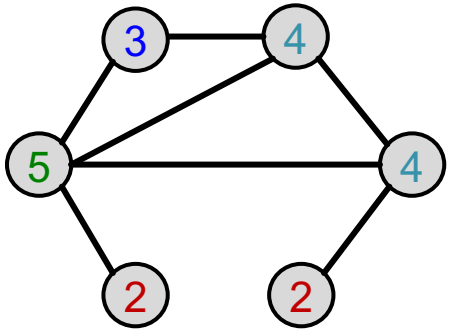
1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

Color Refinement (3)

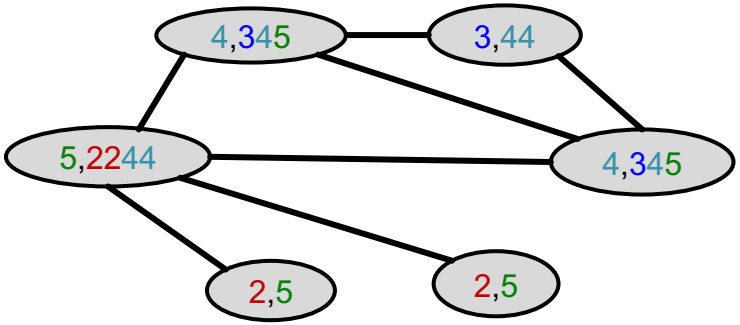
Example of color refinement given two graphs

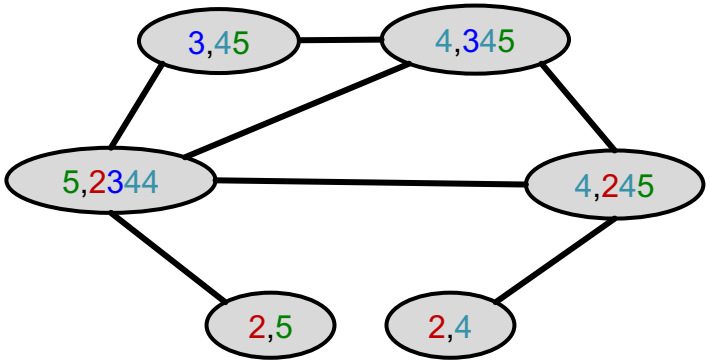
- Aggregated colors





- Hash aggregated colors

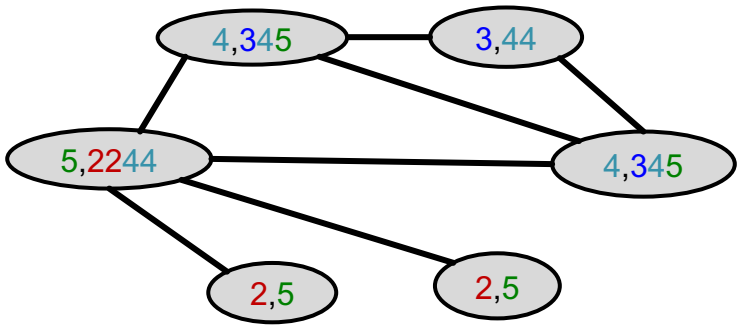


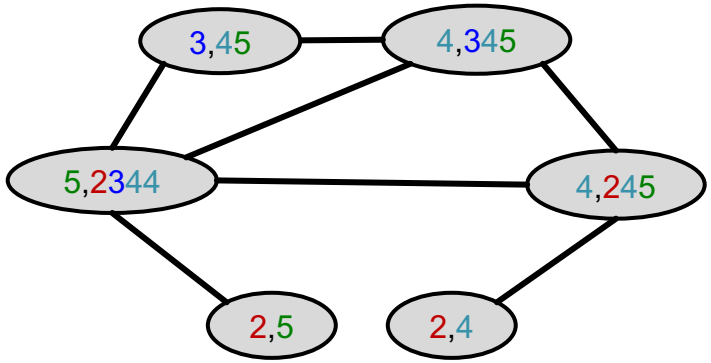


Color Refinement (4)

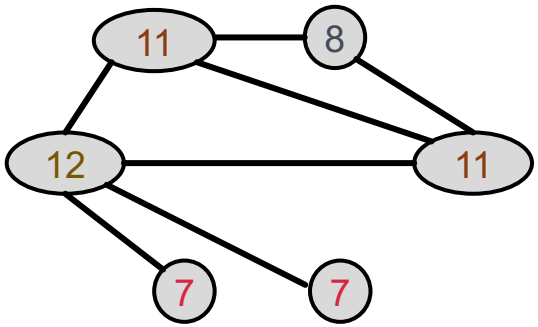
Example of color refinement given two graphs

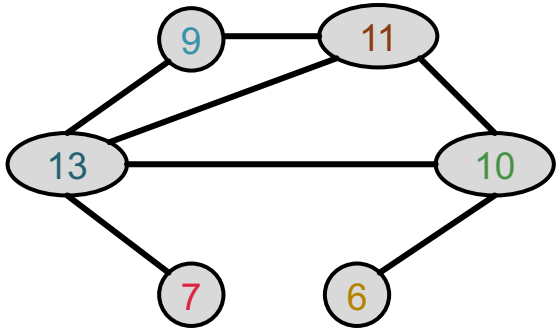
- Aggregated colors





- Hash aggregated colors



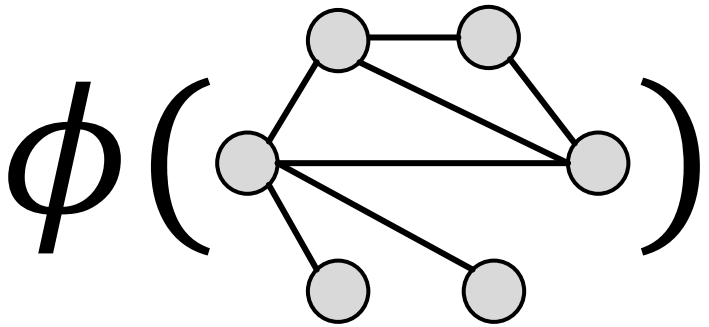


Hash table

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.

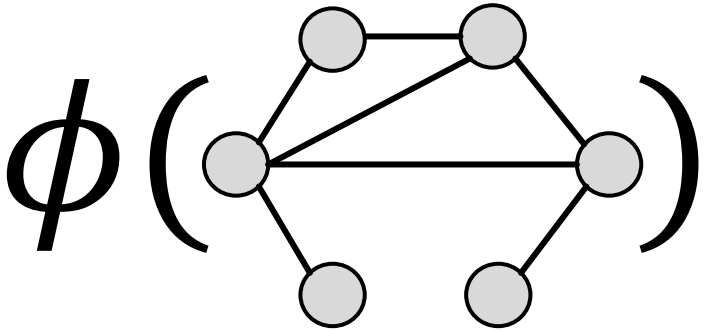


Colors

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

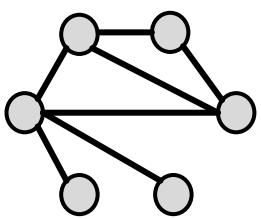
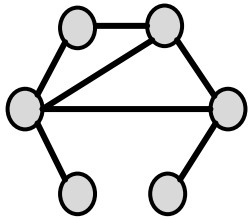


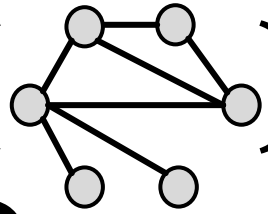
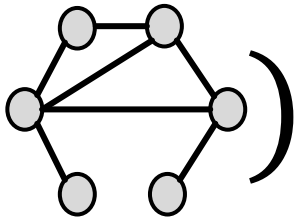
1,2,3,4,5,6,7,8,9,10,11,12,13

= [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:

$$K(\,\text{  }\,,\,\text{  }\,)$$

$$= \phi(\,\text{  }\,)^{T}\,\phi(\,\text{  }\,)$$

$$= 49$$

Weisfeiler-Lehman Kernel

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

Graph-Level Features: Summary

- **Graphlet Kernel**
  - Graph is represented as **Bag-of-graphlets**
  - **Computationally expensive**
- **Weisfeiler-Lehman Kernel**
  - Apply K-step color refinement algorithm to enrich node colors
    - Different colors capture different K-hop neighborhood structures
  - Graph is represented as **Bag-of-colors**
  - **Computationally efficient**
  - Closely related to Graph Neural Networks (as we will see!)

Today's Summary

- **Traditional ML Pipeline**
  - Hand-crafted feature + ML model
- **Hand-crafted features for graph data**
  - **Node-level:**
    - Node degree, centrality, clustering coefficient, graphlets
  - **Link-level:**
    - Distance-based feature
    - local/global neighborhood overlap
  - **Graph-level:**
    - Graphlet kernel, WL kernel

Stanford CS224W: Node Embeddings

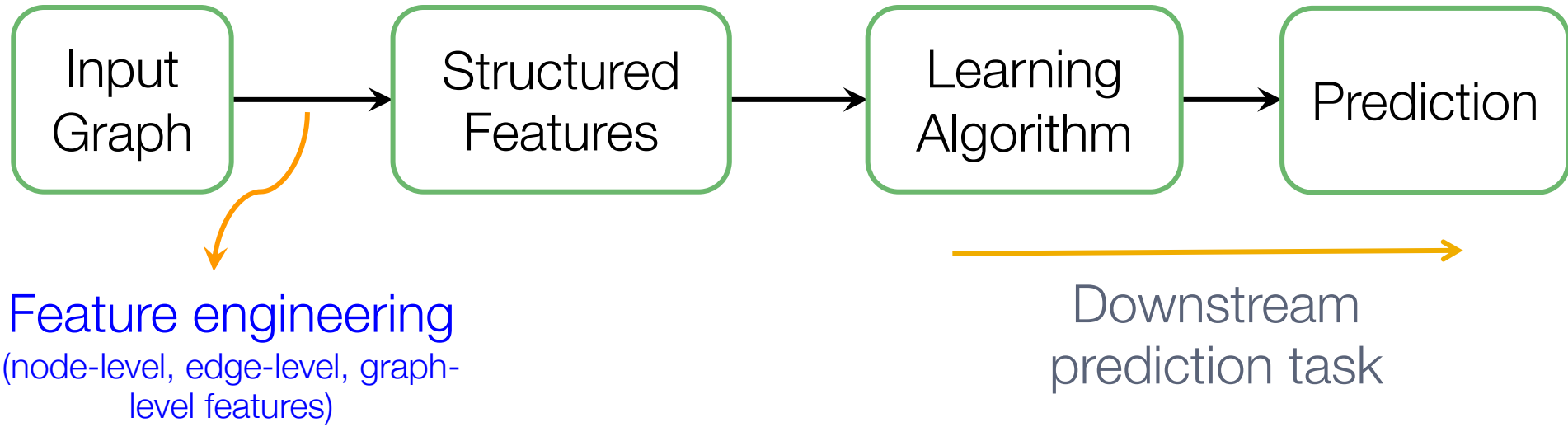
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

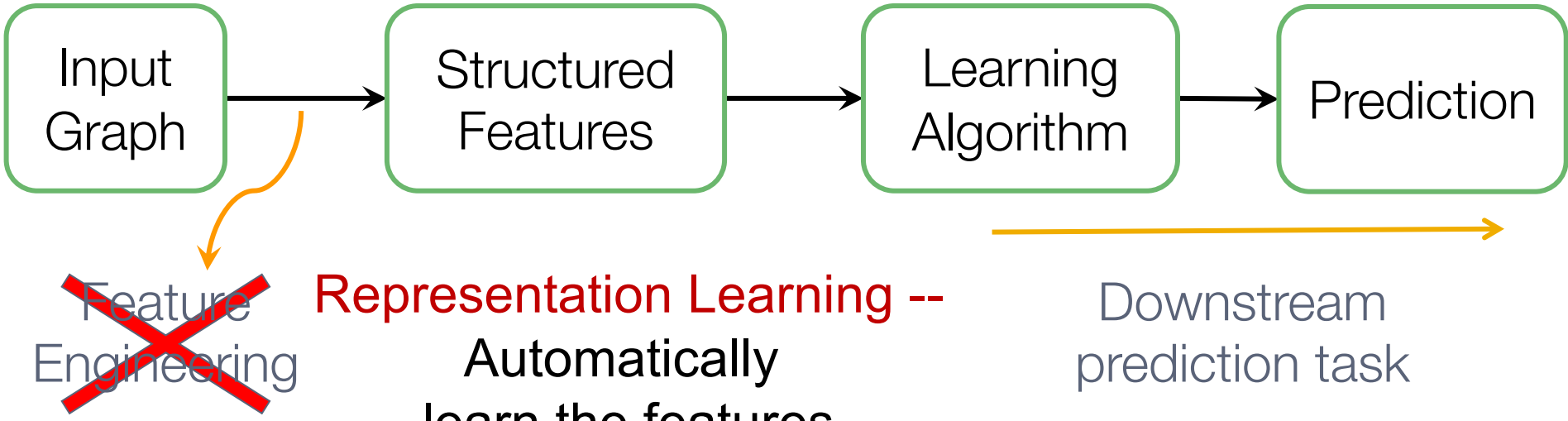
Recap: Traditional ML for Graphs

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



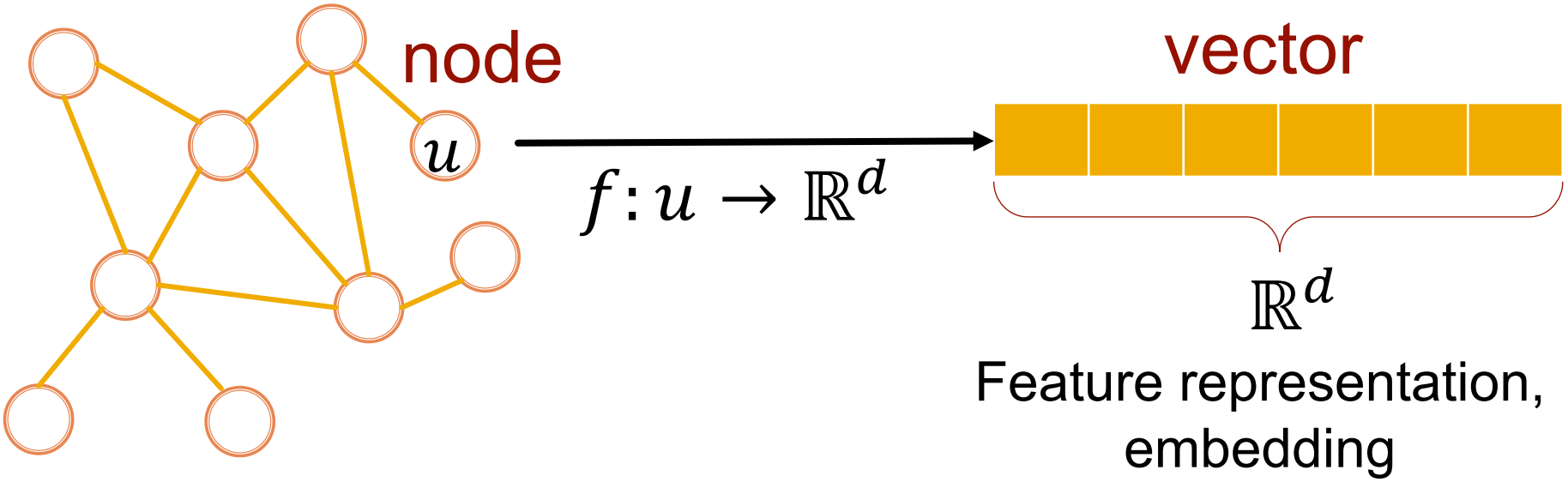
Graph Representation Learning

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



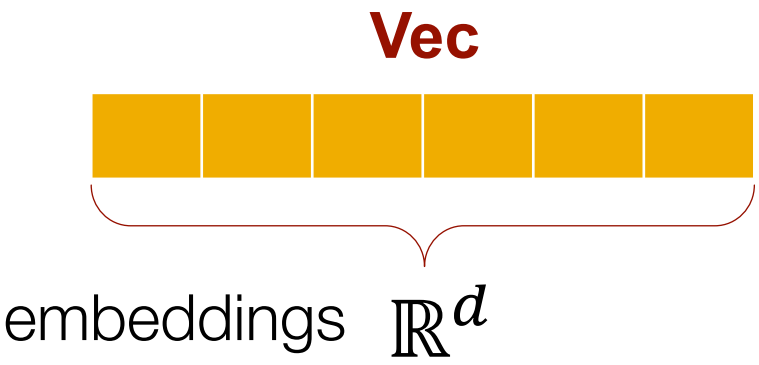
Graph Representation Learning

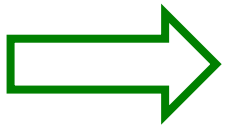
Goal: Efficient task-independent feature learning for machine learning with graphs!



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
- ….

Example Node Embedding

- **2D embedding of nodes of the Zachary's Karate Club network:**

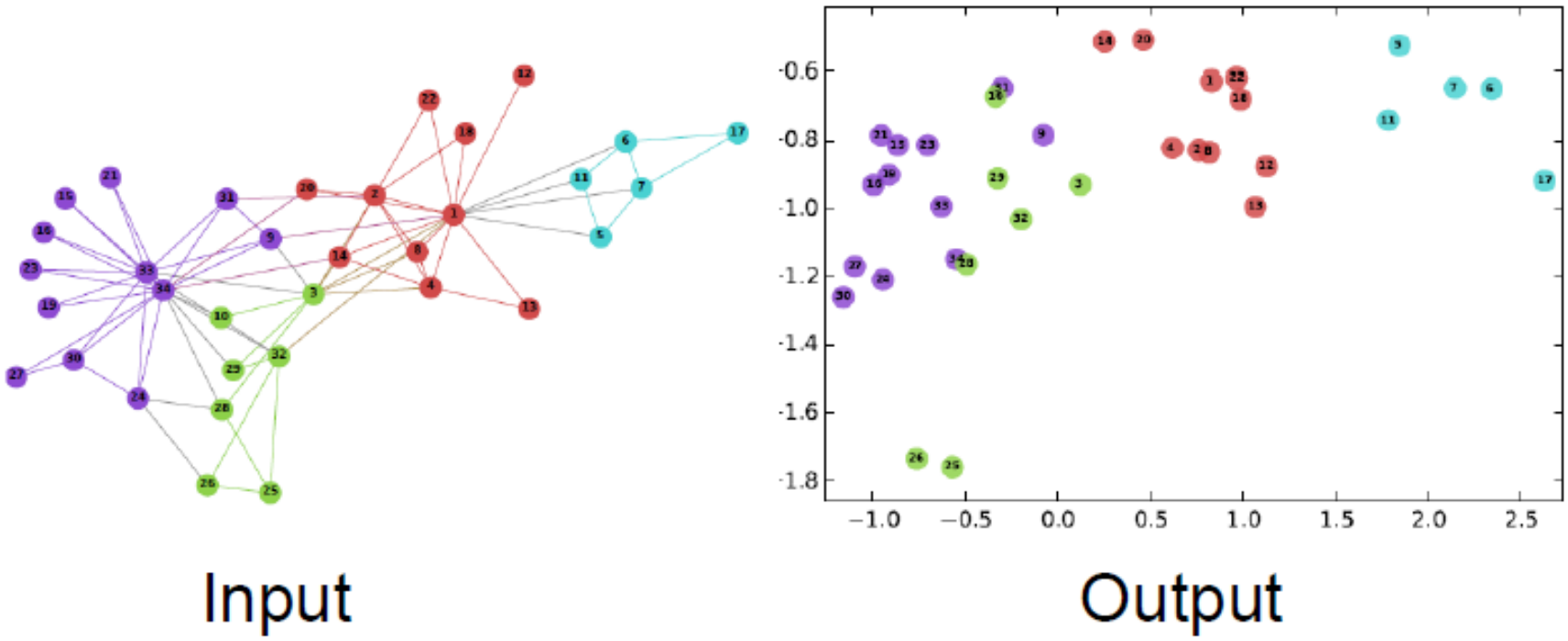


Image from: Perozzi et al. DeepWalk: Online Learning of Social Representations. *KDD 2014.*

Stanford CS224W: Node Embeddings: Encoder and Decoder

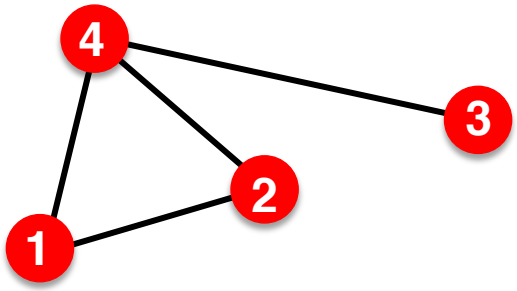
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

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**

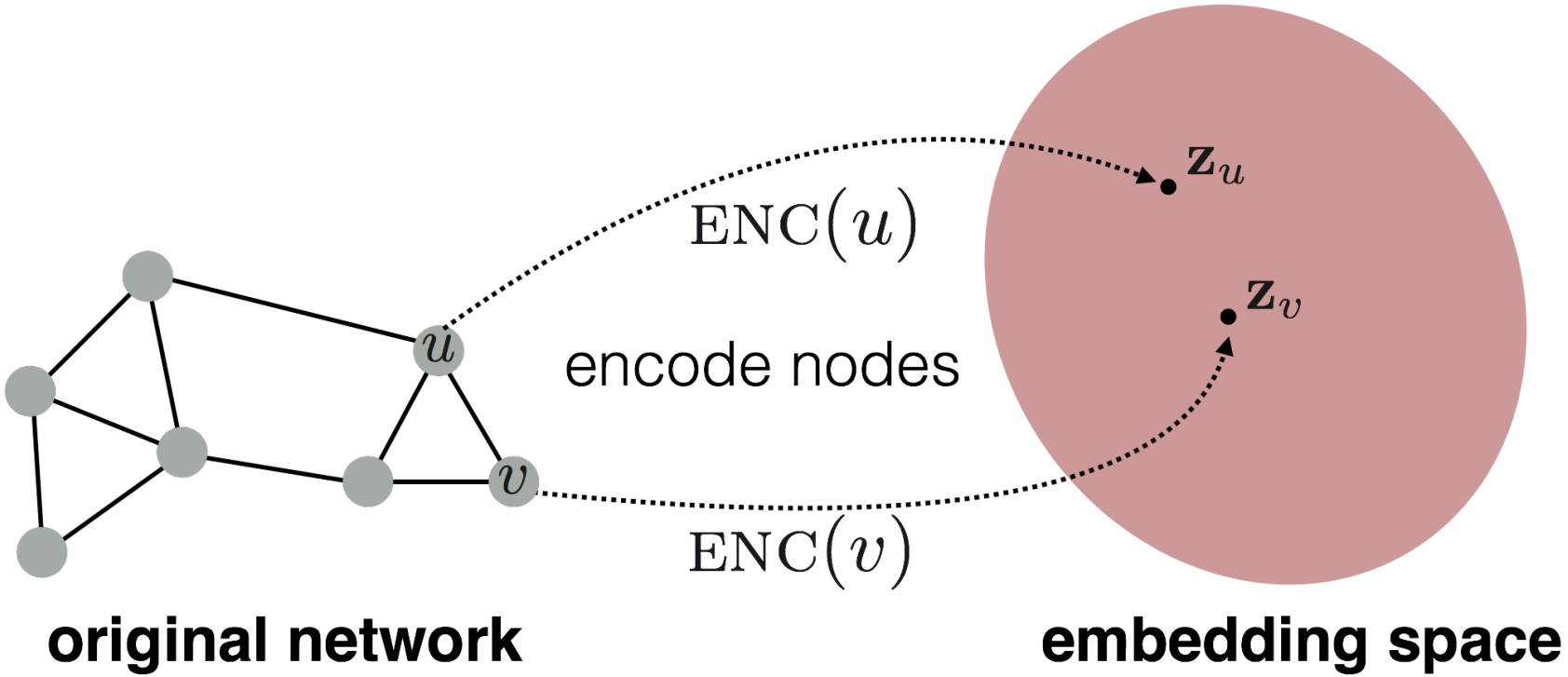


V: {1, 2, 3, 4}

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

Embedding Nodes

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



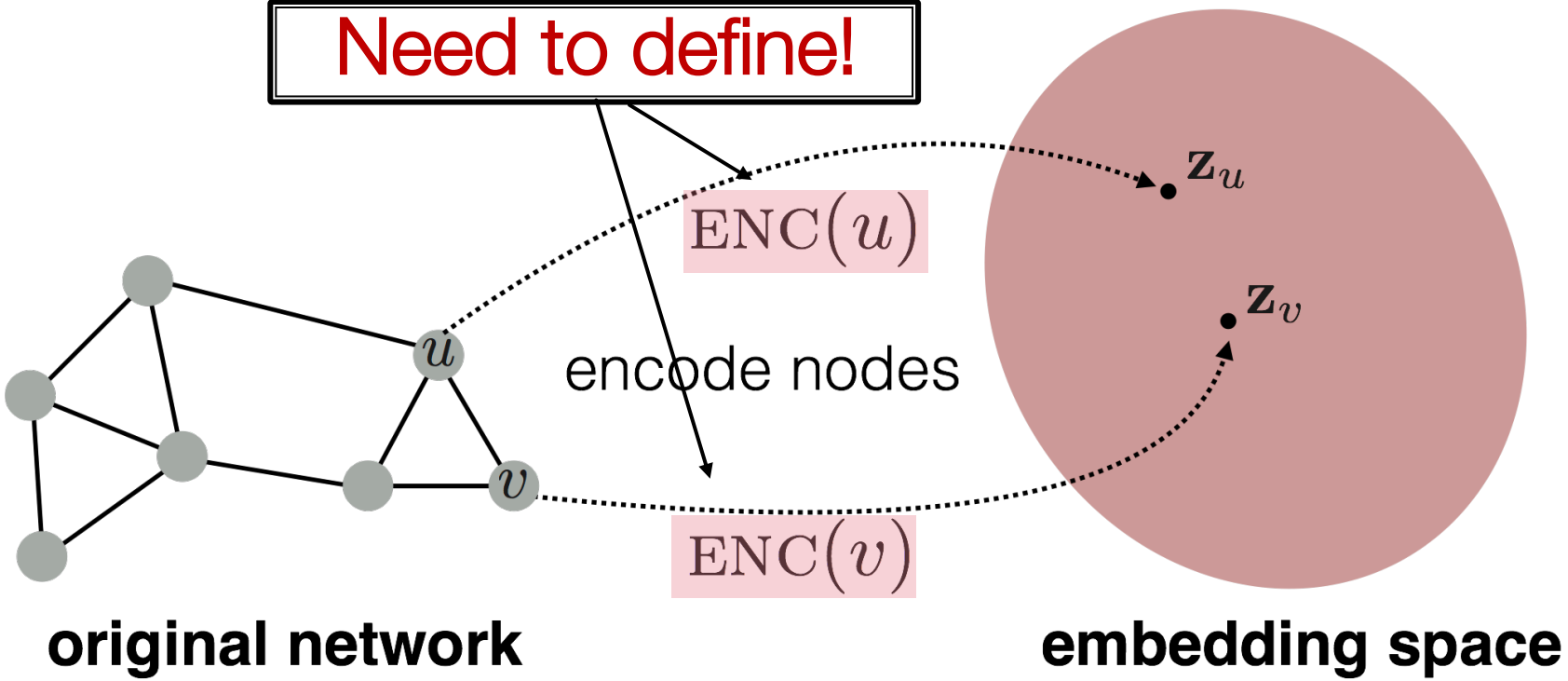
Embedding Nodes

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

similarity(u, v) in the original network

$\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$ 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** $\mathbf{DEC}$ maps from embeddings to the similarity score
4. **Optimize the parameters of the encoder so that:**

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

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

in the original network — Similarity of the embedding

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

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

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**

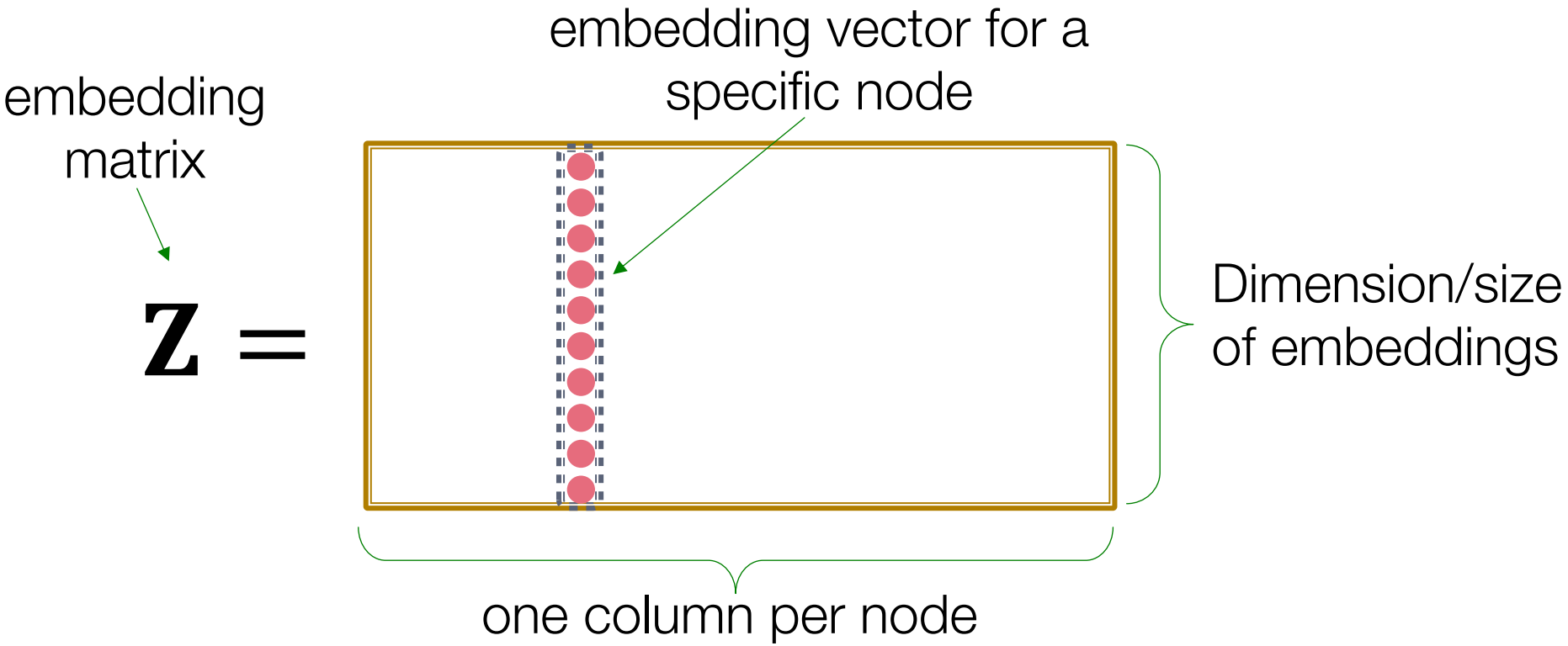
$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

$\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ 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

Framework Summary

- **Encoder + Decoder Framework**
  - Shallow encoder: embedding lookup
  - Parameters to optimize: $\mathbf{Z}$ which contains node embeddings $\mathbf{z}_u$ for all nodes $u \in V$
  - We will cover deep encoders (GNNs) in Lecture 6

  - **Decoder:** based on node similarity.
  - **Objective:** maximize $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$ for node pairs (u, v) that are **similar**

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

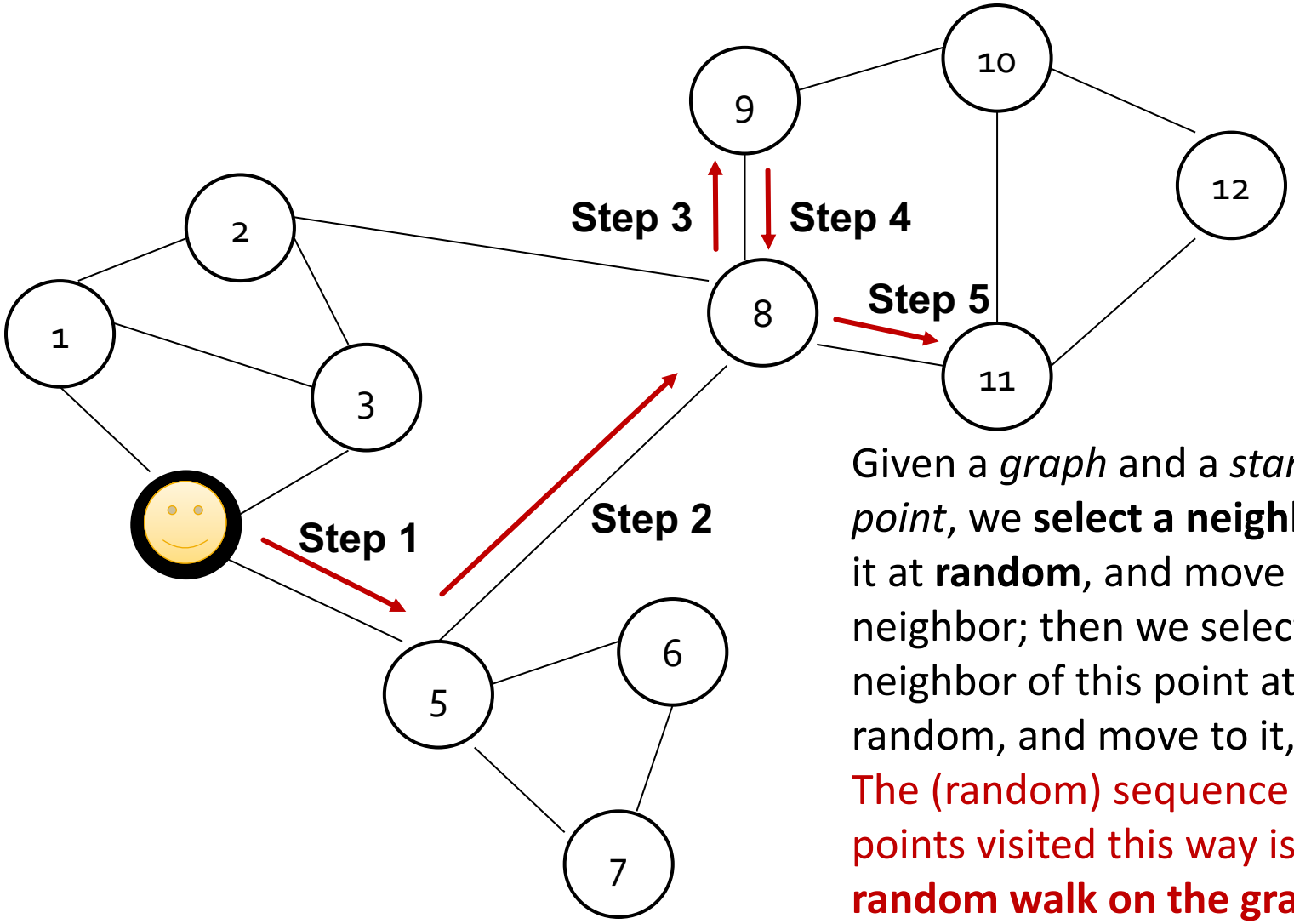
Notation

- **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: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$.
- **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**.

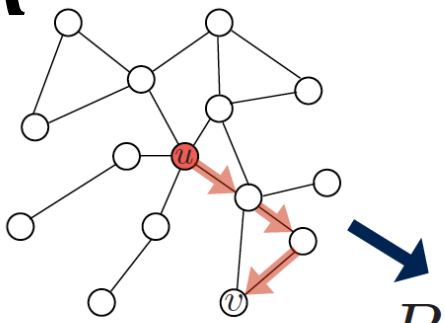
Random-Walk Embeddings

$$\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v \approx$$

probability that u and v co-occur on a random walk over the graph

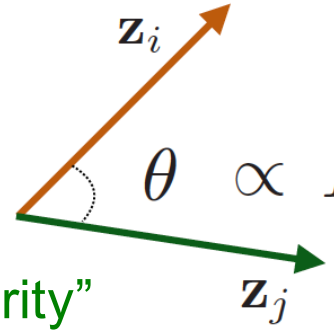
Random-Walk Embeddings

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



$$P_R(v|u)$$

2. **Optimize embeddings to encode these random walk statistics:**



$$\theta \propto P_R(v|u)$$

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:

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

  - $N_R(u)$ 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 $N_{\mathrm{R}}(u)$

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 $N_{\mathrm{R}}(u)$

$$\max_f \sum_{u \in V} \log \mathrm{P}(N_{\mathrm{R}}(u) | \mathbf{z}_u) \Longrightarrow$$ 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 z_u to maximize the likelihood of random walk co-occurrences
- **Parameterize $P(v|\mathbf{z}_u)$ using softmax:**

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

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:

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

sum over all nodes u

sum over nodes v seen on random walks starting from u

predicted probability of u and v co-occuring 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!

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

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

Random Walk Optimization

But doing this naively is too expensive!

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

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

Negative Sampling

- **Solution:** Negative sampling

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

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

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

sigmoid function
(makes each term a "probability" between 0 and 1)

random distribution over nodes

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

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

random distribution over nodes

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

- 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 z_u at some randomized value for all nodes u.
  - Iterate until convergence:
    - For all u, compute the derivative $\frac{\partial \mathcal{L}}{\partial z_u}$.
    - For all u, make a step in reverse direction of derivative: $z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$.

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.*

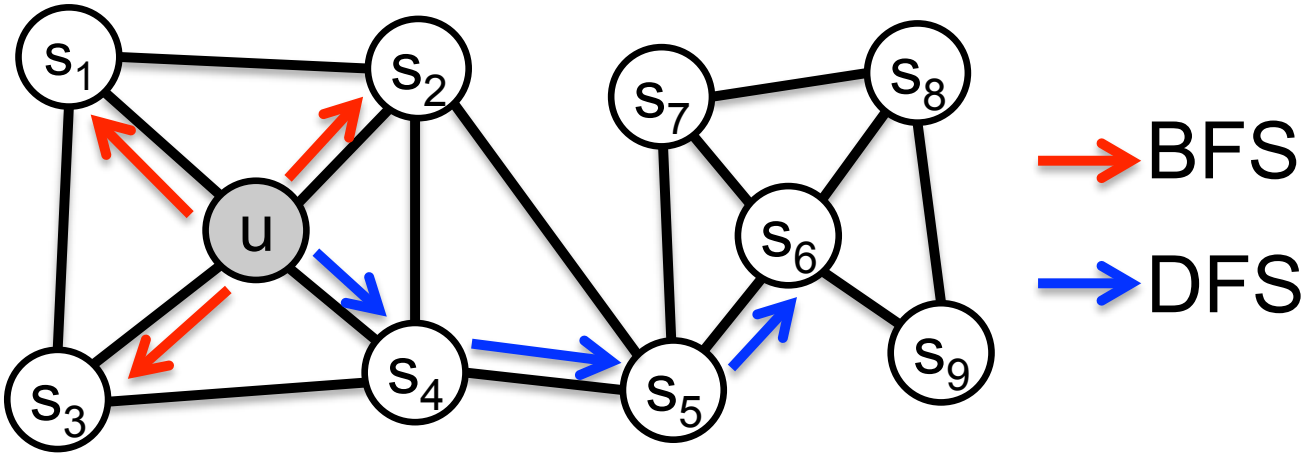
Overview of node2vec

- **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.*

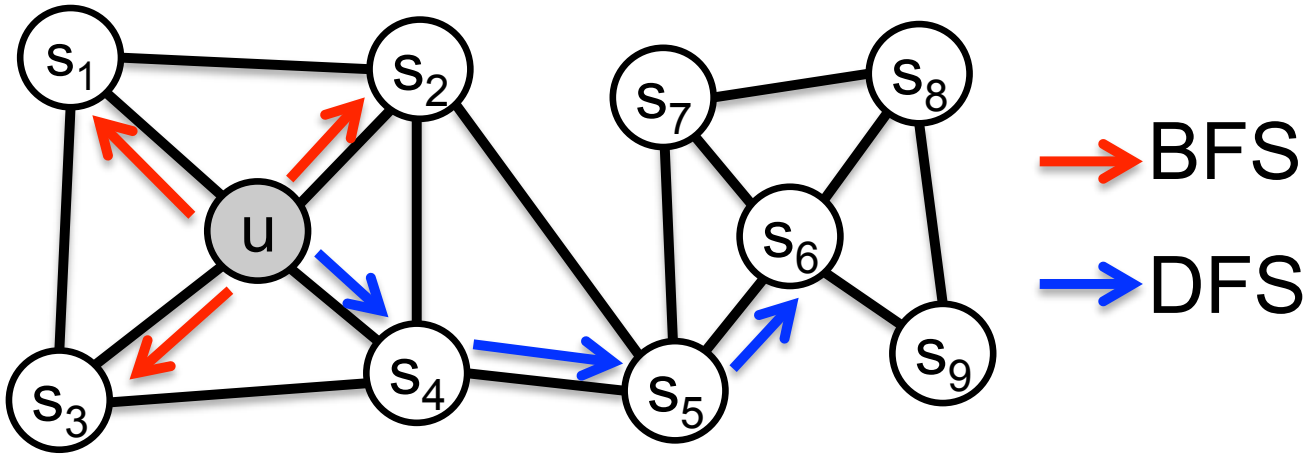
node2vec: Biased Walks

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:

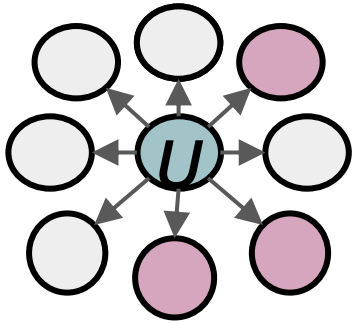


Walk of length 3 ($N_R(u)$ of size 3):

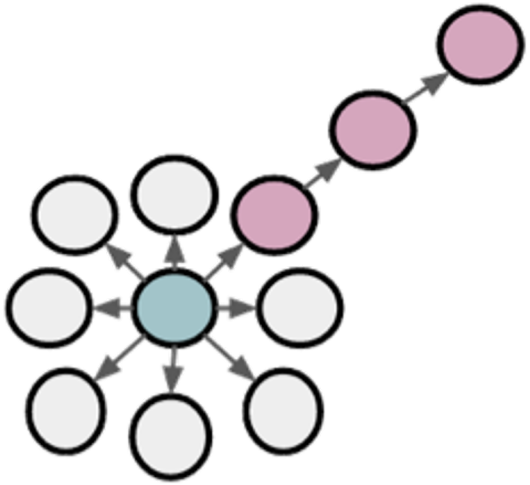
$N_{BFS}(u) = \{s_1, s_2, s_3\}$ Local microscopic view

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

BFS vs. DFS



BFS:
Micro-view of neighbourhood



DFS:
Macro-view of neighbourhood

Interpolating BFS and DFS

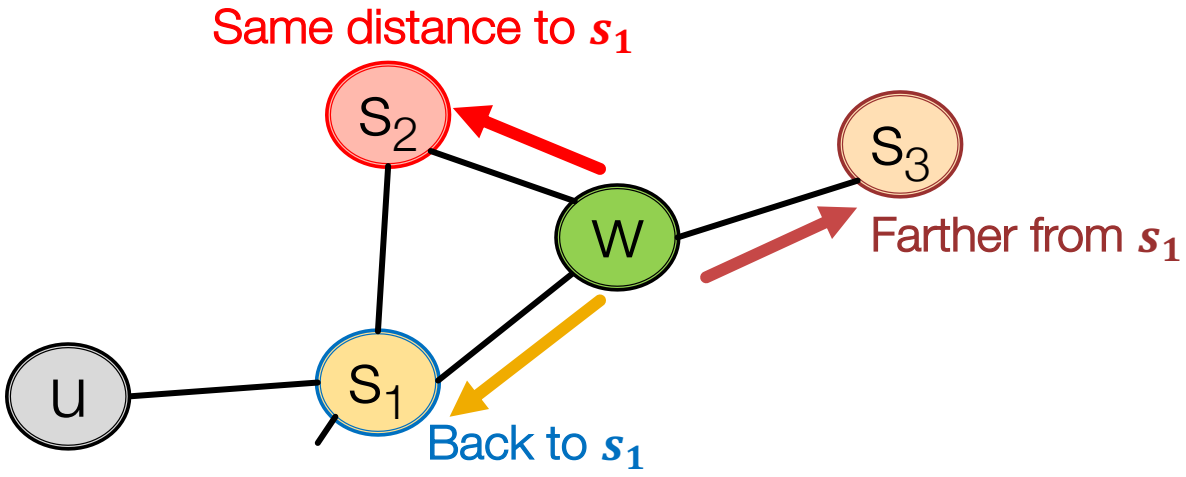
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- 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:

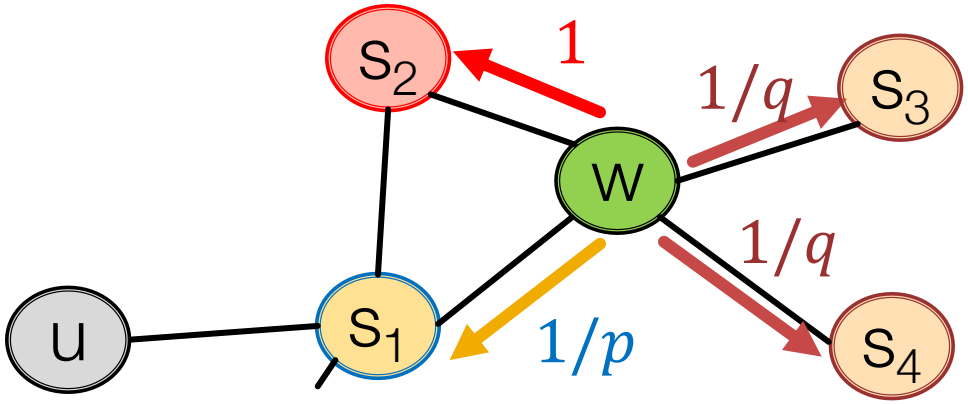
- Rnd. walk just traversed edge (s_1, w) and is now at w
- **Insight:** Neighbors of w can only be:



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?**

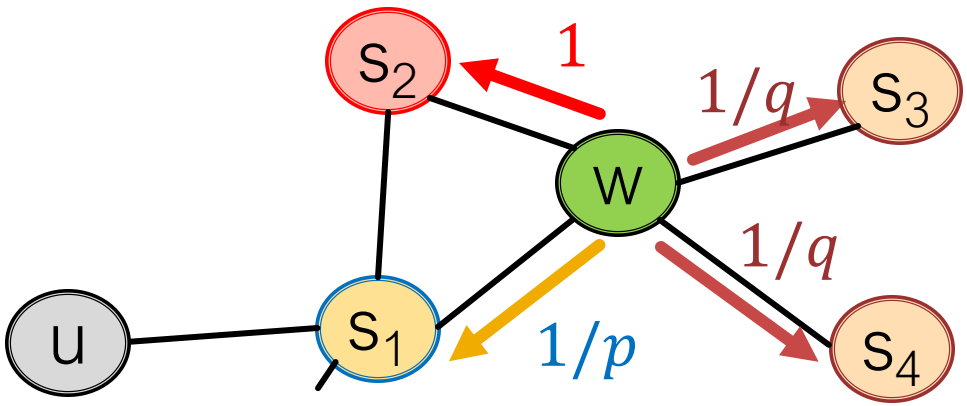


$1/p, 1/q, 1$ are unnormalized probabilities

- 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?**



$w \rightarrow$

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

Other Random Walk Ideas

- **Different kinds of biased random walks:**
  - Based on node attributes (Dong et al., 2017).
  - Based on learned weights (Abu-El-Haija et al., 2017)
- **Alternative optimization schemes:**
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (as in LINE from Tang et al. 2015).
- **Network preprocessing techniques:**
  - Run random walks on modified versions of the original network (e.g., Ribeiro et al. 2017's struct2vec, Chen et al. 2016's HARP).

Summary so far

- **Core idea:** Embed nodes so that distances in embedding space reflect node similarities in the original network.
- **Different notions of node similarity:**
  - Naïve: similar if 2 nodes are connected
  - Neighborhood overlap (covered in Lecture 2)
  - Random walk approaches **(covered today)**

Summary so far

- **So what method should I use..?**
- No one method wins in all cases….
  - E.g., node2vec performs better on node classification while alternative methods perform better on link prediction (Goyal and Ferrara, 2017 survey)
- Random walk approaches are generally more efficient
- **In general:** Must choose definition of node similarity that matches your application!

Stanford CS224W: Embedding Entire Graphs

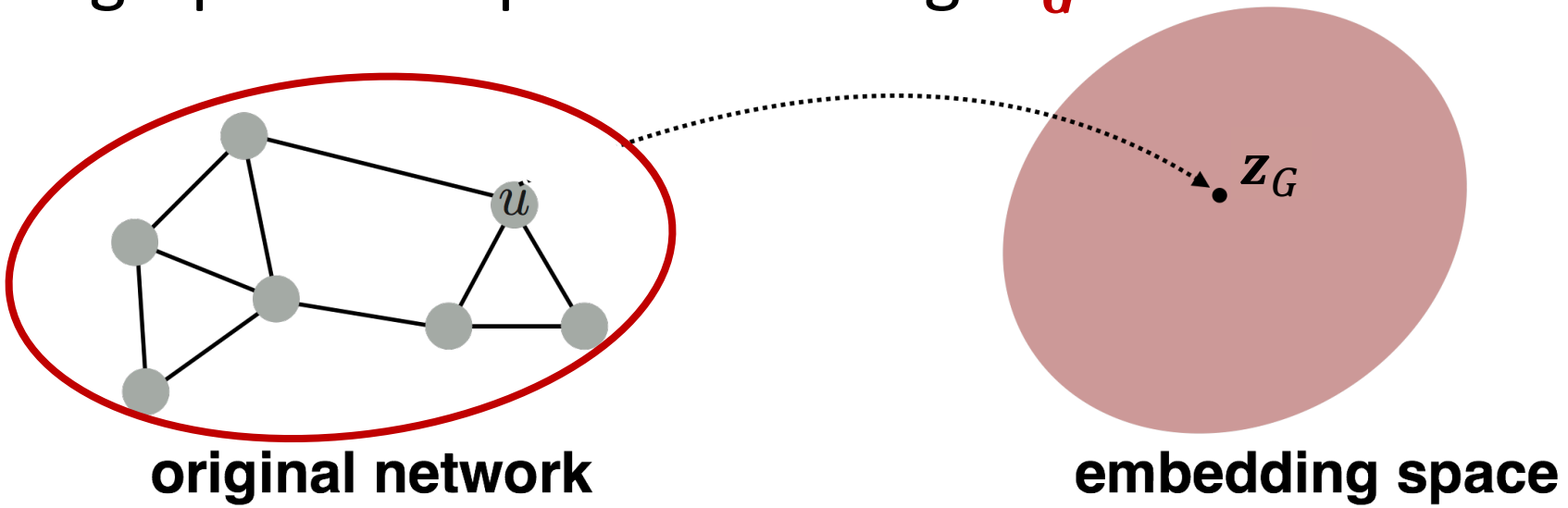
CS224W: Machine Learning with Graphs

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:

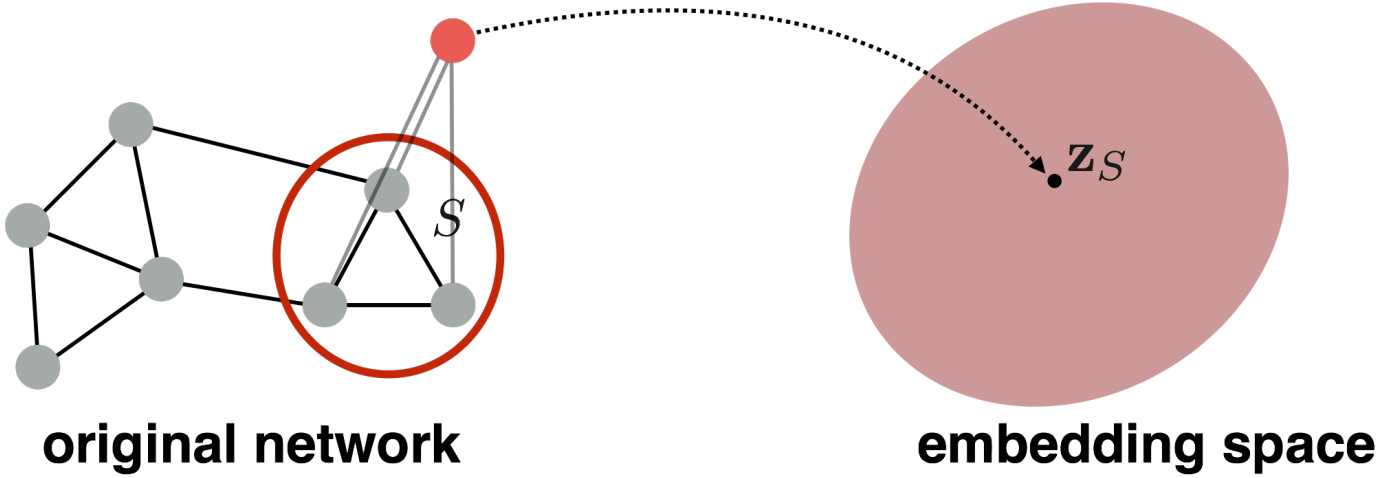
- Run a standard graph embedding technique *on* the (sub)graph G.
- Then just sum (or average) the node embeddings in the (sub)graph G.

$$\boldsymbol{z_G} = \sum_{v \in G} \boldsymbol{z_v}$$

- Used by Duvenaud et al., 2016 to classify molecules based on their graph structure

Approach 2

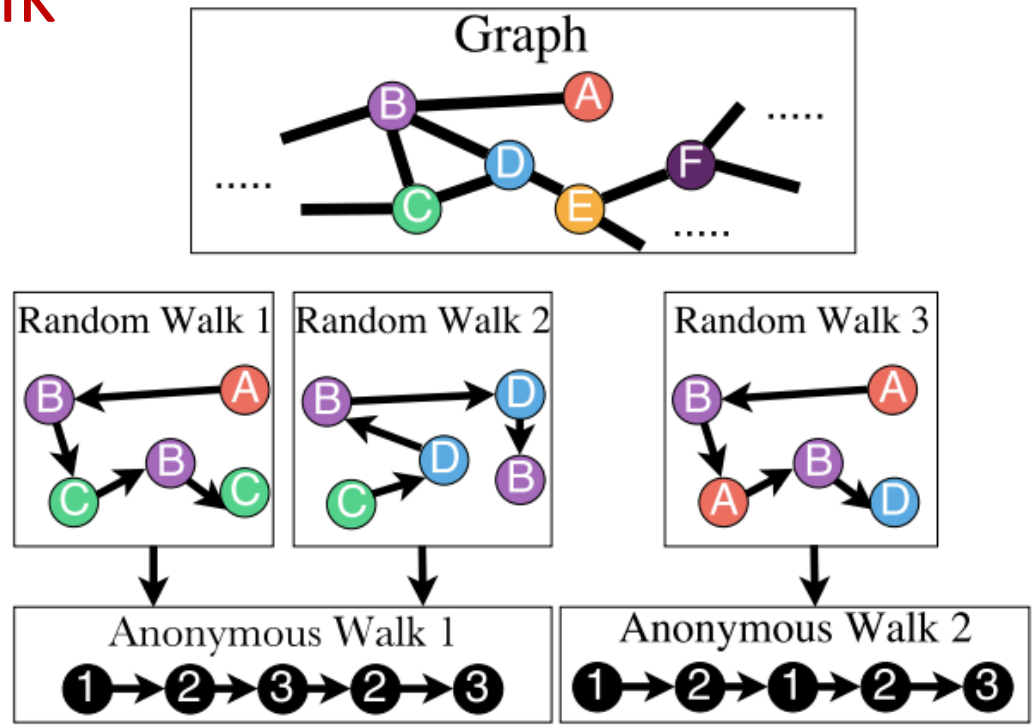
- **Idea 2:** Introduce a **"virtual node"** to represent the (sub)graph and run a standard graph embedding technique



- Proposed by Li et al., 2016 as a general technique for subgraph embedding

Approach 3: Anonymous Walk Embeddings

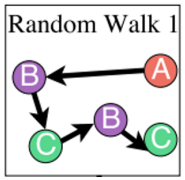
States in **anonymous walks** correspond to the index of the **first time** we visited the node in a random walk



Anonymous Walk Embeddings, ICML 2018 https://arxiv.org/pdf/1805.11921.pdf

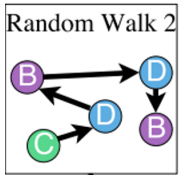
Approach 3: Anonymous Walk Embeddings

- Agnostic to the identity of the nodes visited (hence anonymous)
- Example RW1:

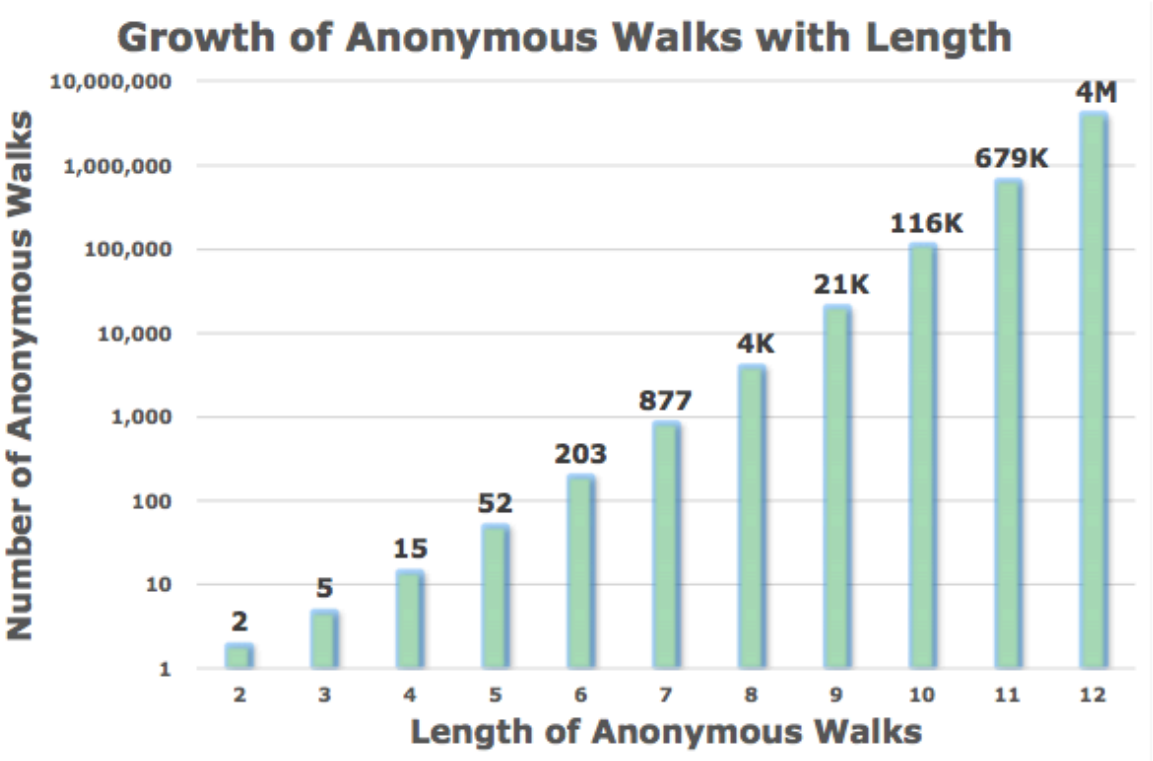


- Step **1**: node A → node 1
- Step **2**: node B → node 2 (different from node 1)
- Step **3**: node C → node 3 (different from node 1, 2)
- Step **4**: node B → node 2 (same as the node in step 2)
- Step **5**: node C → node 3 (same as the node in step 3)

- Note: RW2 gives the same anonymous walk



Number of Walks Grows



Number of anonymous walks grows exponentially:

- There are 5 anon. walks w_i of length 3:
  w_1=111, w_2=112, w_3= 121, w_4= 122, w_5= 123

Simple Use of Anonymous Walks

- Simulate anonymous walks w_i of l steps and record their counts
- **Represent the graph as a probability distribution over these walks**

- **For example:**
  - Set $l = 3$
  - Then we can represent the graph as a 5-dim vector
    - Since there are 5 anonymous walks w_i of length 3: 111, 112, 121, 122, 123
  - $\boldsymbol{Z}_G[i]$ = probability of anonymous walk w_i in G

Sampling Anonymous Walks

- **Sampling anonymous walks:** Generate independently a set of m random walks.
- Represent the graph as a probability distribution over these walks.

- How many random walks m do we need?
  - We want the distribution to have error of more than ε with prob. less than δ:

$$m = \left\lceil \frac{2}{\varepsilon^2} \left(\log(2^\eta - 2) - \log(\delta) \right) \right\rceil$$

where: η is the total number of anon. walks of length l.

For example:
There are $\eta = 877$ anonymous walks of length $l = 7$. If we set $\varepsilon = 0.1$ and $\delta = 0.01$ then we need to generate m=122,500 random walks

New idea: Learn Walk Embeddings

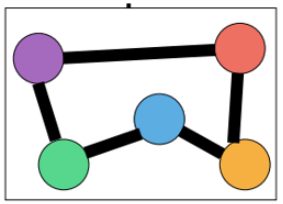
Rather than simply representing each walk by the fraction of times it occurs, we **learn embedding z_i of anonymous walk w_i.**

- **Learn a graph embedding z_G together with all the anonymous walk embeddings z_i**
  $Z = \{z_i : i = 1 \ldots \eta\}$, where η is the number of sampled anonymous walks.

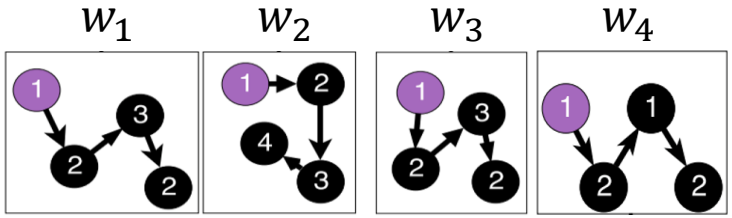
How to embed walks?

- **Idea:** Embed walks s.t. the next walk can be predicted.

Learn Walk Embeddings



- Output: A vector $\boldsymbol{z_G}$ for input graph $\boldsymbol{G}$
  - The embedding of entire graph to be learned
- Starting from **node 1**: Sample anonymous random walks, e.g.



- **Learn to predict walks that co-occur in Δ-size window** (e.g., predict w_3 given w_1, w_2 if $\Delta = 2$)
- Objective:

$$\max_{\boldsymbol{z_G}} \sum_{t=\Delta+1}^{T} \log P(w_t | w_{t-\Delta}, \ldots, w_{t-1}, \boldsymbol{z_G})$$

  - Where T is the total number of walks

Learn Walk Embeddings

- **Run T different random walks from u each of length l:**

$$N_R(u) = \{w_1^u, w_2^u \dots w_T^u\}$$

- **Learn to predict walks that co-occur in Δ-size window**
- Estimate embedding $\boldsymbol{z}_i$ of anonymous walk w_i.
  Let η be number of all possible walk embeddings.

Objective: $$\max_{\boldsymbol{z}_i, \boldsymbol{z}_G} \frac{1}{T} \sum_{t=\Delta}^{T} \log P(w_t | \{w_{t-\Delta}, \dots, w_{t-1}, \boldsymbol{z}_G\})$$

- $P(w_t | \{w_{t-\Delta}, \dots, w_{t-1}, \boldsymbol{z}_G\}) = \frac{\exp(y(w_t))}{\sum_{i=1}^{\eta} \exp(y(w_i))}$
- $y(w_t) = b + U \cdot \left(cat(\frac{1}{\Delta} \sum_{i=1}^{\Delta} \boldsymbol{z}_i, \boldsymbol{z}_G) \right)$

All possible anonymous walks (requires negative sampling)

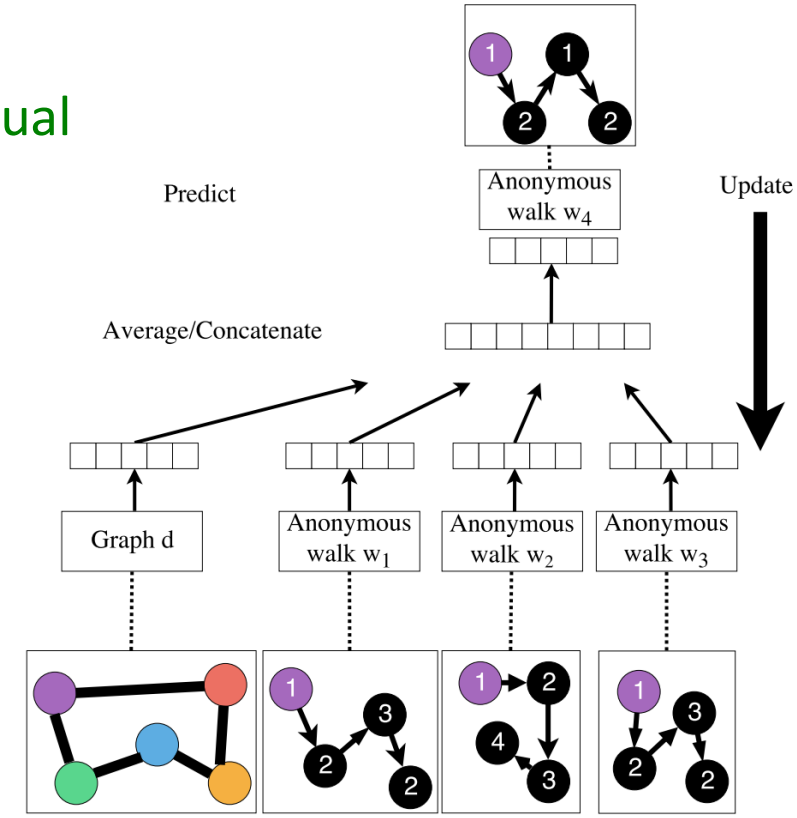
  - $cat(\frac{1}{\Delta} \sum_{i=1}^{\Delta} \boldsymbol{z}_i, \boldsymbol{z}_G)$ means an average of anonymous walk embeddings $\boldsymbol{z}_i$ in the window, concatenated with the graph embedding $\boldsymbol{z}_G$.
  - $b \in \mathbb{R}$, $U \in \mathbb{R}^D$ are learnable parameters. This represents a linear layer.

Anonymous Walk Embeddings, ICML 2018 https://arxiv.org/pdf/1805.11921.pdf

Learn Walk Embeddings

- We obtain the graph embedding z_G (learnable parameter) after the optimization.
  - Is z_G simply the sum over walk embeddings z_i? Or is z_G the residual embedding next to z_i?
  - According to the paper, z_G is a separately optimized vector parameter, just like other z_i's.
- Use z_G to make predictions (e.g., graph classification):
  - **Option1**: Inner product Kernel $z_{G_1}^T z_{G_2}$ (Lecture 2)
  - **Option2**: Use a neural network that takes z_G as input to classify G.

Overall Architecture



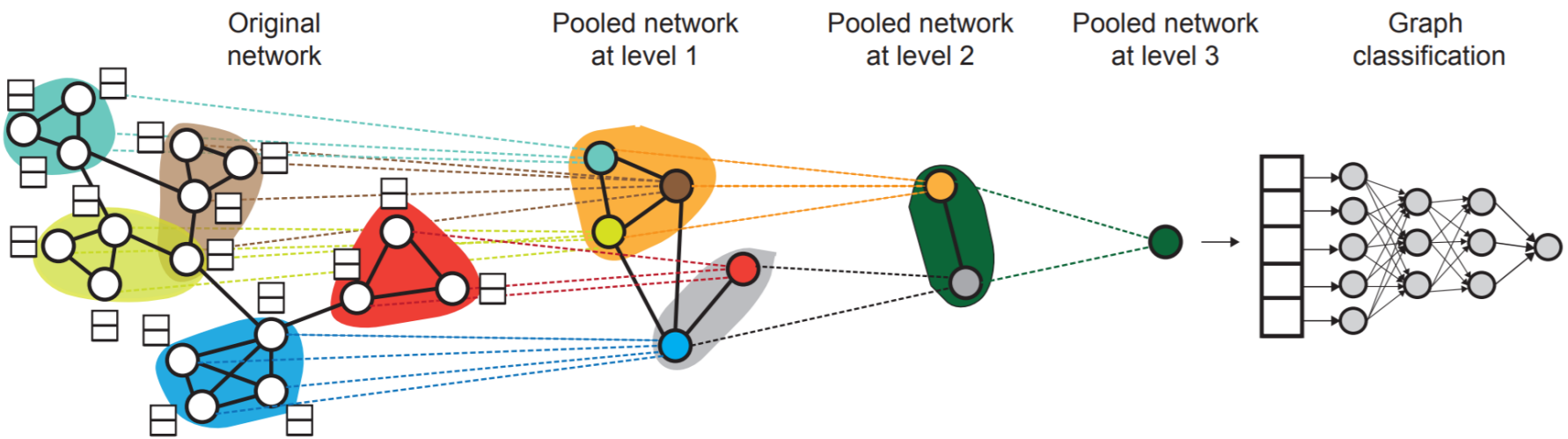
Summary

We discussed 3 ideas to graph embeddings:

- **Approach 1:** Embed nodes and sum/avg them
- **Approach 2:** Create super-node that spans the (sub) graph and then embed that node.
- **Approach 3:** Anonymous Walk Embeddings
  - Idea 1: Sample the anon. walks and represent the graph as fraction of times each anon walk occurs.
  - Idea 2: Learn graph embedding together with anonymous walk embeddings.

Preview: Hierarchical Embeddings

- We will discuss more advanced ways to obtain graph embeddings in Lecture 8.
- We can **hierarchically** cluster nodes in graphs, and **sum/avg** the node embeddings according to these clusters.



How to Use Embeddings

- **How to use embeddings z_i of nodes:**
  - **Clustering/community detection:** Cluster points $\boldsymbol{z_i}$
  - **Node classification:** Predict label of node i based on $\boldsymbol{z_i}$
  - **Link prediction:** Predict edge (i, j) based on $(\boldsymbol{z_i}, \boldsymbol{z_j})$
    - Where we can: concatenate, avg, product, or take a difference between the embeddings:
      - Concatenate: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g([\boldsymbol{z}_i, \boldsymbol{z}_j])$
      - Hadamard: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(\boldsymbol{z}_i * \boldsymbol{z}_j)$ (per coordinate product)
      - Sum/Avg: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(\boldsymbol{z}_i + \boldsymbol{z}_j)$
      - Distance: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(||\boldsymbol{z}_i - \boldsymbol{z}_j||_2)$
  - **Graph classification**: Graph embedding $\boldsymbol{z_G}$ via aggregating node embeddings or anonymous random walks. Predict label based on graph embedding $\boldsymbol{z}_G$.

Today's Summary

We discussed **graph representation learning**, a way to learn **node and graph embeddings** for downstream tasks, **without feature engineering**.

- **Encoder-decoder framework:**
  - **Encoder: embedding lookup**
  - **Decoder: predict score based on embedding to match node similarity**
- **Node similarity measure: (biased) random walk**
  - **Examples: DeepWalk, Node2Vec**
- **Extension to Graph embedding: Node embedding aggregation and Anonymous Walk Embeddings**

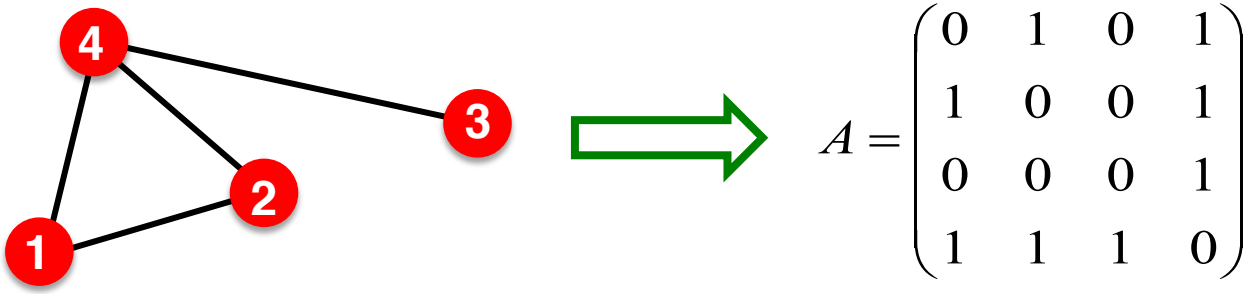
Stanford CS224W: Graph as Matrix: PageRank, Random Walks and Embeddings

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!**



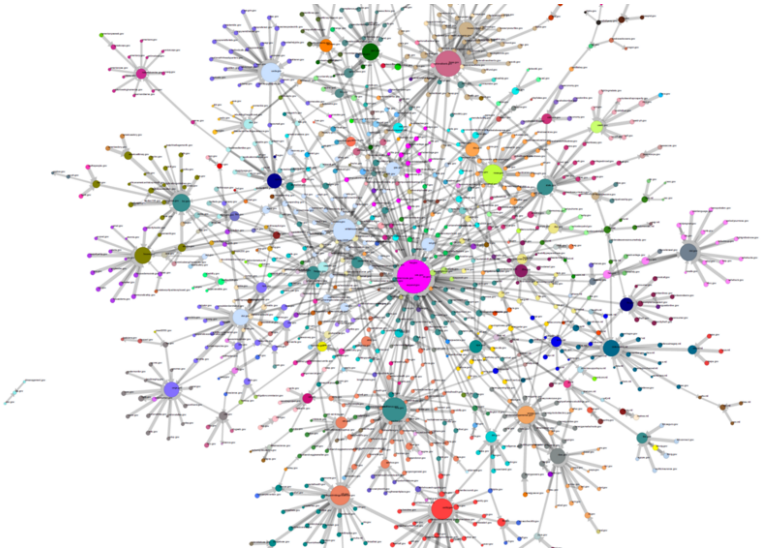
Stanford CS224W: PageRank (aka the Google Algorithm)

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

Example: The Web as a Graph

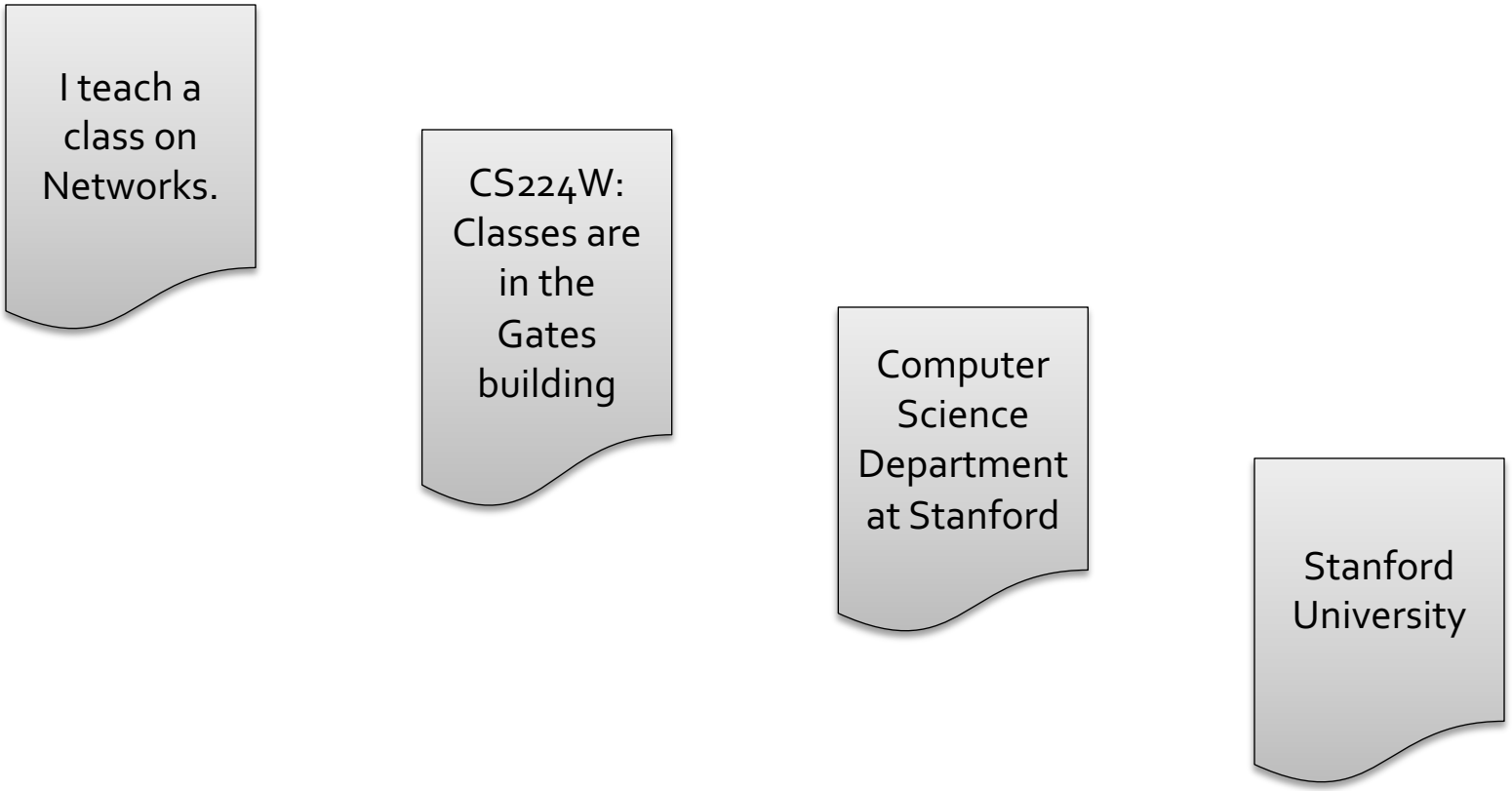
Q: What does the Web "look like" at a global level?

- **Web as a graph:**
  - Nodes = web pages
  - Edges = hyperlinks

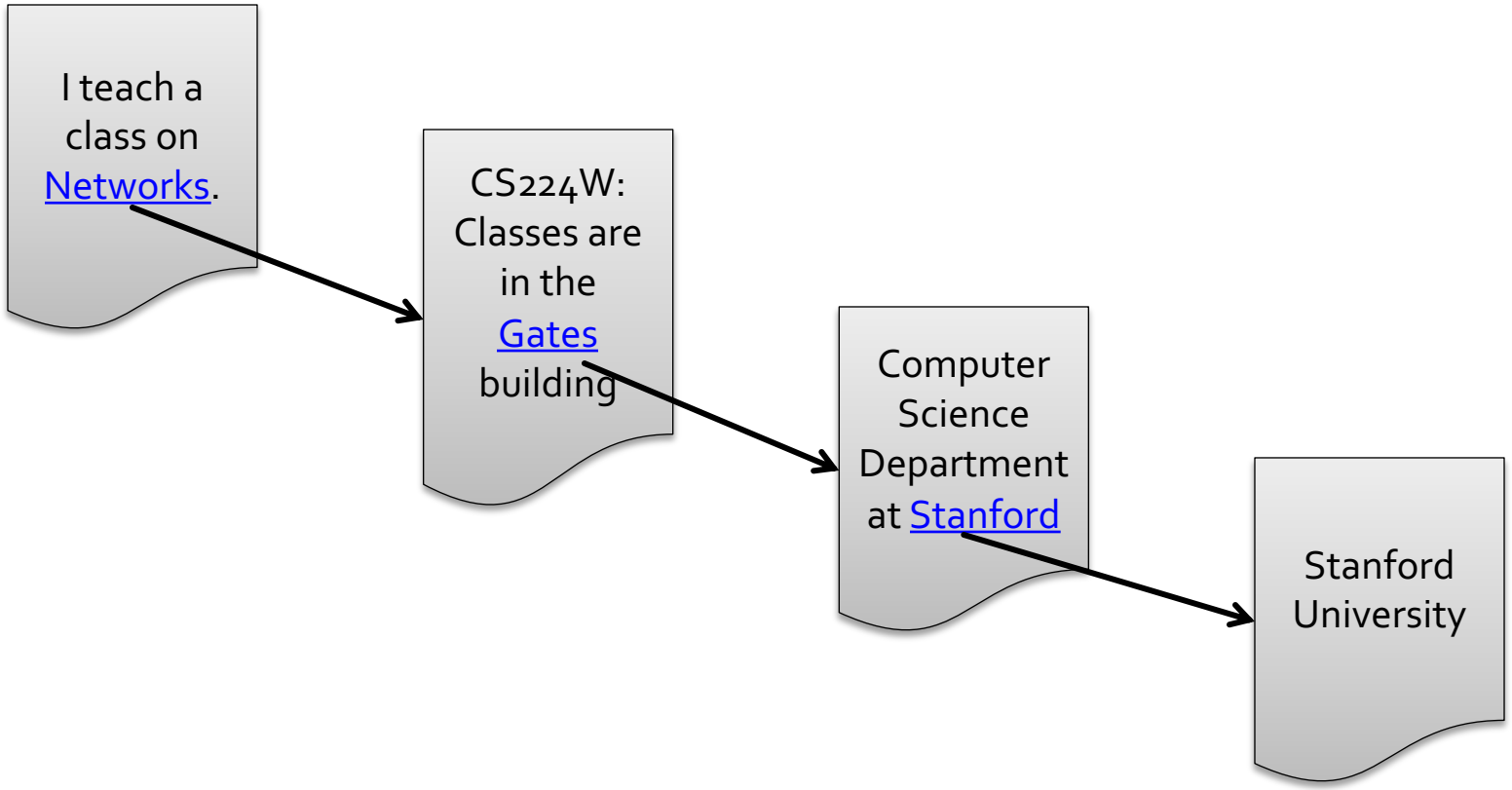


  - **Side issue:** What is a node?
    - Dynamic pages created on the fly
    - "dark matter" – inaccessible database generated pages

The Web as a Graph

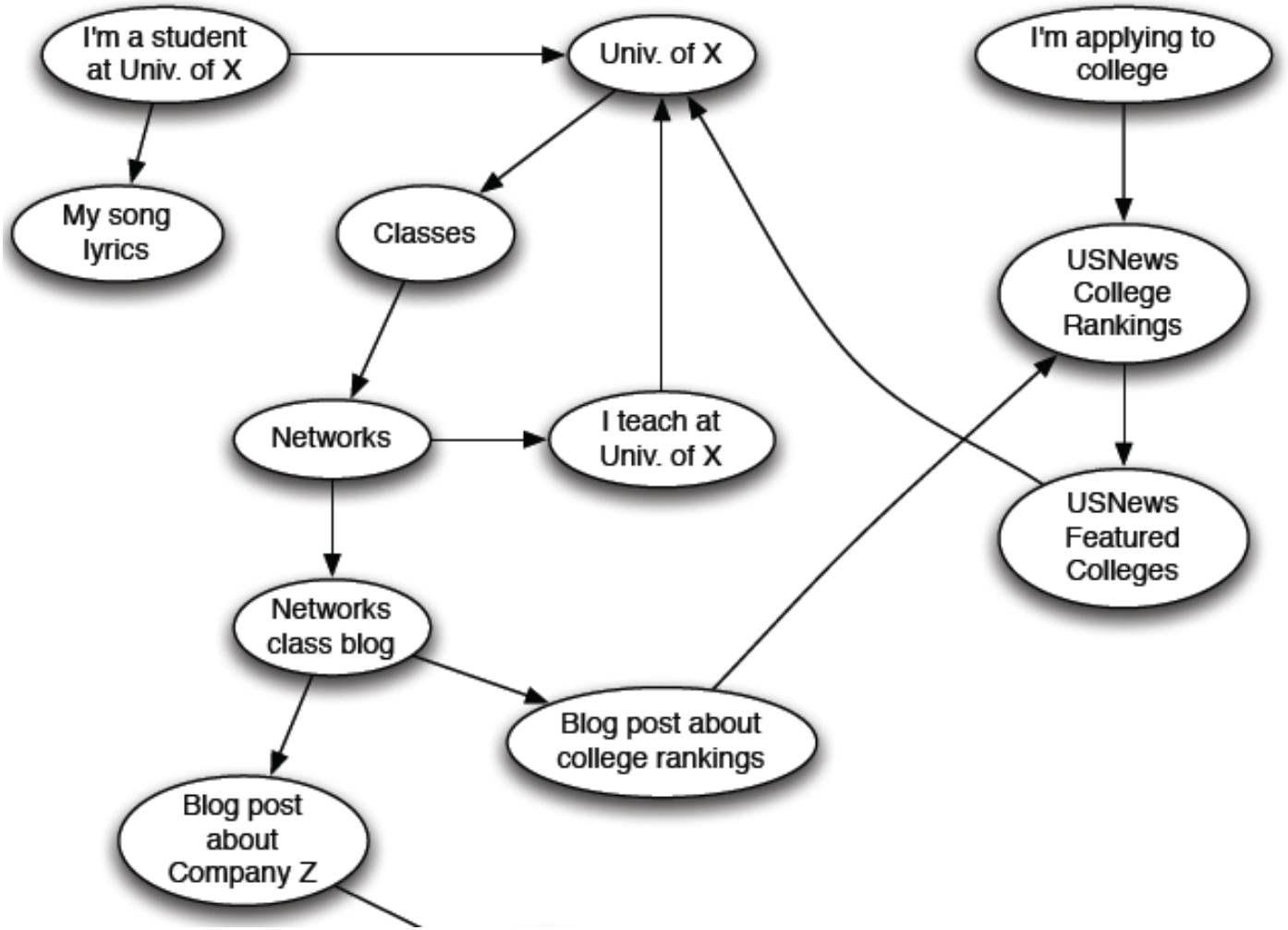


The Web as a Graph

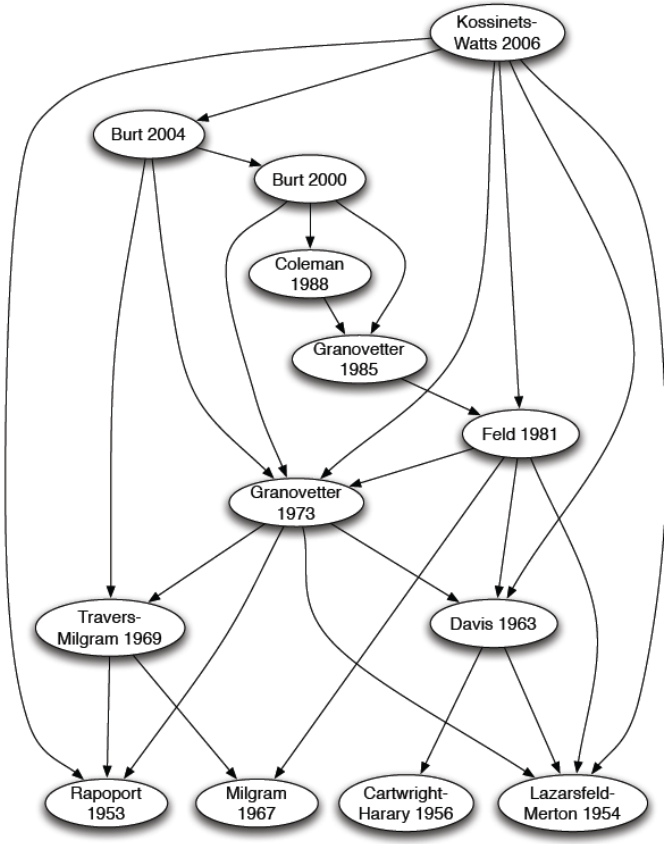


- In early days of the Web links were **navigational**
- Today many links are **transactional** (used not to navigate from page to page, but to post, comment, like, buy, …)

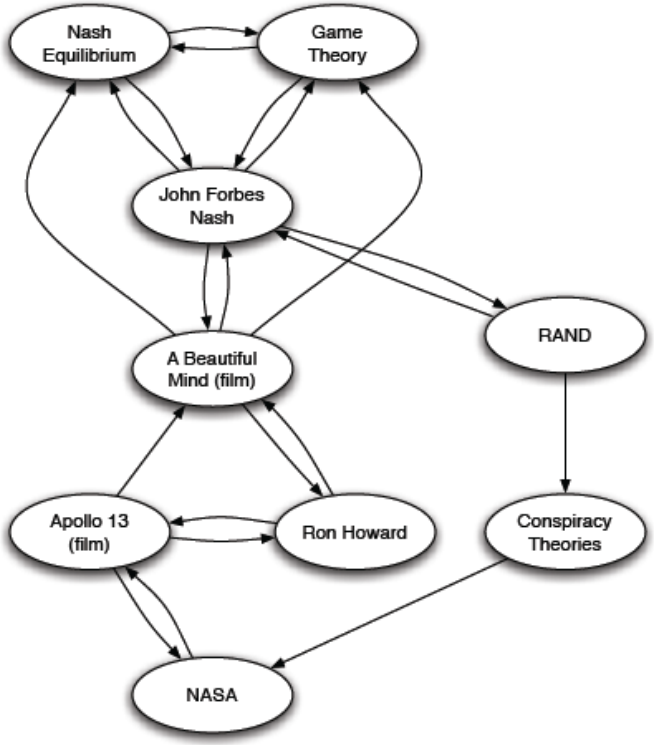
The Web as a Directed Graph



Other Information Networks



Citations



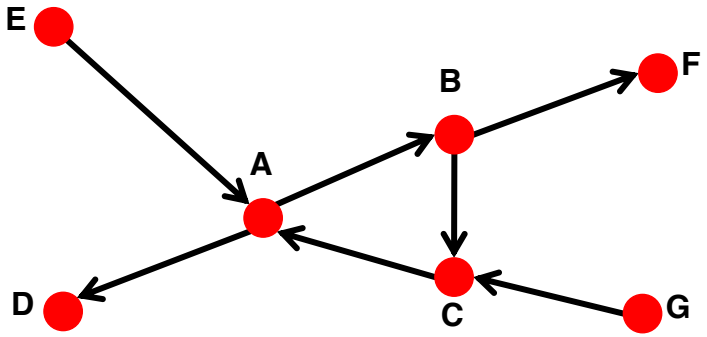
References in an Encyclopedia

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?

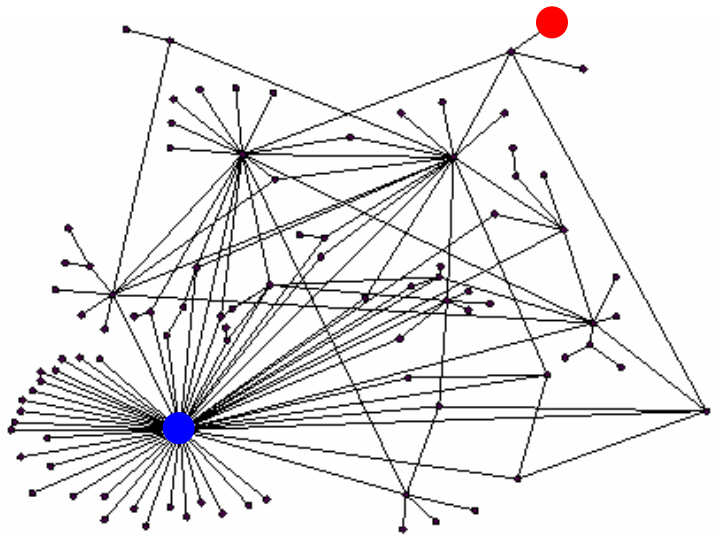


Ranking Nodes on the Graph

- **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

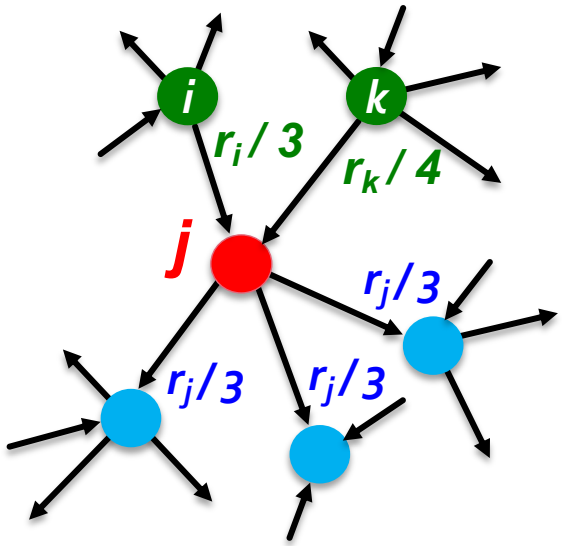
- We will cover the following **Link Analysis approaches** to compute the **importance** of nodes in a graph:
  - PageRank
  - Personalized PageRank (PPR)
  - Random Walk with Restarts

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



$r_j = r_i/3 + r_k/4$

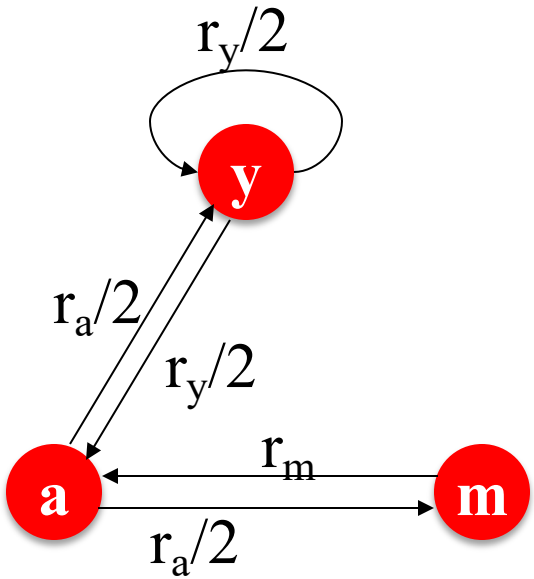
PageRank: The "Flow" Model

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

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

d_i … out-degree of node i

The web in 1839



"Flow" equations:

$r_y = r_y/2 + r_a/2$

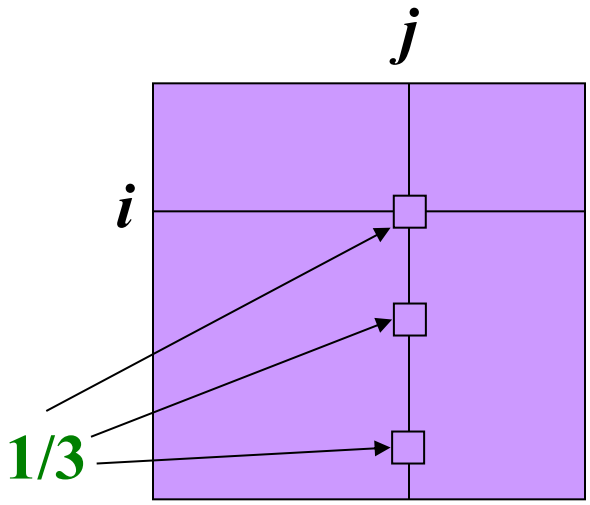
$r_a = r_y/2 + r_m$

$r_m = r_a/2$

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

PageRank: Matrix Formulation

- **Stochastic adjacency matrix M**
  - Let page j have d_j out-links
  - If $j \rightarrow i$, then $M_{ij} = \frac{1}{d_j}$
    - M is a **column stochastic matrix**
      - **Columns** sum to **1**



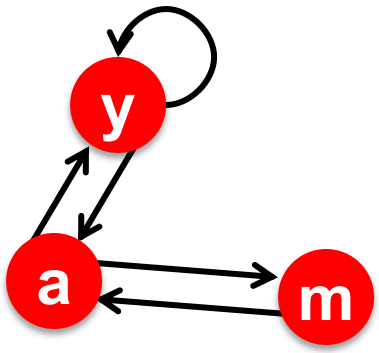
- **Rank vector r:** An entry per page
  - r_i is the importance score of page i
  - $\sum_i r_i = 1$
- **The flow equations can be written**

$$r = M \cdot r$$

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

M

Example: Flow Equations & M



	r_y	r_a	r_m
r_y	½	½	0
r_a	½	0	1
r_m	0	½	0

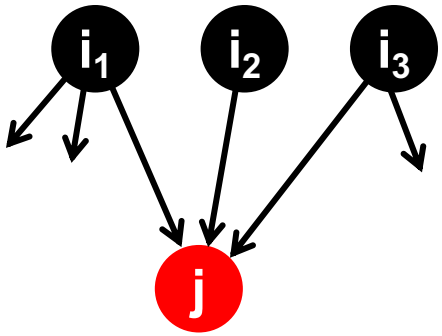
$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix}$$

$$r \qquad M \qquad r$$

Connection to Random Walk

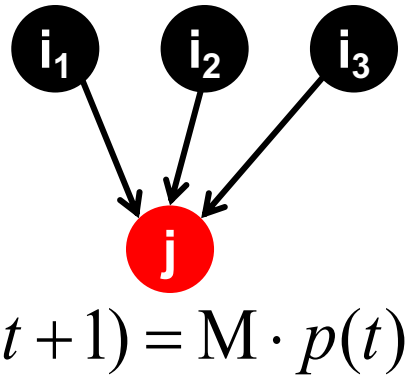
- **Imagine a random web surfer:**
  - At any time t, surfer is on some page i
  - At time $t+1$, the surfer follows an out-link from i uniformly at random
  - Ends up on some page j linked from i
  - Process repeats indefinitely



$$r_j = \sum_{i \to j} \frac{r_i}{\mathrm{d}_{\mathrm{out}}(\mathrm{i})}$$

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

The Stationary Distribution



- **Where is the surfer at time $t+1$?**
  - Follow a link uniformly at random

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

- Suppose the random walk reaches a state

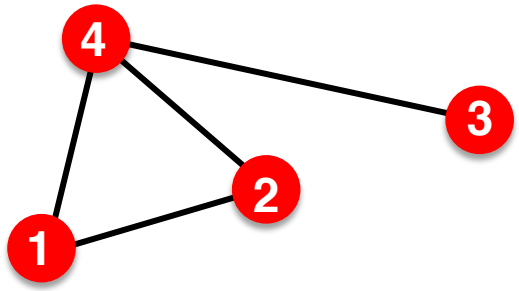
$$\boldsymbol{p(t+1) = M \cdot p(t) = p(t)}$$

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

- **Our original rank vector $\boldsymbol{r}$** satisfies $\boldsymbol{r = M \cdot r}$
  - **So, $\boldsymbol{r}$ is a stationary distribution for the random walk**

Recall Eigenvector of A Matrix

- Recall from lecture 2 (eigenvector centrality), let $\boldsymbol{A \in \{0,1\}^{n \times n}}$ 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 $\boldsymbol{\lambda c = Ac}$
- $\boldsymbol{c}$: eigenvector; $\boldsymbol{\lambda}$: eigenvalue

- Note:
  - This is the definition of eigenvector centrality (for undirected graphs).
  - PageRank is defined for directed graphs

Eigenvector Formulation

- **The flow equation:**

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

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix}$$

$\boldsymbol{r}$ $\quad$ $\boldsymbol{M}$ $\quad$ $\boldsymbol{r}$

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

PageRank: Summary

- **PageRank:**
  - Measures importance of nodes in a graph using the link structure of the web
  - Models a random web surfer using the stochastic adjacency matrix $\boldsymbol{M}$
  - PageRank solves $\boldsymbol{r} = \boldsymbol{M}\boldsymbol{r}$ where $\boldsymbol{r}$ can be viewed as both the principle eigenvector of $\boldsymbol{M}$ and as the stationary distribution of a random walk over the graph

Stanford CS224W: PageRank: How to solve?

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

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

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

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: $\boldsymbol{r}^0 = [1/N, \ldots., 1/N]^T$
  - Iterate: $\boldsymbol{r}^{(t+1)} = \boldsymbol{M} \cdot \boldsymbol{r}^t$
  - Stop when $|\boldsymbol{r}^{(t+1)} - \boldsymbol{r}^t|_1 < \varepsilon$

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

d_i …. out-degree of node i

$|x|_1 = \sum_1^N |x_1|$ is the **L1** norm
Can use any other vector norm, e.g., Euclidean

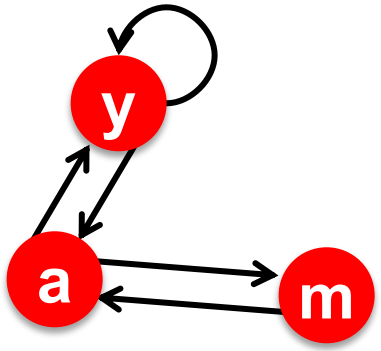
About 50 iterations is sufficient to estimate the limiting solution.

PageRank: How to solve?

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

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

Iteration 0, 1, 2, …

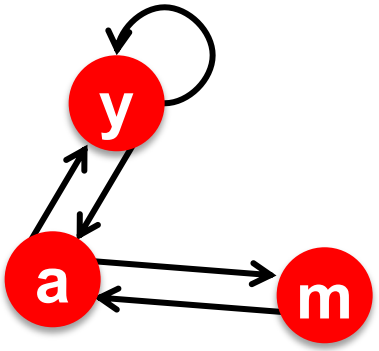


	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + rm$$
$$r_m = r_a/2$$

PageRank: How to solve?

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



	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + rm$$
$$r_m = r_a/2$$

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

Iteration 0, 1, 2, …

PageRank: Three Questions

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

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

PageRank: Problems

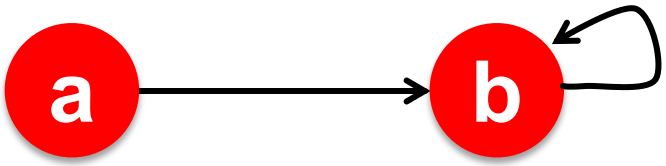
Two problems:

- **(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:**



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

- **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:**

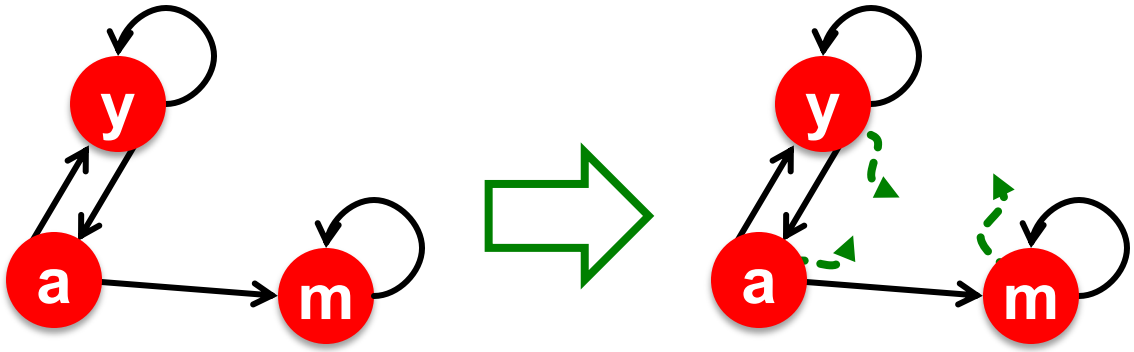
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

- **Example:**

Iteration:	0,	1,	2,	3…
r_a	1	0	0	0
r_b	0	1	0	0

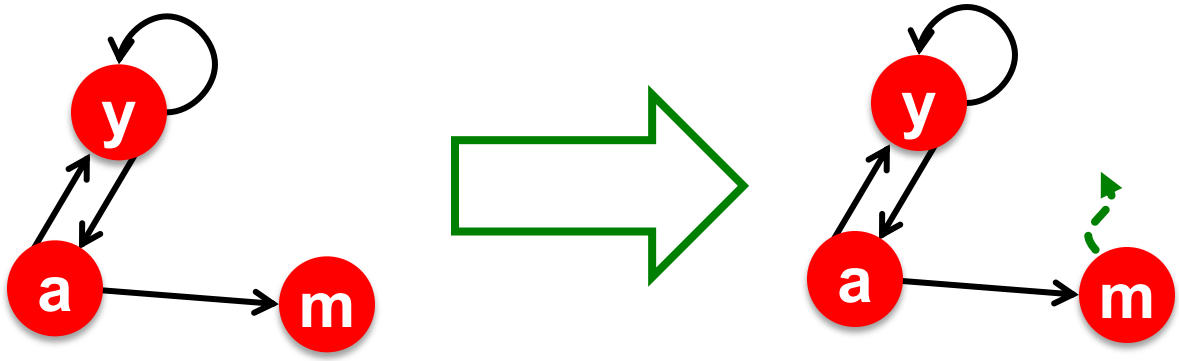
Solution to Spider Traps

- **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-β,** 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	½	½	0
a	½	0	0
m	0	½	0

	y	a	m
y	½	½	**⅓**
a	½	0	**⅓**
m	0	½	**⅓**

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 $1\text{-}\beta$, jump to some random page

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

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

d_i … out-degree of node i

This formulation assumes that $\boldsymbol{M}$ has no dead ends. We can either preprocess matrix $\boldsymbol{M}$ 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]

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

- **The Google Matrix *G*:**

$[1/N]_{NxN}$…N by N matrix where all entries are 1/N

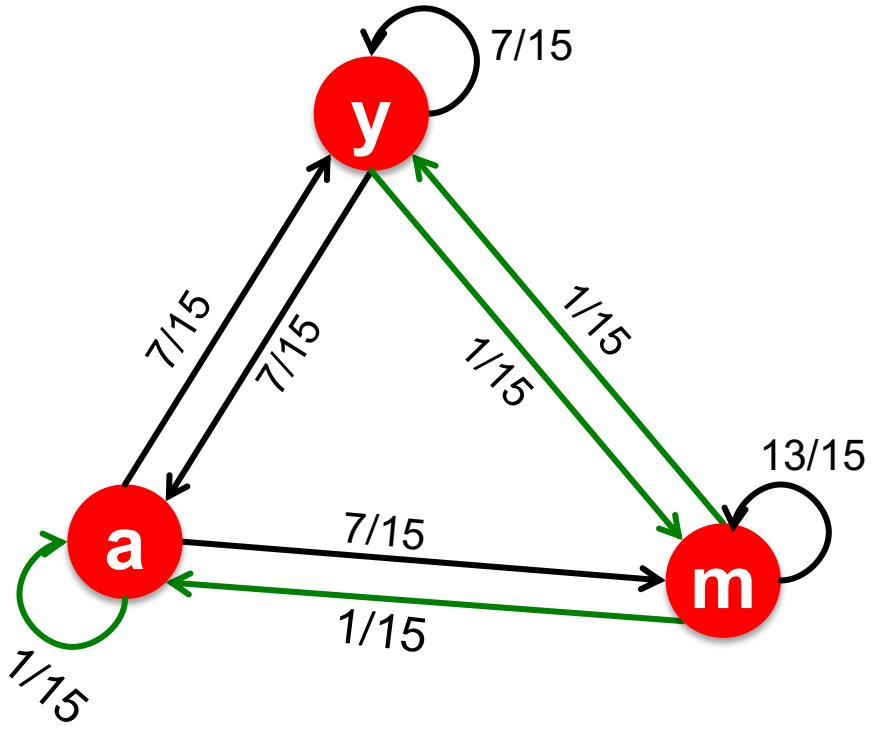
$$G = \beta\, M + (1 - \beta) \left[\frac{1}{N}\right]_{N \times N}$$

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

  And the Power method still works!

- **What is β?**
  - In practice $\beta = 0.8, 0.9$ (make 5 steps on avg., jump)

Random Teleports ($\beta = 0.8$)



M

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

$[1/N]_{NxN}$

G

$$\begin{array}{c} \\ y \\ a \\ m \end{array} \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix}$$

$$\begin{array}{c} y \\ a \\ m \end{array} = \begin{array}{c} 1/3 \\ 1/3 \\ 1/3 \end{array} \begin{array}{c} 0.33 \\ 0.20 \\ 0.46 \end{array} \begin{array}{c} 0.24 \\ 0.20 \\ 0.52 \end{array} \begin{array}{c} 0.26 \\ 0.18 \\ 0.56 \end{array} \ldots \begin{array}{c} 7/33 \\ 5/33 \\ 21/33 \end{array}$$

PageRank Example

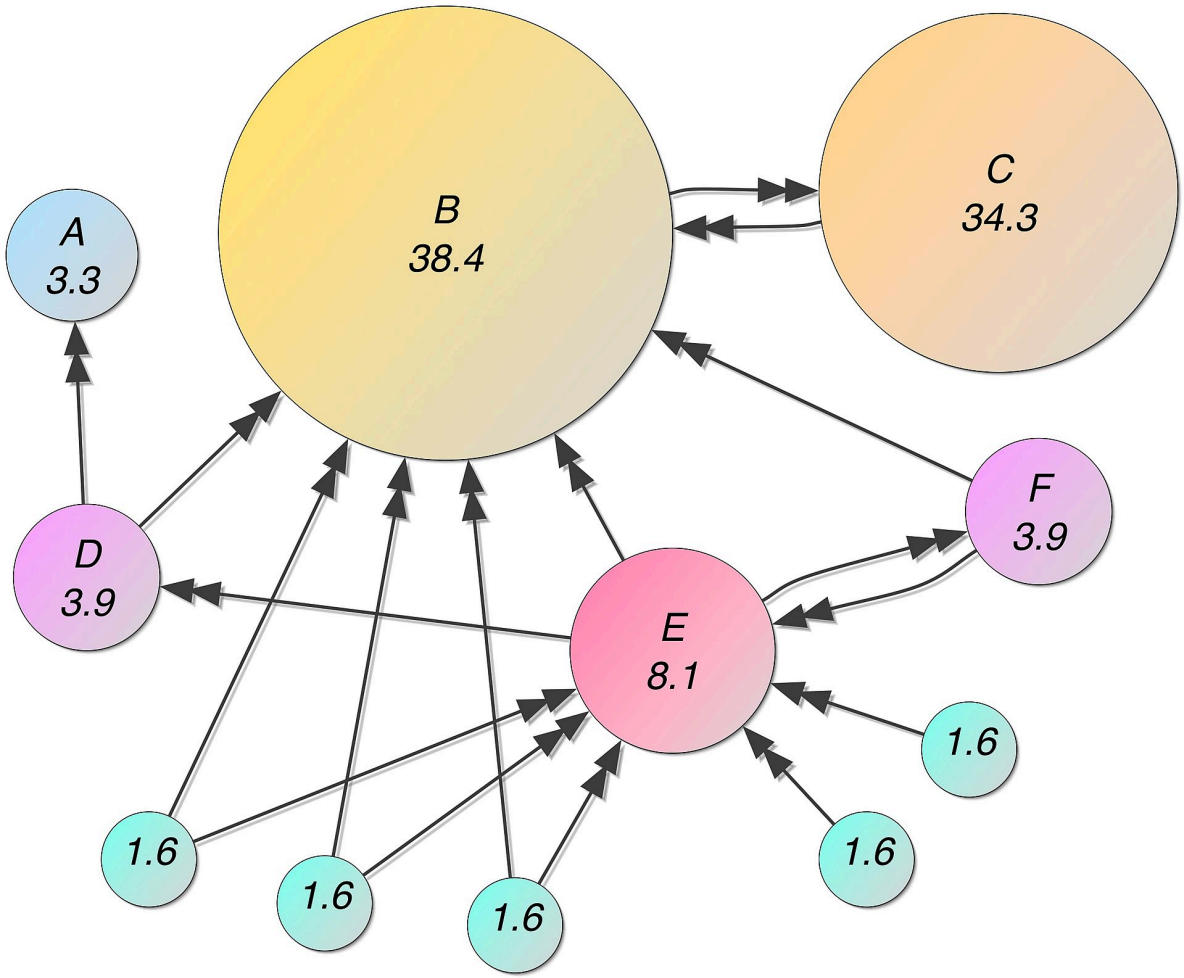


Image credit: Wikipedia

Solving PageRank: Summary

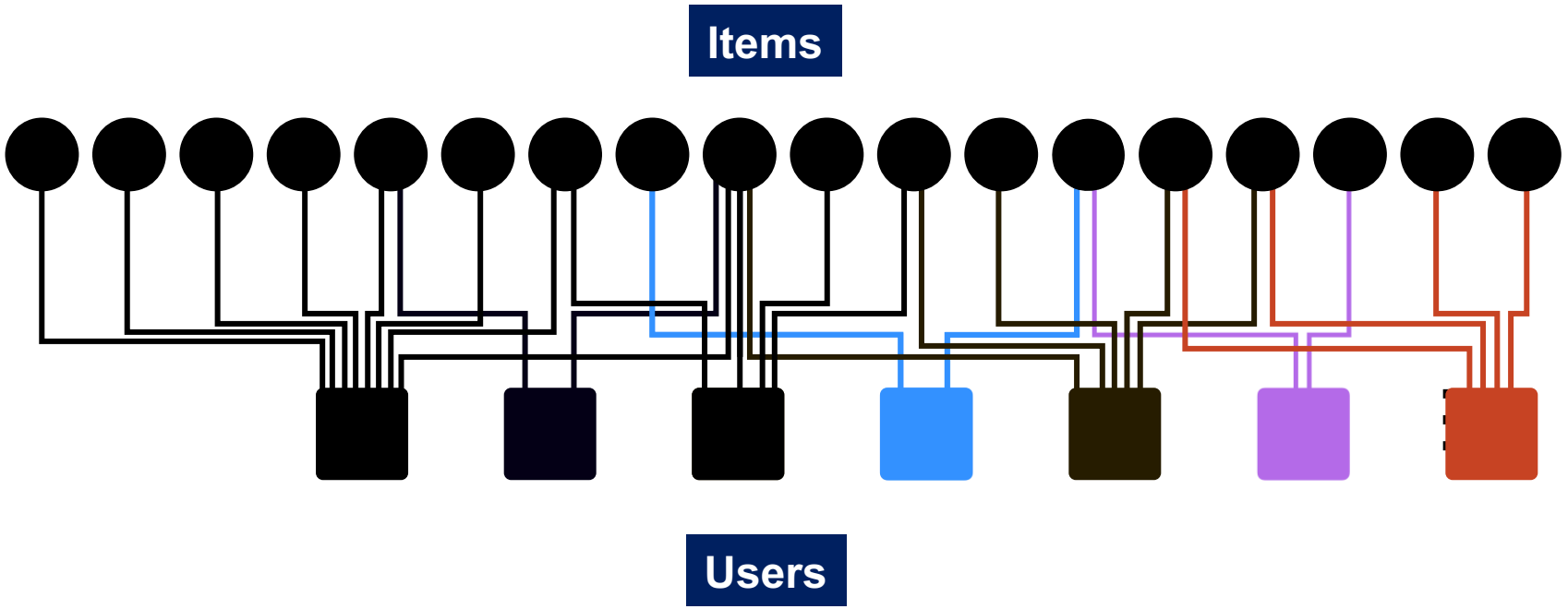
- PageRank solves for $\boldsymbol{r} = \boldsymbol{G}\boldsymbol{r}$ and can be efficiently computed by power iteration of the stochastic adjacency matrix ($\boldsymbol{G}$)
- Adding random uniform teleportation solves issues of dead-ends and spider-traps

Stanford CS224W: Random Walk with Restarts and Personalized PageRank

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

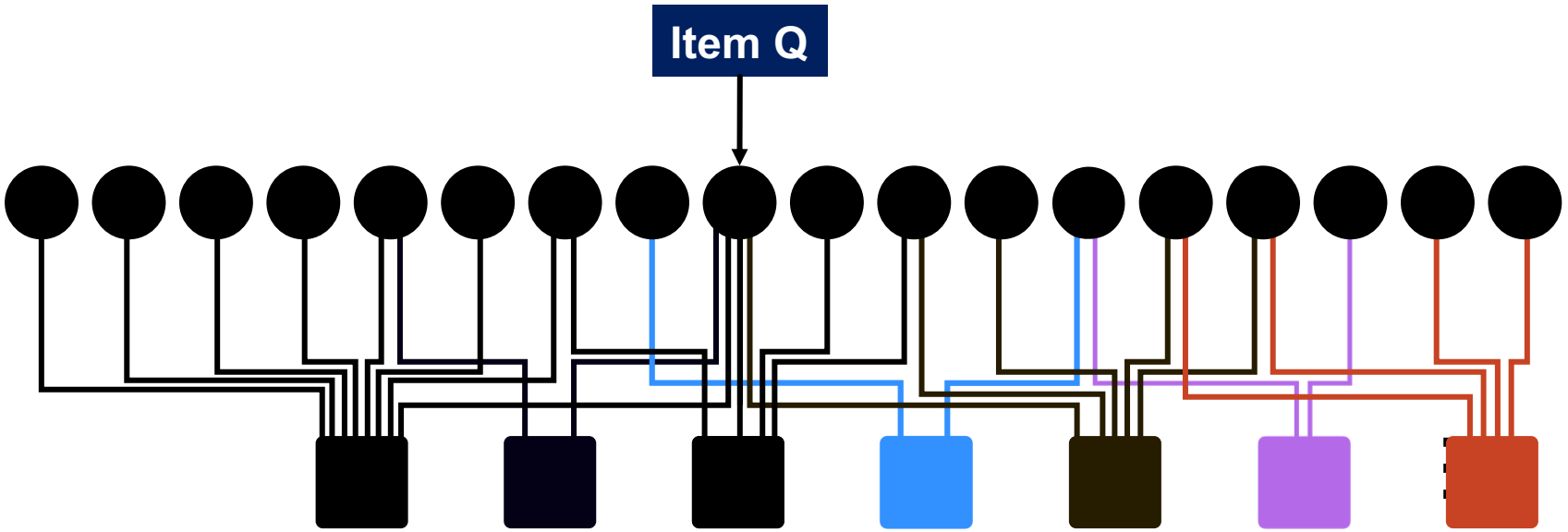
Example: Recommendation

- **Given:**
  A bipartite graph representing user and item interactions (e.g. purchase)



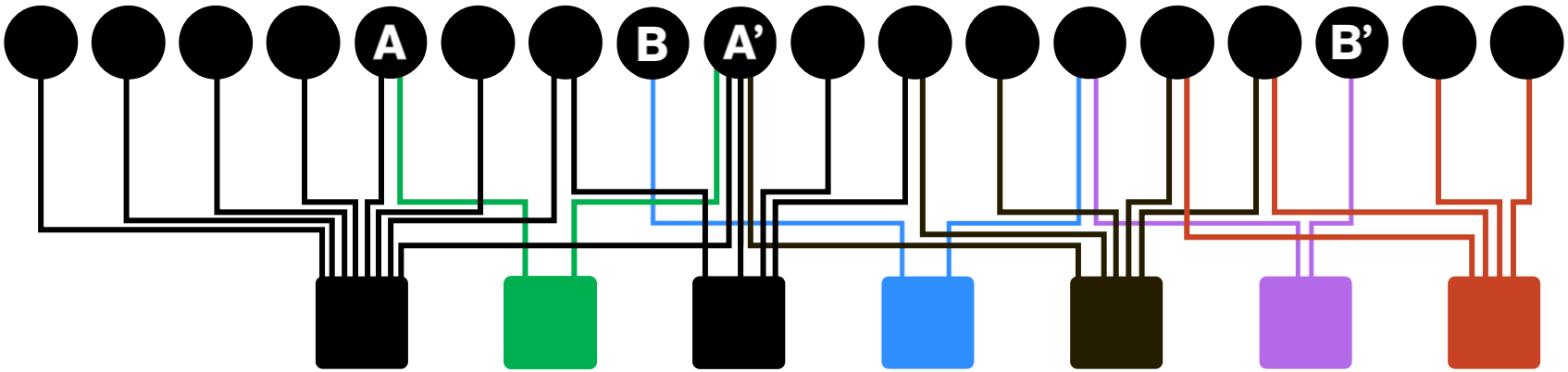
Bipartite User-Item Graph

- **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



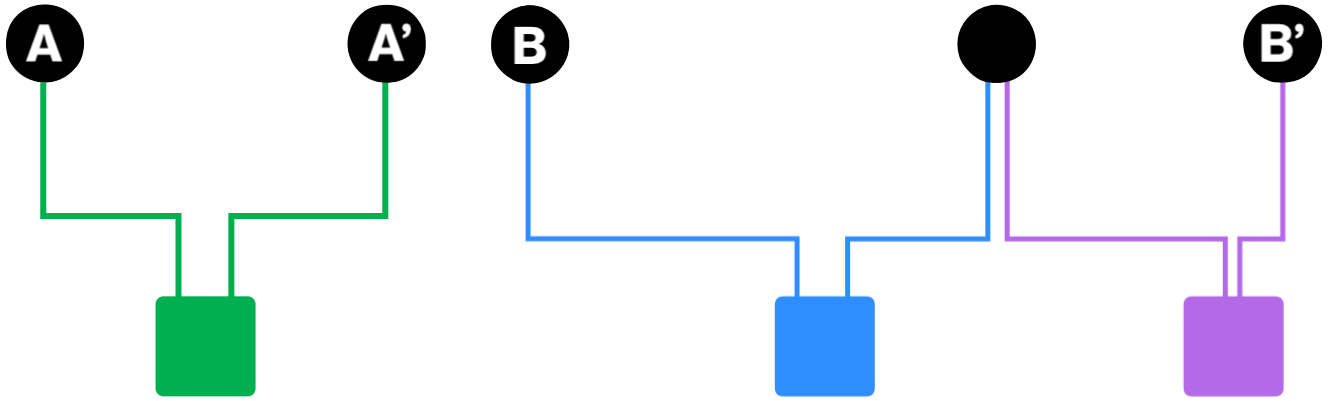
Bipartite User-to-Item Graph

- Which is more related A,A’ or B,B’?



Node proximity Measurements

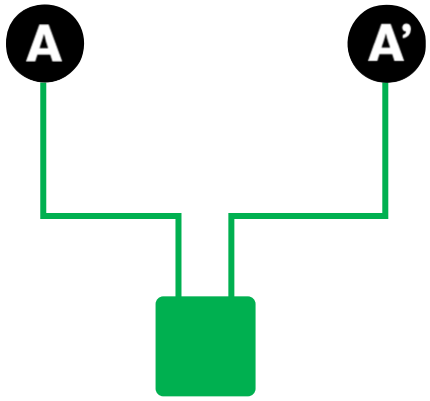
- Which is more related A,A', B,B' or C,C'?



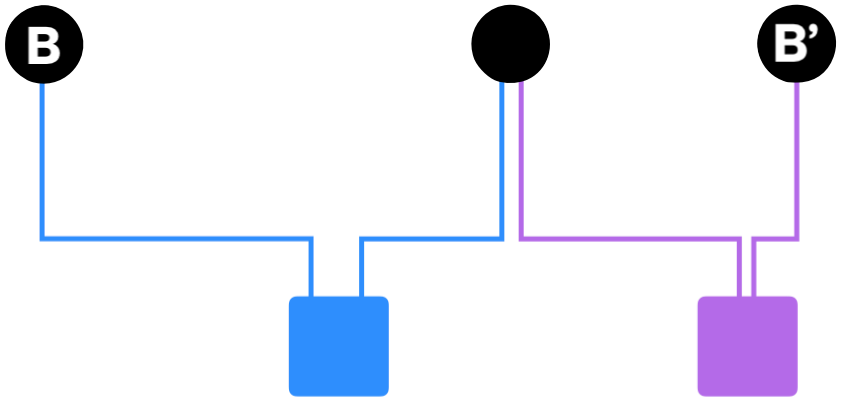
Shortest path

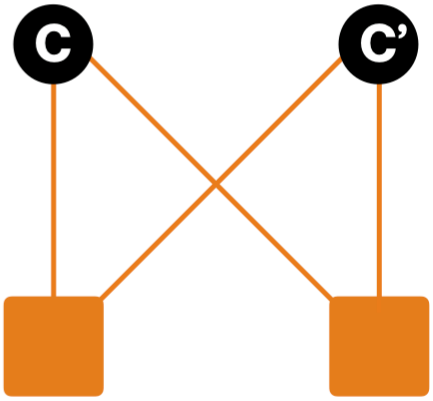
Node proximity Measurements

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



Shortest path

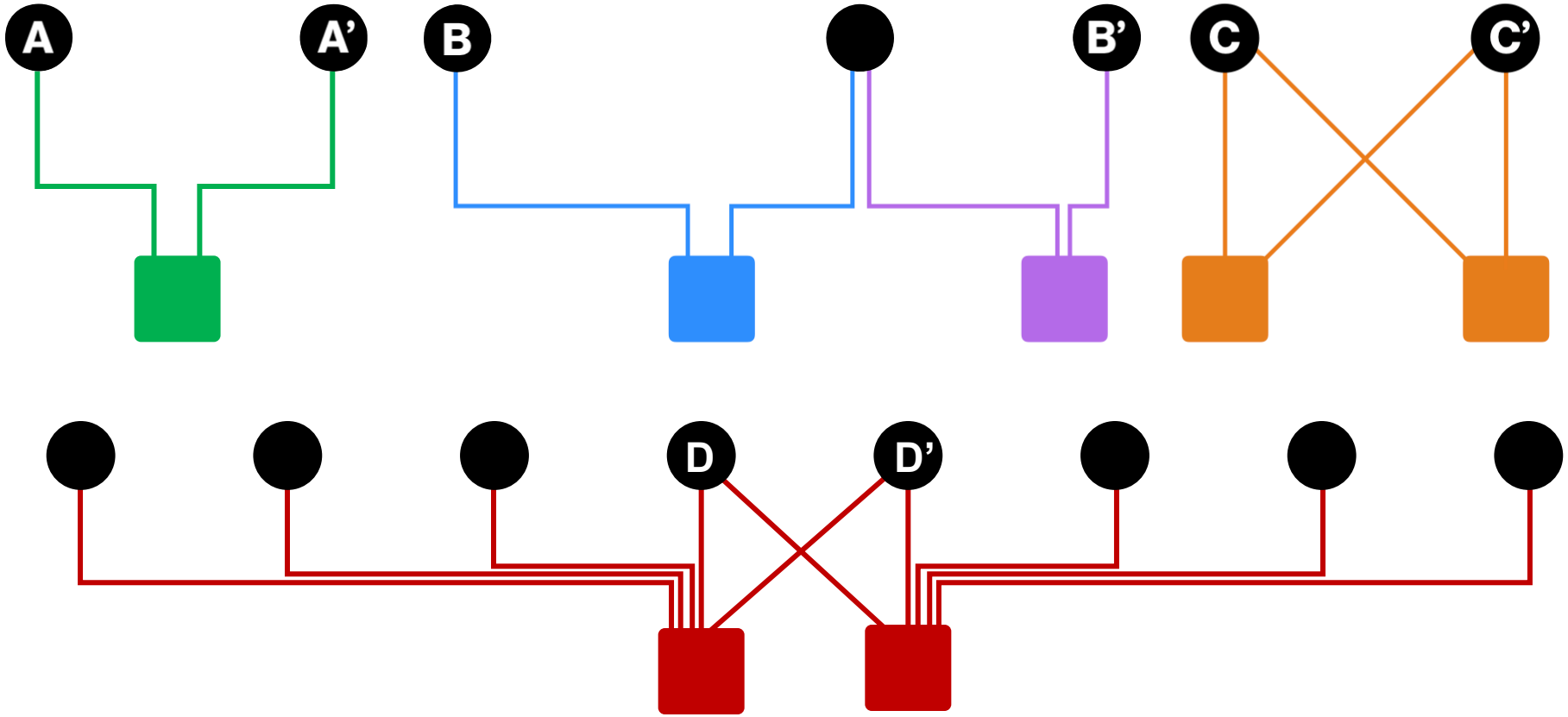




Common Neighbors

Node proximity Measurements

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



Personalized Page Rank/Random Walk with Restarts

Proximity on Graphs

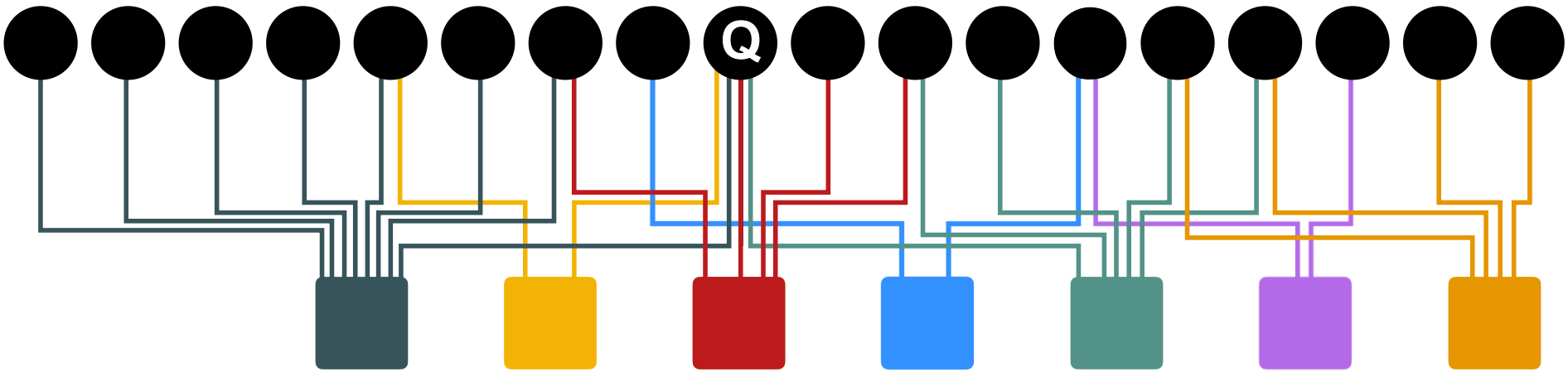
- **PageRank:**
  - Ranks nodes by "importance"
  - Teleports with uniform probability to any node in the network
- **Personalized PageRank:**
  - Ranks proximity of nodes to the teleport nodes $\boldsymbol{S}$
- **Proximity on graphs:**
  - **Q:** What is most related item to **Item Q**?
  - **Random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

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:

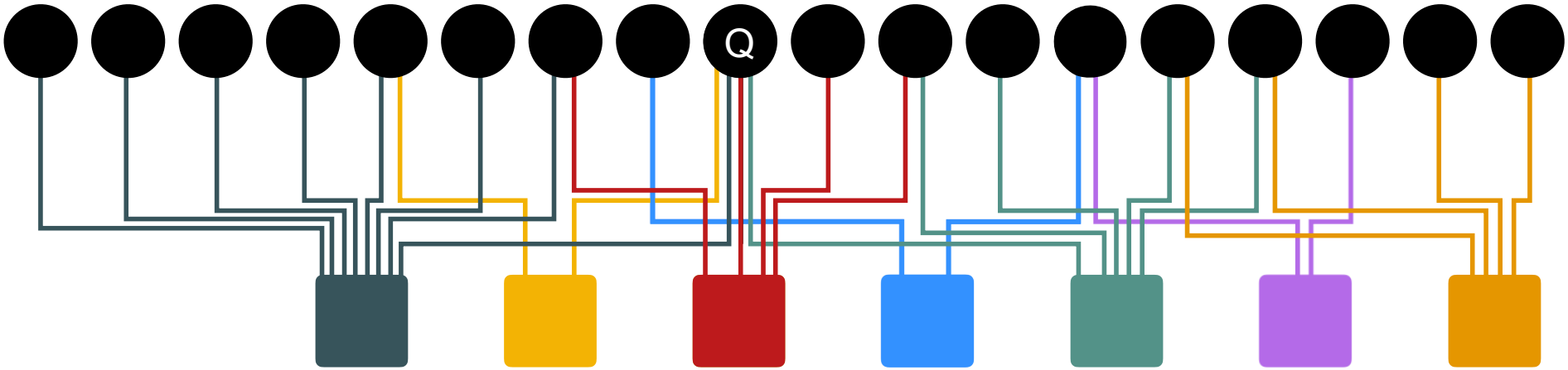


Random Walk Algorithm

- Proximity to query node(s) ***Q***:

```
ALPHA = 0.5
QUERY_NODES = { 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 ( )
```



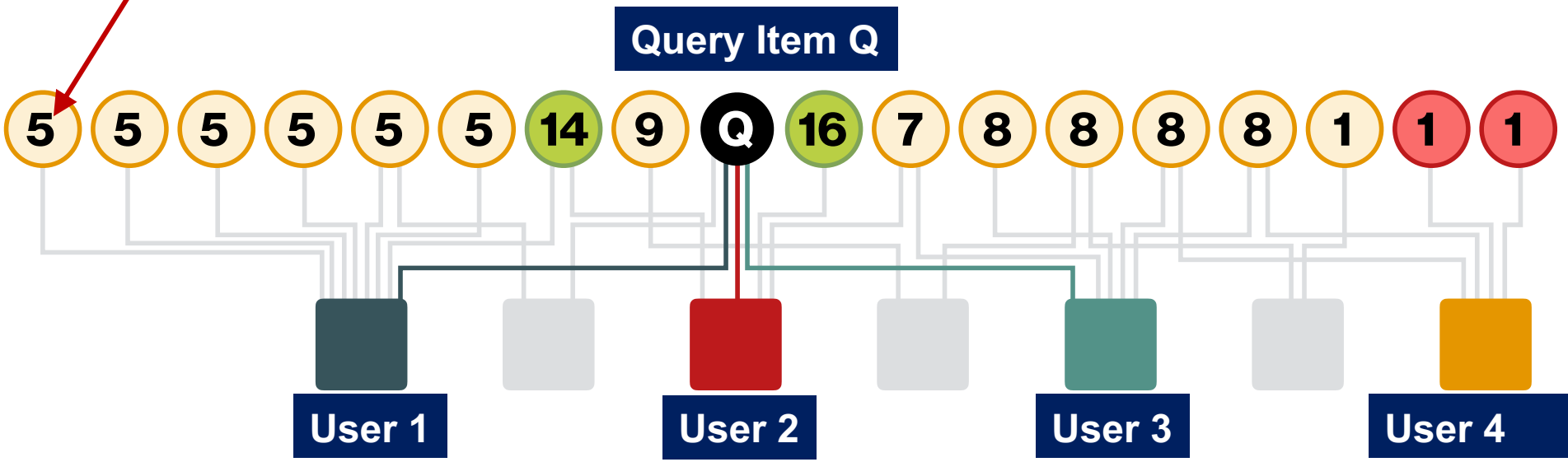
Random Walk Algorithm

- Proximity to query node(s) ***Q***:

```
ALPHA = 0.5
QUERY_NODES = { Q }
```

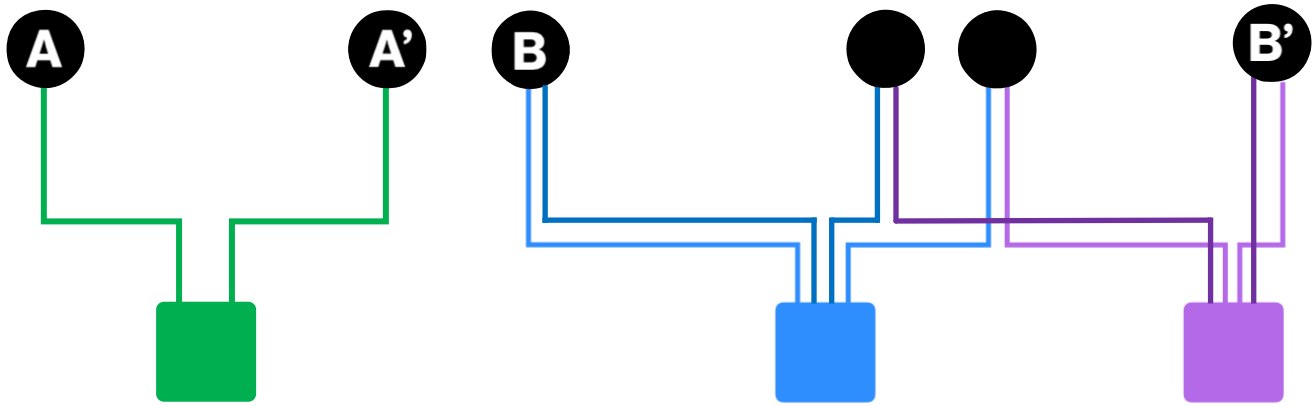
```
item = QUERY_NODES.sample_by_weight( )
for i in range( N_STEPS ):
    user = item.get_random_neighbor( )
    item = user.get_random_neighbor( )
    item.visit_count += 1
    if random( ) < ALPHA:
        item = QUERY_NODES.sample.by_weight ( )
```

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:
    $$\boldsymbol{S} = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$$
- **Topic-Specific PageRank** aka **Personalized PageRank:**
  - Teleports to a specific set of nodes
  - Nodes can have different probabilities of the surfer landing there:
    $$\boldsymbol{S} = [0.1, 0, 0, 0.2, 0, 0, 0.5, 0, 0, 0.2]$$
- **Random Walk with Restarts:**
  - Topic-Specific PageRank where teleport is always to the same node:
    $$\boldsymbol{S} = [0, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0]$$

Summary

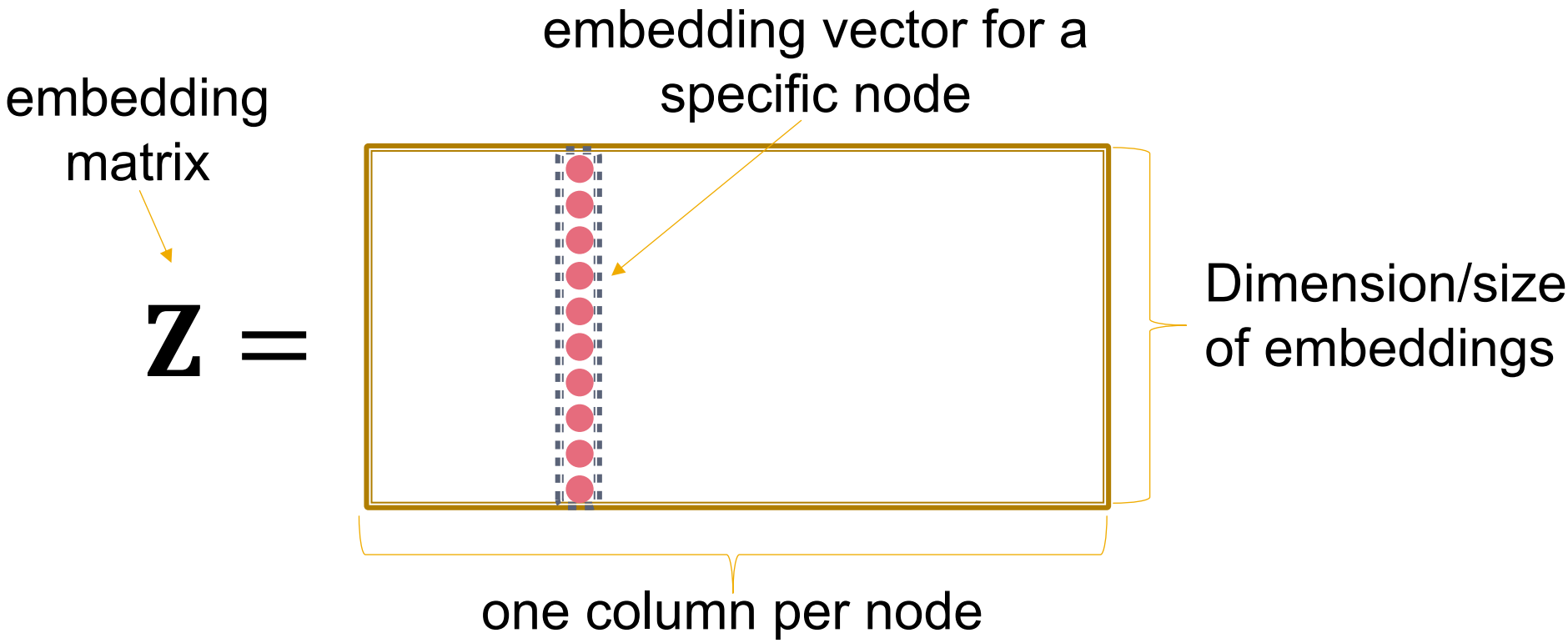
- A graph is naturally represented as a matrix
- We defined a random walk process over the graph
  - Random surfer moving across the links and with random teleportation
  - Stochastic adjacency matrix M
- PageRank = Limiting distribution of the surfer location represented node importance
  - Corresponds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

Stanford CS224W: Matrix Factorization and Node Embeddings

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

Embeddings & Matrix Factorization

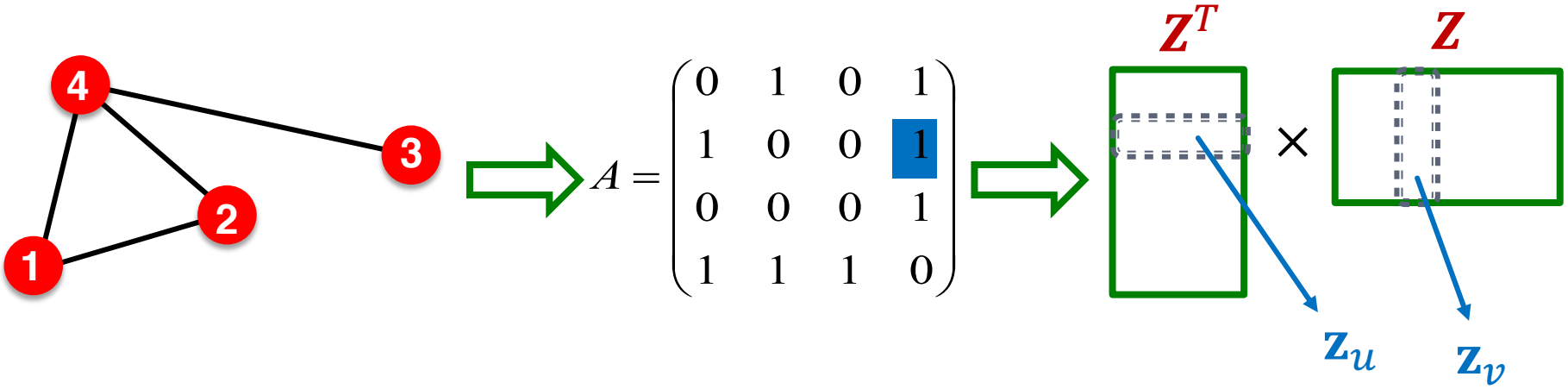
- **Recall:** encoder as an embedding lookup



Objective: maximize $\mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u$ for node pairs (u, v) that are **similar**

Connection to Matrix Factorization

- Simplest **node similarity**: Nodes u, v are similar if they are connected by an edge
- This means: $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u = A_{u,v}$ which is the (u, v) entry of the graph adjacency matrix A
- Therefore, $\boldsymbol{Z}^T \boldsymbol{Z} = A$



Matrix Factorization

- The embedding dimension d (number of rows in $\boldsymbol{Z}$) is much smaller than number of nodes n.
- Exact factorization $A = Z^T Z$ is generally not possible
- However, we can learn $\boldsymbol{Z}$ approximately
- **Objective**: $\min_{Z} \| A - \boldsymbol{Z}^T \boldsymbol{Z} \|_2$
  - We optimize $\boldsymbol{Z}$ such that it minimizes the L2 norm (Frobenius norm) of $A - \boldsymbol{Z}^T \boldsymbol{Z}$
  - Note in lecture 3 we used softmax instead of L2. But the goal to approximate A with $\boldsymbol{Z}^T \boldsymbol{Z}$ is the same.
- Conclusion: **inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization of A**

Random Walk-based Similarity

- **DeepWalk** and **node2vec** have a more complex **node similarity** definition based on random walks
- **DeepWalk** is equivalent to matrix factorization of the following complex matrix expression:

$$log\left(vol(G)\left(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^{r}\right)D^{-1}\right) - \log b$$

- Explanation in next slide

Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec, WSDM 18

Random Walk-based Similarity

Volume of graph

$$vol(G) = \sum_i \sum_j A_{i,j}$$

Diagonal matrix D

$$D_{u,u} = \deg(u)$$

$$\log\left(vol(G)\left(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r\right)D^{-1}\right) - \log b$$

context window size

See Lec 3 slide 30:

$T = |N_R(u)|$

Power of normalized adjacency matrix

Number of negative samples

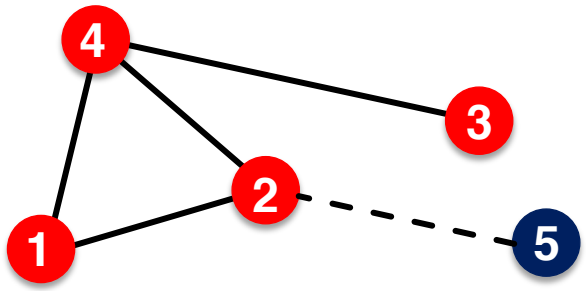
- **Node2vec** can also be formulated as a matrix factorization (albeit a more complex matrix)
- Refer to the paper for more detailed proofs.

Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec, WSDM 18

Limitations (1)

Limitations of node embeddings via matrix factorization and random walks

- Cannot obtain embeddings for nodes not in the training set



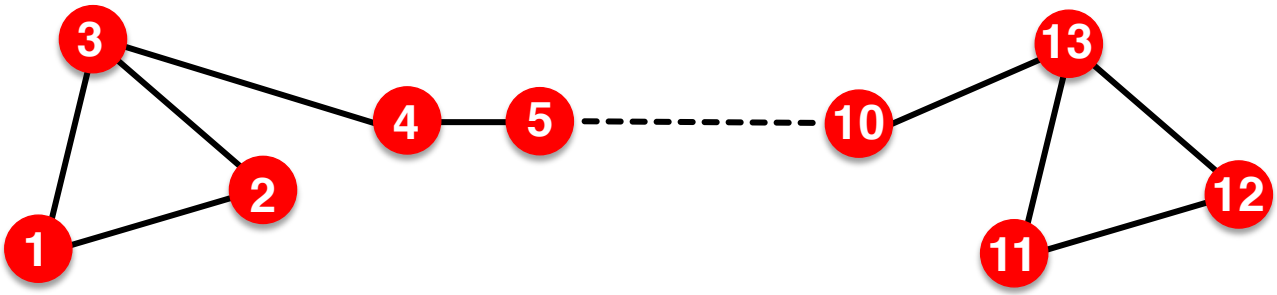
Training set

A newly added node 5 at test time (e.g. new user in a social network)

Cannot compute its embedding with DeepWalk / node2vec. Need to recompute all node embeddings.

Limitation (2)

- Cannot capture **structural similarity**:

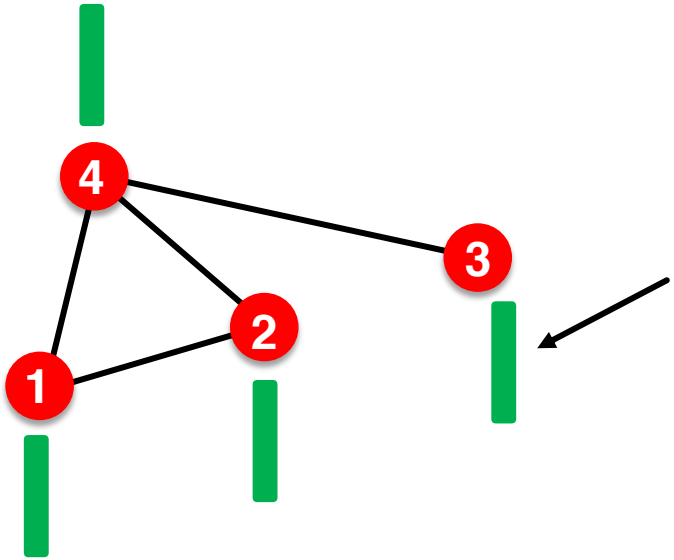


- Node 1 and 11 are **structurally similar** – part of one triangle, degree 2
- However, they have very **different** embeddings
  - It's unlikely that a random walk will reach node 11 from node 1

- **DeepWalk and node2vec do not capture structural similarity**

Limitations (3)

- Cannot utilize node, edge and graph features



Feature vector
(e.g. protein properties in a protein-protein interaction graph)

DeepWalk / node2vec embeddings do not incorporate such node features

Solution to these limitations: Deep Representation Learning and Graph Neural Networks
(To be covered in depth next week)

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!**

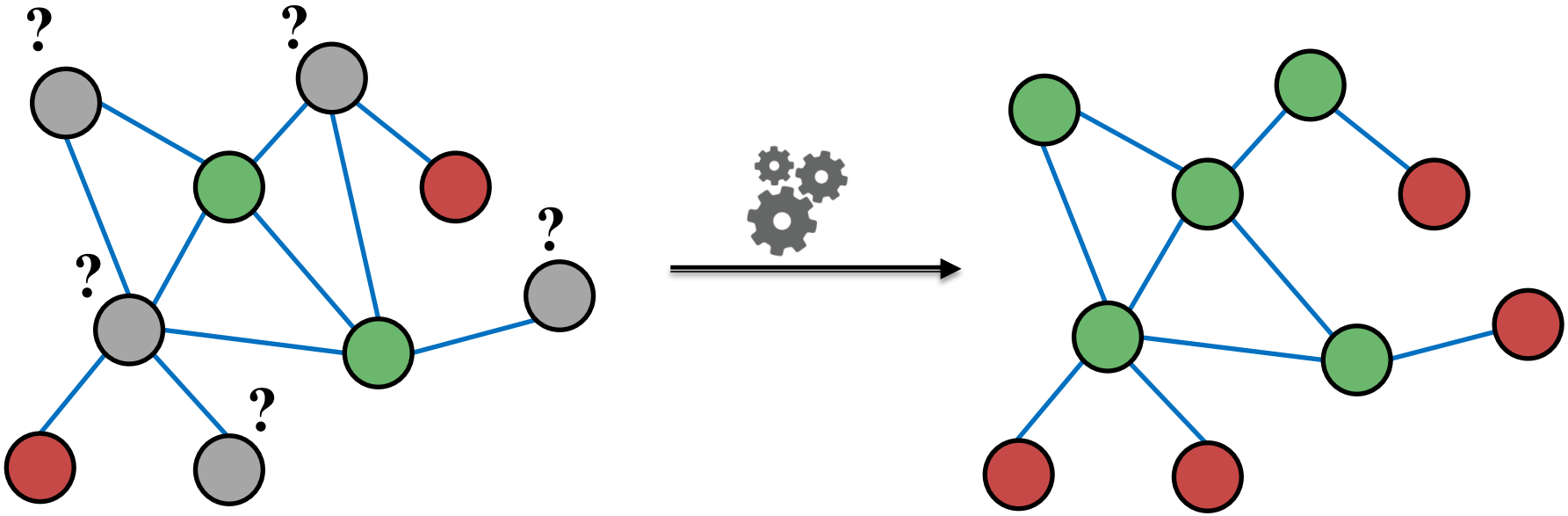
Stanford CS224W: Message Passing and Node Classification

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

Outline

- **Main question today:** Given a network with labels on some nodes, how do we assign labels to all other nodes in the network?
- **Example:** In a network, some nodes are fraudsters and some other nodes are fully trusted. **How do you find the other fraudsters and trustworthy nodes?**
- We already discussed node embeddings as a method to solve this in Lecture 3

Example: Node Classification



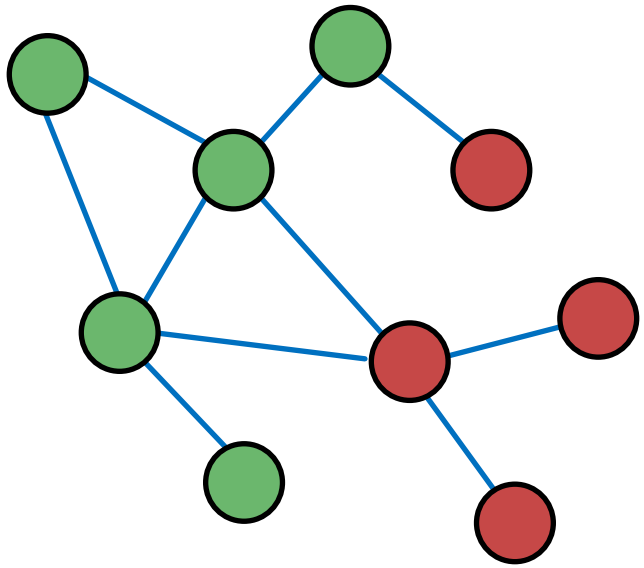
- Given labels of some nodes
- Let's predict labels of unlabeled nodes
- This is called semi-supervised node classification

Outline

- **Main question today:** Given a network with labels on some nodes, how do we assign labels to all other nodes in the network?
- **Today we will discuss an alternative framework: message passing**
- **Intuition: Correlations** exist in networks.
  - In other words: Similar nodes are connected
  - **Key concept** is **collective classification**: Idea of assigning labels to all nodes in a network together
- **We will look at three techniques today:**
  - **Relational classification**
  - **Iterative classification**
  - **Belief propagation**

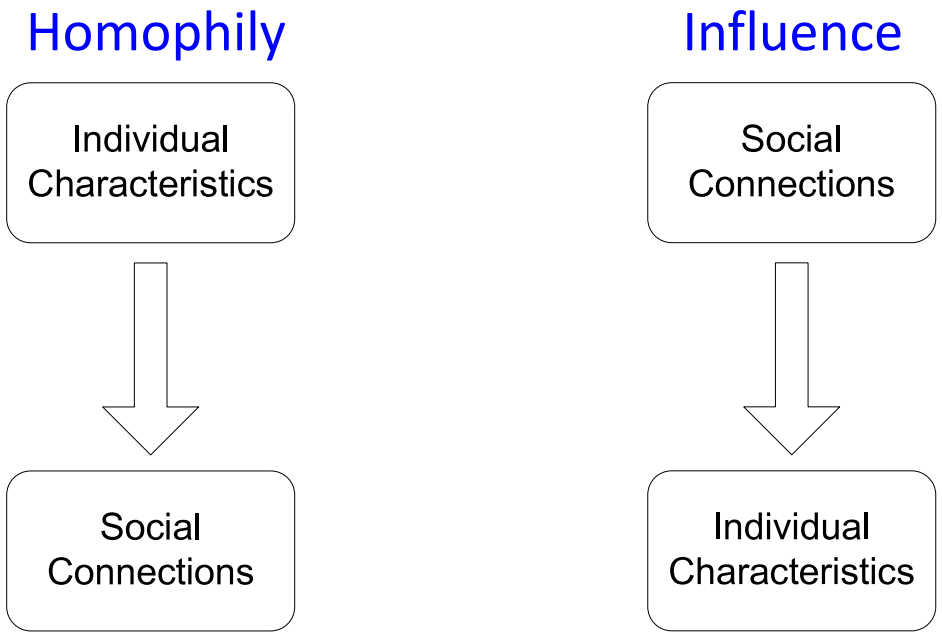
Correlations Exists in Networks

- Individual behaviors are **correlated** in the network
- **Correlation**: nearby nodes have the same color (belonging to the same class)



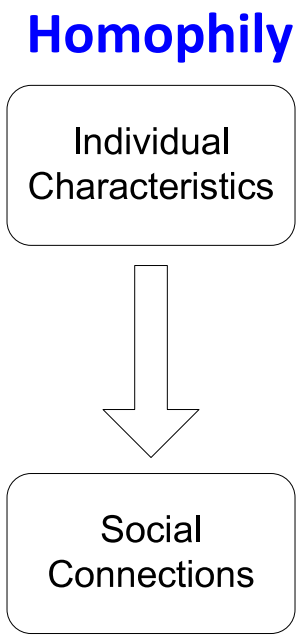
Correlations Exist in Networks

- **Main types of dependencies that lead to correlation:**



Homophily

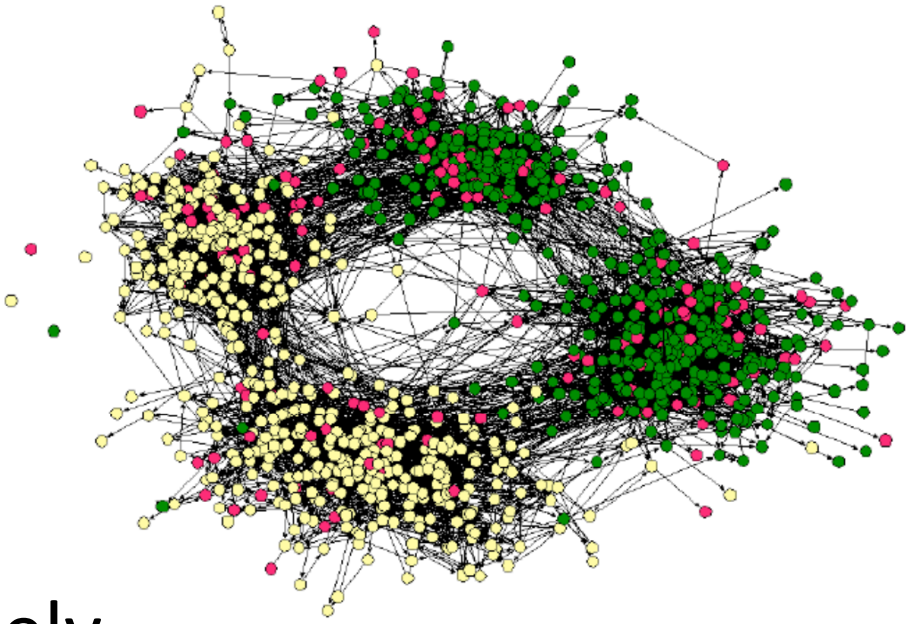
- **Homophily**: The tendency of individuals to associate and bond with similar others
  - *"Birds of a feather flock together"*
  - It has been observed in a vast array of network studies, based on a variety of attributes (e.g., age, gender, organizational role, etc.)
  - **Example**: Researchers who focus on the same research area are more likely to establish a connection (meeting at conferences, interacting in academic talks, etc.)



Homophily: Example

Example of homophily

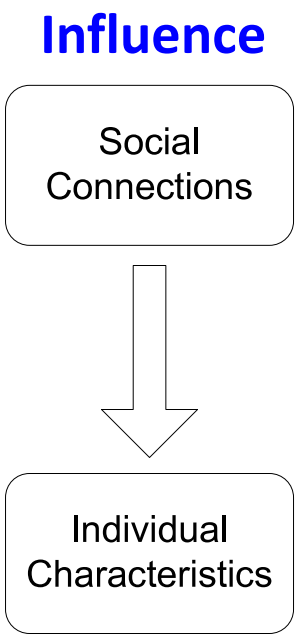
- Online social network
  - Nodes = people
  - Edges = friendship
  - Node color = interests (sports, arts, etc.)
- People with the same interest are more closely connected due to homophily



(Easley and Kleinberg, 2010)

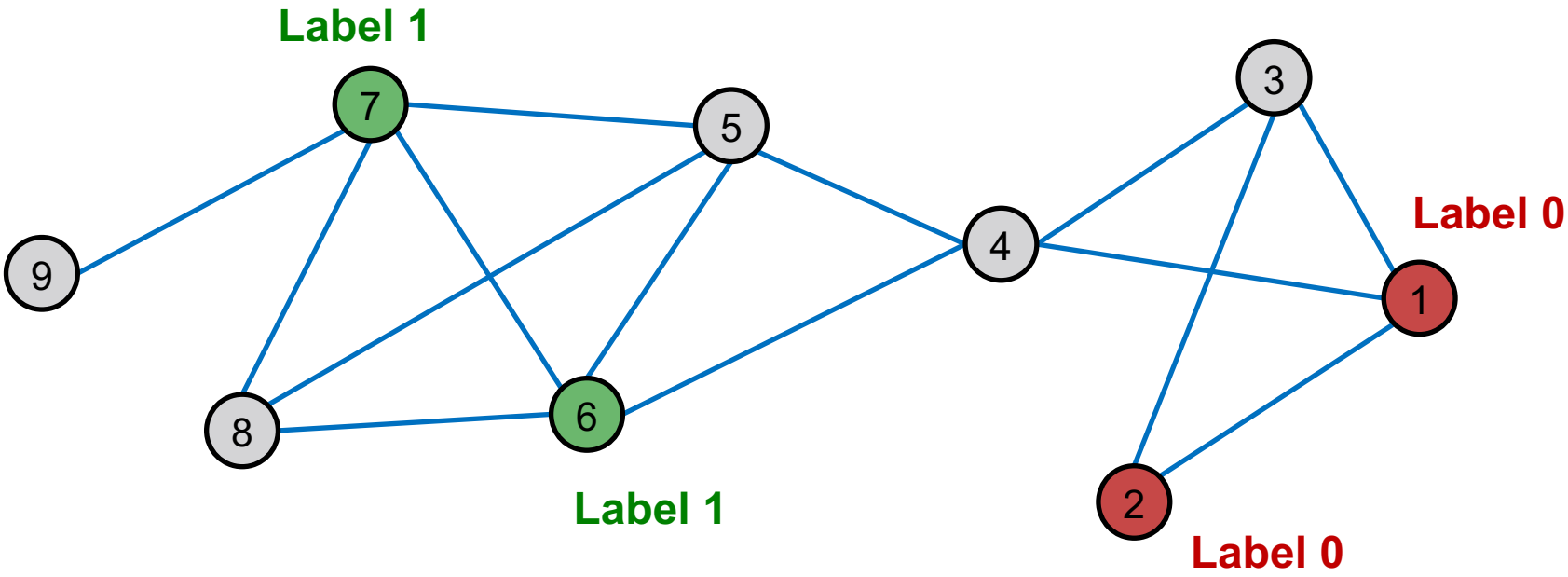
Influence

- **Influence**: Social connections can influence the individual characteristics of a person.
  - **Example**: I recommend my musical preferences to my friends, until one of them grows to like my same favorite genres!



Classification with Network Data

- How do we leverage this correlation observed in networks to help predict node labels?



How do we predict the labels for the nodes in grey?

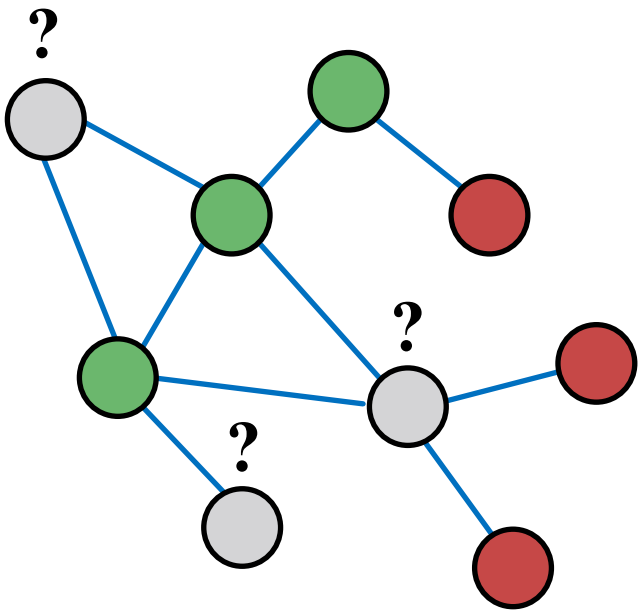
Motivation (1)

- **Similar nodes are typically close together or directly connected in the network:**
  - **Guilt-by-association**: If I am connected to a node with label X, then I am likely to have label X as well.
  - **Example: Malicious/benign web page:** Malicious web pages link to one another to increase visibility, look credible, and rank higher in search engines

Motivation (2)

- **Classification label** of a node v in network may depend on:
  - Features of v
  - Labels of the nodes in v's neighborhood
  - Features of the nodes in v's neighborhood

Semi-supervised Learning (1)



Given:

- Graph
- Few labeled nodes

Find: class (red/green) of remaining nodes

Main assumption: There is homophily in the network

Semi-supervised Learning (2)

Example task:

- Let $\boldsymbol{A}$ be a $n \times n$ adjacency matrix over n nodes
- Let $\mathrm{Y} = \{0, 1\}^n$ be a vector of **labels**:
  - $\mathrm{Y}_v = 1$ belongs to Class 1
  - $\mathrm{Y}_v = 0$ belongs to Class 0
  - There are unlabeled node needs to be classified
- **Goal:** Predict which **unlabeled** nodes are likely **Class 1**, and which are likely **Class 0**

Approach: Collective Classification

- **Many applications:**
  - Document classification
  - Part of speech tagging
  - Link prediction
  - Optical character recognition
  - Image/3D data segmentation
  - Entity resolution in sensor networks
  - Spam and fraud detection

Collective Classification Overview (1)

- **Intuition**: Simultaneous classification of interlinked nodes using correlations
- Probabilistic framework
- Markov Assumption: *the label Y_v of one node v depends on the labels of its neighbors N_v*

$$P(Y_v) = P(Y_v|N_v)$$

- Collective classification involves 3 steps:

Local Classifier	Relational Classifier	Collective Inference
• Assign initial labels	• Capture correlations between nodes	• Propagate correlations through network

Collective Classification Overview (2)

Local Classifier

- Assign initial labels

Relational Classifier

- Capture correlations between nodes

Collective Inference

- Propagate correlations through network

Local Classifier: Used for initial label assignment

- Predicts label based on node attributes/features
- Standard classification task
- Does not use network information

Relational Classifier: Capture correlations

- Learns a classifier to label one node based on the labels and/or attributes of its neighbors
- This is where network information is used

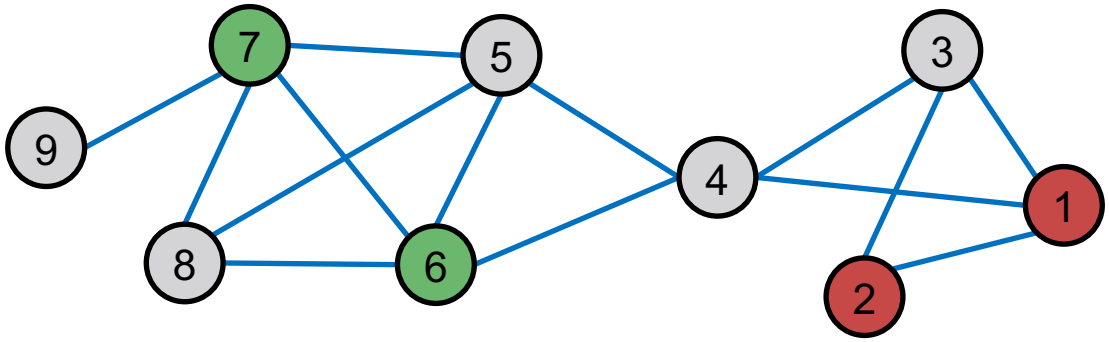
Collective Inference: Propagate the correlation

- Apply relational classifier to each node iteratively
- Iterate until the inconsistency between neighboring labels is minimized
- Network structure affects the final prediction

Problem Setting

- How to predict the labels Y_v for the unlabeled nodes v (in grey color)?
- Each node v has a feature vector f_v
- Labels for some nodes are given (1 for green, 0 for red)
- **Task:** Find $P(Y_v)$ given all features and the network

$P(Y_v) = ?$



Overview of What is Coming

- We focus on semi-supervised node classification
- Intuition is based on **homophily**: Similar nodes are typically close together or directly connected
- **Three techniques we will introduce:**
  - **Relational classification**
  - **Iterative classification**
  - **Belief propagation**

Stanford CS224W: Relational Classification and Iterative Classification

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

Collective Classification Models

- **Relational classifiers**
- Iterative classification
- Loopy belief propagation

Probabilistic Relational Classifier (1)

- **Basic idea:** Class probability Y_v of node v is a weighted average of class probabilities of its neighbors
- For **labeled nodes** v, initialize label Y_v with ground-truth label Y_v^*
- For **unlabeled nodes**, initialize $Y_v = 0.5$
- **Update** all nodes in a random order until convergence or until maximum number of iterations is reached

Probabilistic Relational Classifier (2)

- **Update** for each node v and label c (e.g. 0 or 1)

$$P(Y_v = c) = \frac{1}{\sum_{(v,u)\in E} A_{v,u}} \sum_{(v,u)\in E} A_{v,u}\, P(Y_u = c)$$

  - If edges have strength/weight information, $A_{v,u}$ can be the edge weight between v and u
  - $P(Y_v = c)$ is the probability of node v having label c

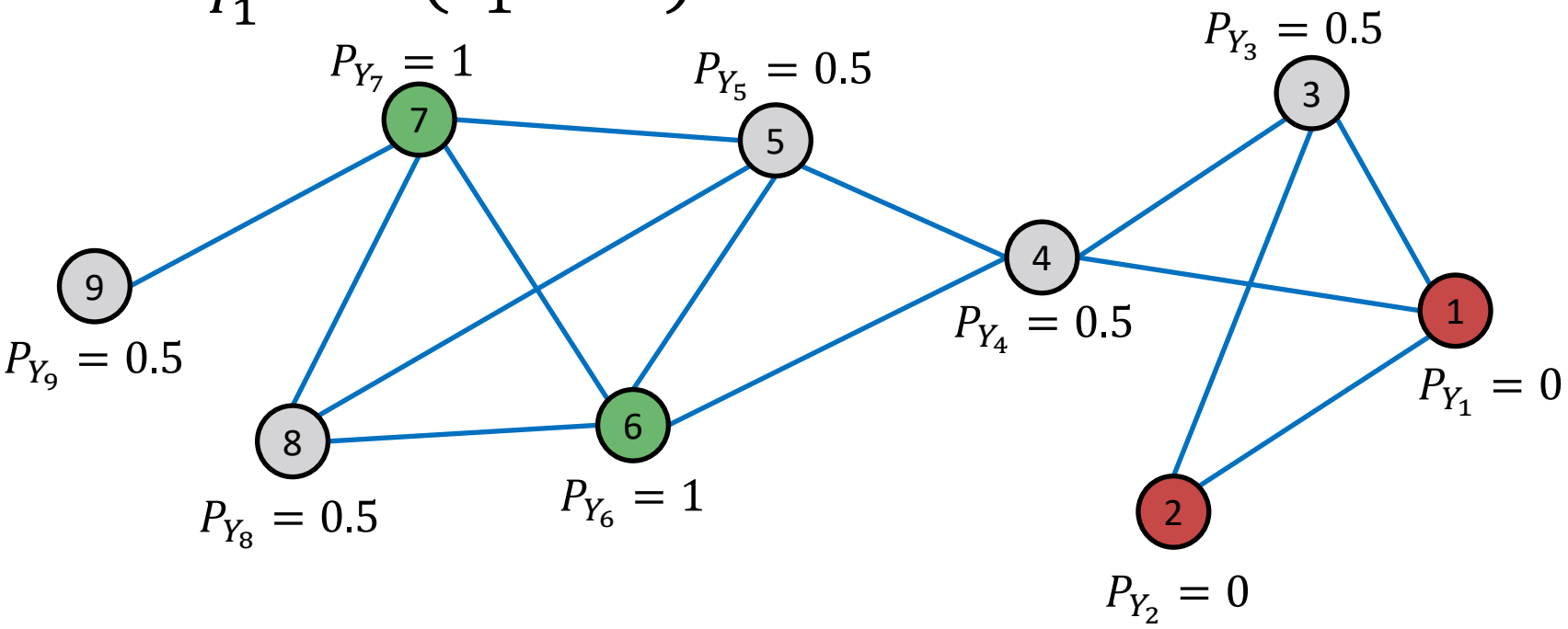
- Challenges:
  - Convergence is not guaranteed
  - Model cannot use node feature information

Example: Initialization

Initialization:

- All labeled nodes with their labels
- All unlabeled nodes 0.5 (belonging to class 1 with probability 0.5)

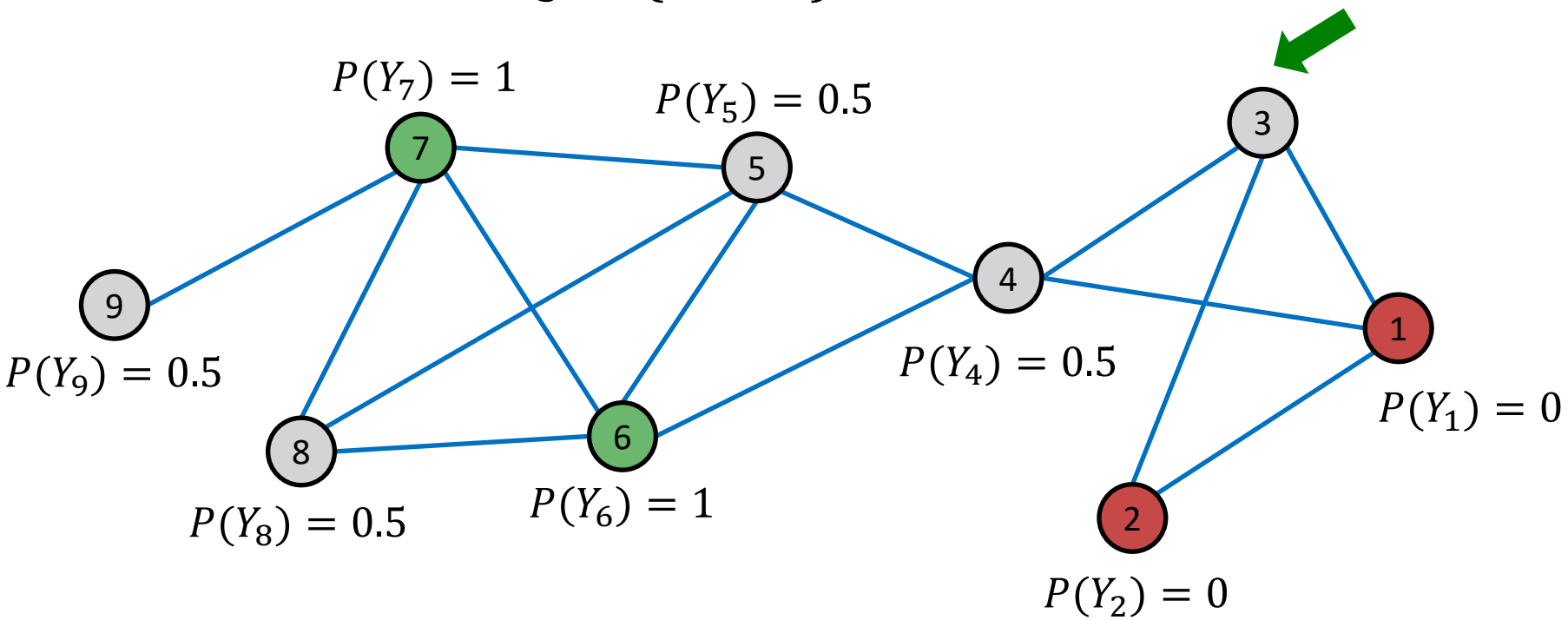
Let $P_{Y_1} = P(Y_1 = 1)$



Example: 1st Iteration, Update Node 3

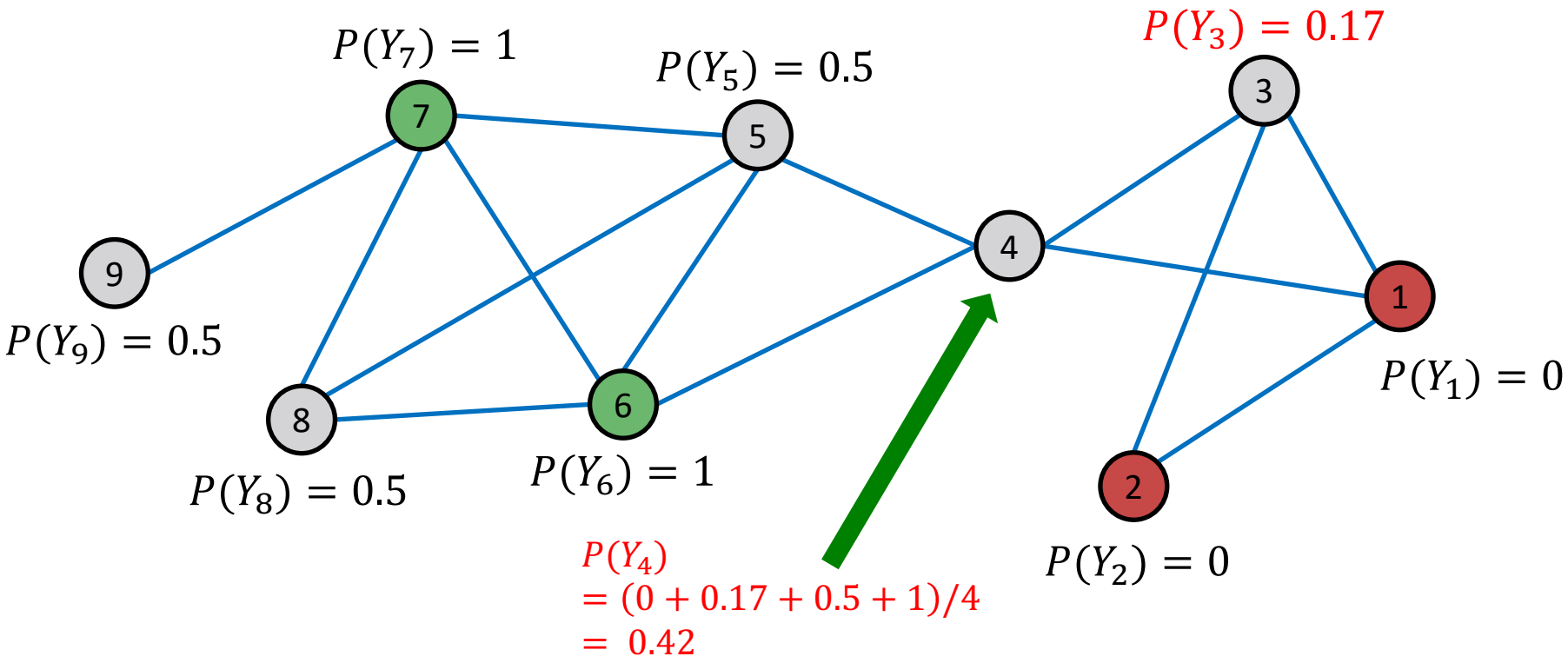
- Update for the 1st Iteration:
  - For node 3, $N_3 = \{1, 2, 4\}$

$P(Y_3) = (0 + 0 + 0.5)/3 = 0.17$



Example: 1st Iteration, Update Node 4

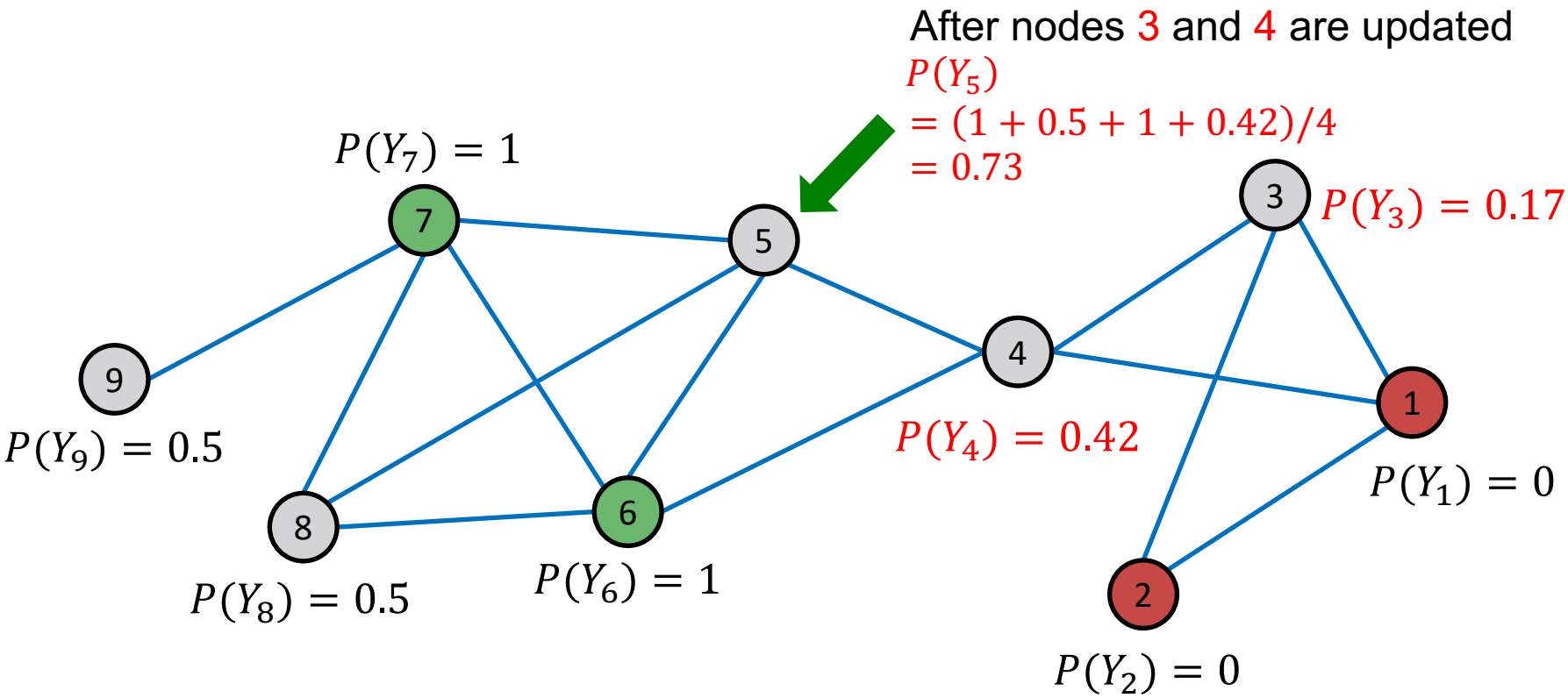
- Update for the 1st Iteration:
  - For node 4, $N_4 = \{1, 3, 5, 6\}$



After Node 3 is updated

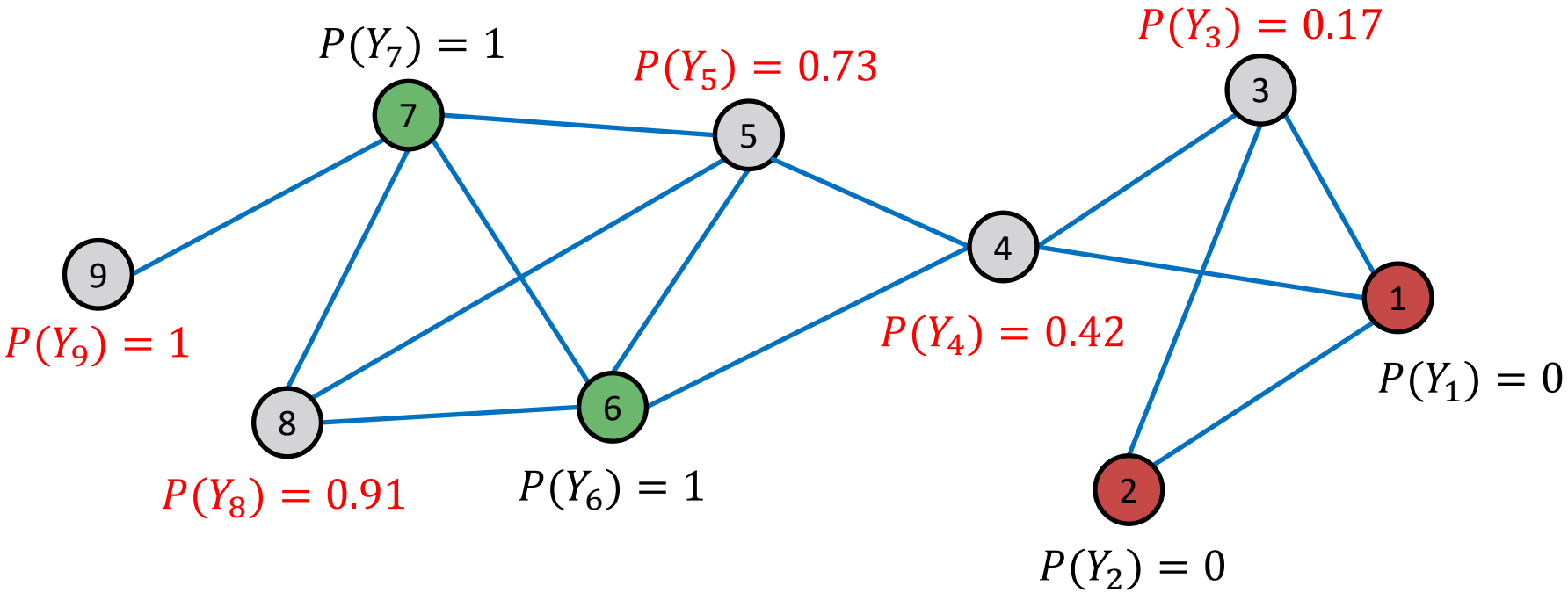
Example: 1st Iteration, Update Node 5

- Update for the 1st Iteration:
  - For node 5, $N_5 = \{4, 6, 7, 8\}$



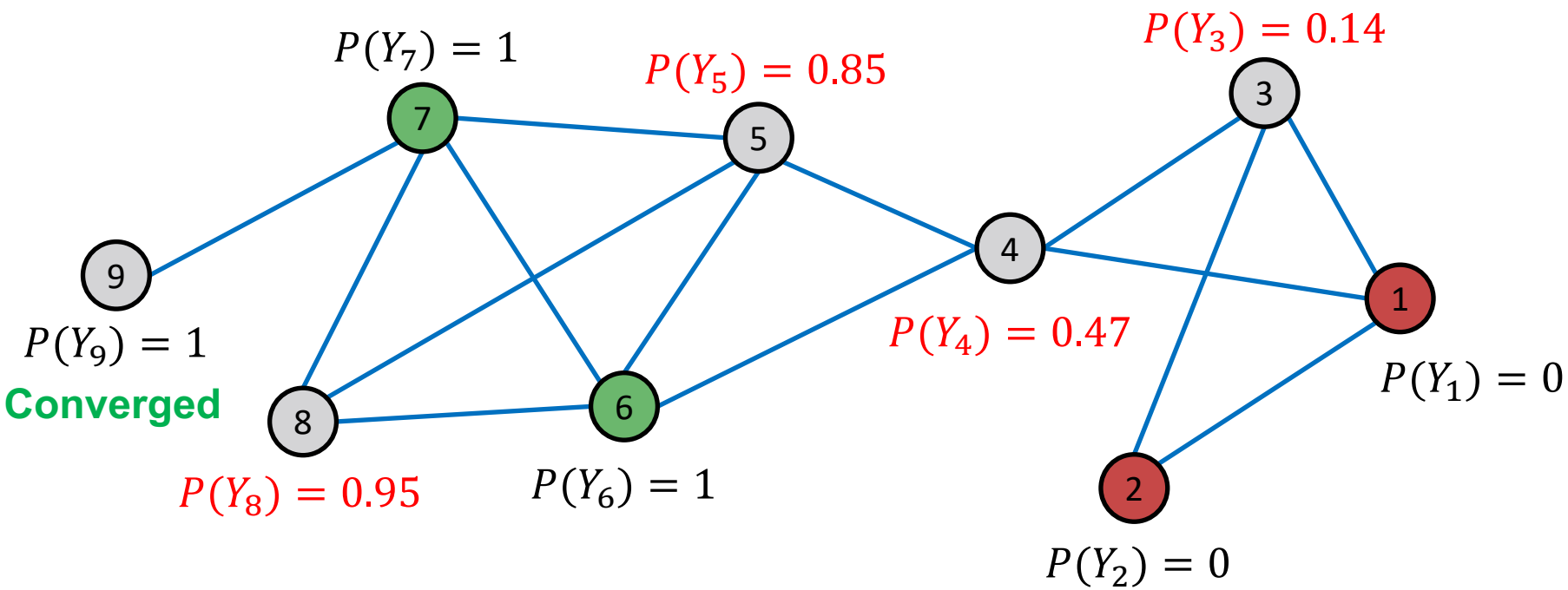
Example: After 1st Iteration

After Iteration 1 (a round of updates for all unlabeled nodes)



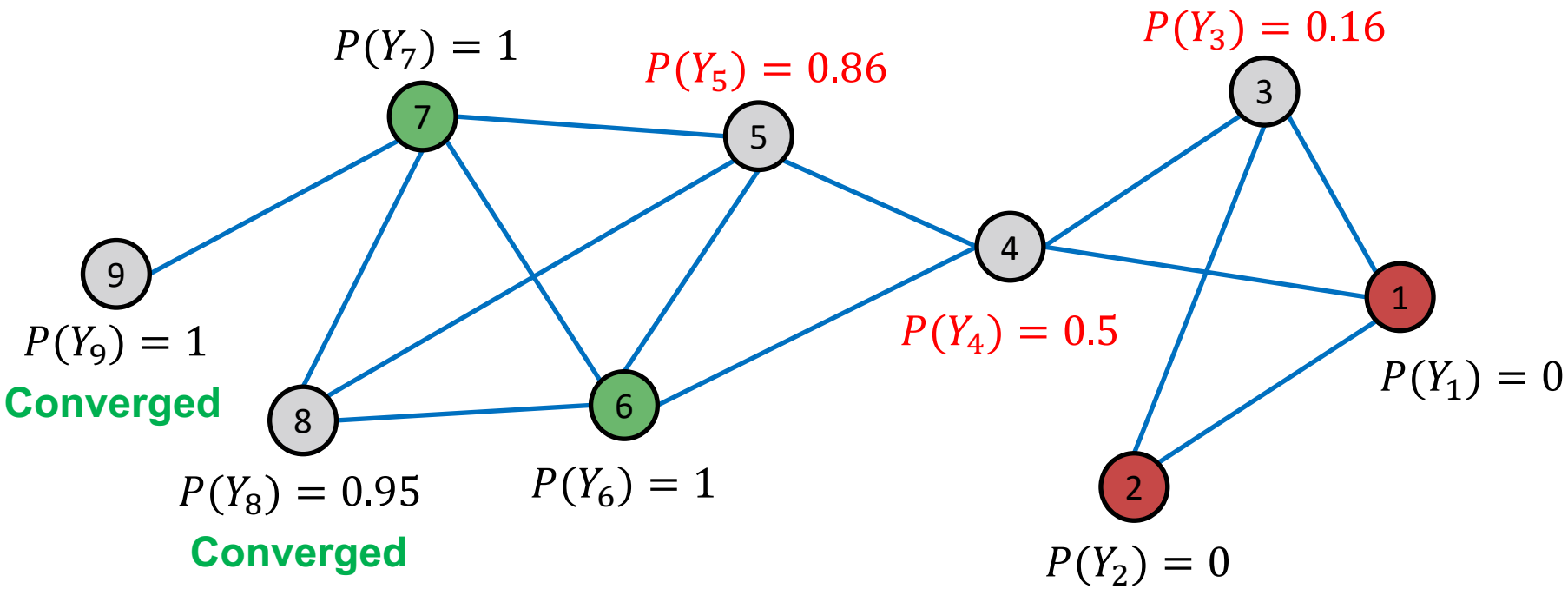
Example: After 2nd Iteration

After Iteration 2



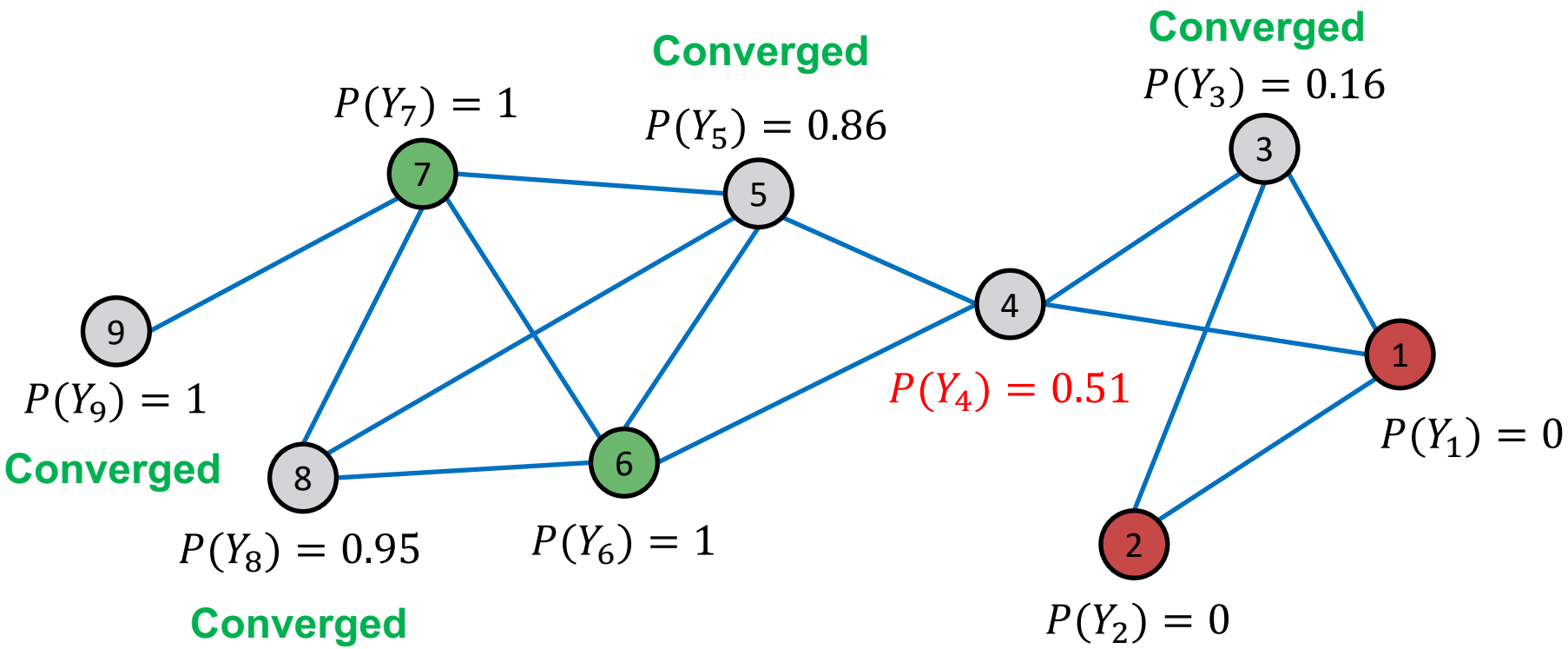
Example: After 3rd Iteration

After Iteration 3



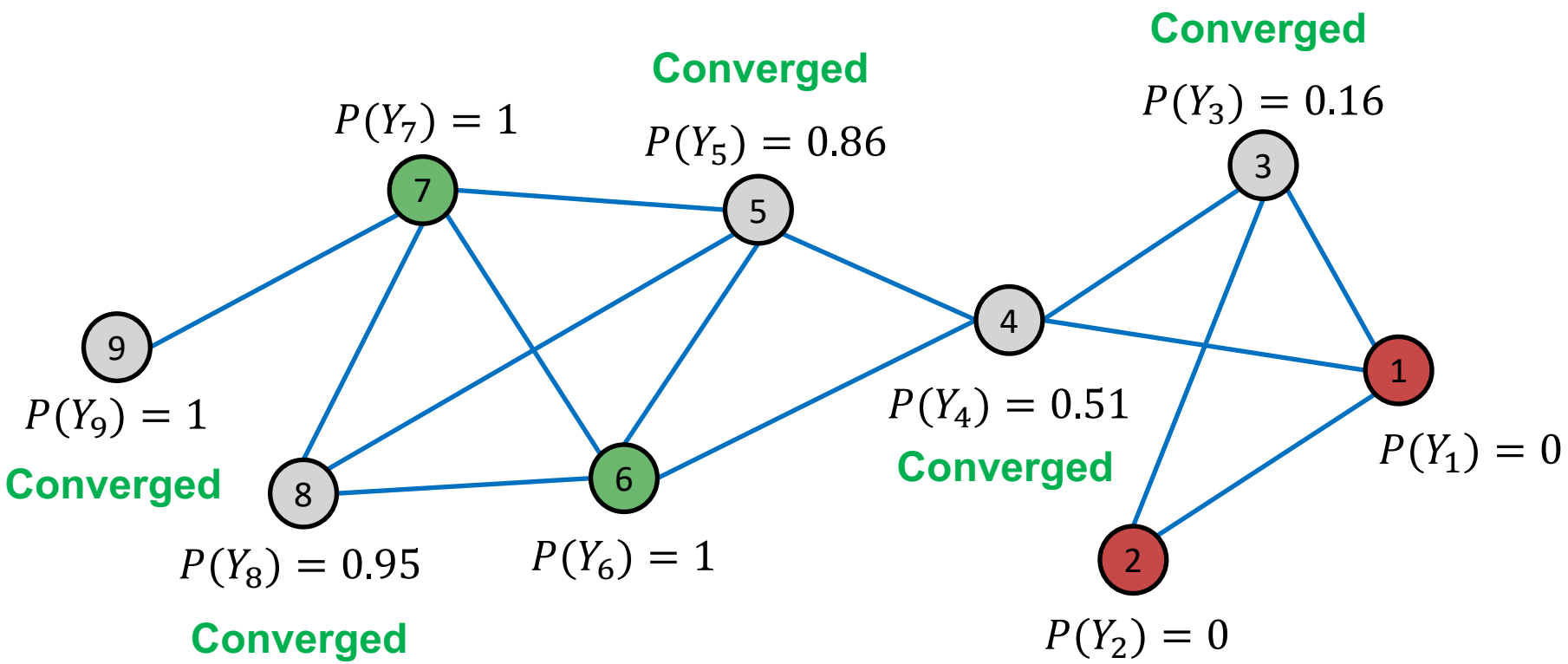
Example: After 4th Iteration

After Iteration 4



Example: Convergence

- All scores stabilize after 4 iterations. We therefore predict:
  - **Nodes 4, 5, 8, 9 belong to class 1** ($P_{Y_v} > 0.5$)
  - **Nodes 3 belong to class 0** ($P_{Y_v} < 0.5$)



Collective Classification Models

- Relational classifiers
- **Iterative classification**
- Loopy belief propagation

Iterative Classification

- **Relational classifiers do not use node attributes. How can one leverage them?**

- **Main idea of iterative classification:** Classify node v based on its **attributes f_v** as well as **labels z_v** of neighbor set N_v

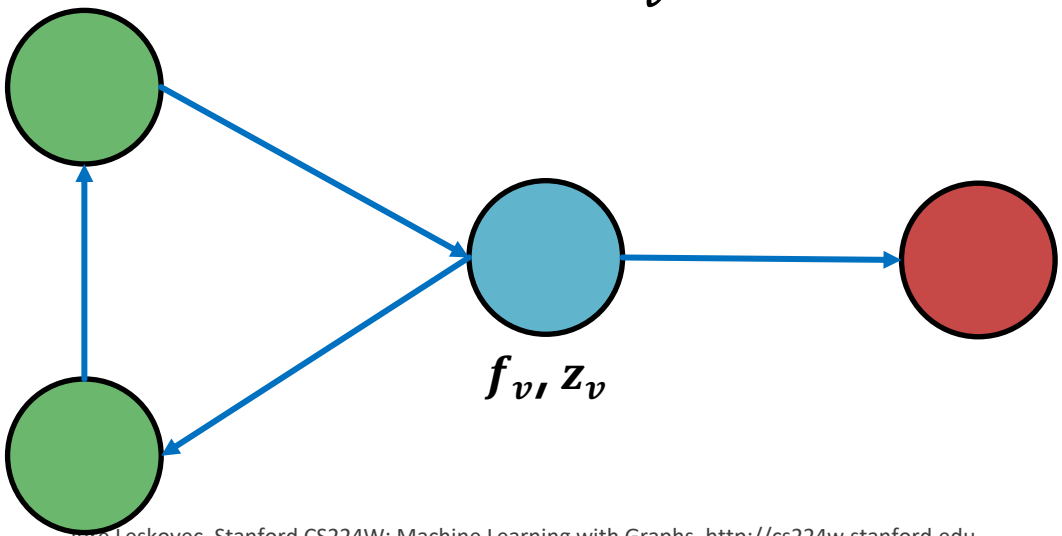
Iterative Classification

- **Input: Graph**
  - f_v : feature vector for node v
  - Some nodes v are labeled with Y_v
- **Task:** Predict label of unlabeled nodes
- **Approach: Train two classifiers:**
- $\phi_1(f_v)$ = Predict node label based on node feature vector f_v
- $\phi_2(f_v, z_v)$ = Predict label based on node feature vector f_v and summary z_v of labels of v's neighbors.

Computing the Summary z_v

How do we compute the summary z_v of labels of v's neighbors N_v?

- Ideas: z_v = **vector**
  - Histogram of the number (or fraction) of each label in N_v
  - Most common label in N_v
  - Number of different labels in N_v



Architecture of Iterative Classifiers

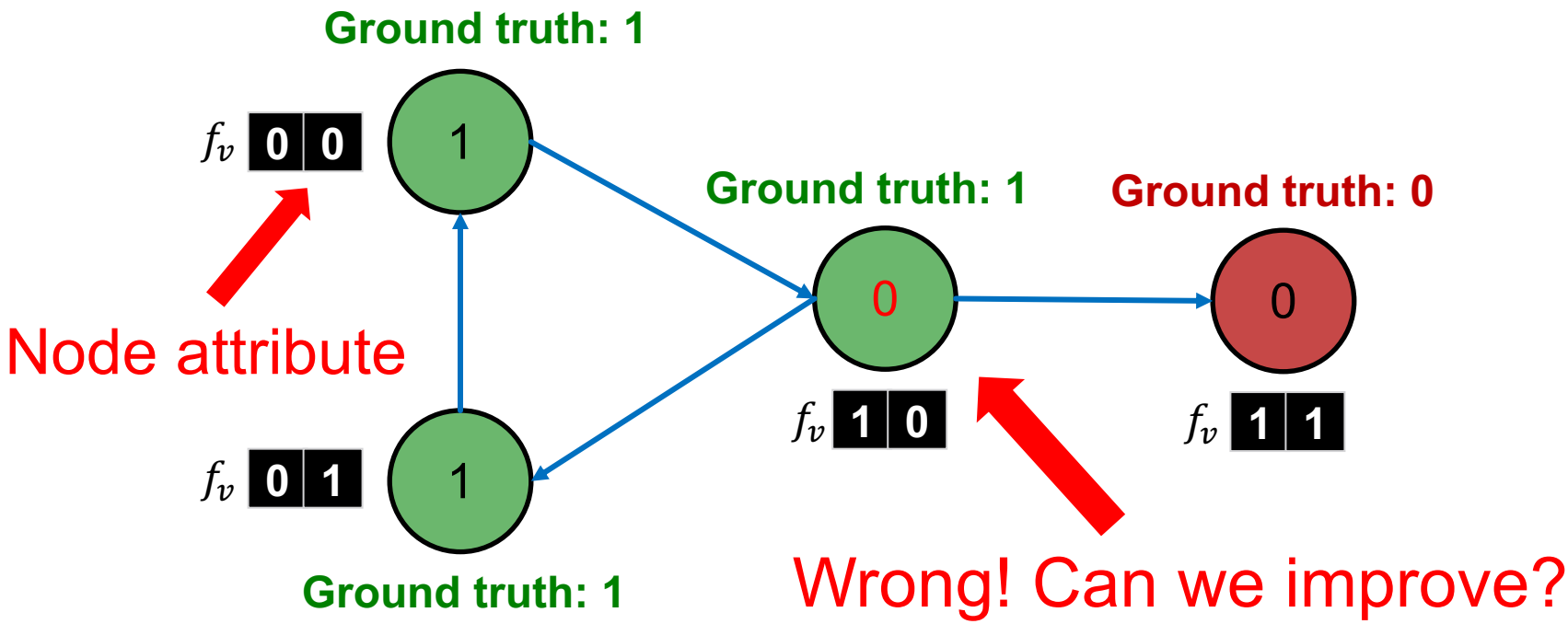
- **Phase 1: Classify based on node attributes alone**
  - On a **training set**, train classifier (e.g., linear classifier, neural networks, …):
  - $\phi_1(f_v)$ to predict Y_v based on f_v
  - $\phi_2(f_v, z_v)$ to predict Y_v based on f_v and summary z_v of labels of v's neighbors
- **Phase 2: Iterate till convergence**
  - On **test set**, set labels Y_v based on the classifier ϕ_1, compute z_v and predict the labels with ϕ_2
  - **Repeat** for each node v
    - Update z_v based on Y_u for all $u \in N_v$
    - Update Y_v based on the new z_v (ϕ_2)
  - Iterate until class labels stabilize or max number of iterations is reached
  - Note: Convergence is not guaranteed

Example: Web Page Classification (1)

- **Input:** Graph of web pages
- **Node:** Web page
- **Edge:** Hyper-link between web pages
  - **Directed edge:** a page points to another page
- **Node features:** Webpage description
  - For simplicity, we only consider 2 binary features
- **Task:** Predict the topic of the webpage

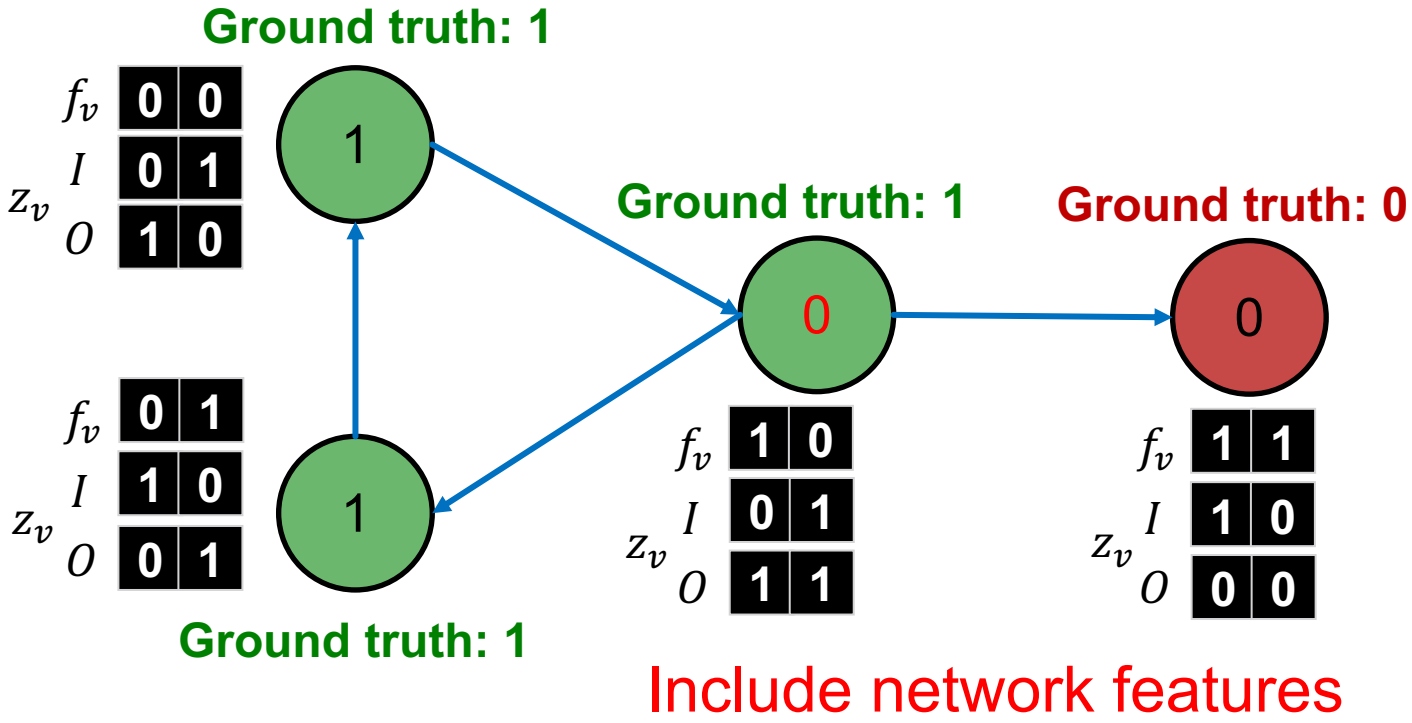
Example: Web Page Classification (2)

- Baseline: train a classifier (e.g., linear classifier) to classify pages based on binary node attributes.



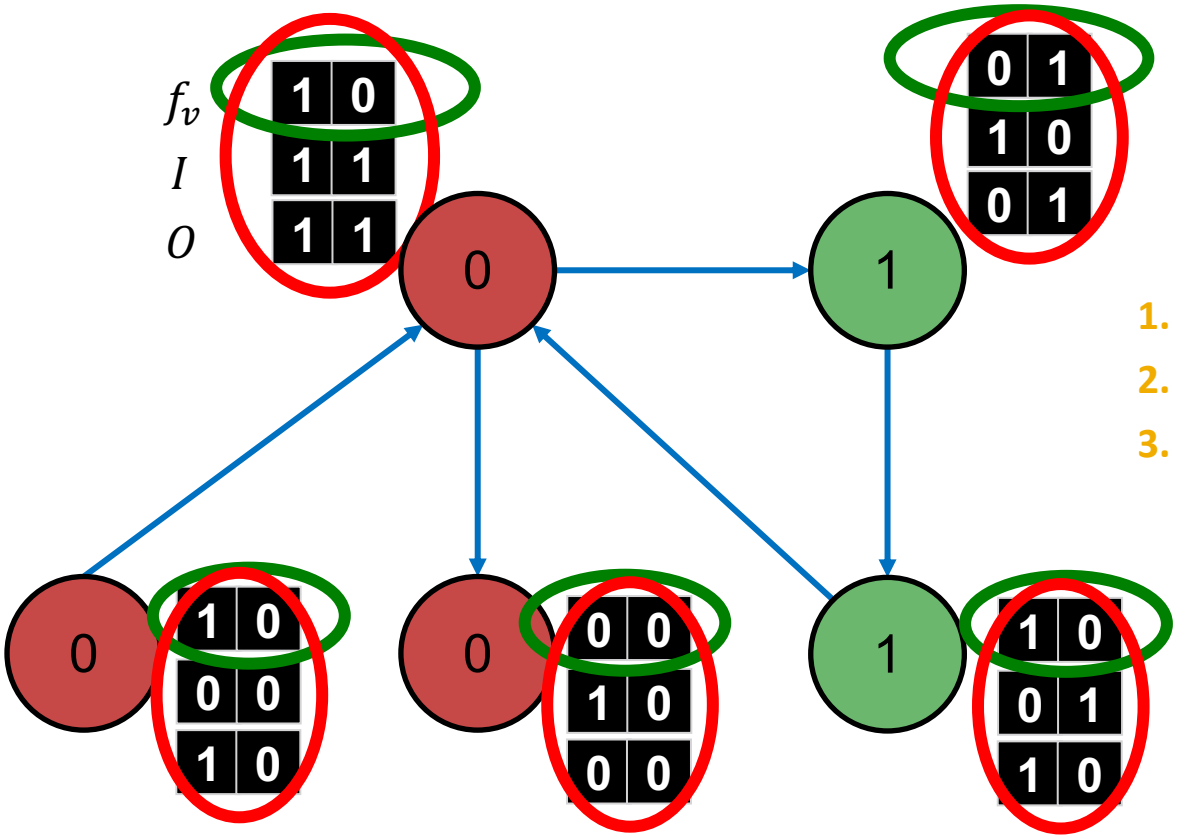
Example: Web Page Classification (3)

- Each node maintains vectors $\boldsymbol{z_v}$ of neighborhood labels:
  I = **Incoming** neighbor label information vector
  O= **Outgoing** neighbor label information vector
- $I_0 = 1$ if at least one of the incoming pages is labelled 0.
  Similar definitions for I_1, O_0, and O_1



Iterative Classifier – Step 1

- On a different **training set**, train two classifiers:
  - Node attribute vector only (green circles): ϕ_1
  - Node attribute and link vectors (red circles): ϕ_2

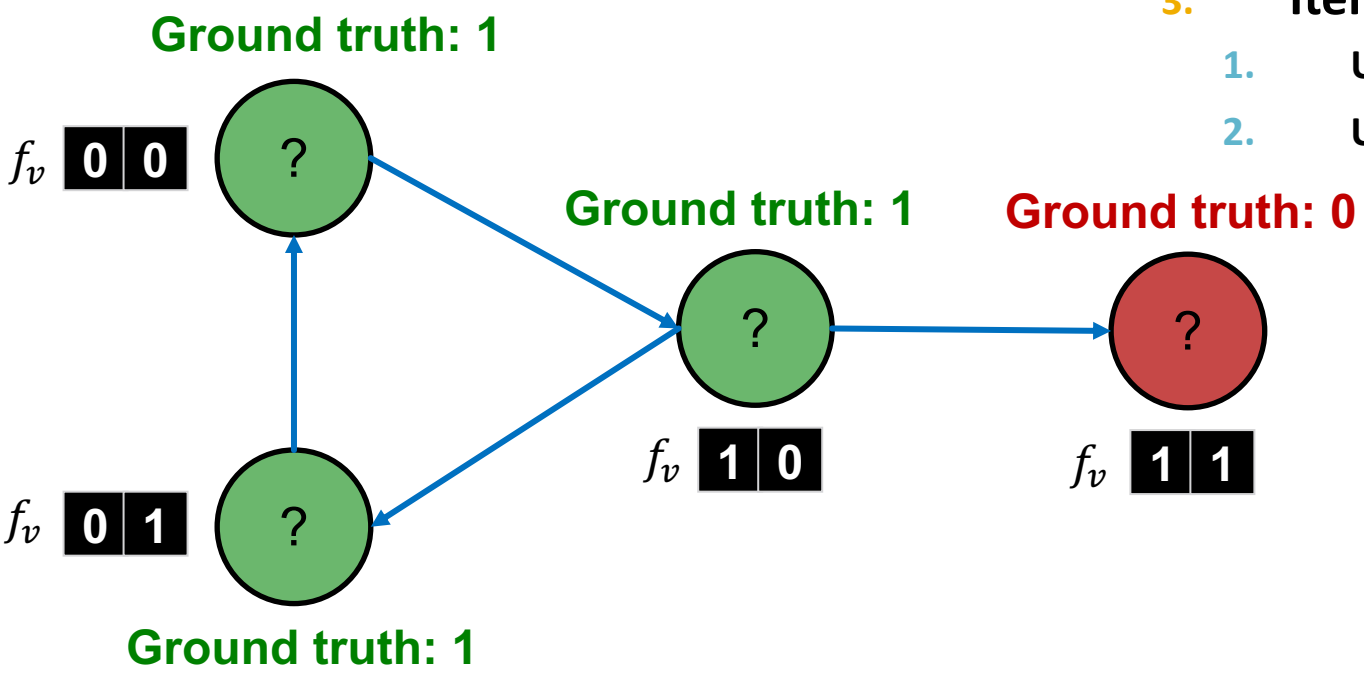


1. **Train classifier**
2. **Apply classifier to test set**
3. **Iterate**
   1. **Update relational features z_v**
   2. **Update label Y_v**

Iterative Classifier – Step 2

- On the **test set**:
  - Use trained node feature vector classifier ϕ_1 to set Y_v

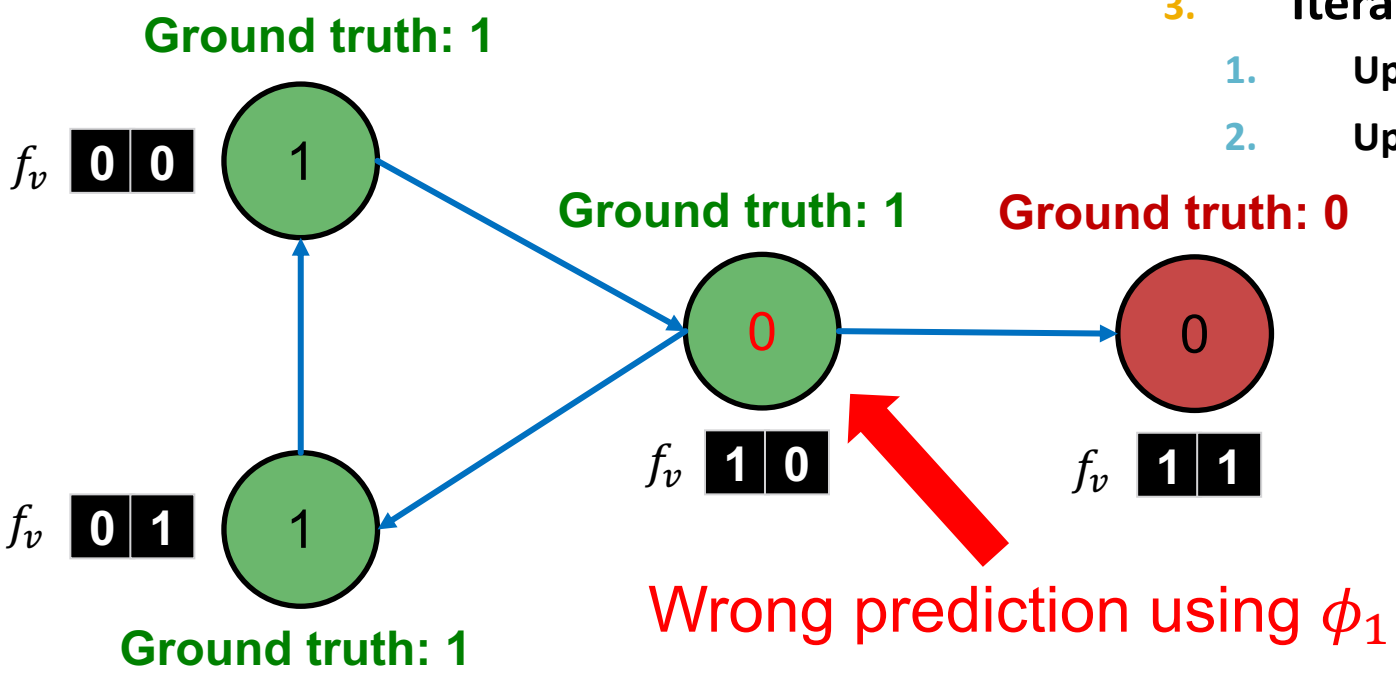
1. Train classifier
2. **Apply classifier to test set**
3. Iterate
   1. Update relational features z_v
   2. Update label Y_v



Iterative Classifier – Step 2

- On the **test set**:
  - Use trained node feature vector classifier ϕ_1 to set Y_v

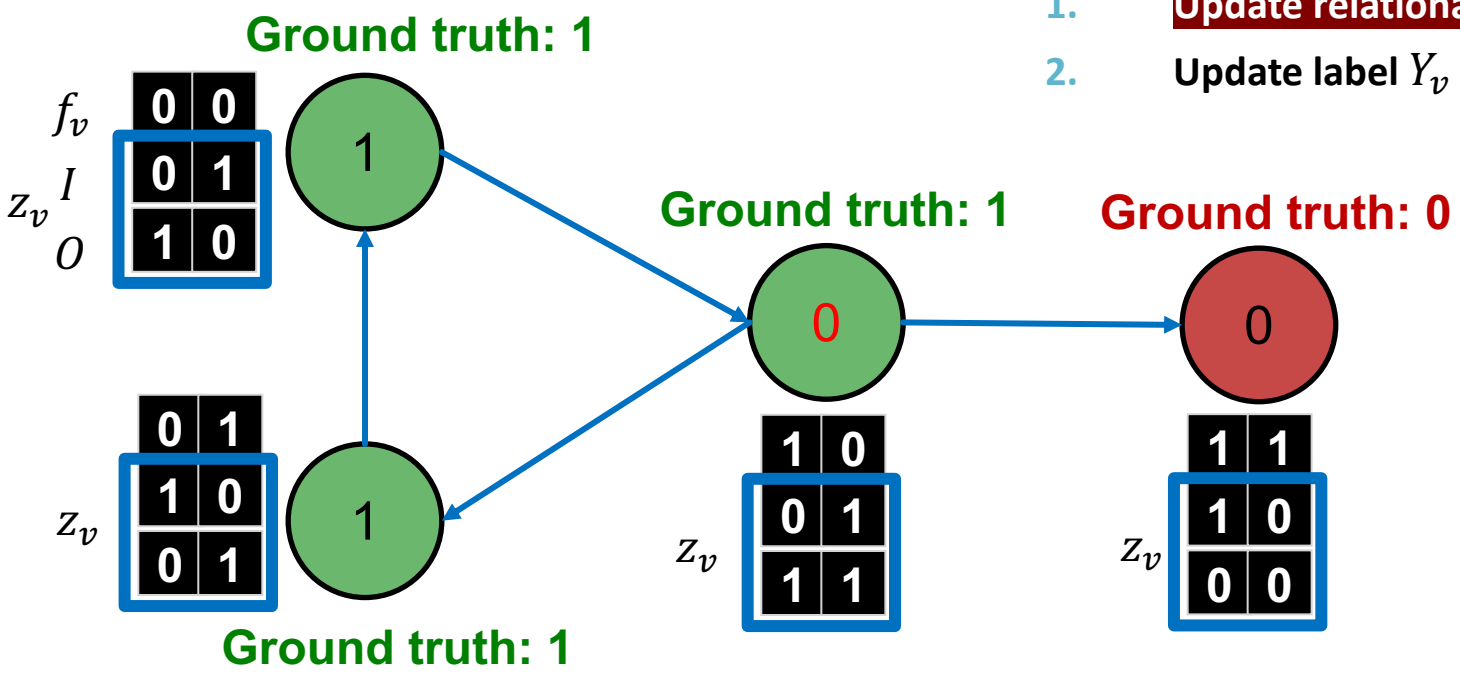
1. Train classifier
2. **Apply classifier to test set**
3. Iterate
   1. Update relational features z_v
   2. Update label Y_v



Iterative Classifier – Step 3.1

- Update z_v for all nodes

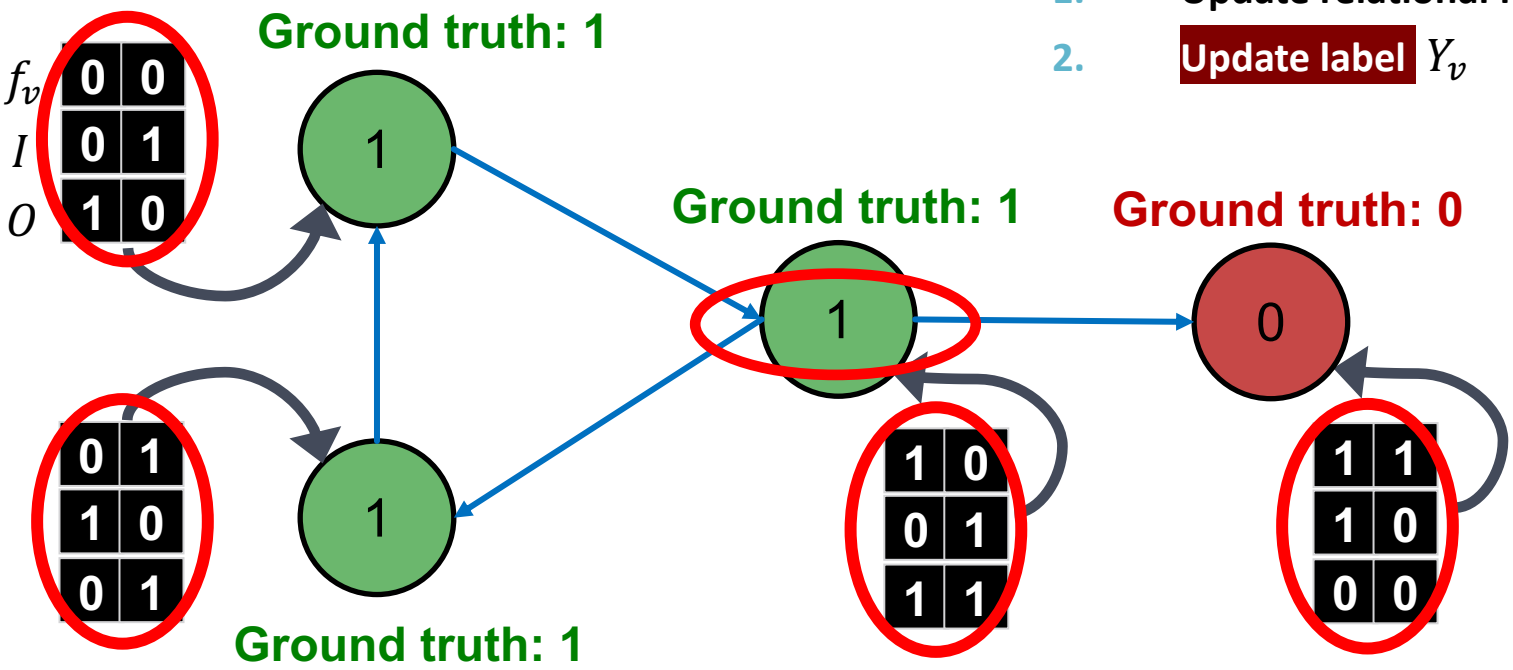
1. Train classifier
2. Apply classifier to test
3. Iterate
   1. Update relational features z_v
   2. Update label Y_v



Iterative Classifier – Step 3.2

- **Re-classify all nodes with ϕ_2**

1. Train classifier
2. Apply classifier to test
3. **Iterate**
   1. **Update relational features z_v**
   2. **Update label Y_v**

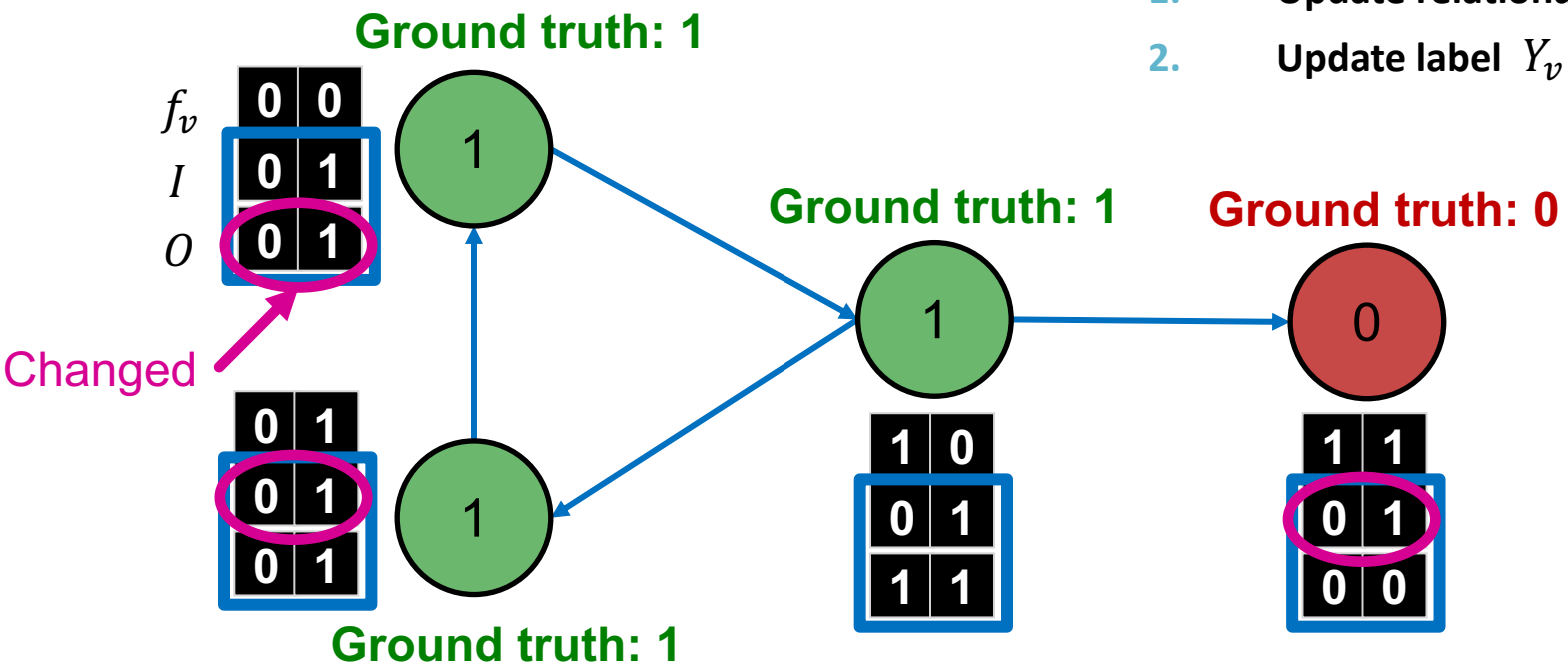


Now it's correct prediction!

Iterative Classifier – Iterate

- Continue until convergence
  - Update z_v
  - Update $Y_v = \phi_2(f_v, z_v)$

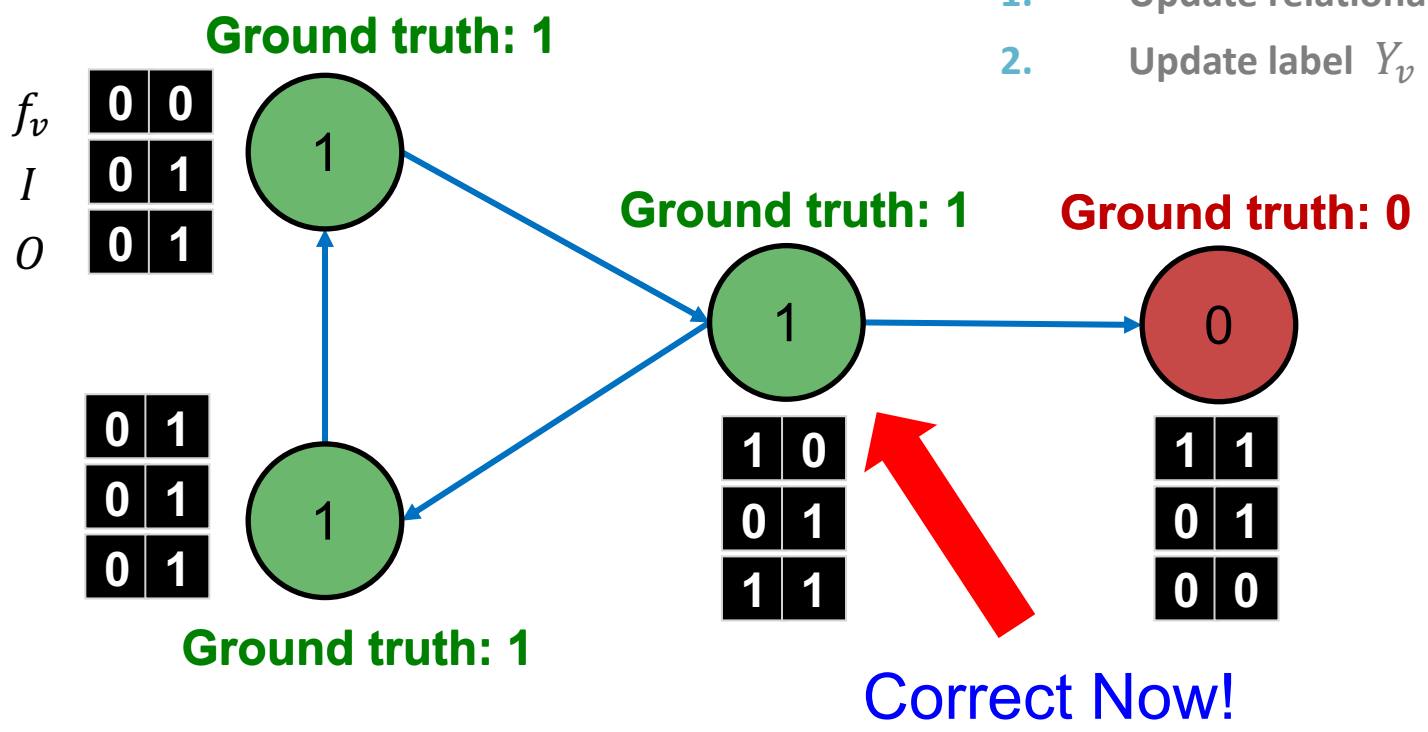
1. Train classifier
2. Apply classifier to test
3. **Iterate**
   1. Update relational features z_v
   2. Update label Y_v



Iterative Classifier – Final Prediction

- Stop iteration
  - After convergence or when maximum iterations are reached

1. Train classifier
2. Apply classifier to test set
3. Iterate
   1. Update relational features z_v
   2. Update label Y_v



Summary

- **We talked about 2 approaches to collective classification**
- **Relational classification**
  - Iteratively update probabilities of node belonging to a label class based on its neighbors
- **Iterative classification**
  - Improve over collective classification to handle attribute/feature information
  - Classify node i based on its **features** as well as **labels** of neighbors

Stanford CS224W: Collective Classification: Belief Propagation

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

Collective Classification Models

- Relational classifiers
- Iterative classification
- **Loopy belief propagation**

Loopy Belief Propagation

- Belief Propagation is a dynamic programming approach to **answering probability queries in a graph** (e.g. probability of node v belonging to class 1)
- Iterative process in which neighbor nodes "talk" to each other, **passing messages**

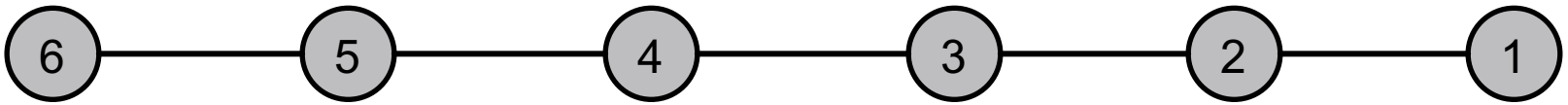
"I (node v) believe you (node u) belong to class 1 with likelihood …"

- **When consensus is reached, calculate final belief**

Message Passing: Basics

- **Task**: Count the number of nodes in a graph*
- **Condition:** Each node can only interact (pass message) with its neighbors

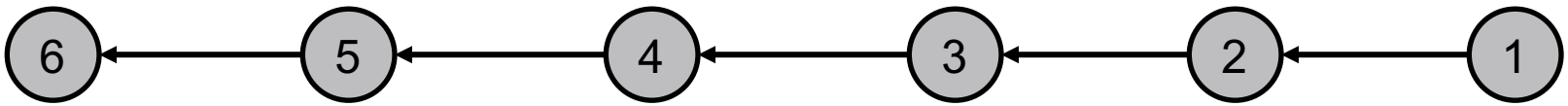
- **Example**: path graph



* Potential issues when the graph contains cycles.
We'll get back to it later!

Message Passing: Algorithm

- **Task**: Count the number of nodes in a graph
- **Algorithm**:
  - Define an ordering of nodes (that results in a path)
  - Edge directions are according to order of nodes
    - Edge direction defines the order of message passing
  - For node i from 1 to 6
    - Compute the message from node i to $i + 1$ (number of nodes counted so far)
    - Pass the message from node i to $i + 1$



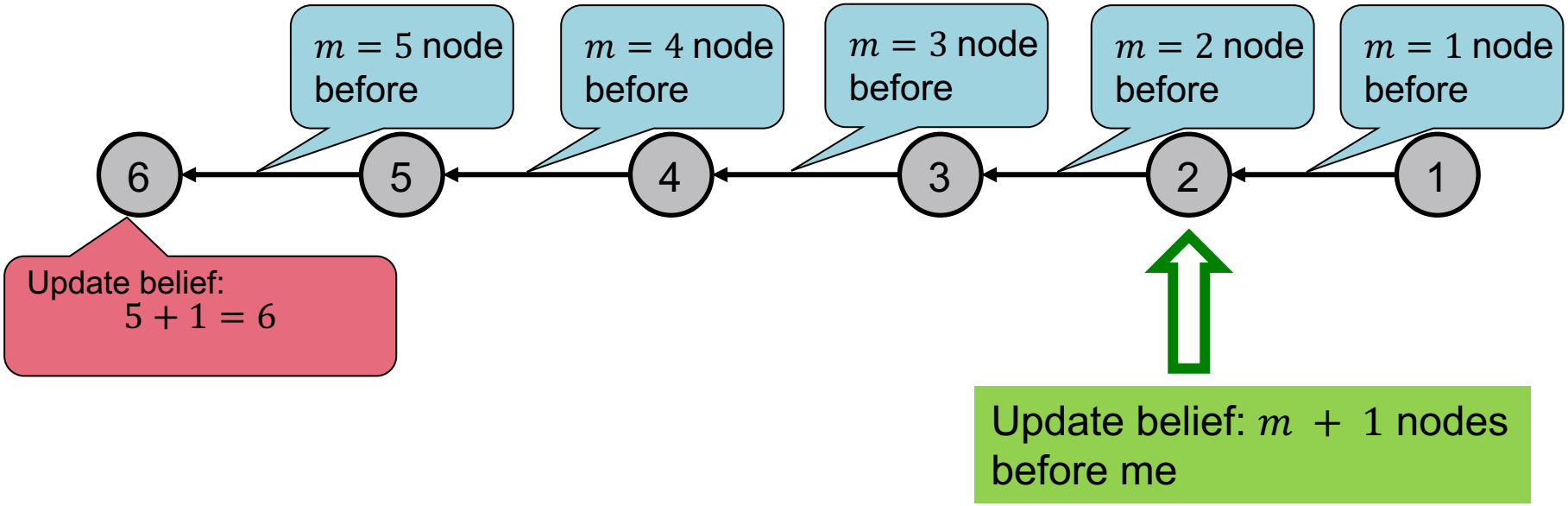
Message Passing: Basics

Task: Count the number of nodes in a graph

Condition: Each node can only interact (pass message) with its neighbors

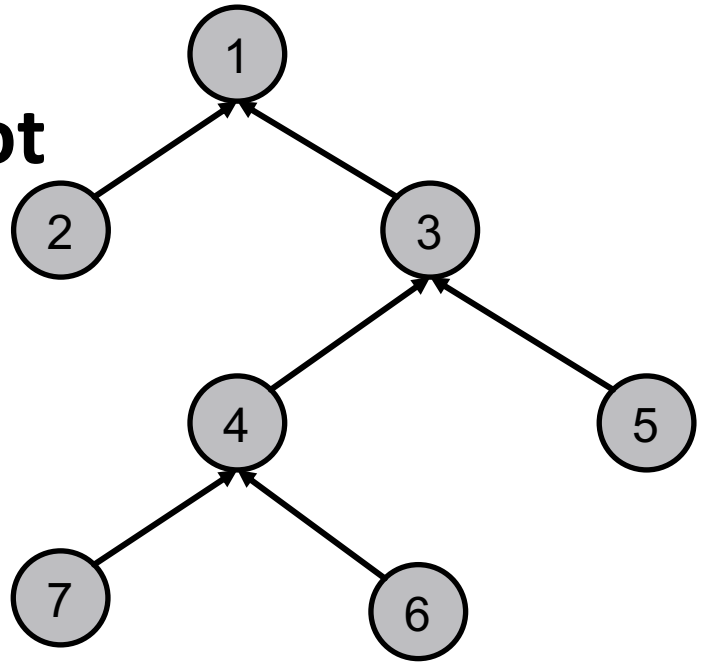
Solution: Each node listens to the message from its neighbor, updates it, and passes it forward

m: the message



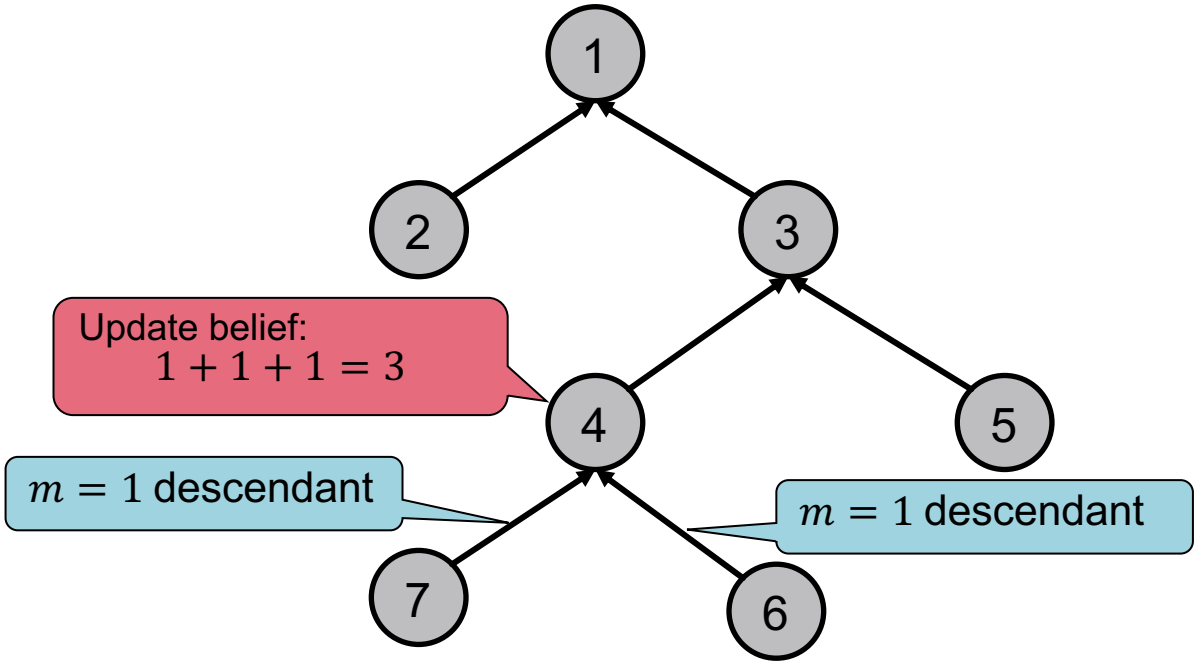
Generalizing to a Tree

- We can perform message passing not only on a path graph, but also on a tree-structured graph
- Define order of message passing from **leaves** to **root**



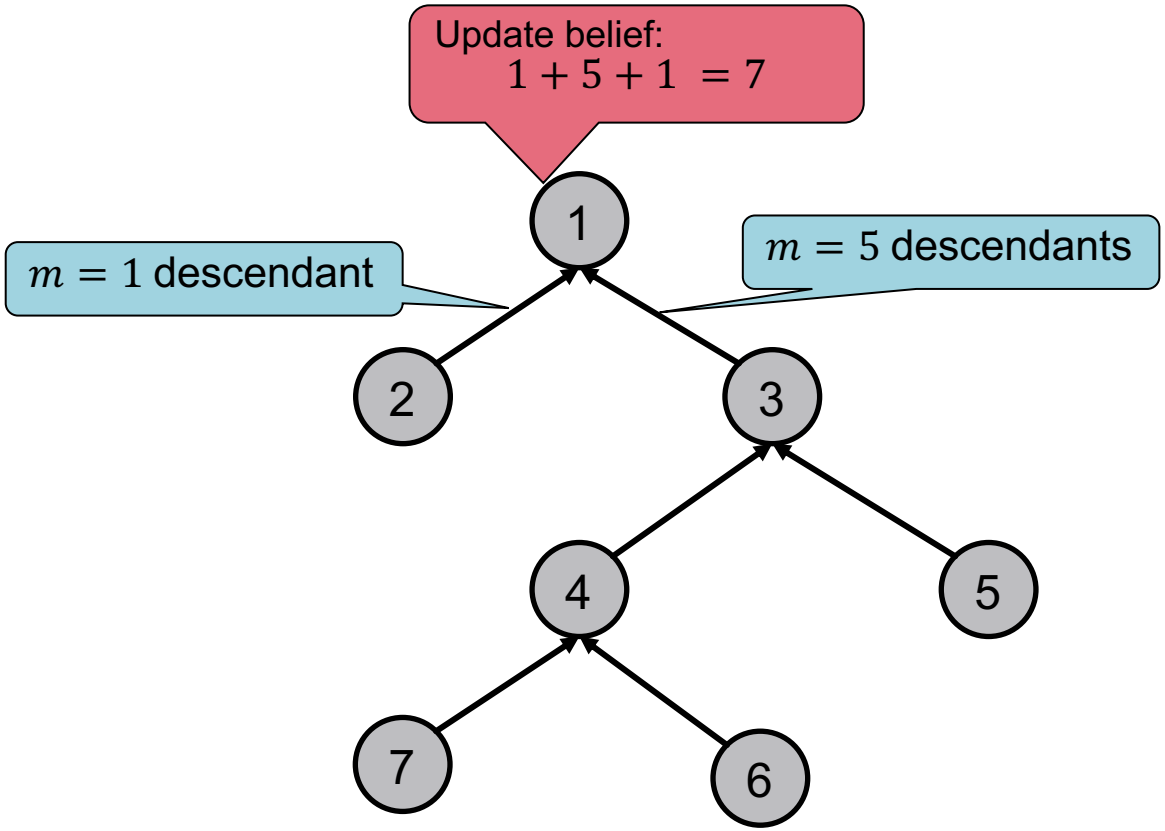
Message passing in a tree

Update beliefs in tree structure

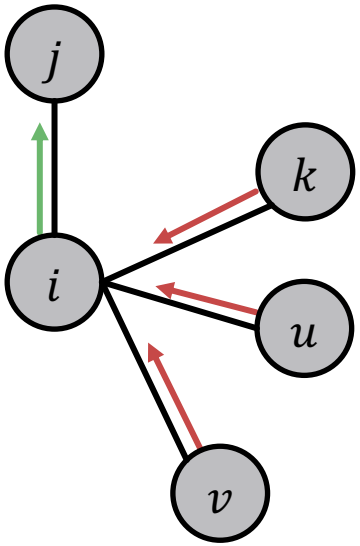


Message passing in a tree

Update beliefs in tree structure



Loopy BP Algorithm



What message will i send to j?

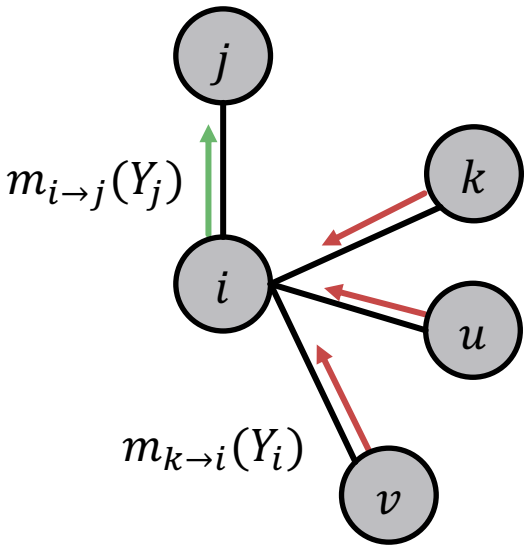
- It depends on what i hears from its neighbors
- Each neighbor passes a message to i its beliefs of the state of i

I (node i) believe that you (node j) belong to class Y_j with probability …

Notation

- ***Label-label potential matrix*** $\boldsymbol{\psi}$: Dependency between a node and its neighbor. $\boldsymbol{\psi}(Y_i, Y_j)$ is proportional to the probability of a node j being in class Y_j given that it has neighbor i in class Y_i.
- ***Prior belief*** $\boldsymbol{\phi}$: $\phi(Y_i)$ is proportional to the probability of node i being in class Y_i.
- $m_{i \to j}(Y_j)$ is i's message / estimate of j being in class Y_j.
- $\mathcal{L}$ is the set of all classes/labels

Loopy BP Algorithm



1. Initialize all messages to 1
2. Repeat for each node:

Label-label potential

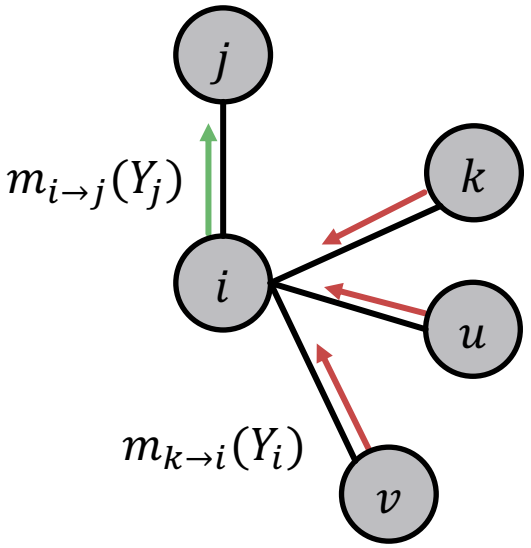
All messages sent by neighbors from previous round

$$m_{i\to j}(Y_j) = \sum_{Y_i\in\mathcal{L}} \psi(Y_i, Y_j)\,\phi_i(Y_i) \prod_{k\in N_i\setminus \mathrm{j}} m_{k\to i}(Y_i), \forall Y_j \in \mathcal{L}$$

Sum over all states

Prior

Loopy BP Algorithm



After convergence:

$b_i(Y_i)$ = node i's belief of being in class Y_i

All messages from neighbors

Prior

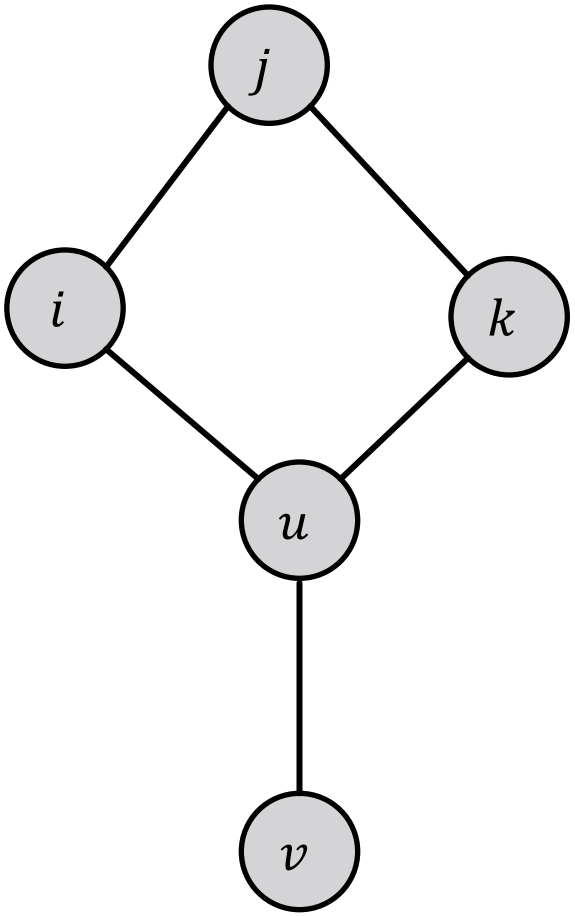
$$b_i(Y_i) = \phi_i(Y_i) \prod_{j \in N_i} m_{j\to i}(Y_i), \quad \forall\, Y_i \in \mathcal{L}$$

Example: Loopy Belief Propagation

- Now we consider a graph with cycles
- There is no longer an ordering of nodes
- We apply the same algorithm as in previous slides:
  - Start from arbitrary nodes
  - Follow the edges to update the neighboring nodes

Example: Loopy Belief Propagation

What if our graph has cycles?

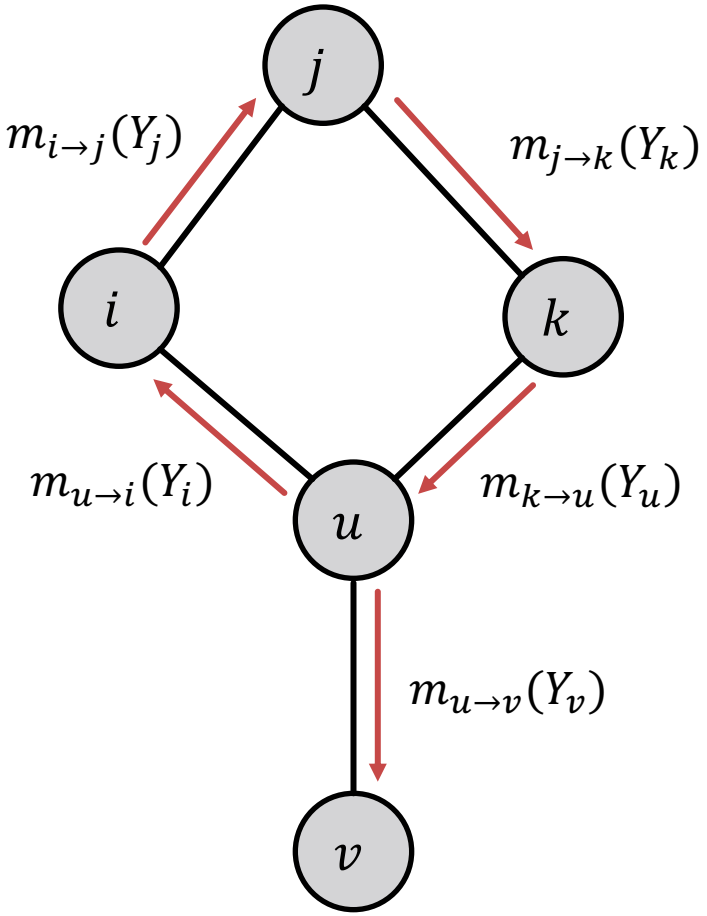


Messages from different subgraphs are **no longer independent**!

But we can still run BP, but it will pass messages in loops.

Example: Loopy Belief Propagation

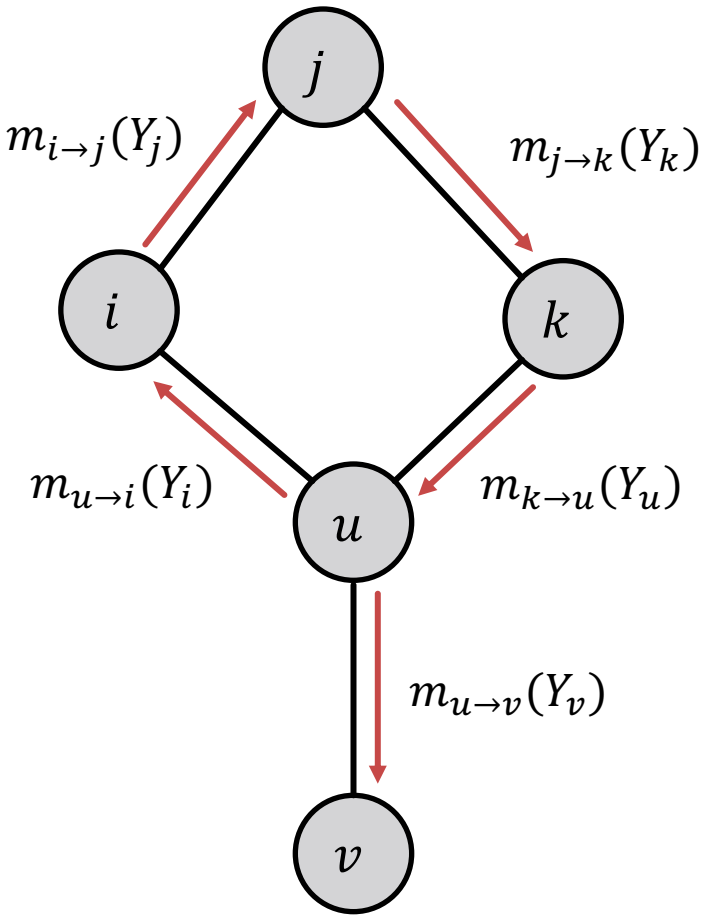
What if our graph has cycles?



Messages from different subgraphs are **no longer independent**!

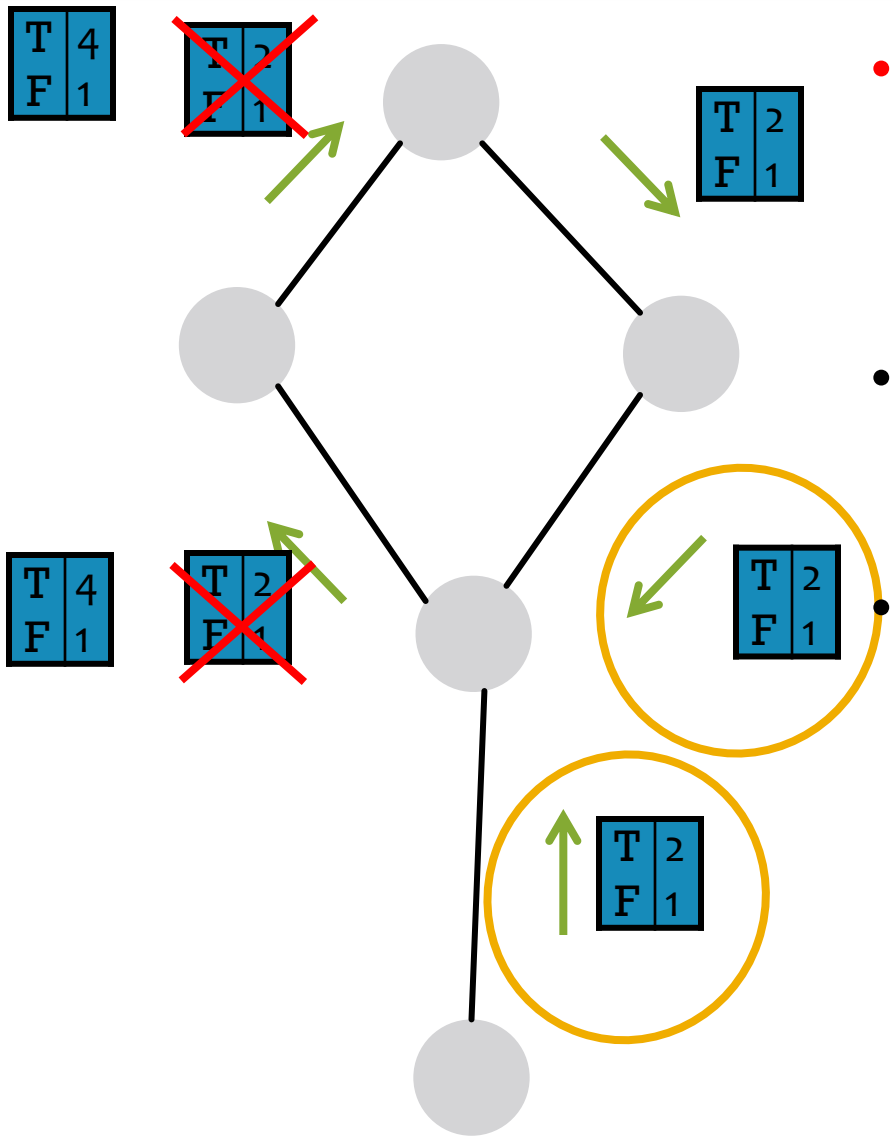
But we can still run BP, but it will pass messages in loops.

What Can Go Wrong?



- **Beliefs may not converge**
  - Message $m_{u\to i}(Y_i)$ is based on initial belief of i, not a **separate evidence** for i
  - The initial belief of i (which could be incorrect) is reinforced by the cycle $i \to j \to k \to u \to i$
- However, in practice, Loopy BP is still a good heuristic for complex graphs which contain many branches.

What Can Go Wrong?



- Messages loop around and around: 2, 4, 8, 16, 32, ... More and more convinced that these variables are T!
- BP incorrectly treats this message as **separate evidence** that the variable is T (true).
- Multiplies these two messages as if they were independent.
  - But they don't actually come from *independent* parts of the graph.
  - One influenced the other (via a cycle).

This is an extreme example. Often in practice, the cyclic influences are weak. (As cycles are long or include at least one weak correlation.)

66

Advantages of Belief Propagation

- **Advantages**:
  - Easy to program & parallelize
  - General: can apply to any graph model with any form of potentials
    - Potential can be higher order: e.g. $\boldsymbol{\psi}(Y_i, Y_j, Y_k, Y_v \ldots)$
- **Challenges**:
  - Convergence is not guaranteed (when to stop), especially if many closed loops
- **Potential functions** (parameters)
  - Require training to estimate

Summary

- We learned how to leverage correlation in graphs to make prediction on nodes
- Key techniques:
  - Relational classification
  - Iterative classification
  - Loopy belief propagation