IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

Week 2: Node Embedding, Link Analysis

Instructor: Thanh H. Nguyen

Many slides are adapted from https://web.stanford.edu/class/cs224w/

Announcement and Reminder

- Week 3: (Tuesday, August 20th, 8 am)
  - Will be held on Zoom
  - Zoom link will be posted on Slack

- Class project
  - Project proposal (deadline: end of week 4)

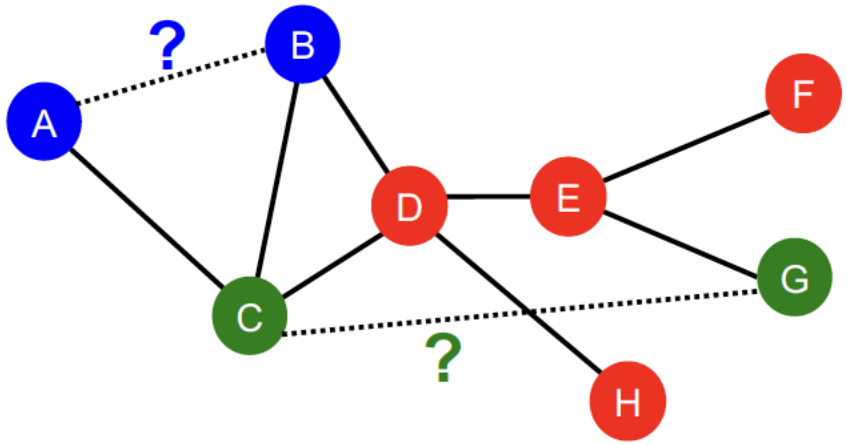
Outline

- Feature engineering: finish traditional link/graph tasks
- Graph representation learning
  - Shallow node embedding
  - Graph embedding
  - Connection to matrix factorization
- Link analysis
  - PageRank
  - Personalized PageRank
  - Random walk with restarts

Link Prediction Task and Features

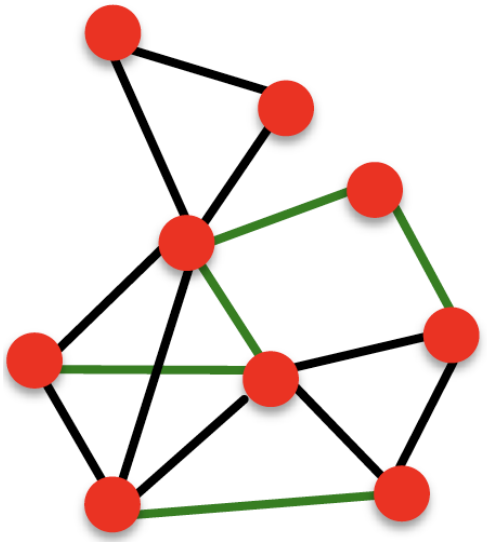
Link-level Prediction Task: Recap

- The task is to predict new/missing/unknown links based on the existing links.
- At test time, node pairs (with no existing links) are ranked, and top K node pairs are predicted.
- Task: Make a prediction for a pair of nodes.



Link Prediction as a Task

- Links missing at random
  - Remove a random set of links and then aim to predict them
- 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:
    - $n = |E_{new}|$: number of edges that appear during the test period $G[t_1, t_1']$
    - Take top n elements of L and count corrected edges

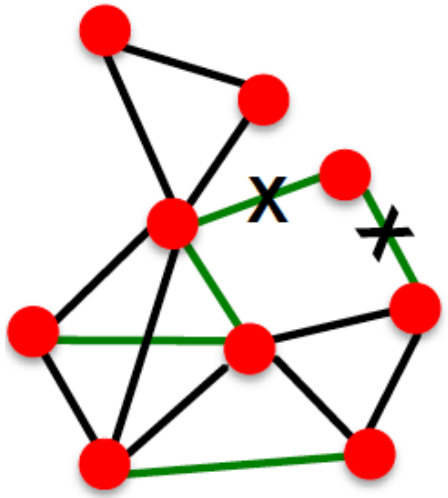


$G[t_0, t_0']$

$G[t_1, t_1']$

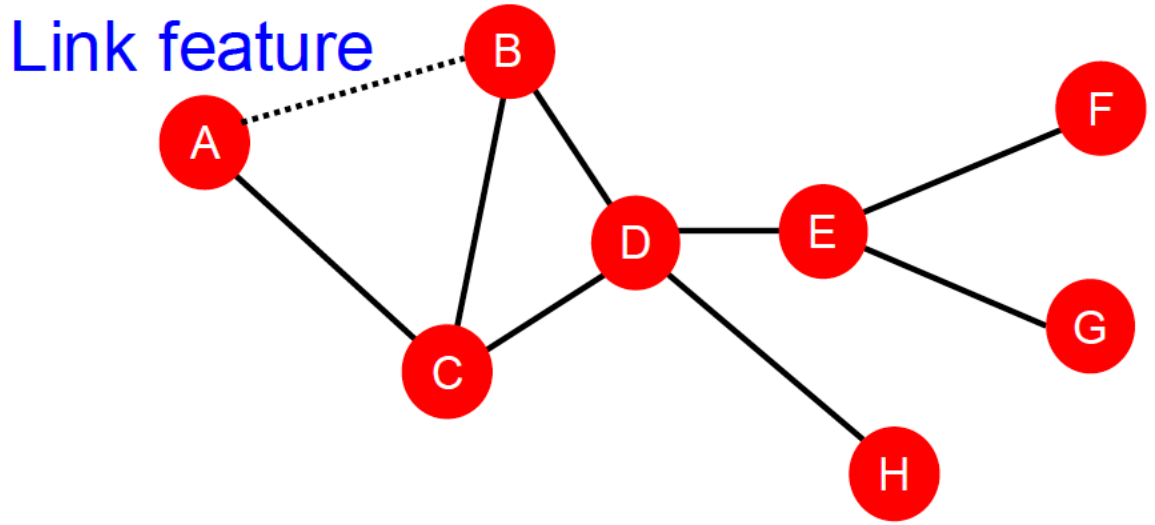
Link Prediction via Proximity

- Methodology:
  - For each pair of nodes (x, y), compute score $c(x, y)$
    - For example: #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 $G[t_1, t_1']$



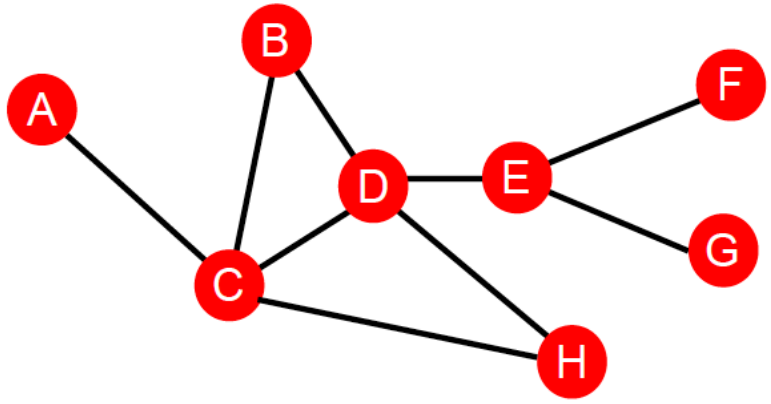
Link-Level Feature: Overview

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



Distance-based Feature

- Shortest path distance between two nodes
  - Example



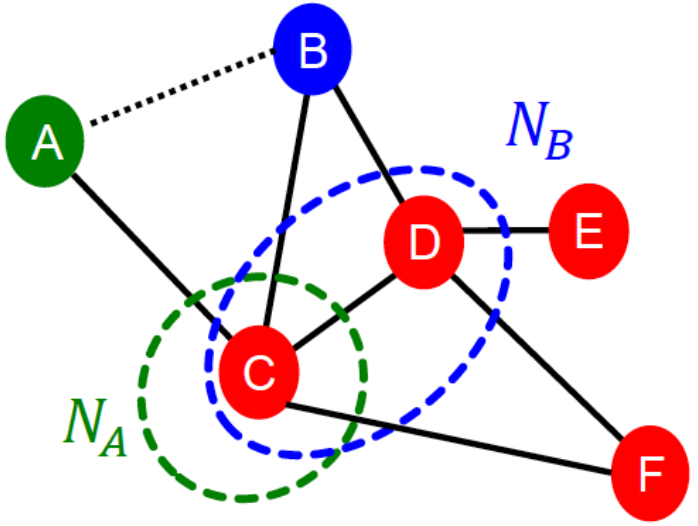
$$S_{BH} = S_{BE} = S_{AB} = 2$$

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

  - However, this does not capture the degree of neighborhood overlap
    - Node pair (B, H) has two shared neighboring nodes
    - Node pair (B, E) and (A, B) only have 1 such nodes

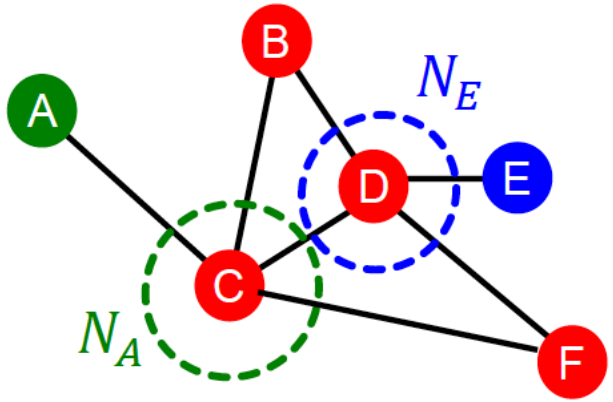
Local Neighborhood Overlap

- Capture #neighboring nodes shared between two nodes
  - Common neighbors: $|N(v_1) \cap N(v_2)|$
    - Example: $|N(A) \cap N(B)| = 1$
  - Jaccard's coefficient: $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
    - Example: $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{1}{2}$
  - 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 overlap
  - Metric is always zero if the two nodes do not have any neighbors in common



$$|N(A) \cap N(E)| = 0$$

  - However, the two nodes may still potentially be connected in the future
- Global neighborhood overlap: resolves the limitation by considering the entire graph

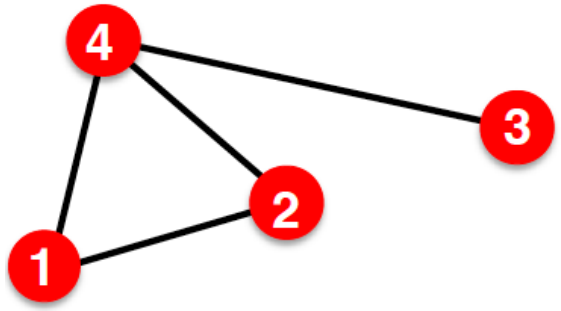
Global Neighborhood Overlap

- Katz index: count the number of walks of all lengths between a given pair of nodes

- Compute #walks:
  - Use powers of the graph adjacency matrix

Intuition: Powers of Adj Matrices

- Compute #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 $P^{(k)} = A^k$
  - $P_{uv}^{(1)} = A_{uv} = \#walks$ of length 1 (direct neighborhood) between u and v



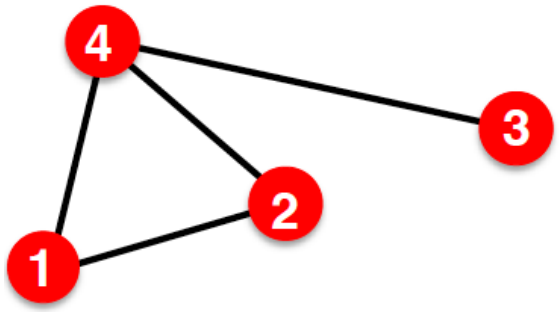
$$\boldsymbol{P_{12}^{(1)} = A_{12}}$$

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

Intuition: Powers of Adj Matrices

- How to compute $P_{uv}^{(2)}$?
  - Step 1: Compute #walks of length 1 between each neighbor of u 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

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

Power of adjacency

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

Global Neighborhood Overlap

- Katz index between v_1 and v_2 is computed as:

Sum over *all walk lengths*

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

$A^l_{v_1 v_2}$: #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

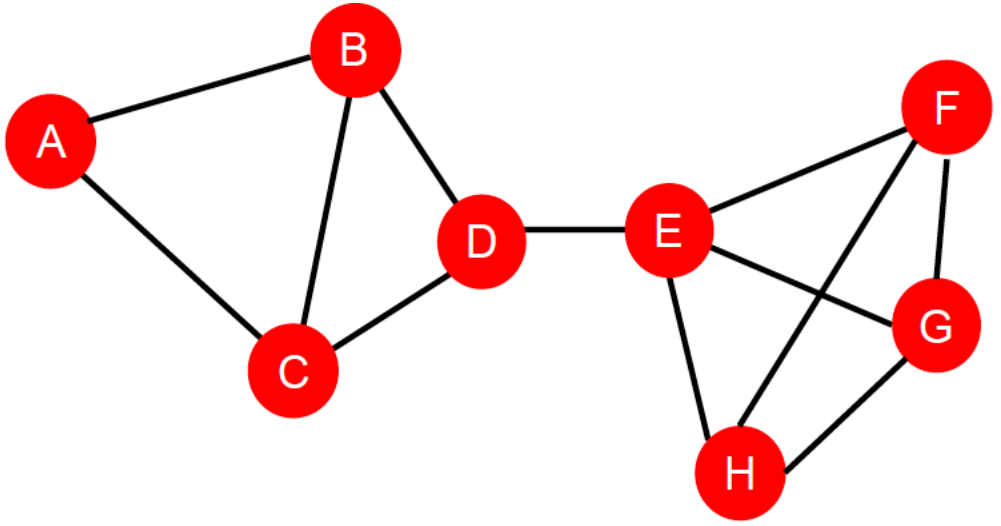
- Distance-based feature
  - Use the shortest path length
  - Does not capture how neighborhood overlaps
- Local neighborhood overlap
  - Capture how many neighboring nodes are shared
  - Become zero when no neighbors are shared
- Global neighborhood overlap
  - Use global graph structure to score two nodes
  - Katz index count #walks of all lengths between two nodes

Graph-Level Features and Graph Kernels

Graph-Level Features

- Goal: characterize structure of an entire graph

- Example:



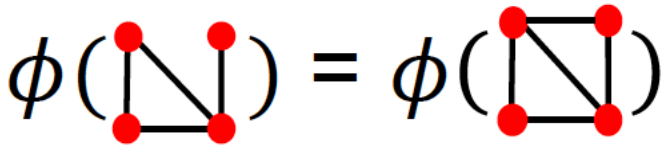
Graph-Level Features: Overview

- Graph kernels: measure similarity between two graphs
  - Graphlet kernel
  - Weisfeiler-Lehman kernel

  - Other kernels (will not be covered in this lecture)
    - Random-walk kernel
    - Shortest-path graph kernel
    - Many more…

Graph Kernel: Ideas

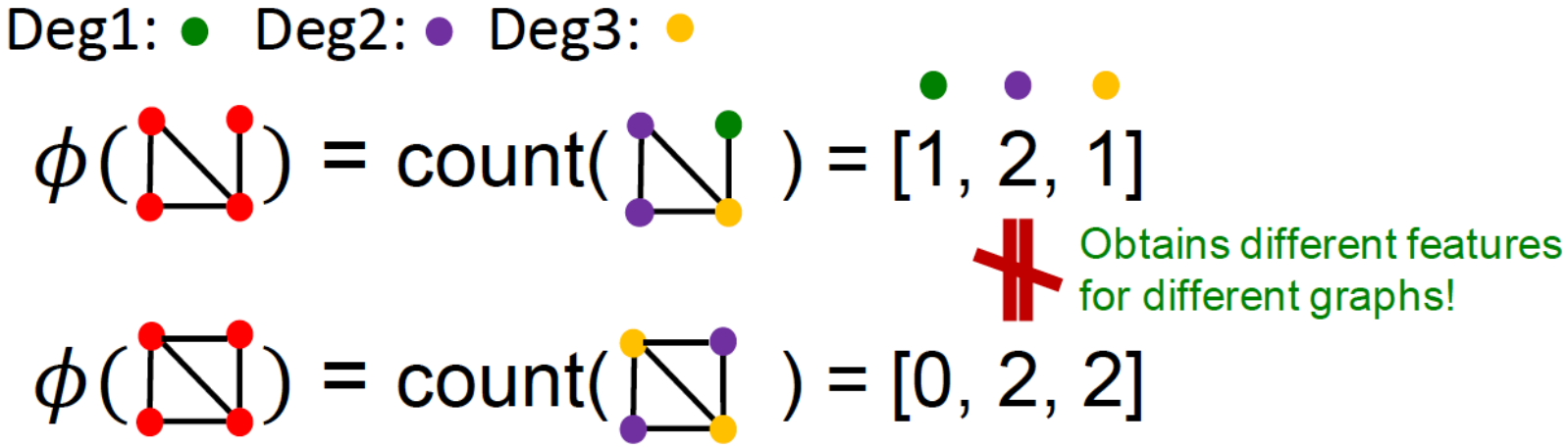
- Goal: design graph feature vector $\phi(G)$
- Key ideas: Bag-of-Words (BoW) for a graph
  - Recall: BoW uses word counts as features for documents (no ordering)
  - Naïve extension to a graph: consider nodes as words
    - Limitation:



    - Since both graphs have 4 nodes, we get the same feature vector for two different graphs

Graph Kernel: Key Ideas

- What if we use Bag of node degrees?



- Both Graphlet kernel and Weisfeier-Lehman (WL) use Bag-of-* representation of graphs.

Graph-Level Graphlet Features

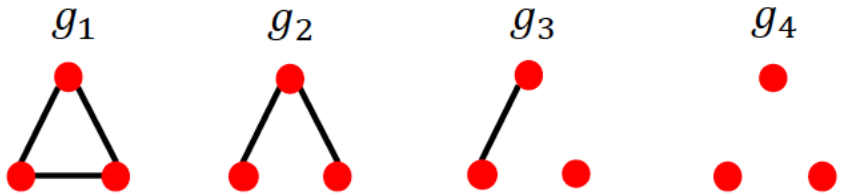
- Key idea: count #different graphlets in a graph

  - Note: definition of graphlets here is slightly different from node-level features

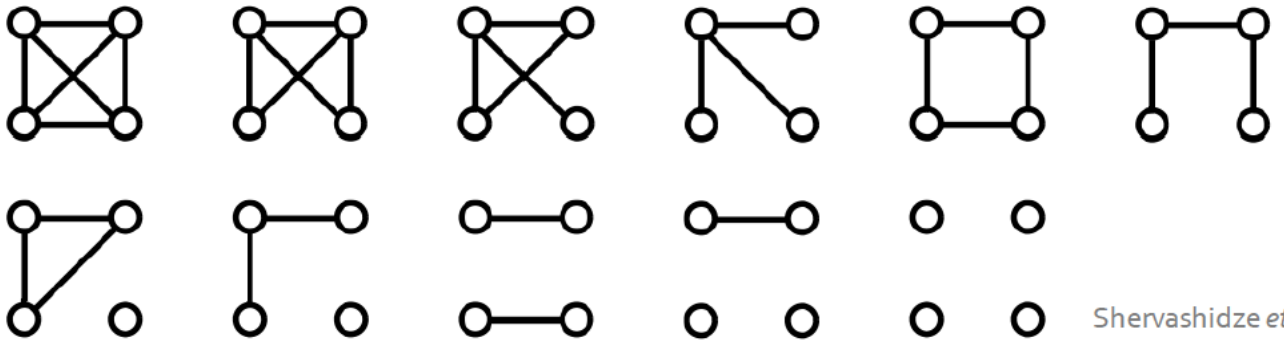
  - Two differences:
    - Nodes in graphlets here do not need to be connected
    - Graphlets here are not rooted

Graph-Level Graphlet Features

- Let $\mathcal{G}_k = (g_1, g_2, \ldots, g_{n_k})$ 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

Graph-Level Graphlet Features

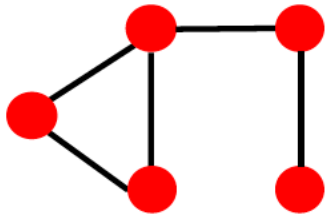
- Given graph G, and a graphlet list $\mathcal{G}_k = (g_1, g_2, \ldots, g_{n_k})$, define the graphlet count vector $f_G \in \mathbb{R}^{n_k}$ as:

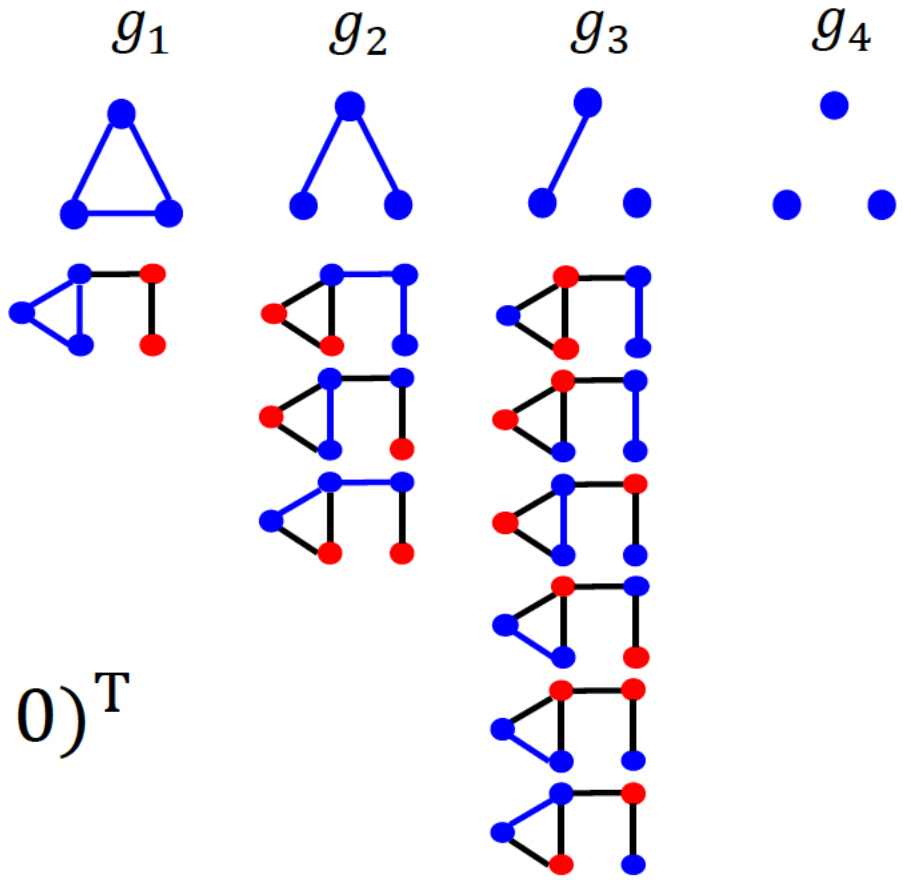
$$(f_G)_i = \#(g_i \in G), \forall i = 1, 2, \ldots, n_k$$

Graph-Level Graphlet Features

- Example: k = 3

G





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

Graph-Level Graphlet Kernel

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

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

- Problem:
  - If G and G' have different sizes, that will greatly skew the value.

- Solution: normalize each feature vector

$$h_G = \frac{f_G}{sum(f_G)} \qquad K(G, G') = h_G^T h_{G'}$$

The Graphlet Kernel

- Limitations: counting graphlets is expensive
  - Counting size-k graphlets for a graph of size n by enumeration takes n^k
  - This is unavoidable in worst case since subgraph isomorphism test (judging if 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 graphlets of size k.

- Can we design a more efficient graph kernel?

Weisfeiler-Lehman Kernel

- Goal: Design an efficient graph feature description $\phi(G)$

- Key idea: Use neighborhood structure to iteratively enrich node vocabulary.
  - Generalized version of Bag of node degree since node degrees are one-hop neighborhood information

- Algorithm: 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

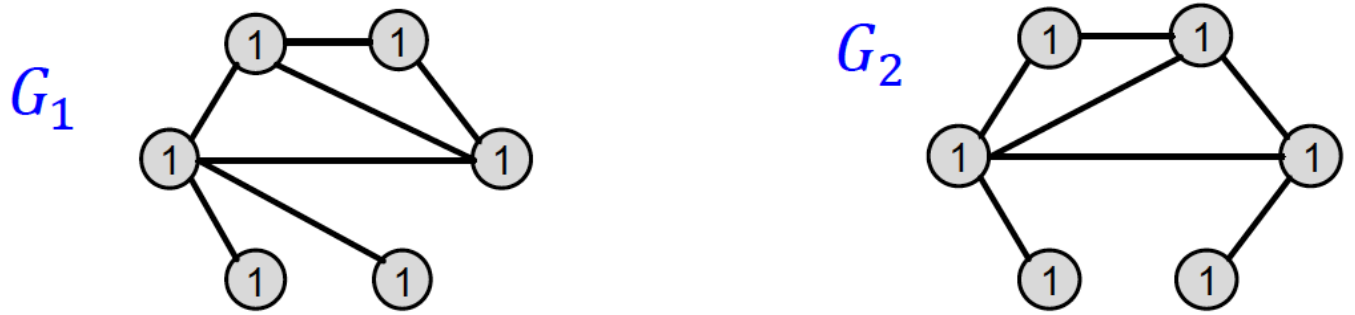
$$c^{(k+1)}(v) = HASH\left(\left\{c^{(k)}(v), \left\{c^{(k)}(u)\right\}_{u \in N(v)}\right\}\right)$$

where HASH maps different inputs to different colors

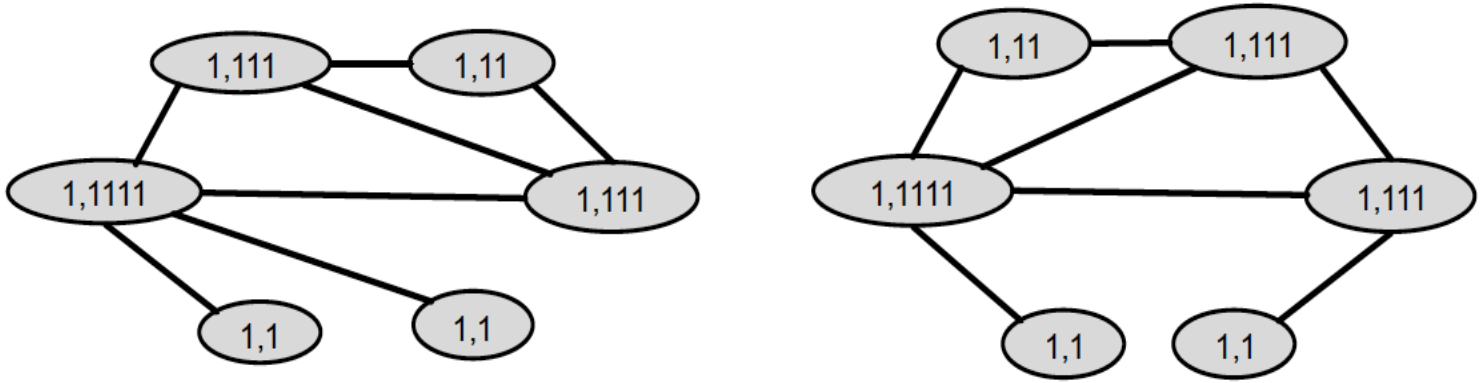
  - After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood.

Color Refinement: Example

- Assign initial colors

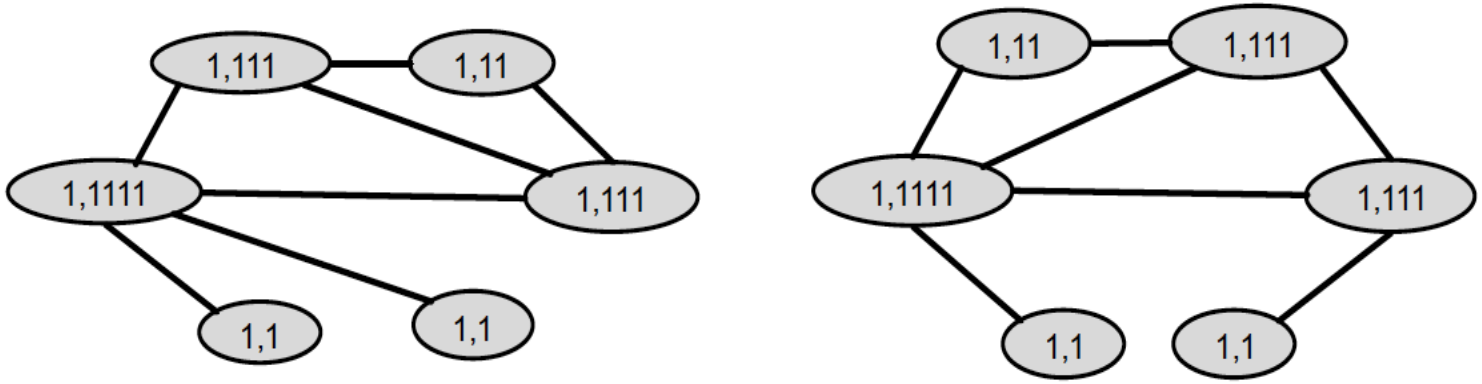


- Aggregate neighboring colors

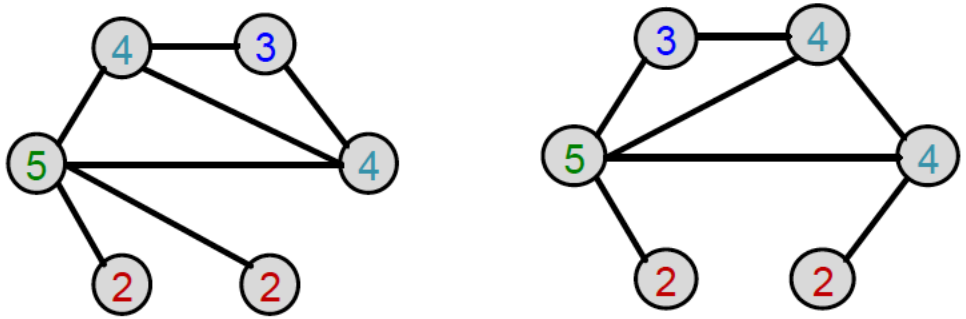


Color Refinement: Example

- Aggregate neighboring colors



- Hash aggregated colors

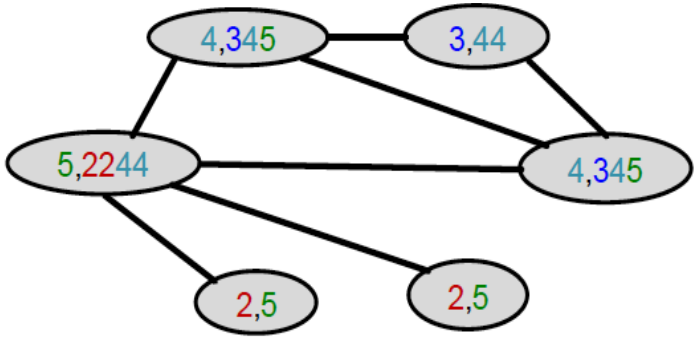


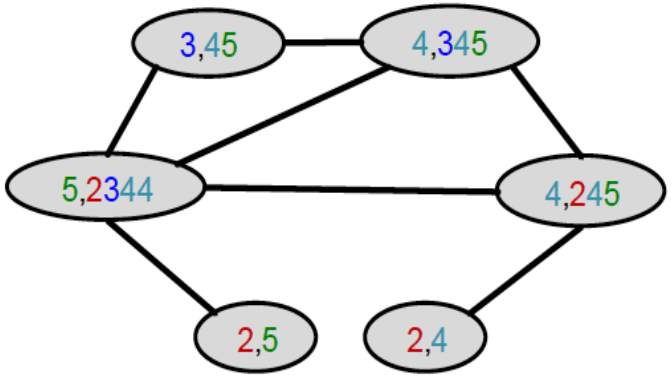
Hash table

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

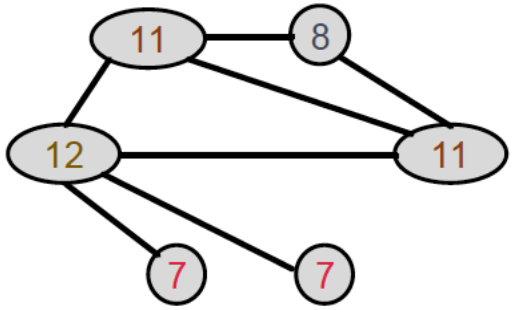
Color Refinement: Example

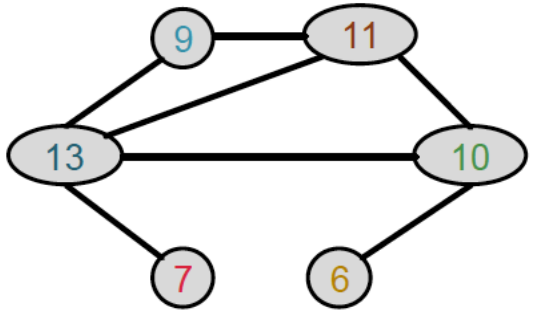
- Aggregate neighboring colors





- Hash aggregated colors



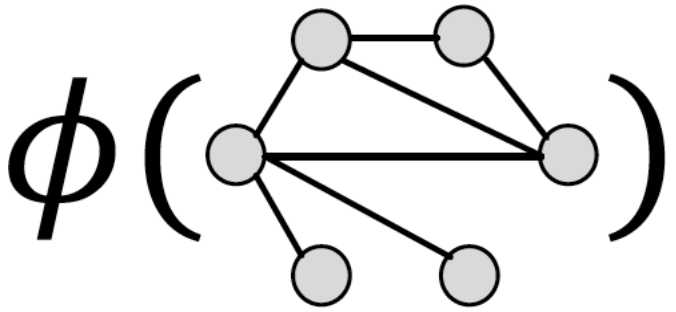


Hash table

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 #nodes with a given color

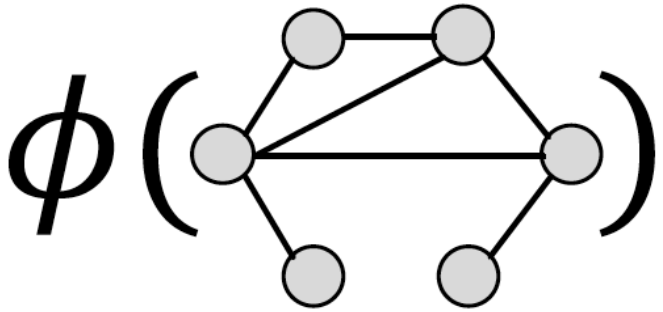


Colors

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

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

Counts

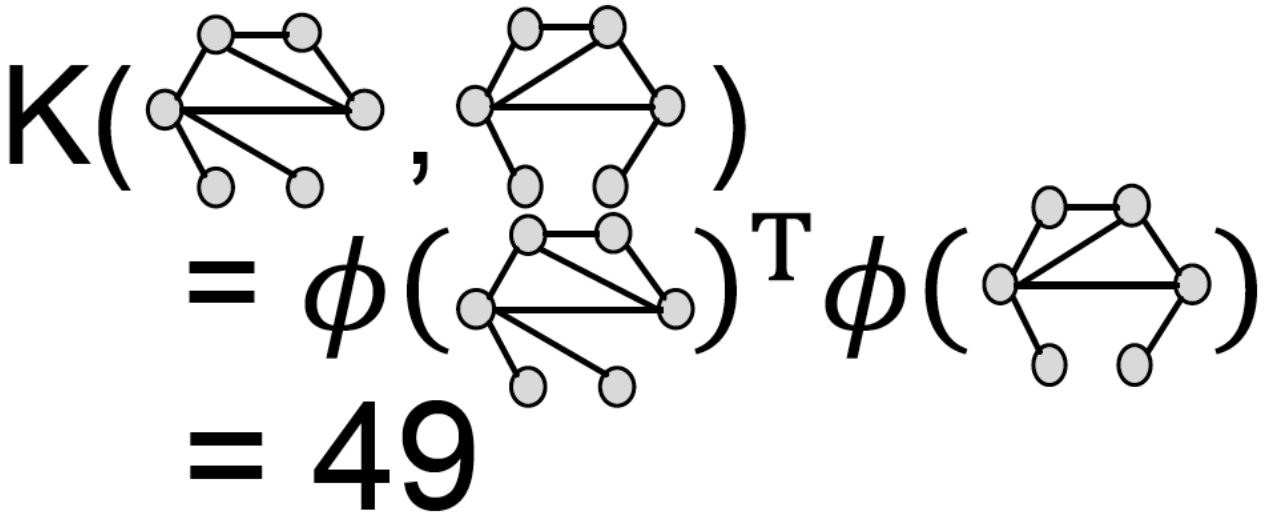


1,2,3,4,5,6,7,8,9,10,11,12,13

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

Weisfeiler-Lehman Kernel

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



Weisfeiler-Lehman Kernel

- Computationally efficient
  - Time complexity for color refinement at each step is linear in #edges.
- When computing a kernel value, only colors appeared in the two graphs need to be tracked
  - Thus, #color 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

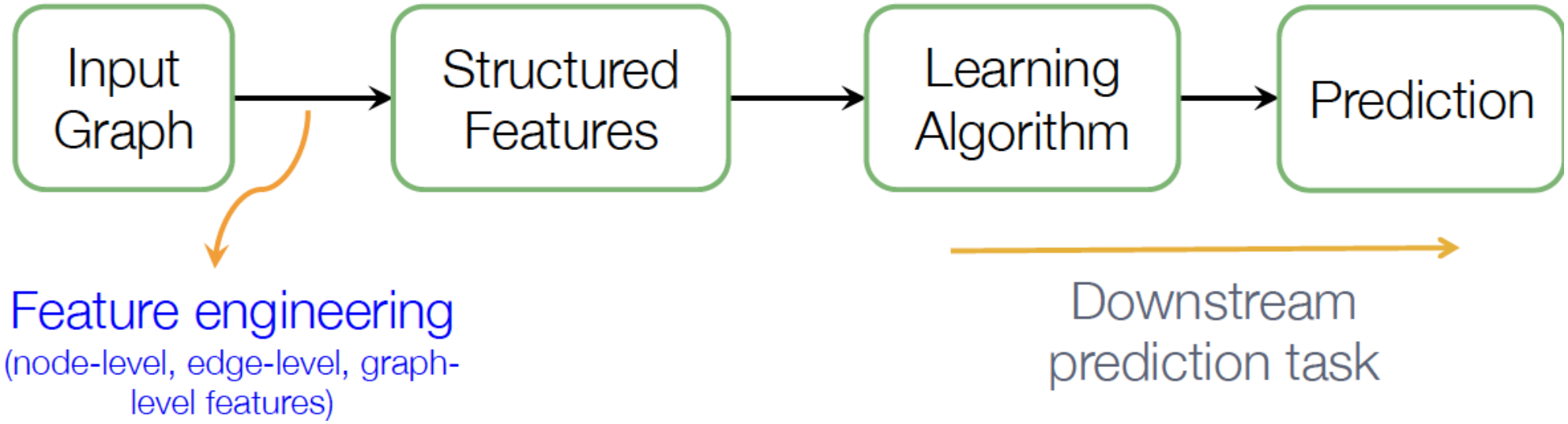
- Weisfeiler-Lehman kernel
  - Apply K-step color refinement algorithm to enrich node colors
    - Different colors capture different K-hop neighborhood structures
  - Graph is represented as Bag-of-colors
  - Computationally efficient
  - Closely related to Graph Neural Nets (will study later)

Summary

- Traditional ML pipeline
  - Hand-crafted (structural) features + ML models
- Hand-crafted features for graph data
  - Node-level: node degree, centrality, clustering coefficient, graphlets
  - Link-level: distance-based features, local/global neighborhood overlap
  - Graph-level: graphlet kernel, WL kernel
- However, we only considered featurizing the graph structure (but not the attribute of nodes and their neighbors)

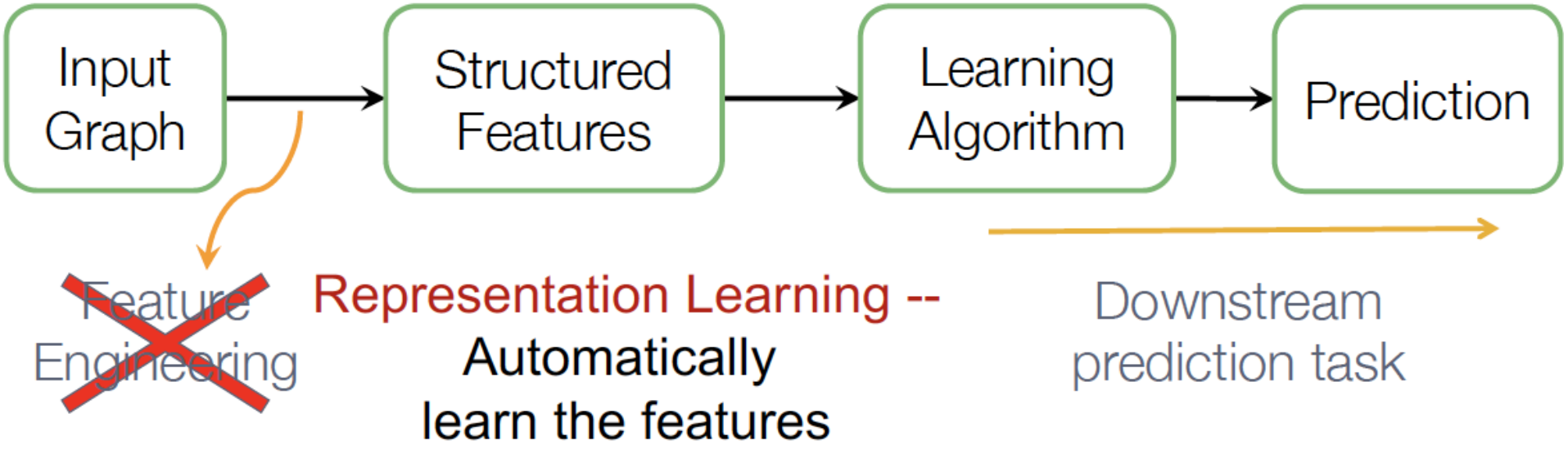
Graph Representation Learning

Recap: Traditional ML for Graphs



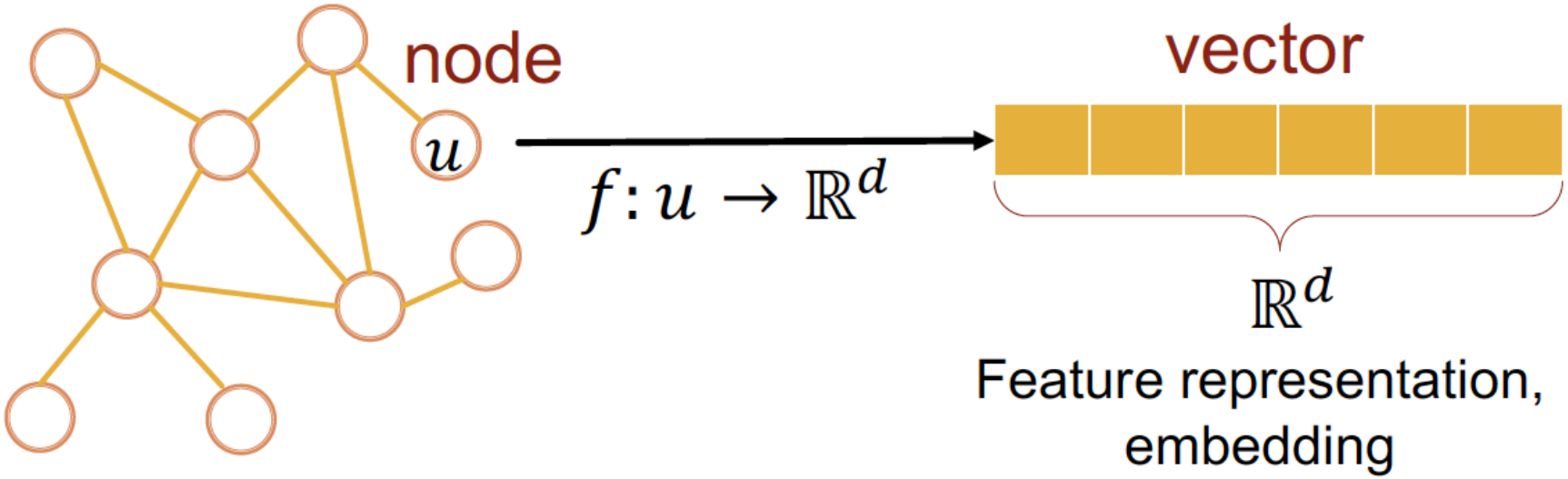
Graph Representation Learning

- Alleviate the need to do feature engineering every single time



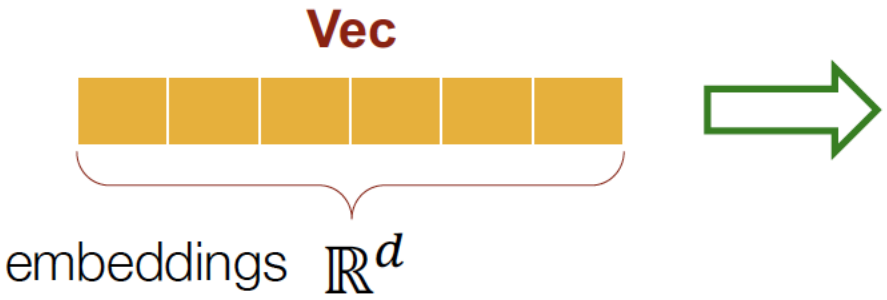
Graph Representation Learning

- Goal: Efficient task-independent feature learning for ML with graphs



Why Embedding

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

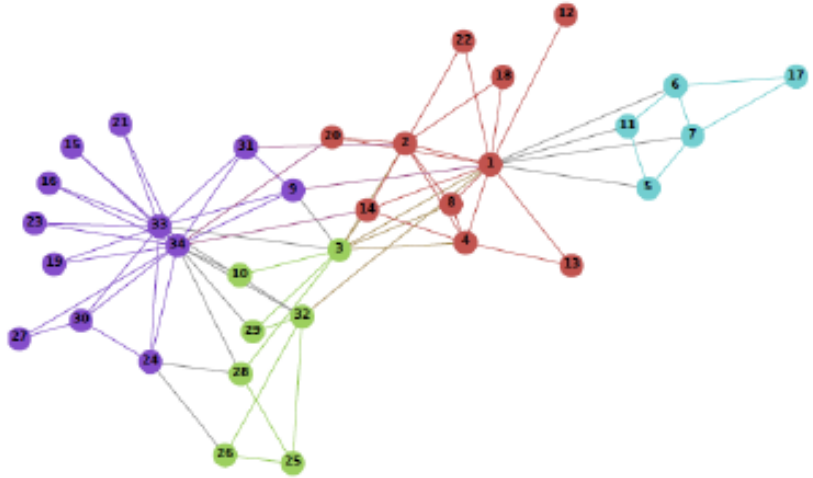


Tasks:

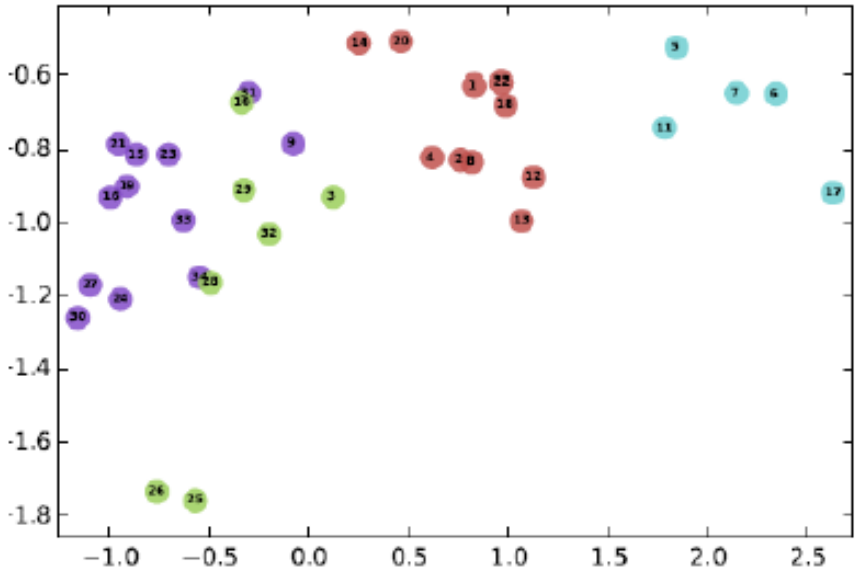
- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering

Example of Node Embedding

- Zachary's Karate Club network



Input

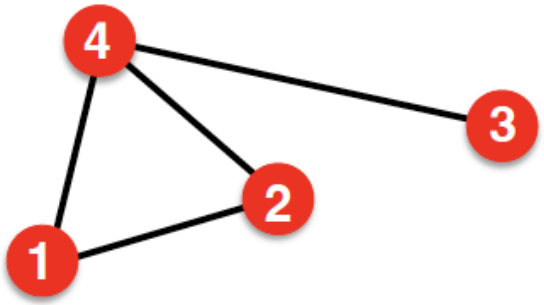


Output

Node Embeddings: Encoder & Decoder

Setup

- Assume an undirected graph G
  - V: vertex set
  - A: adjacency matrix (assuming binary)
  - For simplicity: No node features or extra information

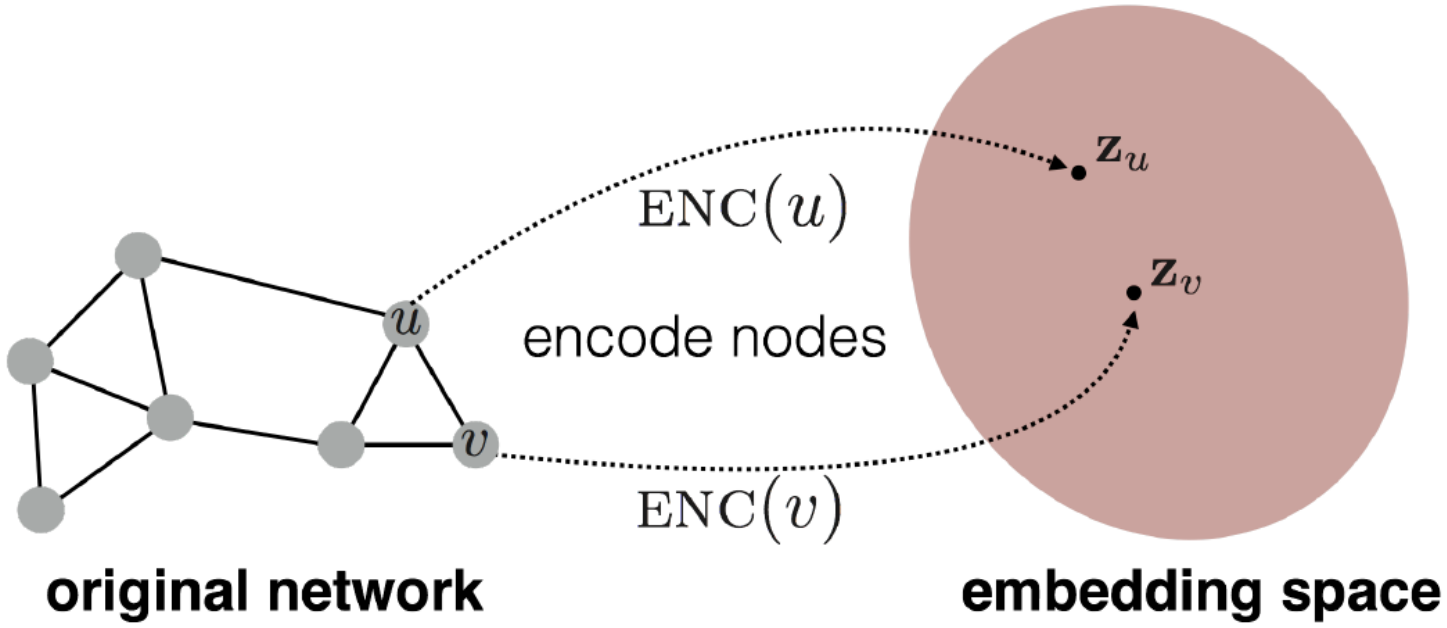


V: {1, 2, 3, 4}

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

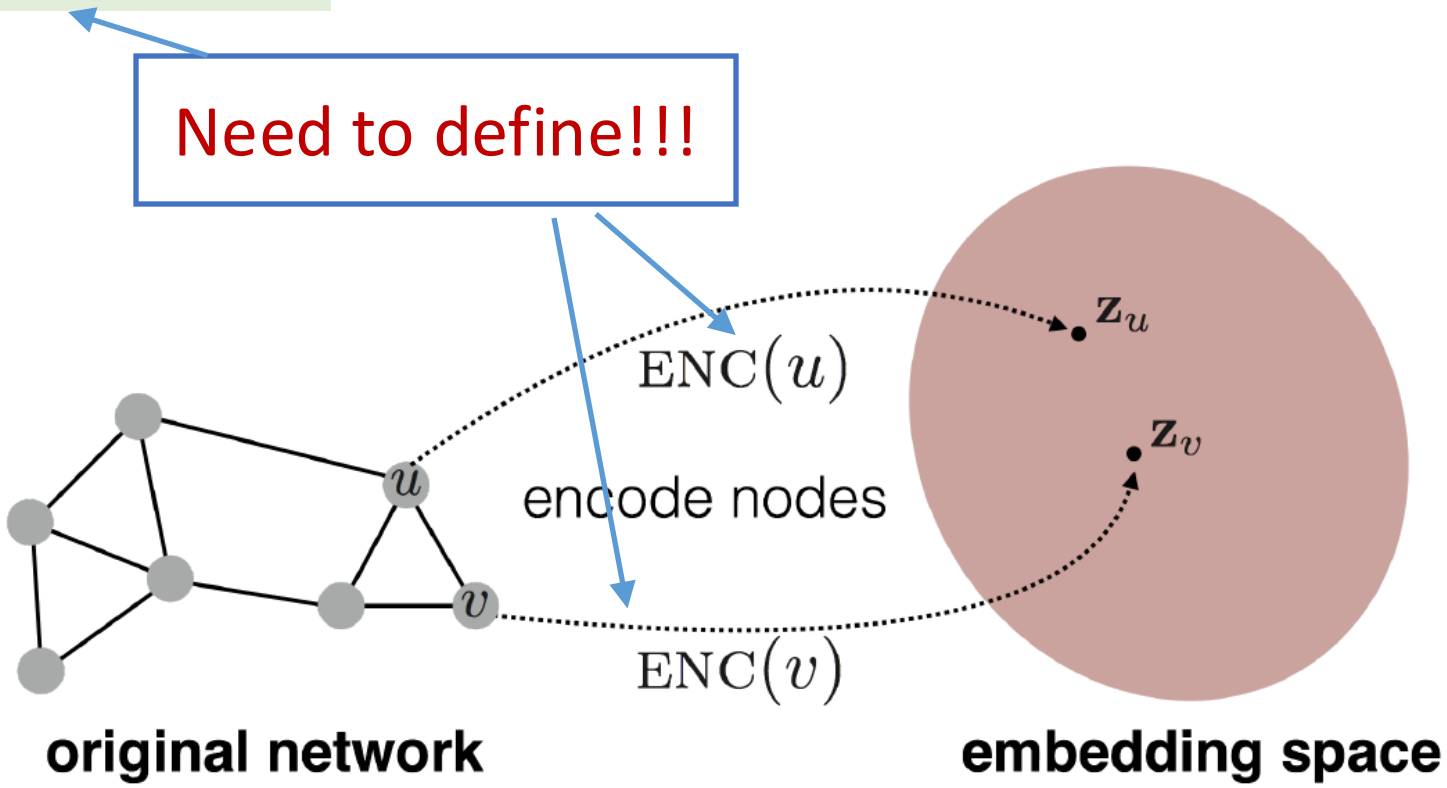
Embedding Nodes

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



Embedding Nodes

- Goal: $similarity(u, v) \approx z_u^T z_v$



Learning Node Embedding

1. Encoder maps from nodes to embeddings
2. Define a node similarity function (e.g., a measure of similarity in the original graph)
3. Decoder (DEC) maps from embeddings to the similarity score
4. Optimize parameters of the encoder s.t:

$$similarity(u, v) \approx z_v^T z_u$$

in original graph

similarity of embeddings

$$DEC(z_v^T z_u)$$

Two Key Components

- Encoder: map each node to a low-dimensional vector

$$\text{ENC}(v) = z_v$$

d-dimensional embedding

- Similarity function: specify mapping of relationships between embedding space and original space

$$similarity(u, v) \approx z_v^T z_u$$

Decoder

"Shallow" Encoding

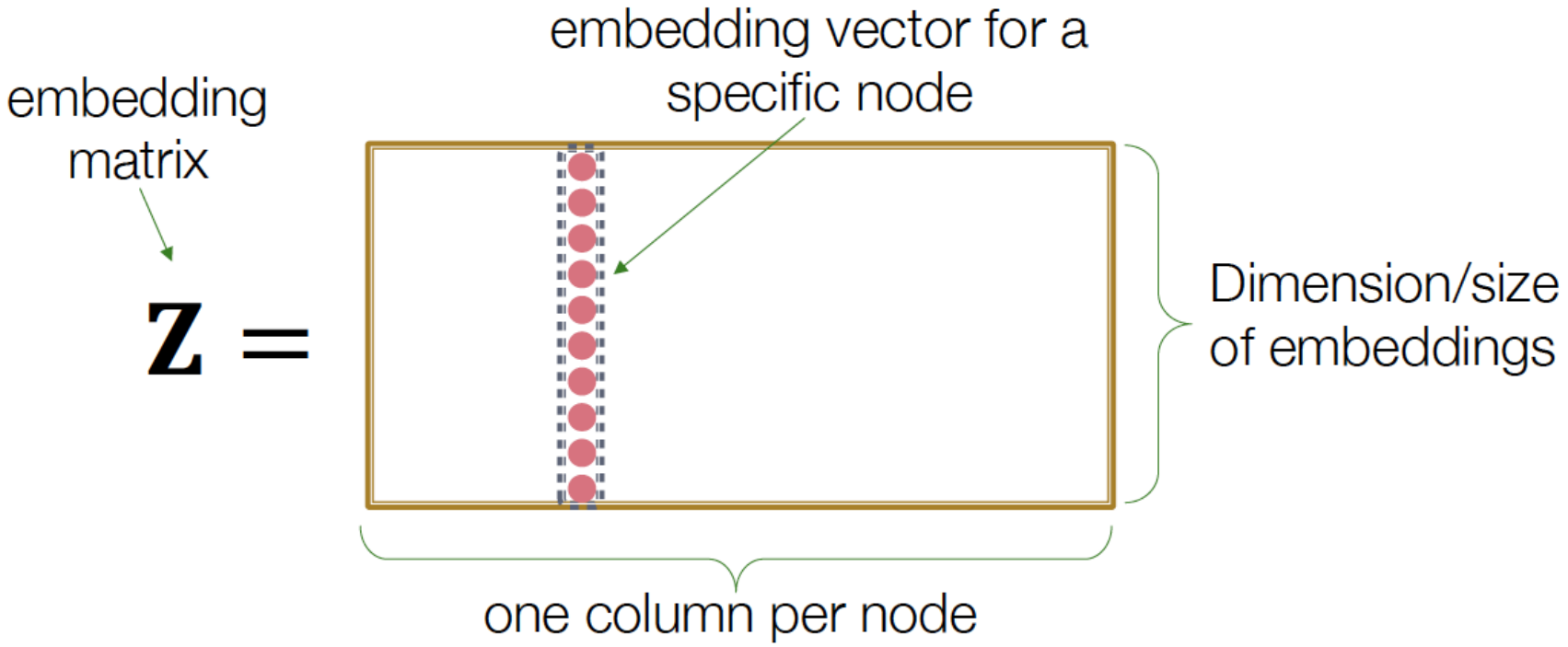
- Simplest encoding approach: Encoder is just an embedding-lookup

$$ENC(v) = z_v = Z \cdot v$$

- $Z \in \mathbb{R}^{d \times |V|}$: matrix, each column is a node embedding (need to optimize)
- $v \in \mathbb{I}^{|V|}$: indicator vector, all zeros 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: Z which contains node embeddings z_u for each node $u \in V$
  - We will cover deep encoders in the GNNs

  - Decoder: based on node similarity
  - Objective: maximize $z_v^T z_u$ for node pairs (u, v) that are similar.

How to Define Node Similarity

- Key choice of methods is how they define 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 walk, and how to optimize embeddings for such a similarity measure

Note on Node Embeddings

- This is unsupervised/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

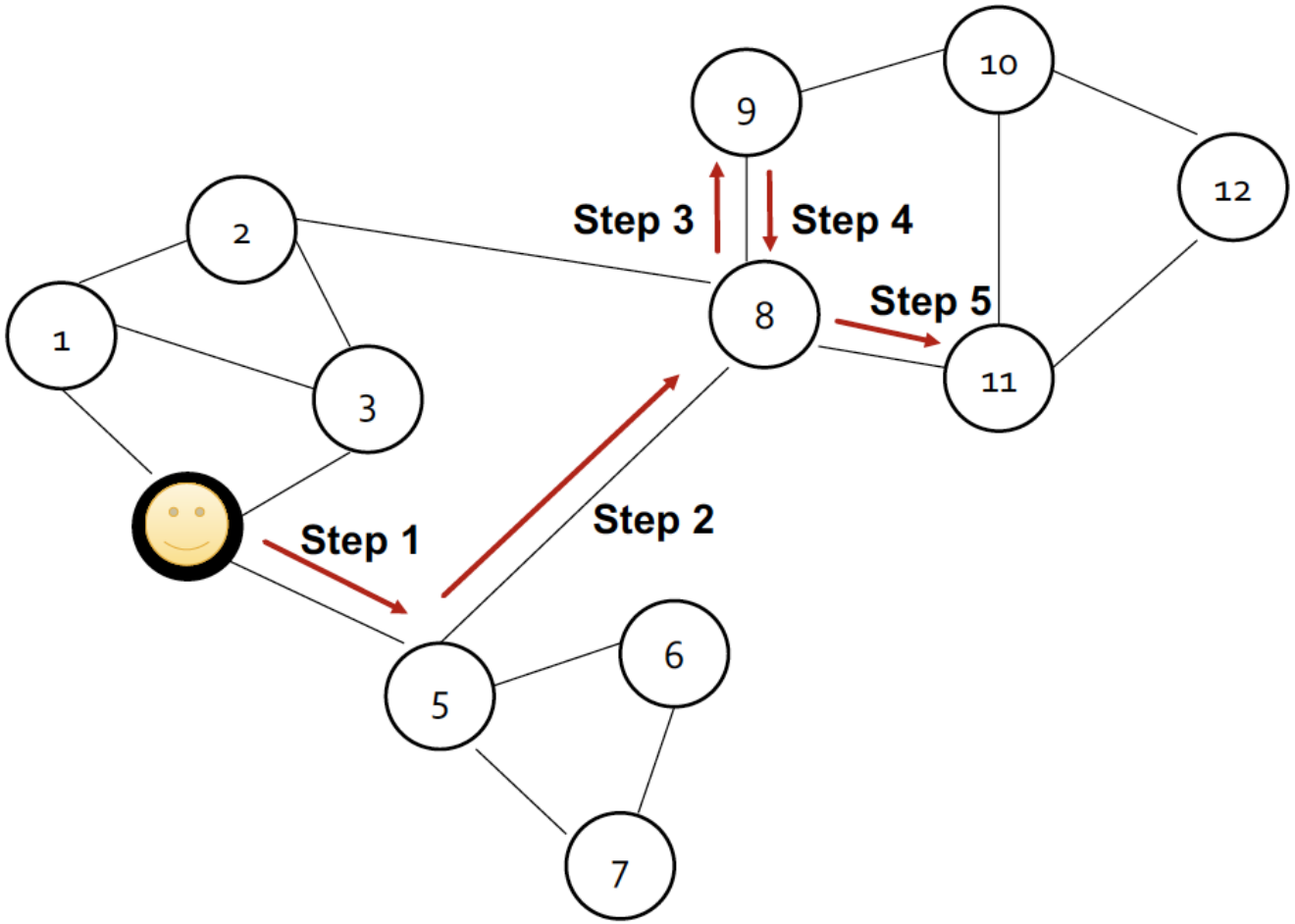
- These embeddings are task independent
  - They are not trained for a specific task but can be used for any task

Random Walk Approach

Notations

- Vector z_u
  - Embedding of node u
- Probability $P(v \mid z_u)$:
  - Our model prediction is based on embedding z_u
  - Predicted probability of visiting node v on random walks starting from u
- Functions to predict probabilities
  - Softmax function
  - Sigmoid function

Random Walk



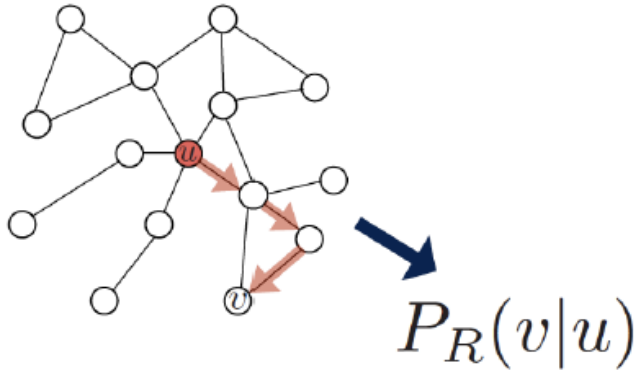
Given a graph and a starting point, we select a neighbor of it at random, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The random sequence of points visited this way is a random walk on graph.

Random-Walk Embeddings

$z_v^T z_u \approx$ Probability that u and v co-occur on a random walk over the graph

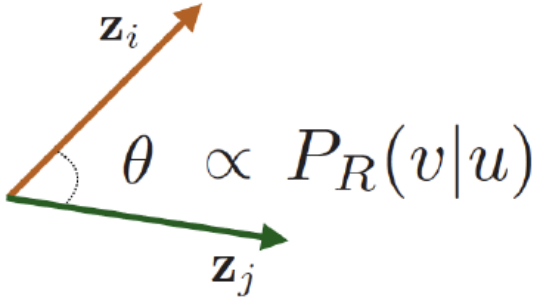
Random-Walk Embeddings

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



2. Optimize embeddings to encode these random walk statistics

Similarity in embedding space encodes random walk similarity



Why Random Walks?

1. Expressivity:
   - Flexible stochastic definition of node similarity that incorporates both local and high-order neighborhood information
   - Idea: if random walk starting from node u visit 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 embeddings of nodes in d-dimensional space that preserve similarity

- Idea: learn node embedding such that nearby nodes are close together in the graph.

- Given a node u, how do we define nearby nodes?
  - $N_R(u)$: neighborhood of u obtained by some random walk strategy R

Feature Learning as Optimization

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

$$\max_z \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

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

Random Walk Optimization

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

$$\max_z \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

Random Walk Optimization

- Equivalently,

$$\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\big(P(v \mid z_u)\big)$$

- Intuition: optimize embedding z_u to minimize the negative log-likelihood of random walk neighborhoods $N_R(u)$
- Parameterize $\mathrm{P}(v \mid z_u)$ using softmax:

$$P(v \mid z_u) = \frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)}$$

  - Goal: v to be most similar to 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

- Optimize random walk embeddings = Find embeddings that minimizes $\mathcal{L}$

Random Walk Optimization

- But doing this naively is too expensive

$$\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)}\right)$$

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

Random Walk Optimization

- But doing this naively is too expensive

$$\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)}\right)$$

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

Negative Sampling

- Solution: negative sampling

$$-\log\left(\frac{\exp(z_u^T z_v)}{\sum_{n\in V}\exp(z_u^T z_n)}\right)$$

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.
Reference: https://arxiv.org/pdf/1402.3722

$$\approx \log\left(\sigma(z_u^T z_v)\right) + \sum_{i=1}^{k}\log\left(\sigma\left(-z_u^T z_{n_i}\right)\right), n_i \sim P_V$$

Sigmoid function

Random distribution over nodes

- Just normalize against k random negative samples n_i
  - Quick neighborhood calculation

Negative Sampling

- Sampling k negative nodes n_i, each with prob. proportional to its degree

- Two considerations for k:
  - Higher k gives more robust estimates
  - Higher k corresponds to higher bias on negative events.
  - Typical choice: $k = 5\ to\ 20$

- Can negative samples be any node or only nodes not on the walk?
  - People often sample any nodes (for efficiency)

Stochastic Gradient Descent

- Evaluate gradient for each individual training example:

1. Initialize z_u at some randomized value for all nodes u

2. Iterate until convergence: $\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log\big(P(v \mid z_u)\big)$

   - Sample a node u, for all v calculate the gradient $\frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$
   - For all v, update: $z_v \leftarrow z_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$

Random Walk Summary

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

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)}\right)$$

We can efficiently approximate this using negative sampling

How Should We Randomly Walk?

- What strategies should we use to run these random walks?
  - Simplest idea: Just run fixed-length, unbiased random walks starting from each node (DeepWalk from Perozzi et al., 2013)
    - The issue is that such notion of similarity is too constrained.

- How can we generalize this?

Overview of node2vec

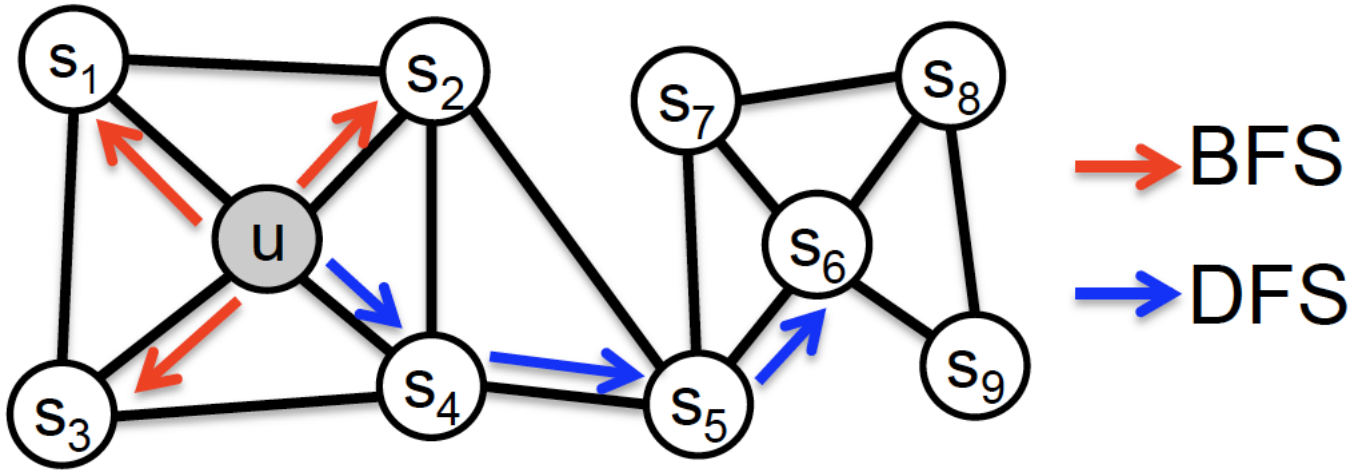
- Goal: Embed nodes with similar network neighborhoods close in embedding space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.

- Key observation: Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 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.

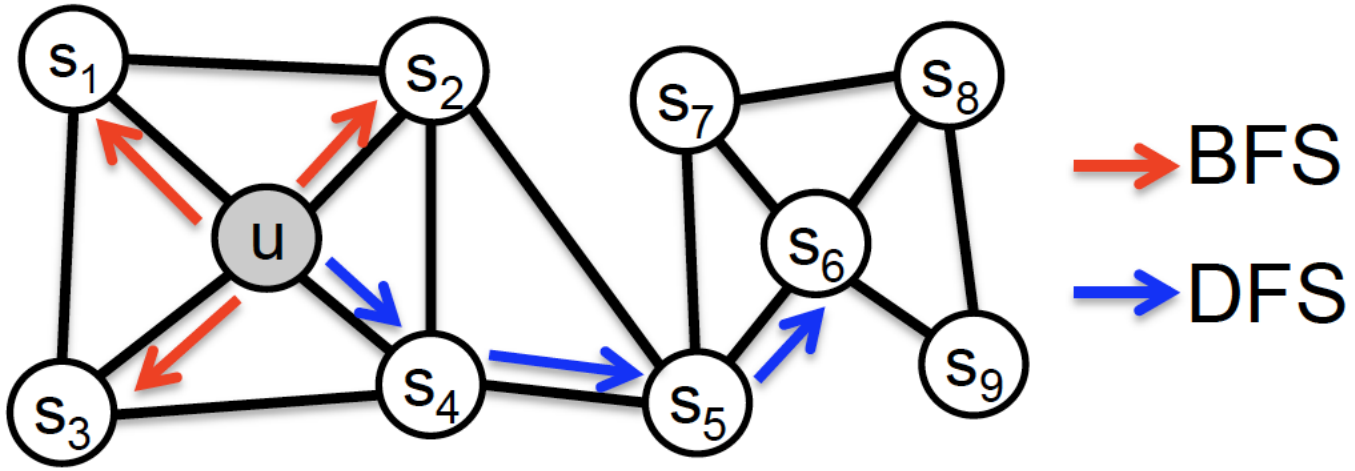
node2vec: Biased Walks

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



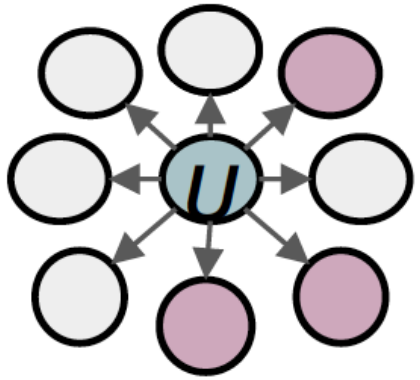
node2vec: Biased Walks

- Two classic strategies to define a neighborhood $N_R(u)$ of a node u.

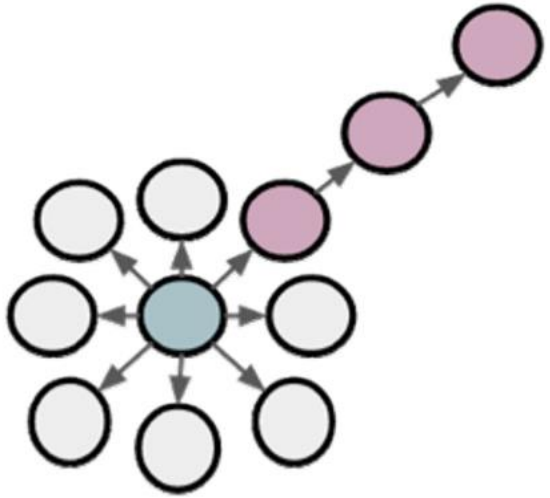


- Walk of length 3 ($N_R(u)$ of size 3)
  - $N_{BFS}(u) = \{s_1, s_2, s_3\}$: Local microscopic view
  - $N_{DFS}(u) = \{s_4, s_5, s_6\}$: Global macroscopic view

BFS versus DFS



BFS: $N_R(\cdot)$ will provide a micro-view of neighborhood



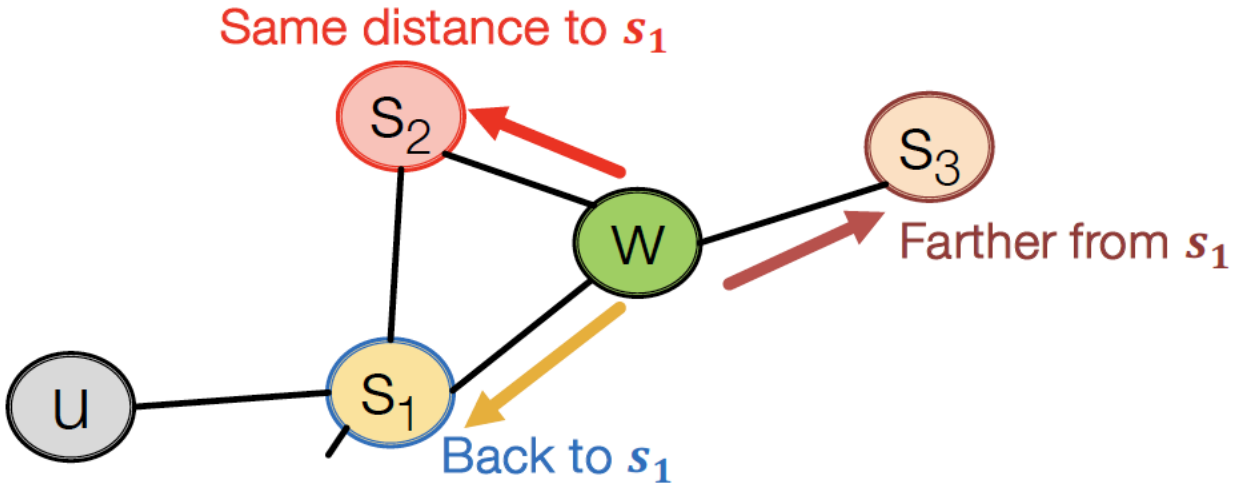
DFS: $N_R(\cdot)$ will provide a macro-view of neighborhood

Interpolating BFS and DFS

- Biased fixed-length random walk R that generates neighborhood $N_R(u)$ for a given node u .
- Random walk has two parameters:
  - Return parameter p:
    - Return to previous node
  - In-out parameter q
    - Moving outwards (DFS) vs inwards (BFS) from previous node
    - Intuitively, q is the ratio of BFS vs DFS
- Next, we specify how a single step of biased random walk is performed
- Random walk is a sequence of these steps

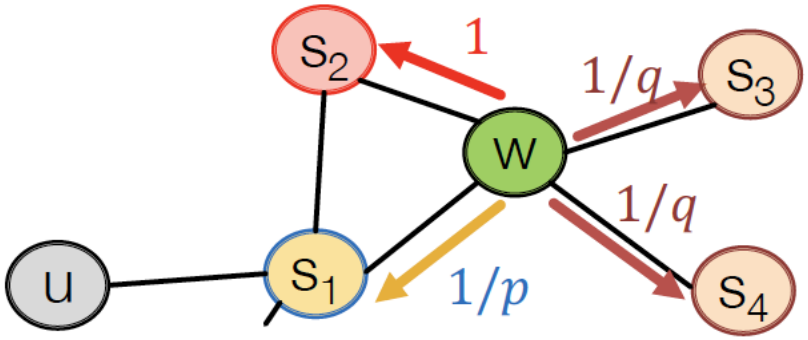
One Step of Biased Random Walk

- Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node w
  - Random walk just traversed edge (s_1, w) and is now at w.
  - We specify edge transition probabilities out of node w
  - Insight: neighbors of w can only be:



One Step of Biased Random Walk

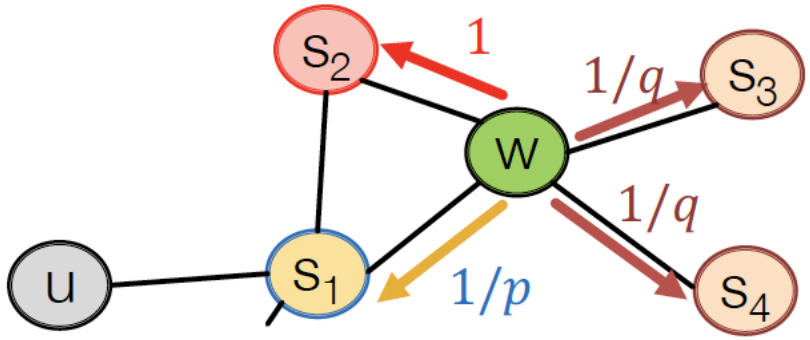
- Walker came over edge (s_1, w) and is now at w. How to set edge transition probabilities?



- p, q model transition probabilities:
  - p: return parameter
  - q: walk-away parameter

One Step of Biased Random Walk

- Walker came over edge (s_1, w) and is now at w. How to set edge transition probabilities?



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

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

node2vec Algorithm

1. Compute edge transition probabilities
   - For each edge (s_1, w) we compute edge walk probabilities (based on p and q) of edges $(w, \cdot)$
2. Simulate r random walks of length l starting from each node u
3. Optimize 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 learnt weights (Abu-El-Haija et al., 2017)
- Alternative optimization schemes
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (LINE from Tang et al. 2015)
- Network preprocessing techniques
  - Random walks on modified versions of original graph (Ribeiro et al. 2017's struct2vec, Chen et al. 2016's HARP)

Summary

- Core idea: embed nodes so that distances in embedding space reflect node similarities in the original graph

- Different notions of node similarity
  - Naïve: similar if two nodes are connected
  - Random walk approaches

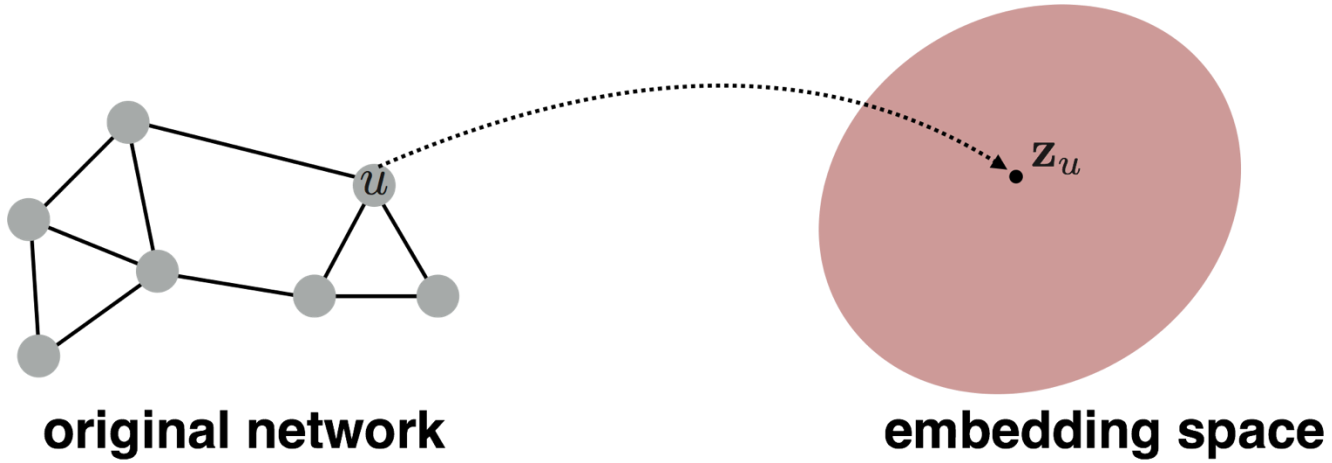
Summary

- No 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

Embedding Entire Graphs

Embedding Entire Graphs

- Goal: want to embed a subgraph or an entire graph G. Graph embedding: z_G



- Tasks:
  - Classifying toxic versus non-toxic molecules
  - Identifying anomalous graphs

Approach 1

Simple but effective approach

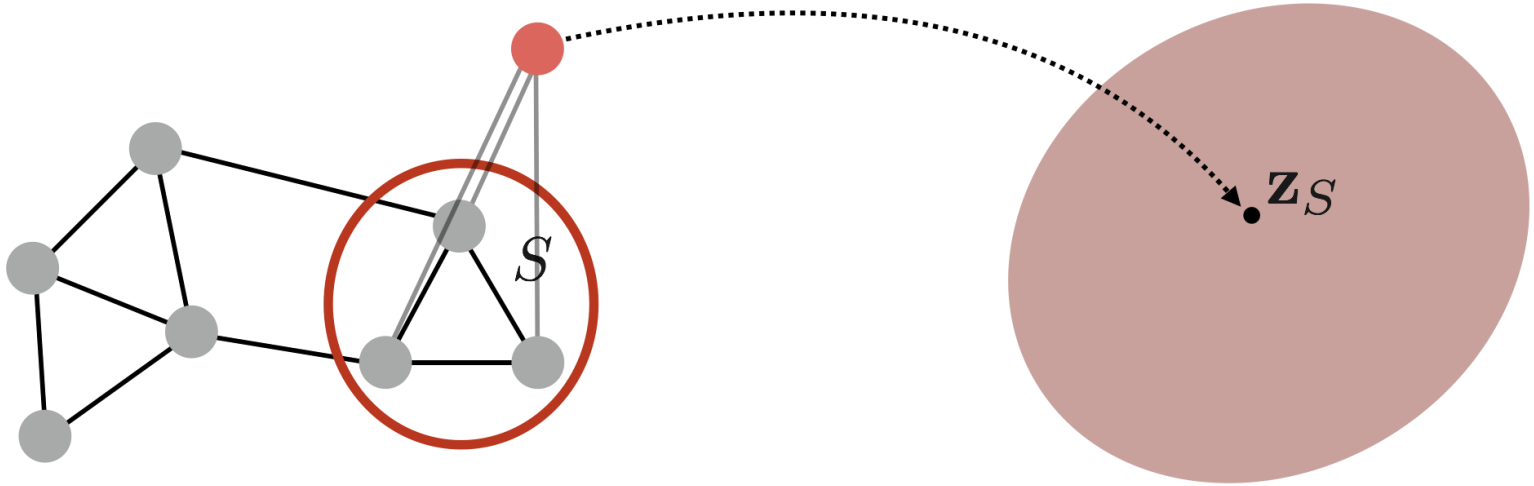
- Run a standard node embedding technique on a (sub)-graph G
- Then just sum (or average) the node embeddings in the sub-graph G:

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

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

Approach 2

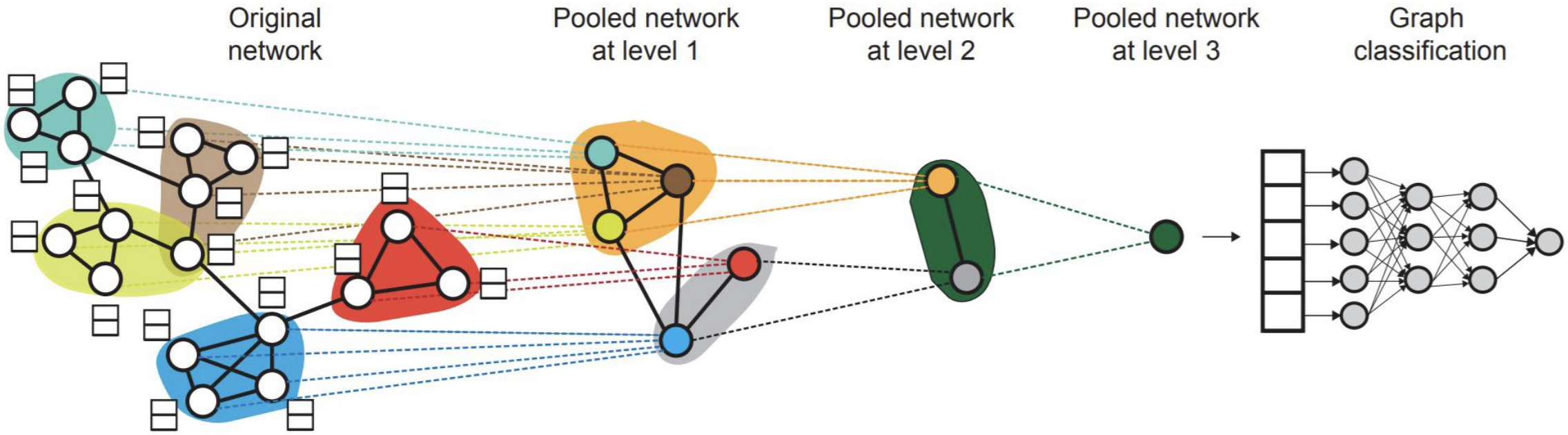
- Introduce a virtual node to represent the sub-graph and run the standard node embedding technique



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

Preview: Hierarchical Embeddings

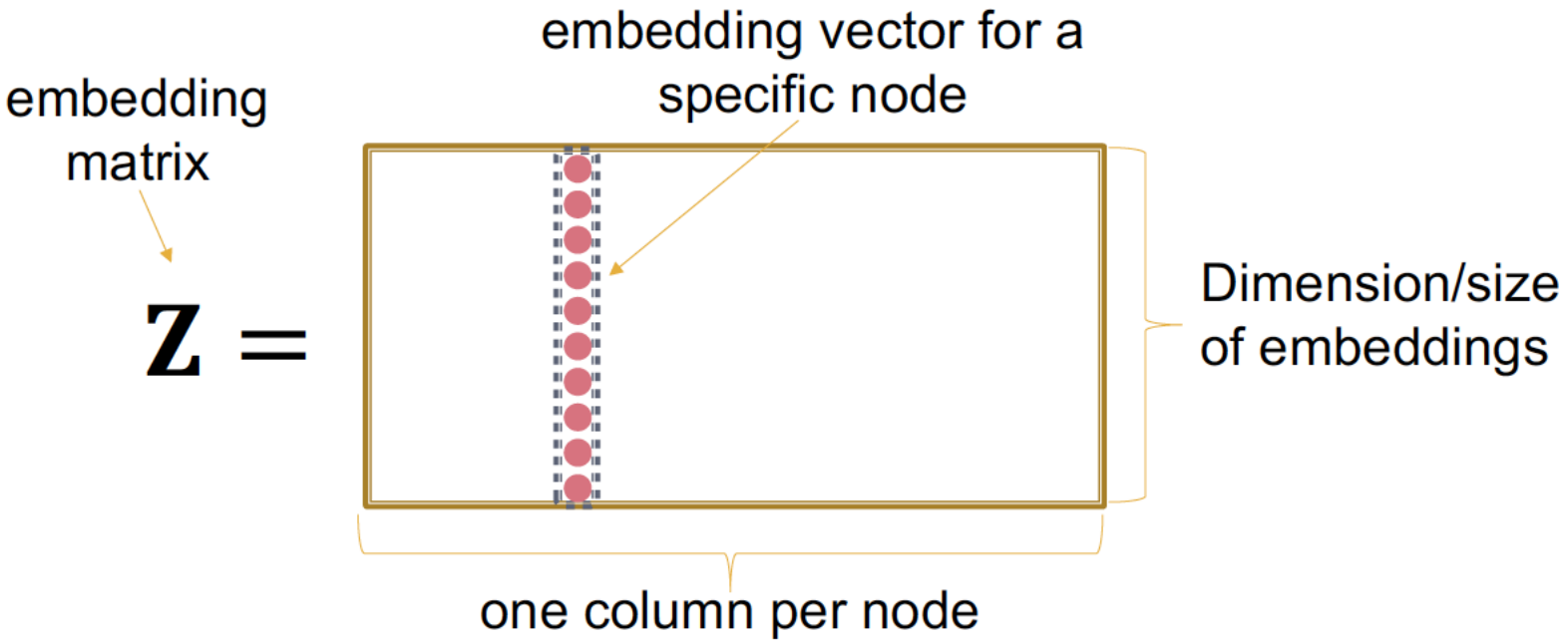
- DiffPool: we can also hierarchically cluster nodes in graphs, and sum/avg the node embeddings according to these clusters



Matrix Factorization and Node Embeddings

Embeddings

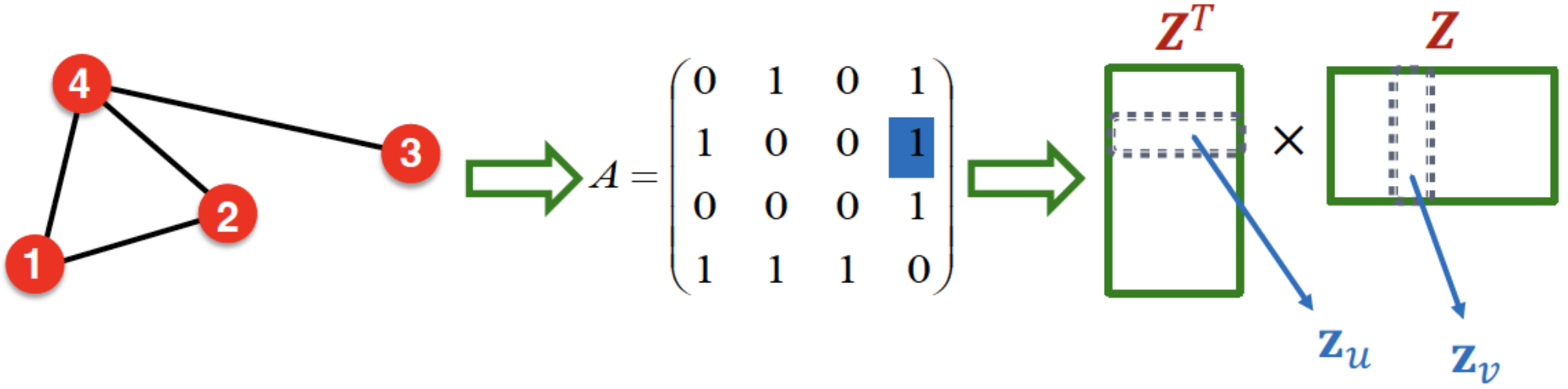
- Recall: encoder as an embedding lookup



- Objective: maximize $z_v^T z_u$ for node pairs (u, v) that are similar

Connection to Matrix Factorization

- Simplest node similarity: node u, v are similar if they are connected by an edge.
- This means: $z_v^T z_u = A_{u,v}$ which is the (u, v) entry of the adjacency matrix A
- Therefore, $Z^T Z = A$



Matrix Factorization

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

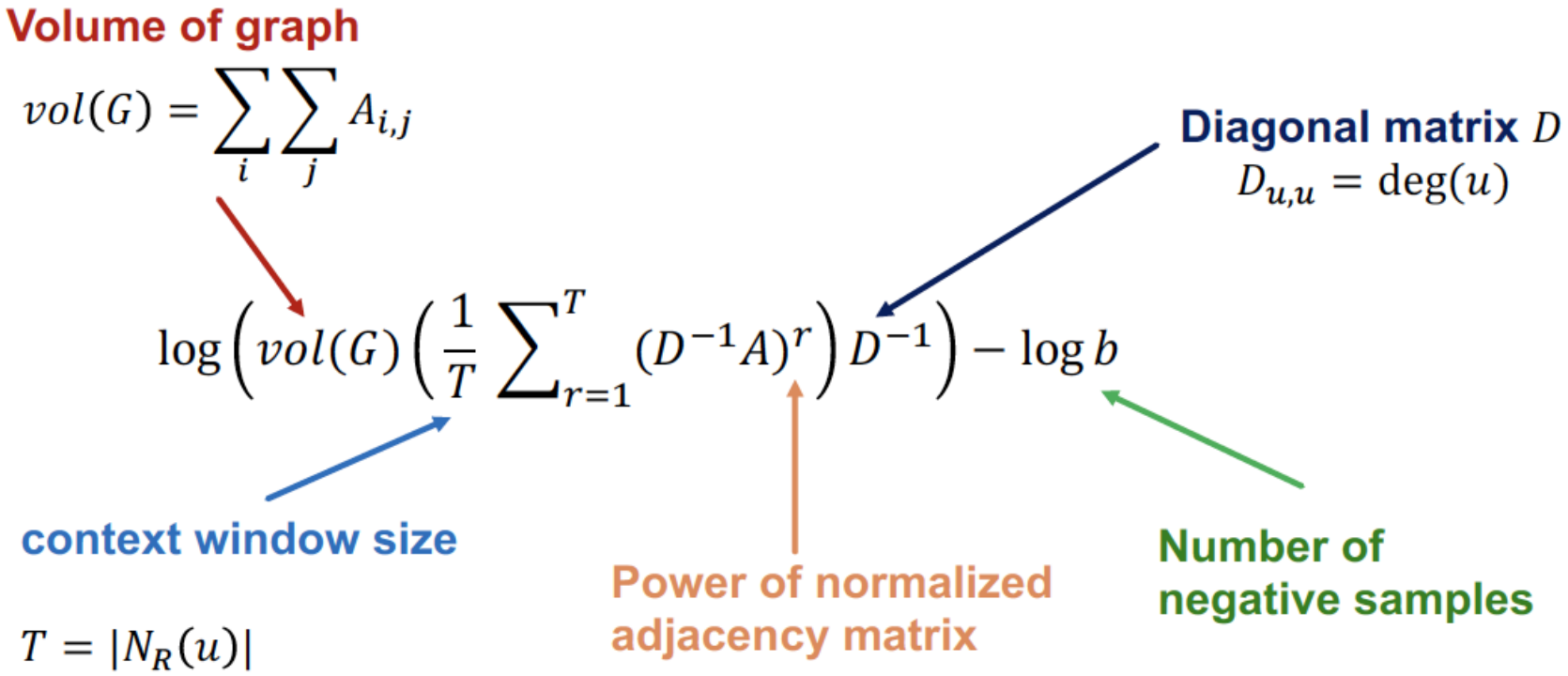
Random Walk-Biased Similarity

- 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:

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

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

Random Walk-Biased Similarity



- Node2vec can also be formulated as a matrix factorization (albeit a more complex matrix)

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

How to Use Embeddings

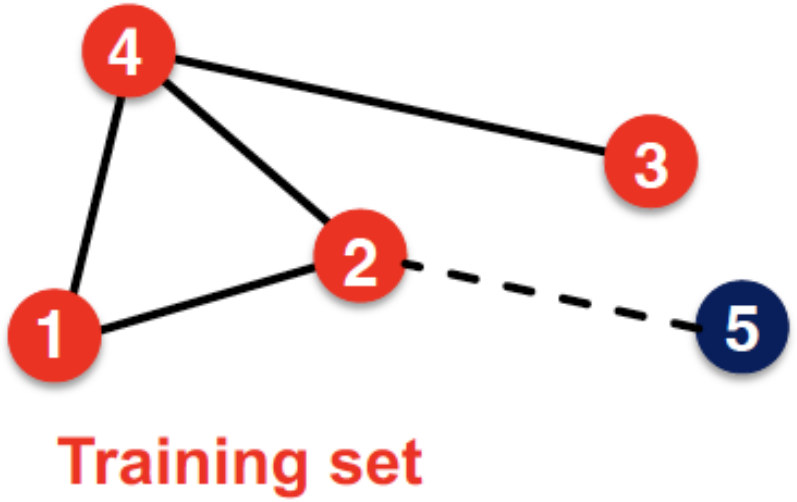
- Clustering/community detection
- Node classification
- Link prediction: predict edge (i, j) based on (z_i, z_j)
  - We can concatenate, avg, product or subtract between the embeddings
    - Concatenate: $f(z_i, z_j) = g([z_i, z_j])$
    - Hadamard: $f(z_i, z_j) = g(z_i * z_j)$
    - Sum/avg: $f(z_i, z_j) = g(z_i + z_j)$
    - Distance: $f(z_i, z_j) = g\left(\|z_i - z_j\|_2\right)$
- Graph classification: predict labels based on graph embedding z_G via aggregating node embeddings or virtual node

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
  - Example: DeepWalk, node2vec
- Extension to graph embedding:
  - Node embedding aggregation

Limitations

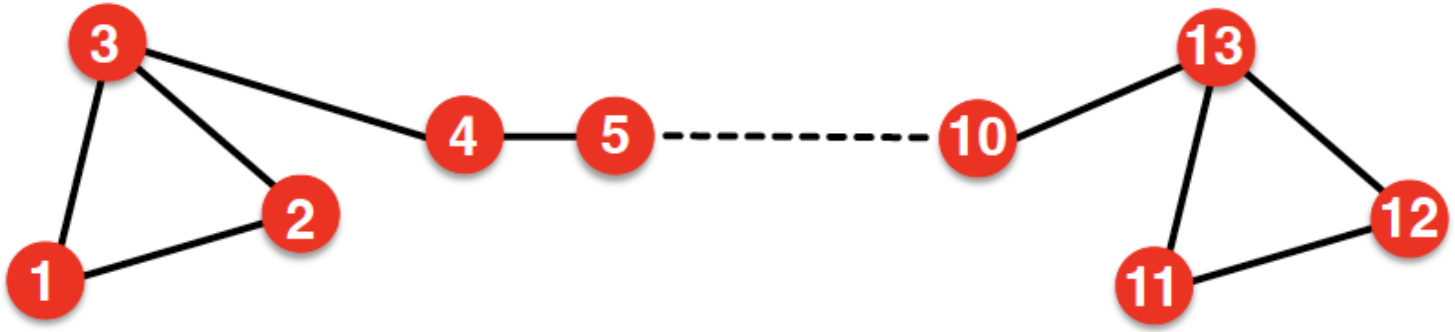
- Transductive (not inductive) method: cannot obtain embeddings for node not in the training set. Cannot apply to new graphs, evolving graphs



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

Limitations

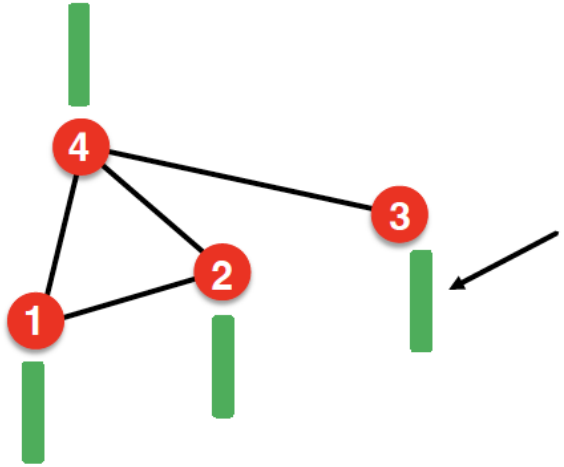
- Cannot capture structural similarity



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

Limitations

- Cannot utilize node, edge, and graph features



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

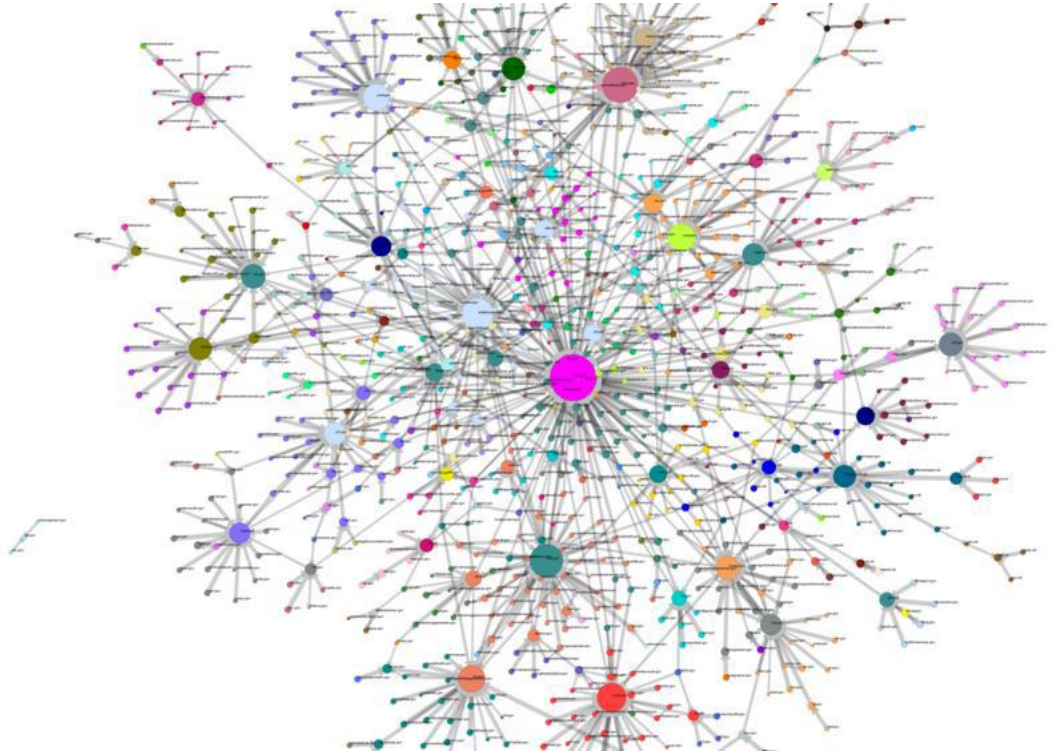
DeepWalk/node2vec embeddings do not incorporate node features

- Solution: Deep Representation Learning with Graph Neural Networks

Link Analysis and PageRank (Google Algorithm)

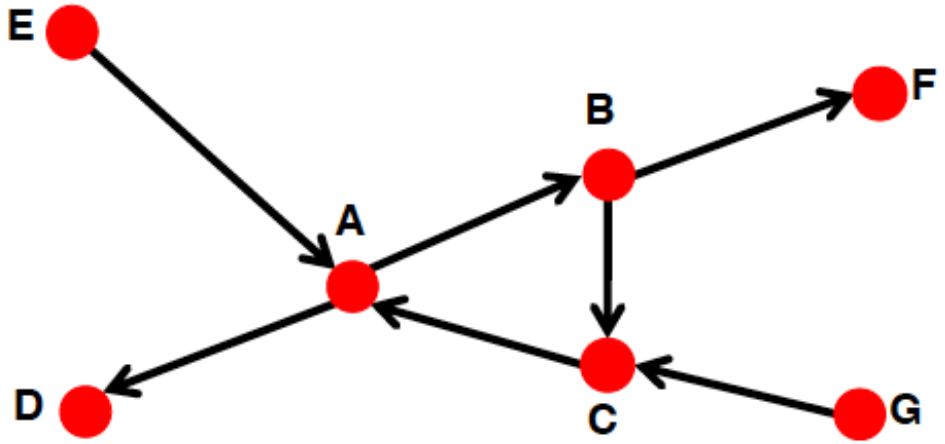
Example: The Web as a Graph

- 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



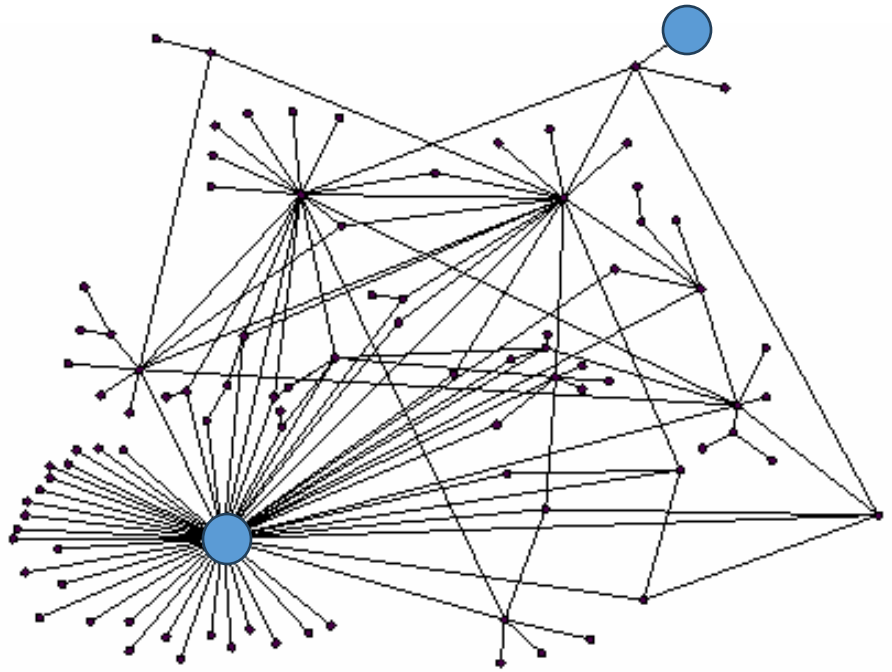
What does The Web Look Like?

- How is the Web linked?
- What is the map of the Web?
- Web as a directed graph:
  - Given a node v, what nodes can v reach?
  - What other nodes can reach v?



Ranking Nodes on the Graph

- All webpages are not equally important
- There is a large diversity in the web-graph node connectivity
- Goal: rank the pages using the web graph link structure



Link Analysis Algorithms

- Compute the importance of nodes in a graph
  - PageRank
  - Personalized PageRank (PPR)
  - Random Walk with Restarts

Links as Votes

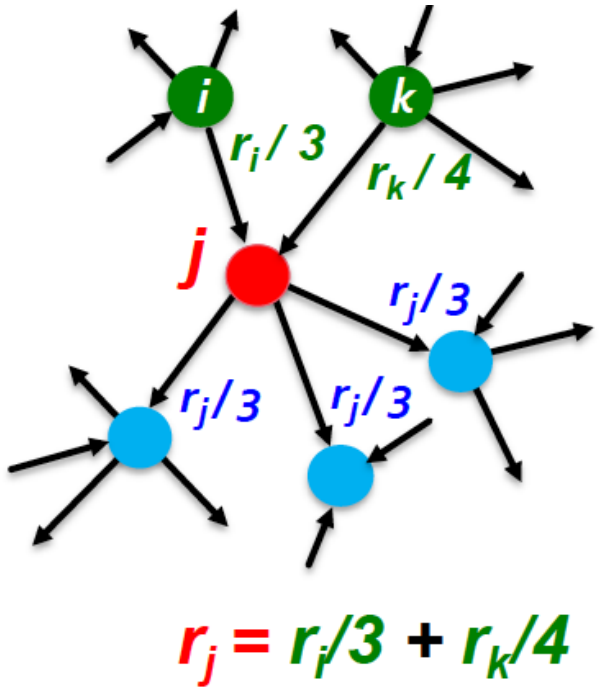
- Ideas: Links as votes
  - Page is more important if it has more links
  - In-coming links? Out-coming links?

- Think of in-links as votes

- 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 out-link gets $\frac{r_i}{d_i}$ votes
  - Page j's own importance r_j is the sum of votes on its in-links



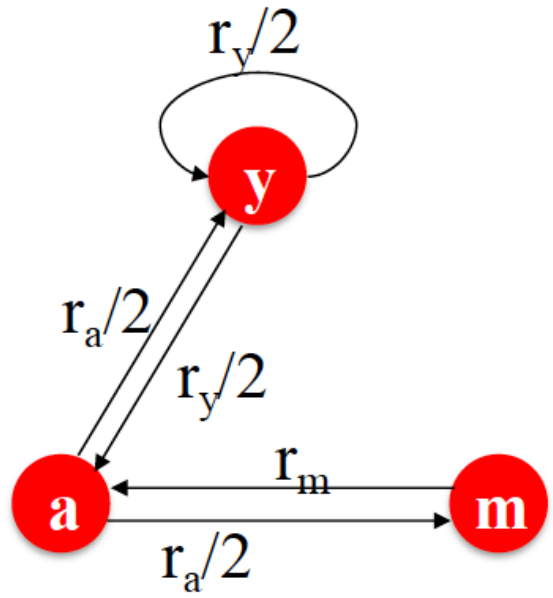
PageRank: The "Flow" Model

- A page is important if it is pointed to by other important pages

- Define rank r_j for node j:

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

d_i: out-degree of node i



"Flow" equations:

$r_y = r_y/2 + r_a/2$

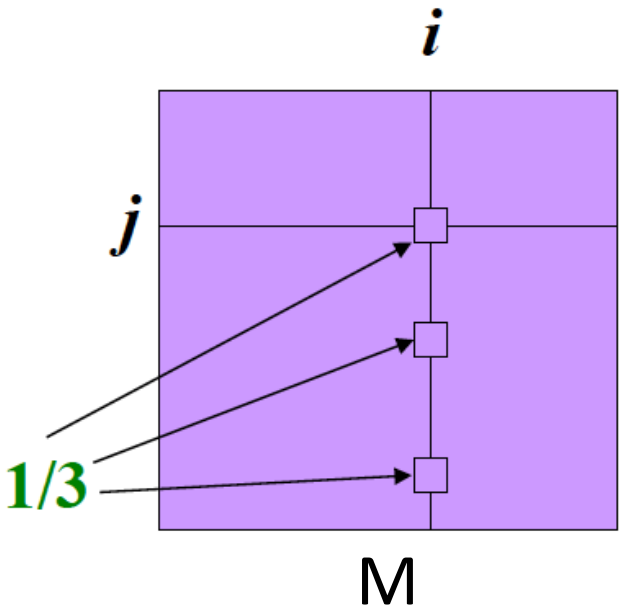
$r_a = r_y/2 + r_m$

$r_m = r_a/2$

PageRank: Matrix Formulation

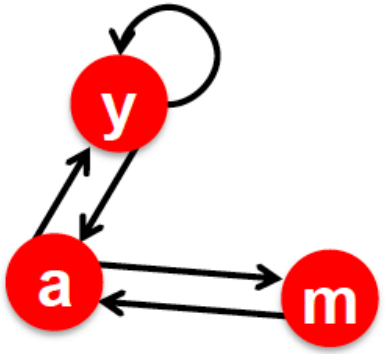
- Stochastic adjacency matrix M
  - d_i: out-degree of node i
  - If $i \to j$, then $M_{ij} = \frac{1}{d_i}$
  - M is column stochastic matrix
    - Columns sum to 1.
- Rank vector r: an entry per page
  - r_i: the importance score of page i
  - $\sum_i r_i = 1$
- Flow equations:

$$r = M \cdot r$$



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

Example: Flow Equations and M



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

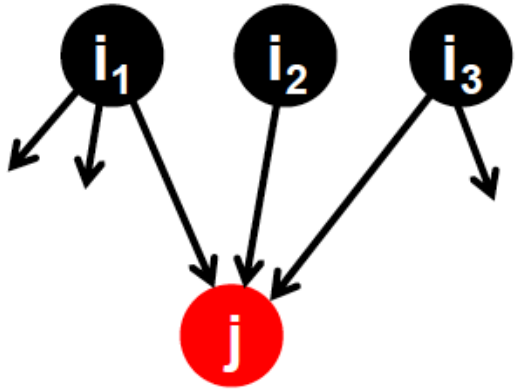
$$r_y = r_y/2 + r_a/2$$
$$r_a = r_y/2 + r_m$$
$$r_m = r_a/2$$

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

r M r

Connection to Random Walk

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



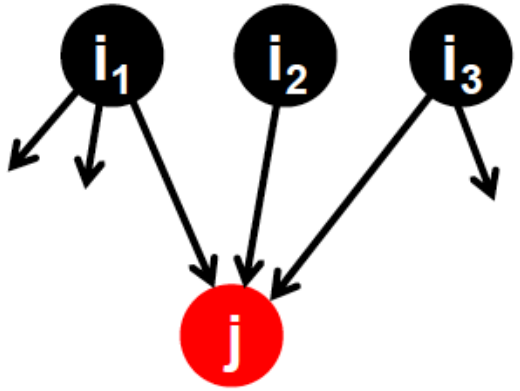
- Let:
  - p(t): a vector --- ith coordinate is prob. the suffer is at page i at time t
  - Essentially, p(t) is the probability distribution over pages

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

Connection to Random Walk: Stationary Distribution

- Where is suffer at time t+1?

$$p(t+1) = M \cdot p(t)$$



- Suppose the random walk reaches a state

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

  - Then p(t) is stationary distribution of random walk

- Our original rank vector r satisfies: $r = M \cdot r$
  - It means r is stationary distribution of random walk

PageRank: Summary

- Measure importance of nodes in a graph using the link structure of the web
- Model a random web surfer using stochastic adjacency matrix M
- Solve $r = Mr$ where r can be viewed as both the principal eigenvector of M and as the stationary distribution of a random walk over the graph

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: $\sum_i \left| r_i^{t+1} - r^t\_i \right| < \epsilon$
    - 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 wit N nodes, where nodes are pages and edges are hyperlinks
- Power iteration: a simple iterative scheme
  - Initialize: $r^{(0)} = \left[\frac{1}{N}, \frac{1}{N}, \cdots, \frac{1}{N}\right]^T$
  - Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
  - Stop when: $\left|r^{(t+1)} - r^{(t)}\right|_1 < \epsilon$

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

- Note: About 50 iterations is sufficient to estimate the limiting solution

PageRank: Three Questions

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

Or equivalently, $r = Mr$

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

PageRank: Problems

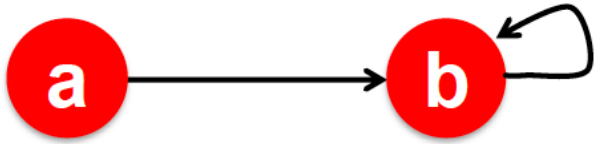
- Two problems:

1. Some pages are dead ends (have no out-links)
   - These 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:



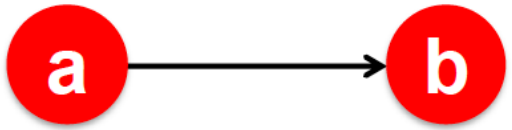
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$$

- PageRank iteration

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:



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

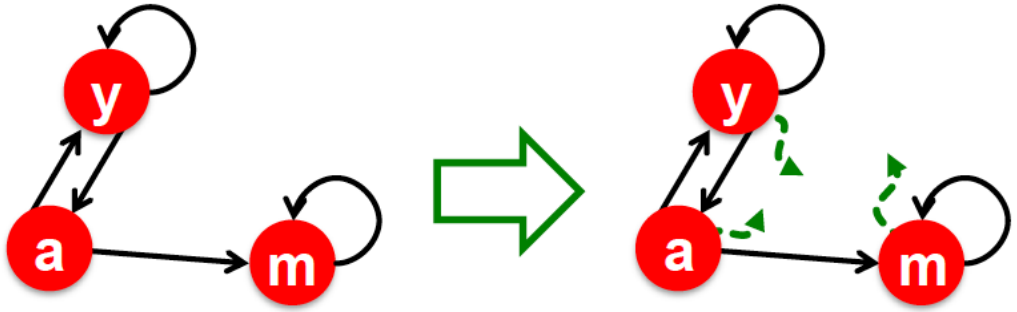
- PageRank iteration

Iteration	0	1	2	3
r_a	1	0	0	0
r_b	0	1	0	0

Solution to Spider Traps

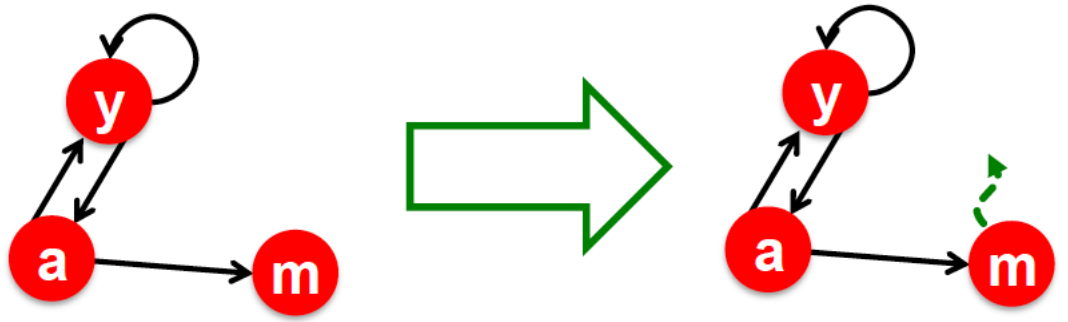
- At each time step, random surfer has two options
  - With prob. β, follow a link at random
  - With prob. $1 - \beta$, jump to a random page
  - Common values for β: $[0.8, 0.9]$

- Surfer will teleport out of spider trap within a few steps



Solution to Dead Ends

- Teleports: Follow random teleport links with total probability 1.0 from dead-ends
  - Adjust adjacency 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 to go

Solution: Random Teleports

- Google's solution does it all
- At each step, random surfer has two options
  - With probability β, follow a link at random
  - With probability $1 - \beta$, jump to some random page
- PageRank equation:

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

The Google Matrix

- PageRank equation:

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

- The Google matrix

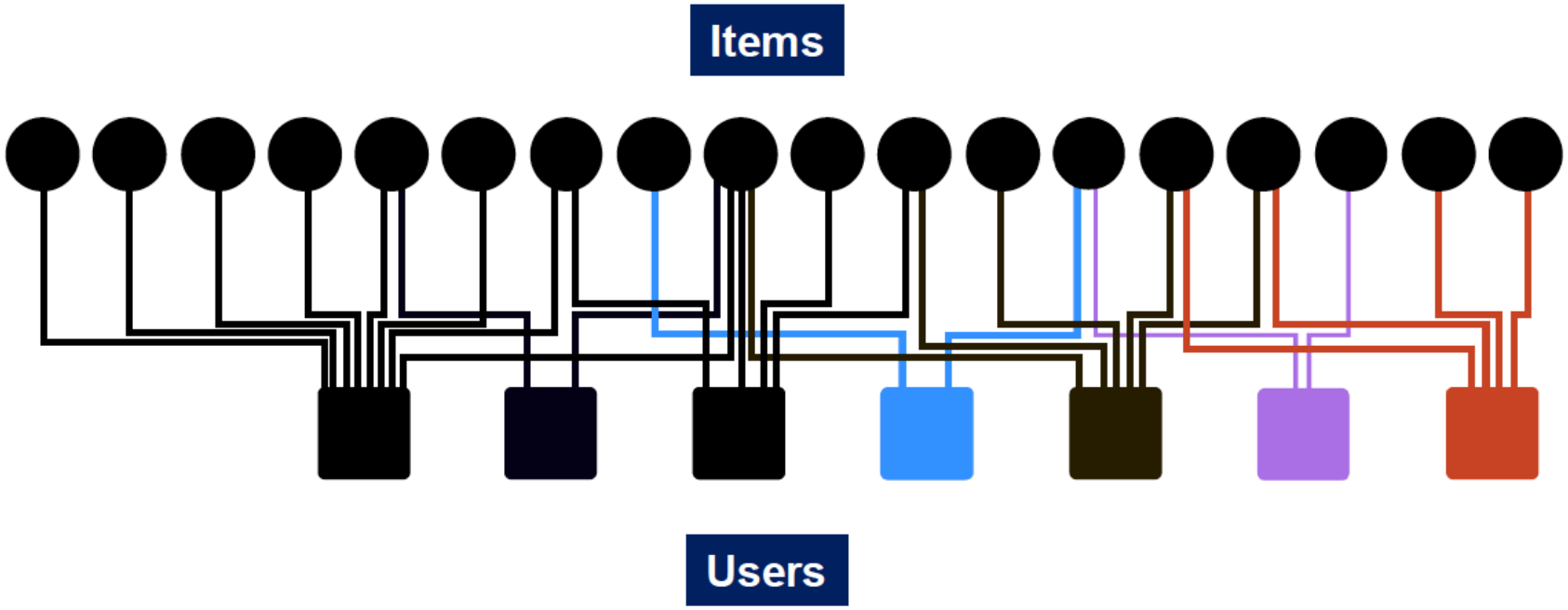
$$G = \beta M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

- We have a recursive problem: $r = Gr$
  - The Power method still works
  - In practice, $\beta \in [0.8, 0.9]$

Random Walk with Restarts and Personalized PageRank

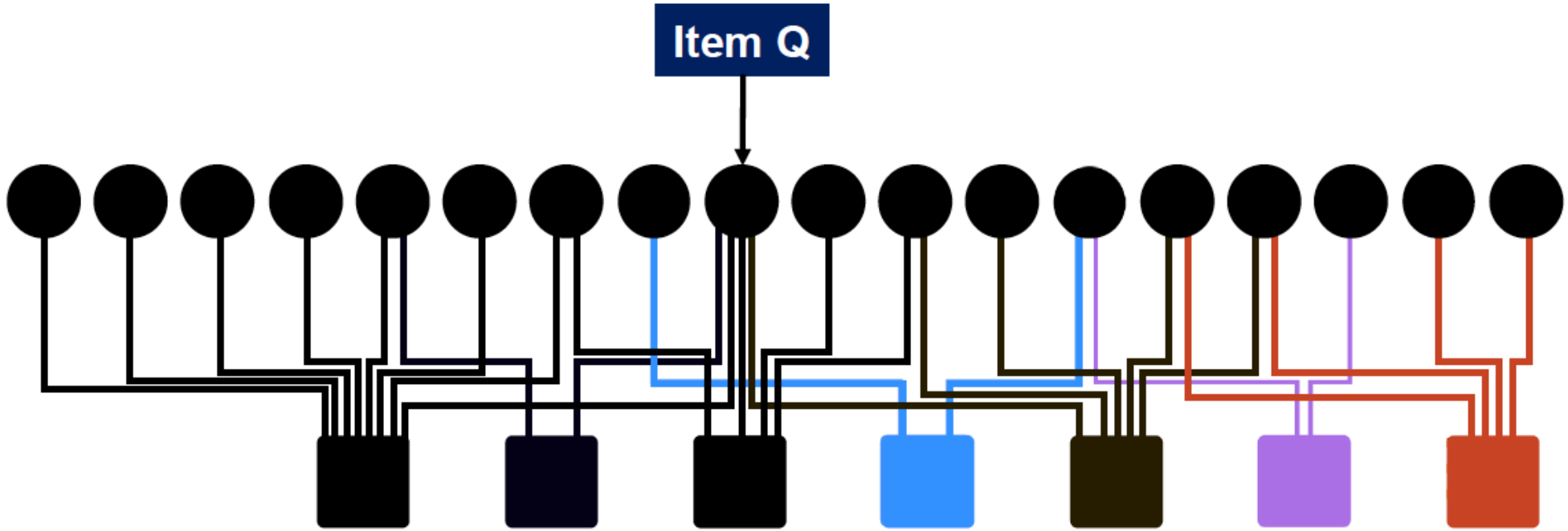
Example: Recommendation

- Given: a bipartite graph representing user and item interactions



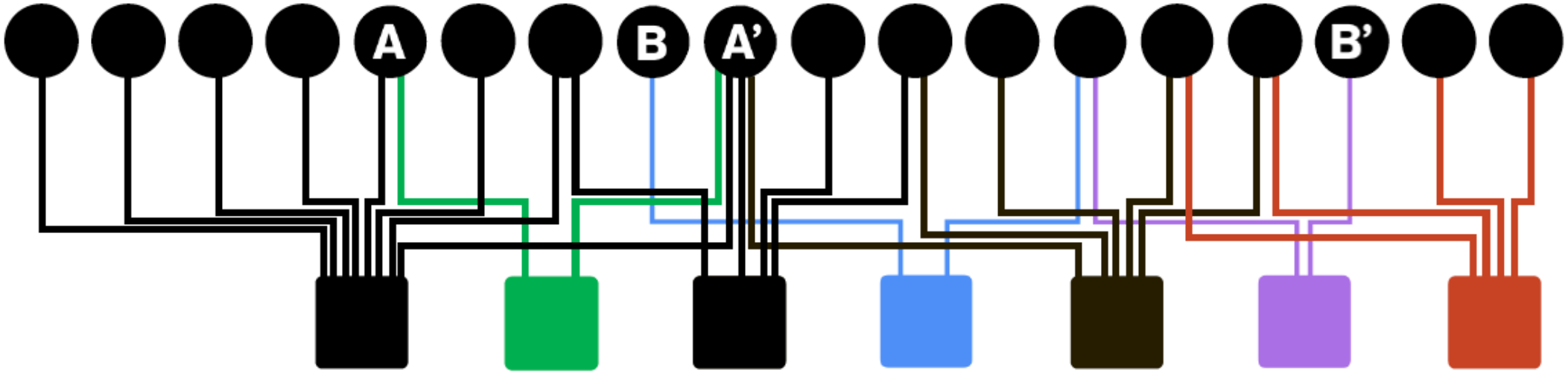
Bipartite User-Item Graph

- 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 to users interacts with Q



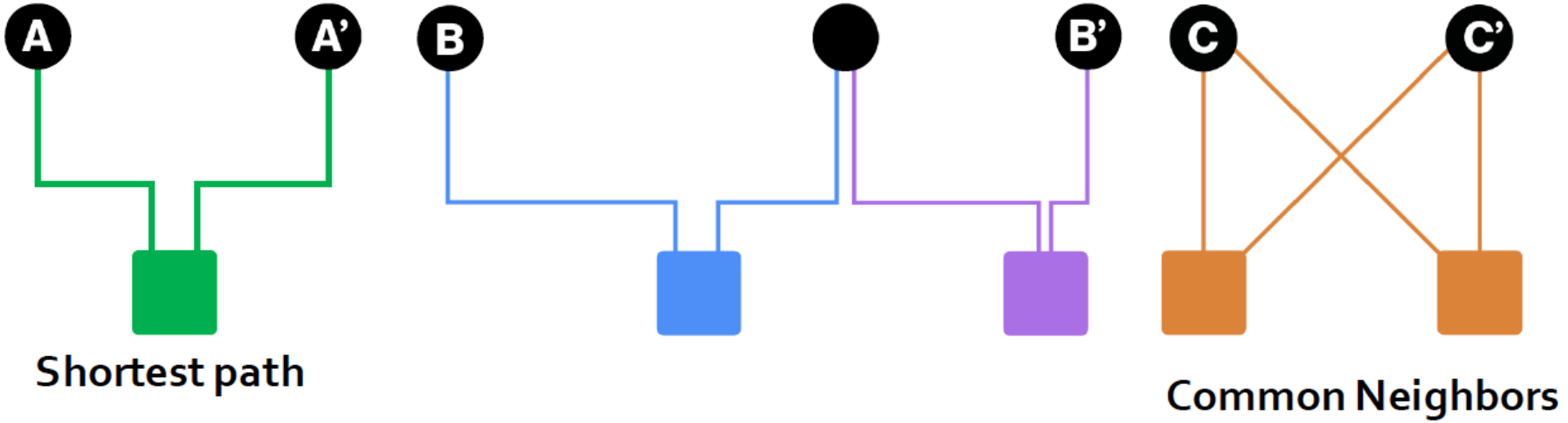
Bipartite User-Item Graph

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



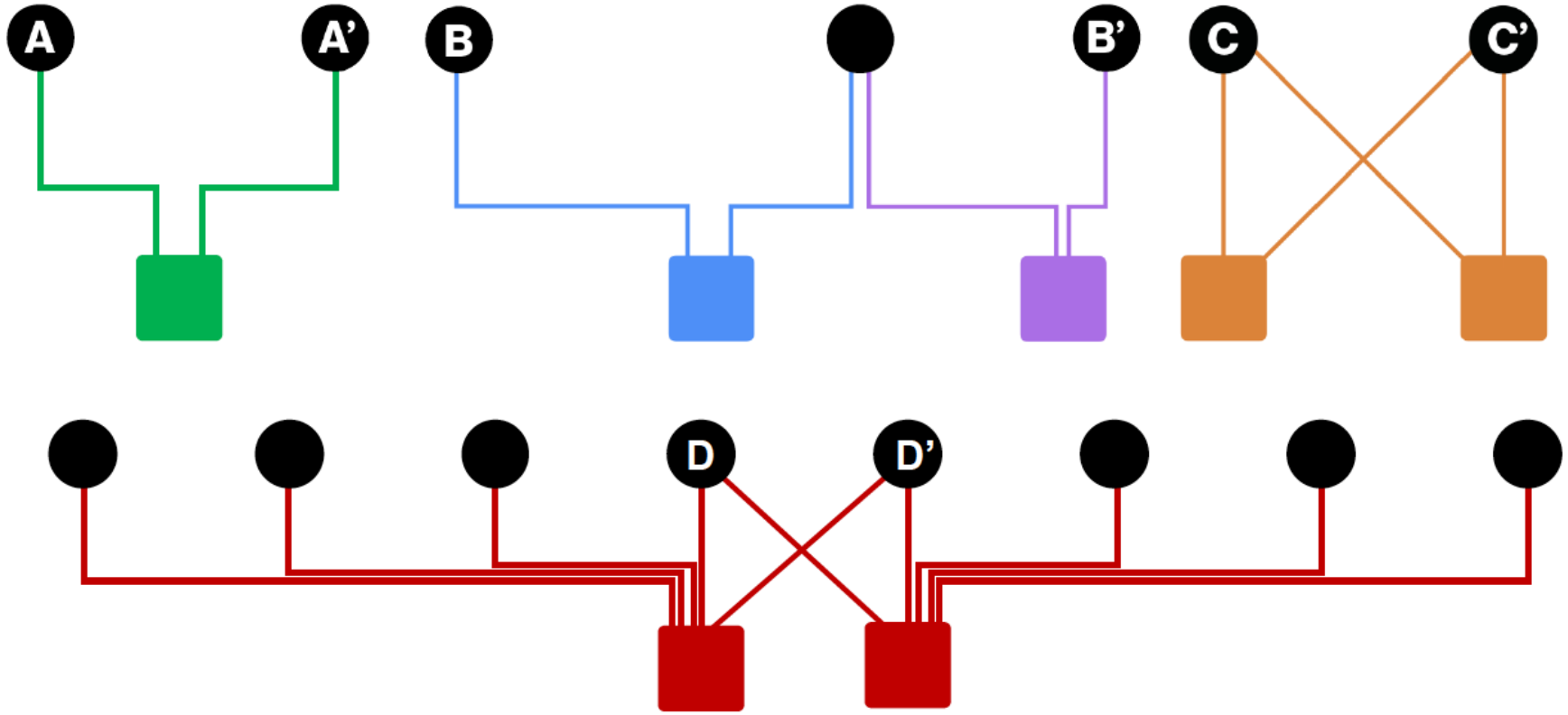
Node Proximity Measurements

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



Node Proximity Measurements

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



Personalized Page Rank/Random Walk with Restarts

Proximity on Graphs

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

Idea: Random Walks

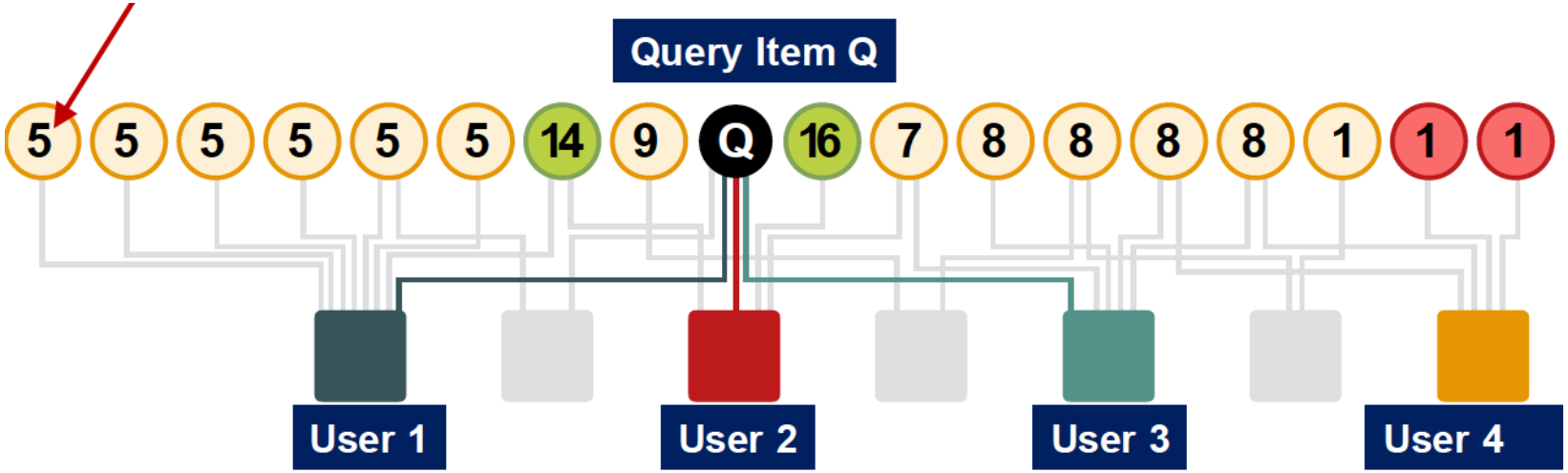
- Idea:
  - Every node has some importance
  - Importance gets evenly split among all edges and pushed to neighbors

- Given a set of QUERY_NODES, we simulate a random walk
  1. Make a step to a random neighbor and record the visit (visit count)
  2. With probability α, restart the walk at one of the query nodes in QUERY_NODES
  3. The nodes with highest visit count have the highest proximity to QUERY_NODES

Random Walk Algorithms

```
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 random visits



Benefits

- Why is this a good solution?

- Because the similarity considers:
  - Multiple connections
  - Multiple paths
  - Direct and undirect connects
  - Degree of the nodes

Summary: PageRank Variants

- PageRank
  - Teleport to any nodes
  - Nodes have the same probability of the suffer landing

$$\boldsymbol{S} = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]$$

- Personalized PageRank
  - Teleport to a specific set of nodes
  - Nodes can have different probabilities of suffer landing

$$\boldsymbol{S} = [0.1, 0, 0, 0.2, 0, 0, 0.5, 0, 0, 0.2]$$

- Random Walk with Restarts
  - 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