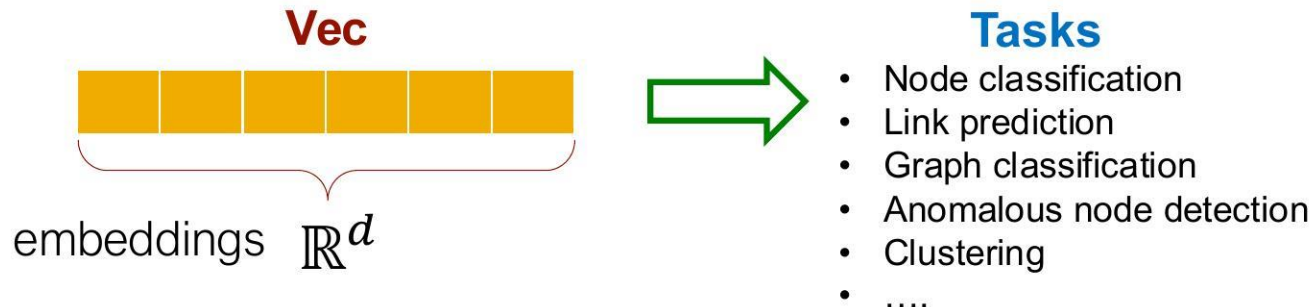


Node Embeddings

Credits to cs224w at Stanford

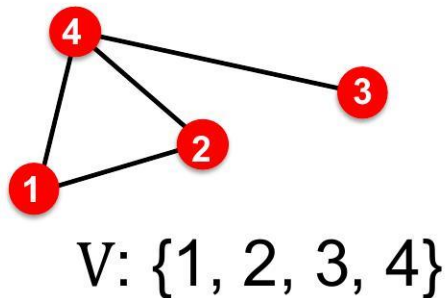
Why Embedding?

- **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



Setup

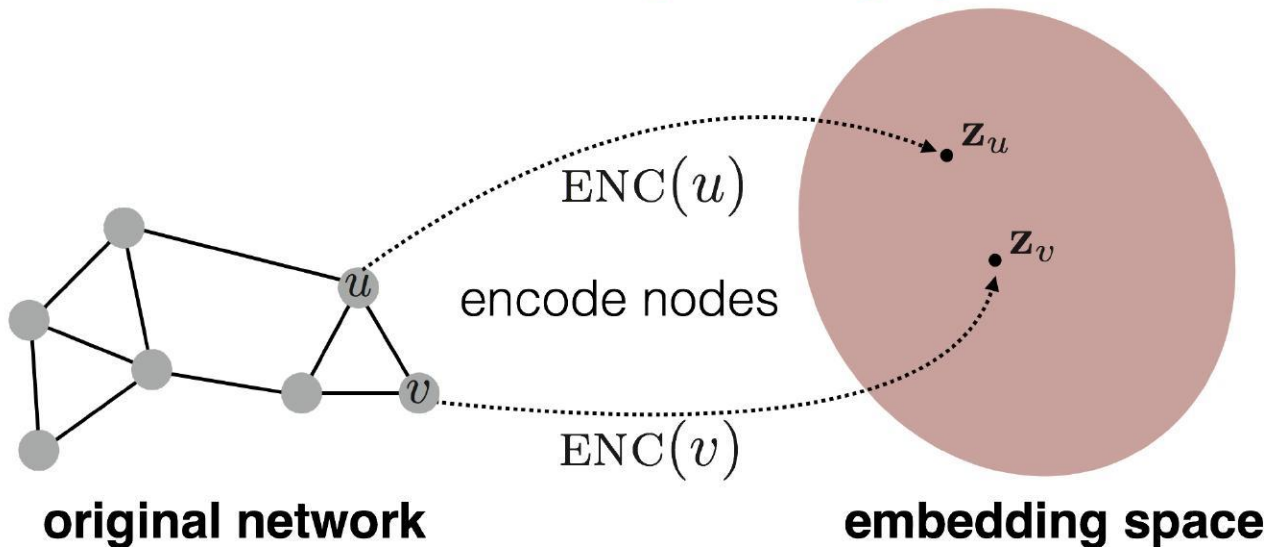
- Assume we have an (undirected) graph G :
 - V is the vertex set.
 - A is the adjacency matrix (assume binary).
 - For simplicity: No node features or extra information is used



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

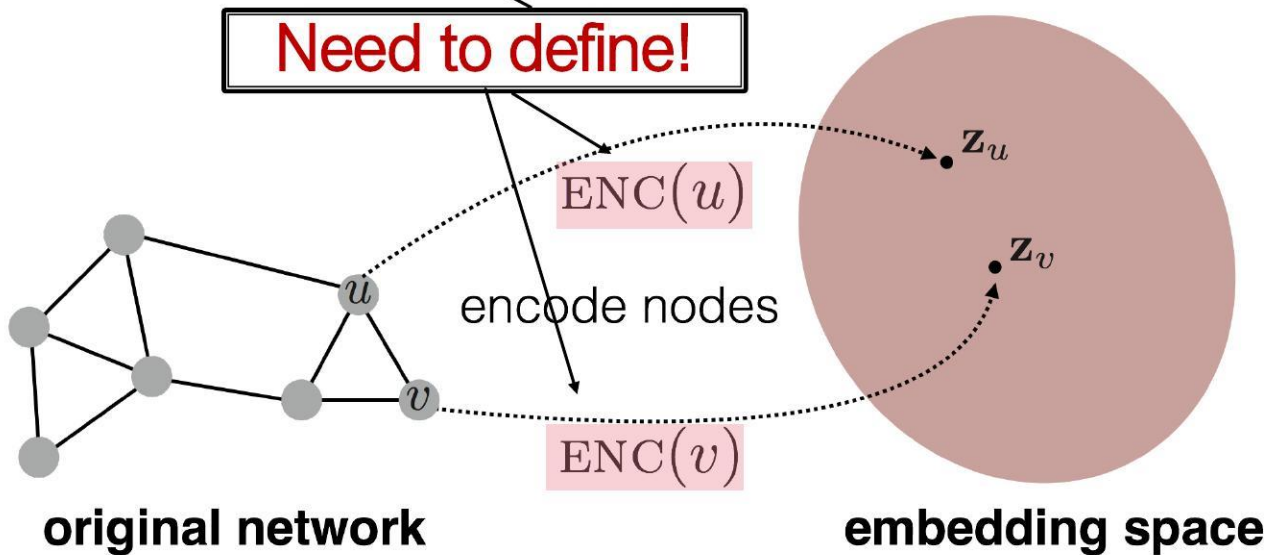
Embedding Nodes

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



Embedding Nodes

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



Learning Node Embeddings

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

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

in the original network

Similarity of the embedding

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

Two Key Components

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

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

node in the input graph

d -dimensional embedding

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

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

Similarity of u and v in the original network

dot product between node embeddings

Decoder

“Shallow” Encoding

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

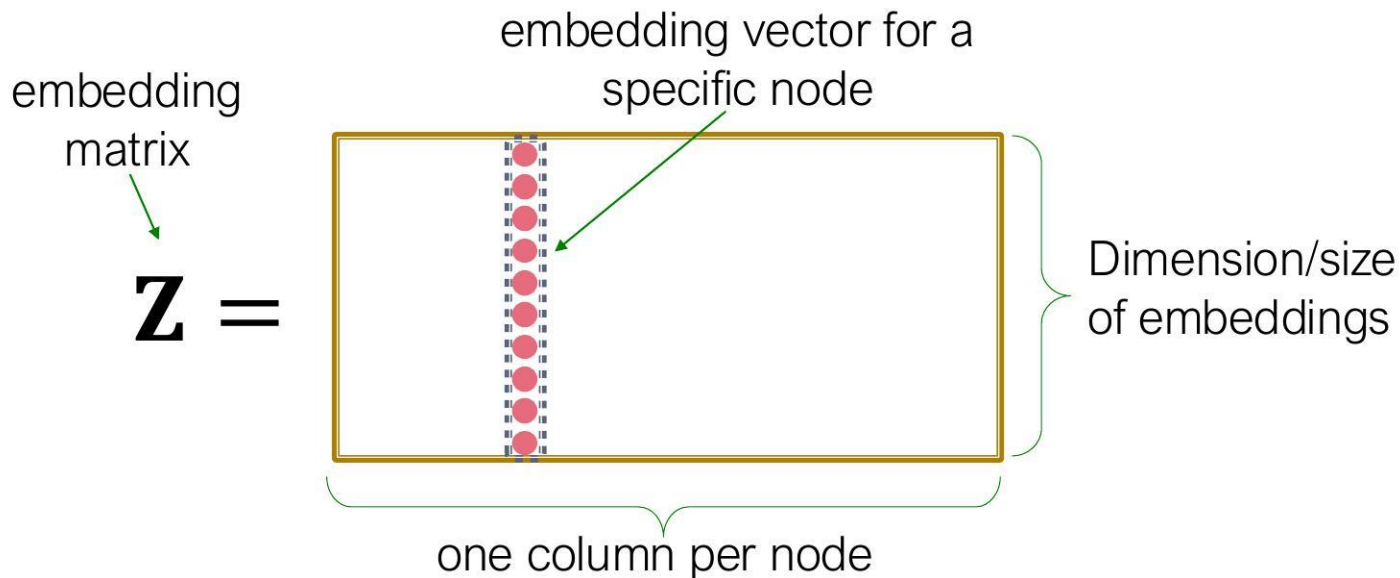
$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

$\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ 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 in the GNNs
- **Decoder:** based on node similarity.
- **Objective:** maximize $\mathbf{z}_v^T \mathbf{z}_u$ 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>



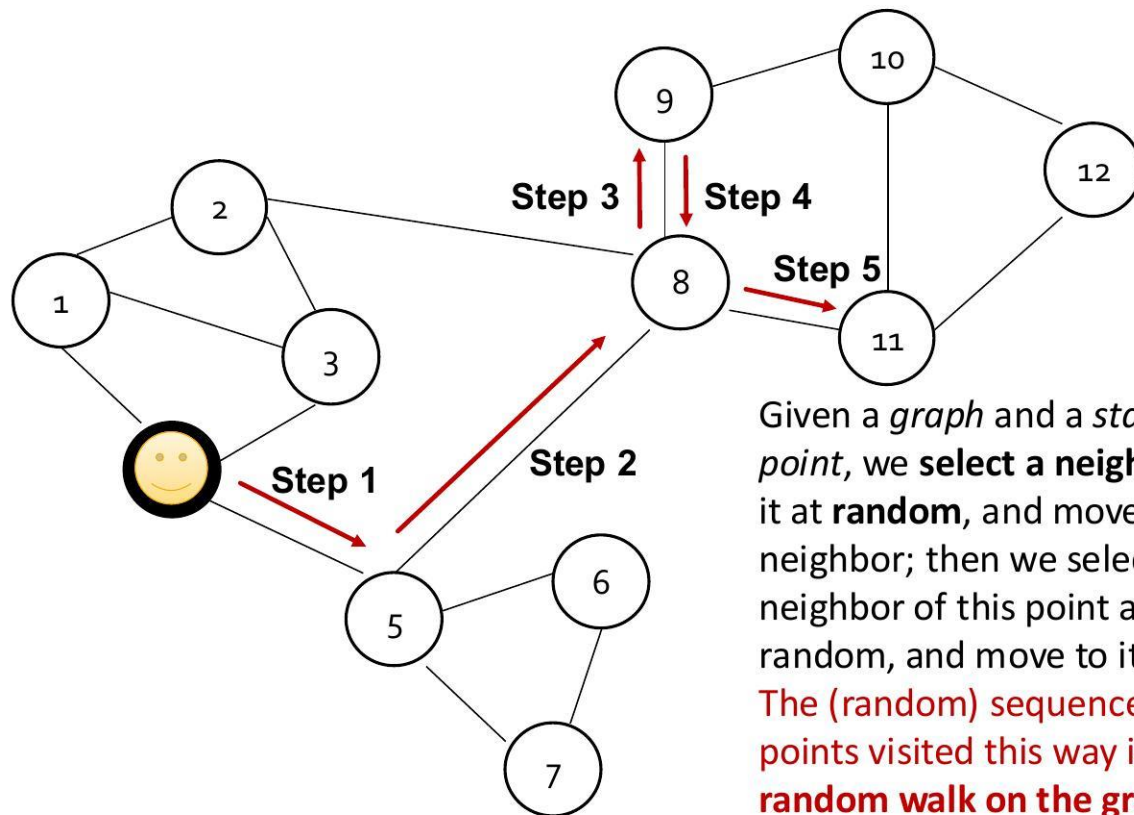
Notation

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

Random Walk



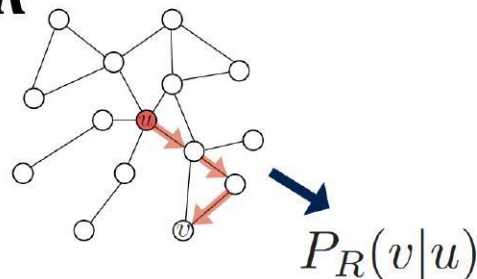
Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The (random) sequence of points visited this way is a **random walk on the graph**.

Random-Walk Embeddings

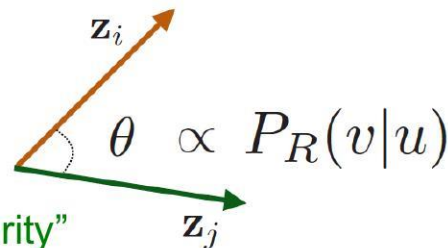
$\mathbf{z}_u^T \mathbf{z}_v \approx$ probability that u
and v co-occur on a
random walk over
the graph

Random-Walk Embeddings

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



2. Optimize embeddings to encode these random walk statistics:



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

Why Random Walks?

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

Unsupervised Feature Learning

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

Feature Learning as Optimization

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

$$f(u) = \mathbf{z}_u$$

- Log-likelihood objective:

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

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

Random Walk Optimization

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

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

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

Random Walk Optimization

Equivalently,

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

- **Intuition:** Optimize embeddings $\mathbf{z}_u$ to **minimize** the negative log-likelihood of random walk neighborhoods $N(u)$.

- **Parameterize $P(v|\mathbf{z}_u)$ using softmax:**

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

Why softmax?

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

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

Random Walk Optimization

Putting it all together:

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

sum over all nodes u

sum over nodes v seen on random walks starting from u

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

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

Random Walk Optimization

But doing this naively is too expensive!

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


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

Random Walk Optimization

But doing this naively is too expensive!

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

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

Negative Sampling

- **Solution: Negative sampling**

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

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

sigmoid function

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

random distribution
over nodes

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

- Negative sampling allows for quick likelihood calculation.

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>

Negative Sampling

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

random distribution
over nodes

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

- Sample k negative nodes n_i 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 sample any node (for efficiency).

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 $\frac{\partial \mathcal{L}}{\partial z_u}$.

η : learning rate

- For all u , make a step in reverse direction of derivative: $z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$.

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: $\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v|z_u))$
 - 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 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 Z 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.](#))
 - The issue is that such notion of similarity is too constrained
- **How can we generalize this?**

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

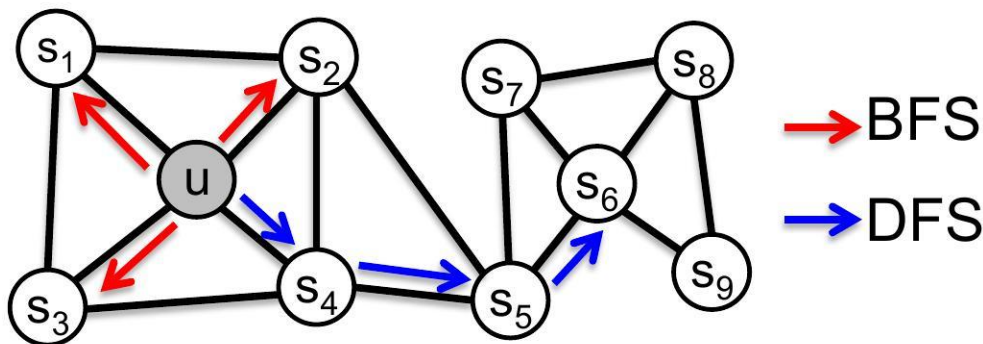
Overview of node2vec

- **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. [node2vec: Scalable Feature Learning for Networks](#). *KDD*.

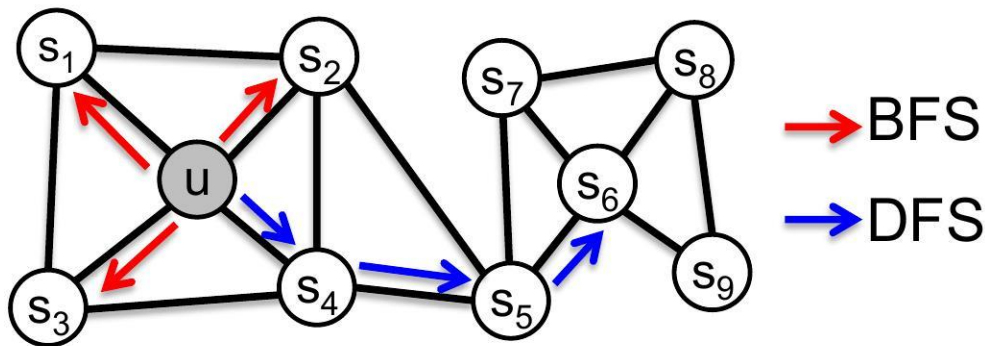
node2vec: Biased Walks

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



node2vec: Biased Walks

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

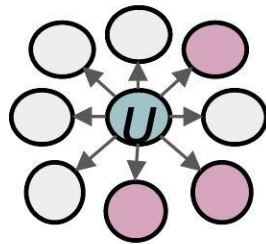


Walk of length 3 ($N_R(u)$ of size 3):

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

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

BFS vs. DFS



BFS:

$N_R(\cdot)$ will provide a
micro-view of
neighbourhood



DFS:

$N_R(\cdot)$ will provide a
macro-view of
neighbourhood

Interpolating BFS and DFS

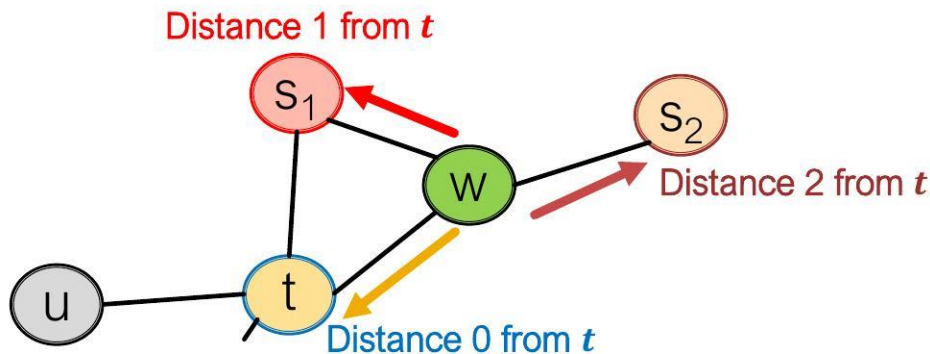
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- Random walk has two parameters:
 - **Return parameter p :**
 - Return back to the previous node
 - **In-out parameter q :**
 - Moving outwards (DFS) vs. inwards (BFS) from the 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 then just a sequence of these steps.

One Step of the Biased Random Walk

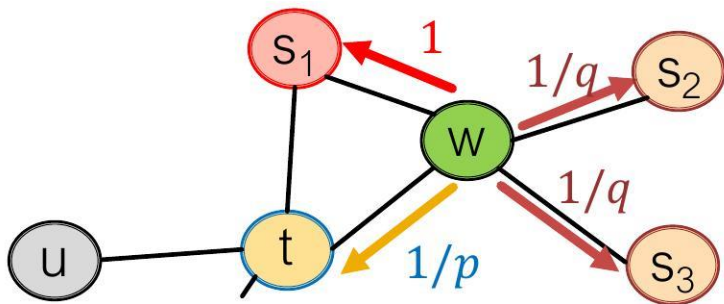
Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node w :

- Rnd. walk **just traversed edge** (t, w) and is **now at** w
- We specify edge transition probs. out of node w
- **Insight:** Neighbors of w can only be:



One Step of the Biased Random Walk

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



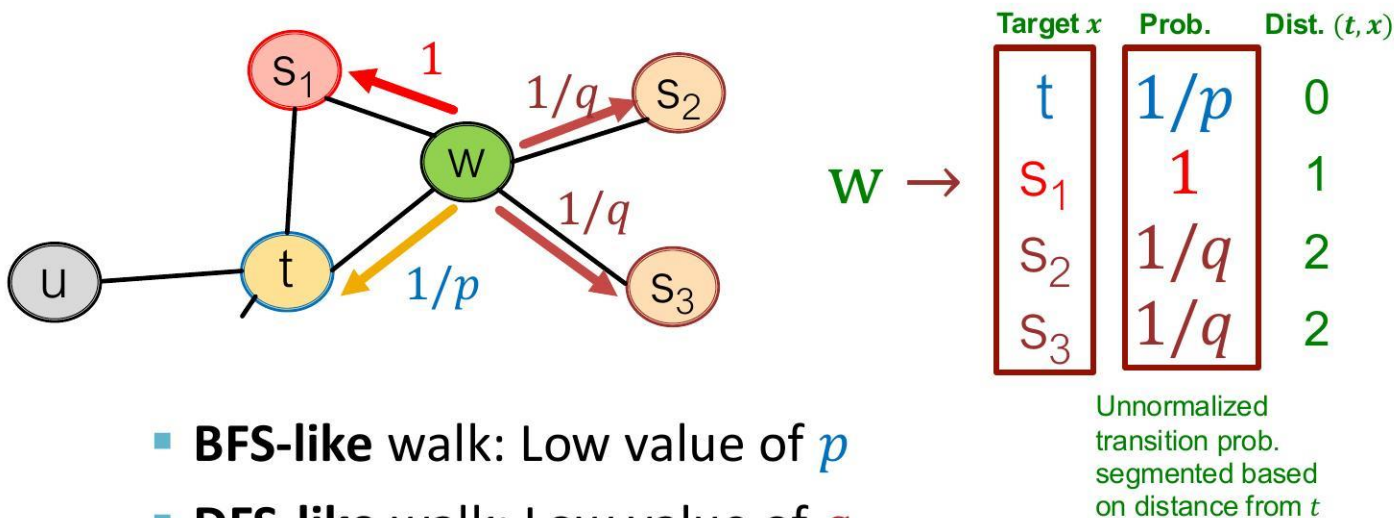
$1/p, 1/q, 1$ are
unnormalized
probabilities

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

One Step of the Biased Random Walk

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

How to set edge transition probabilities?



- 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 edge transition probabilities:
 - For each edge (t, w) we compute edge walk probabilities (based on p, q) of edges $(w, \cdot)$
- 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.](#)).
 - Based on learned weights ([Abu-El-Haija et al.](#))
- **Alternative optimization schemes:**
 - Directly optimize based on 1-hop and 2-hop random walk probabilities (as in [LINE from Tang et al.](#)).
- **Network preprocessing techniques:**
 - Run random walks on modified versions of the original network (e.g., [Ribeiro et al. struct2vec](#), [Chen et al. 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 two nodes are connected
 - 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, survey](#)).
- Random walk approaches are generally more efficient.
- **In general:** Must choose definition of node similarity that matches your application.

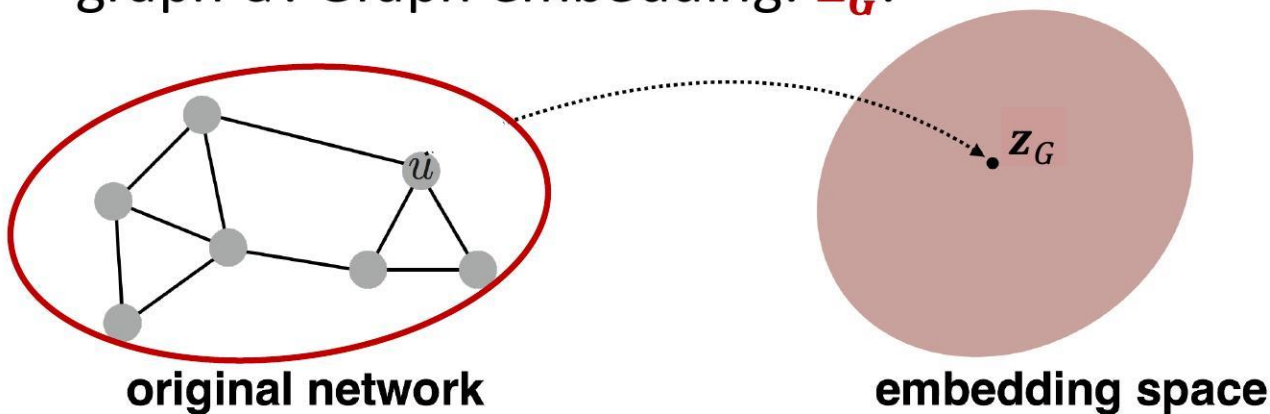
Stanford CS224W: Embedding Entire Graphs

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:

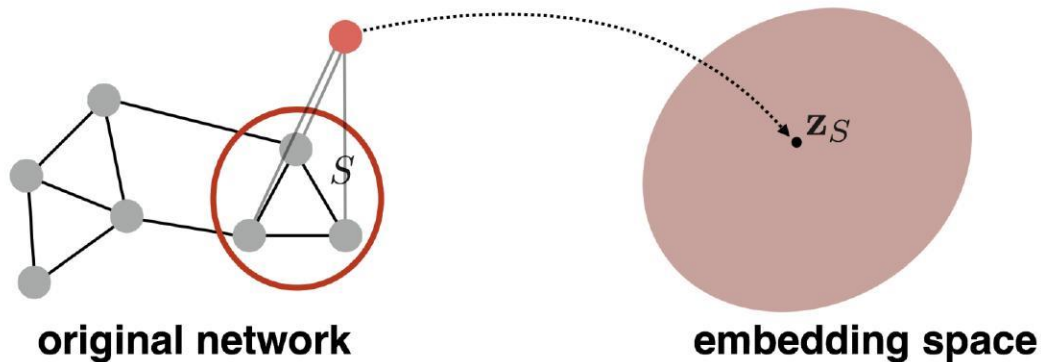
- Run a standard graph embedding technique *on* the (sub)graph G .
- Then just sum (or average) the node embeddings in the (sub)graph G .

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

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

Approach 2

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



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

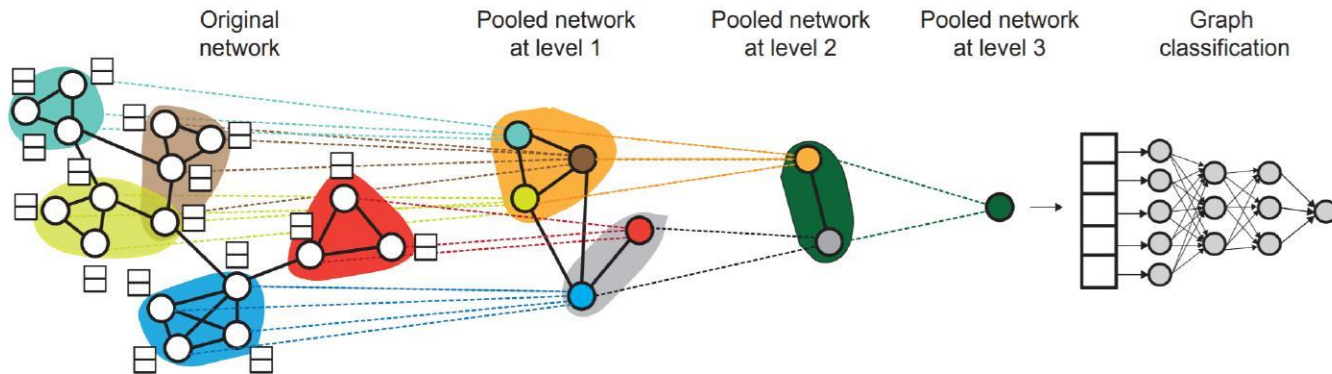
Summary

We discussed 2 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.

Preview: Hierarchical Embeddings

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



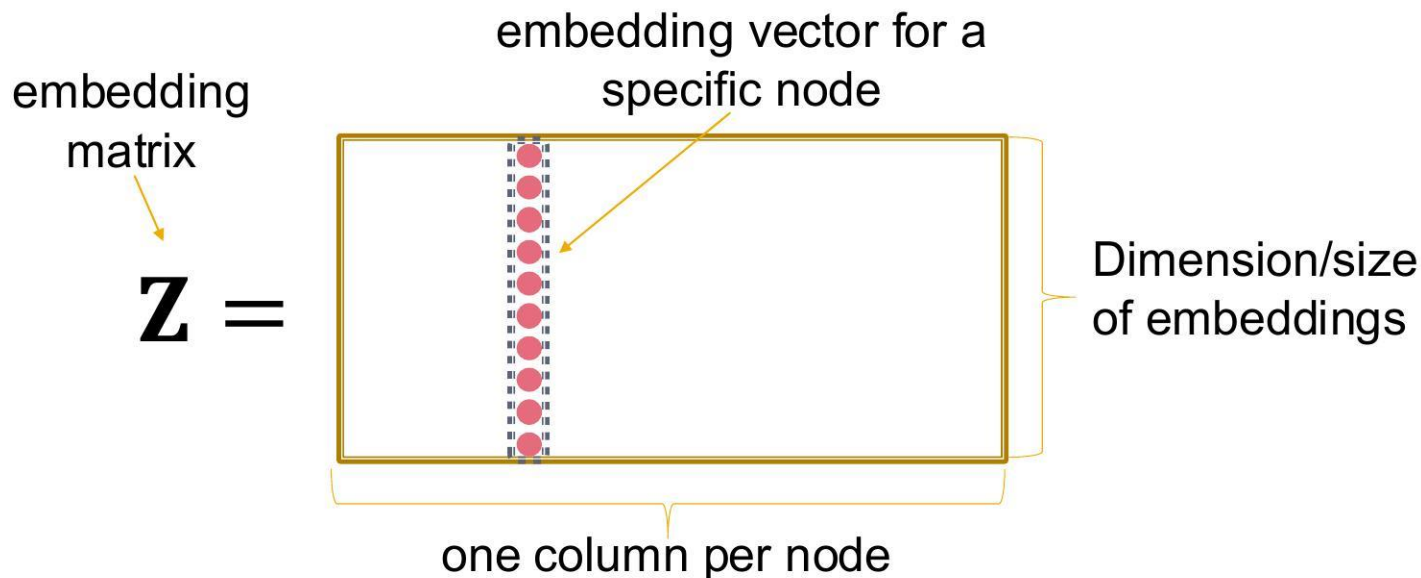
Stanford CS224W: Matrix Factorization and Node Embeddings

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



Embeddings & Matrix Factorization

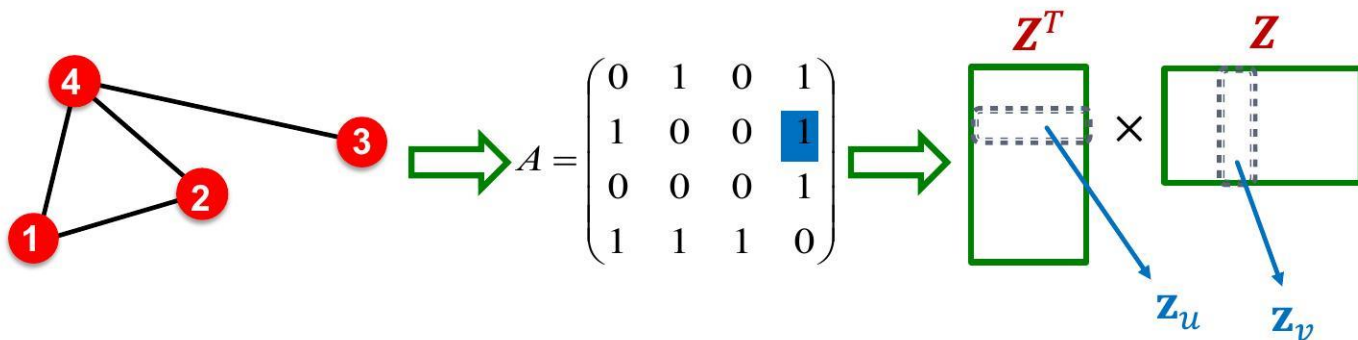
- Recall: encoder as an embedding lookup



Objective: maximize $\mathbf{z}_v^T \mathbf{z}_u$ 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: $\mathbf{z}_v^T \mathbf{z}_u = A_{u,v}$
which is the (u, v) entry of the graph adjacency matrix A
- Therefore, **$\mathbf{Z}^T \mathbf{Z} = A$**



Matrix Factorization

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

Random Walk-based Similarity

- **DeepWalk** and **node2vec** have a more complex **node similarity** definition based on random walks

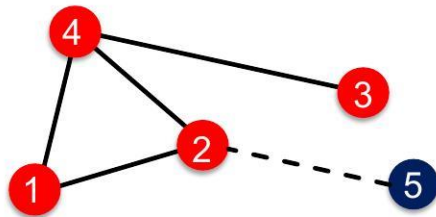
How to Use Embeddings

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

Limitations (1)

Limitations of node embeddings via matrix factorization and random walks

- **Transductive (not inductive) method:**
 - Cannot obtain embeddings for nodes not in the training set
 - Cannot apply to new graphs
 - If you apply DeepWalk to the same graph multiple times, each time you'll get a different embedding each time. Why?



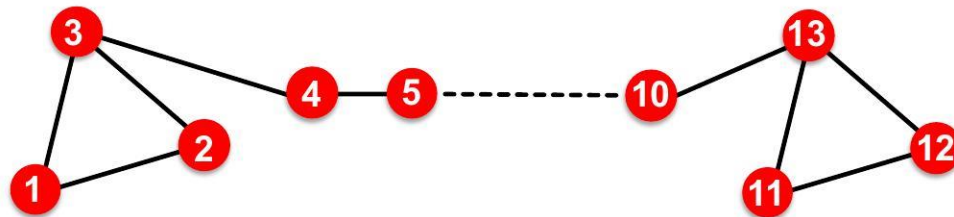
Training set

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)

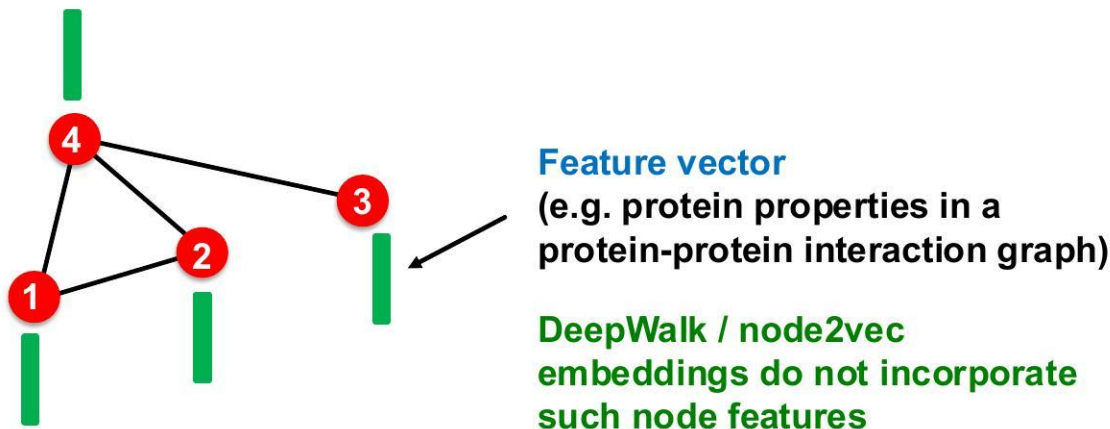
- Cannot capture **structural similarity**:



- 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



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

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

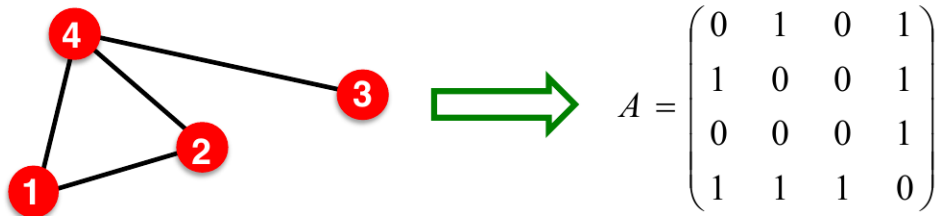
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
<http://cs224w.stanford.edu>



Graph as Matrix

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

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

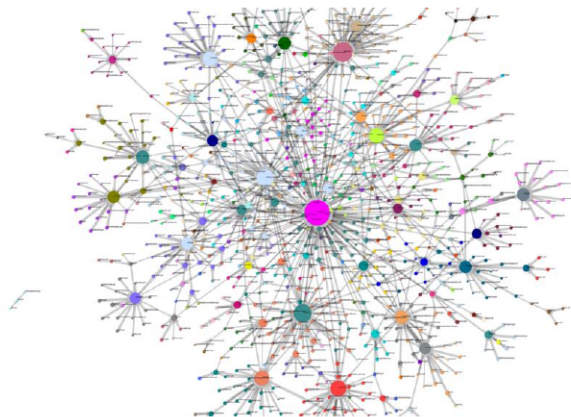


Example: The Web as a Graph

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

- **Web as a graph:**

- Nodes = web pages
- Edges = hyperlinks
- **Side issue: What is a node?**
 - Dynamic pages created on the fly
 - “dark matter” – inaccessible database generated pages



The Web as a Graph

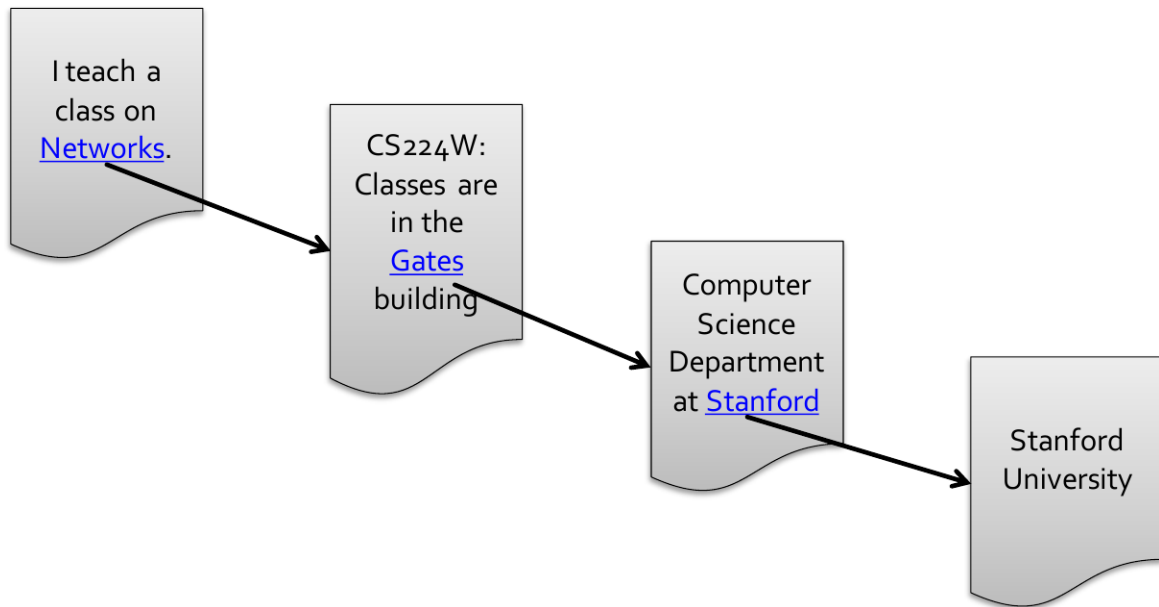
I teach a
class on
Networks.

CS224W:
Classes are
in the
Gates
building

Computer
Science
Department
at Stanford

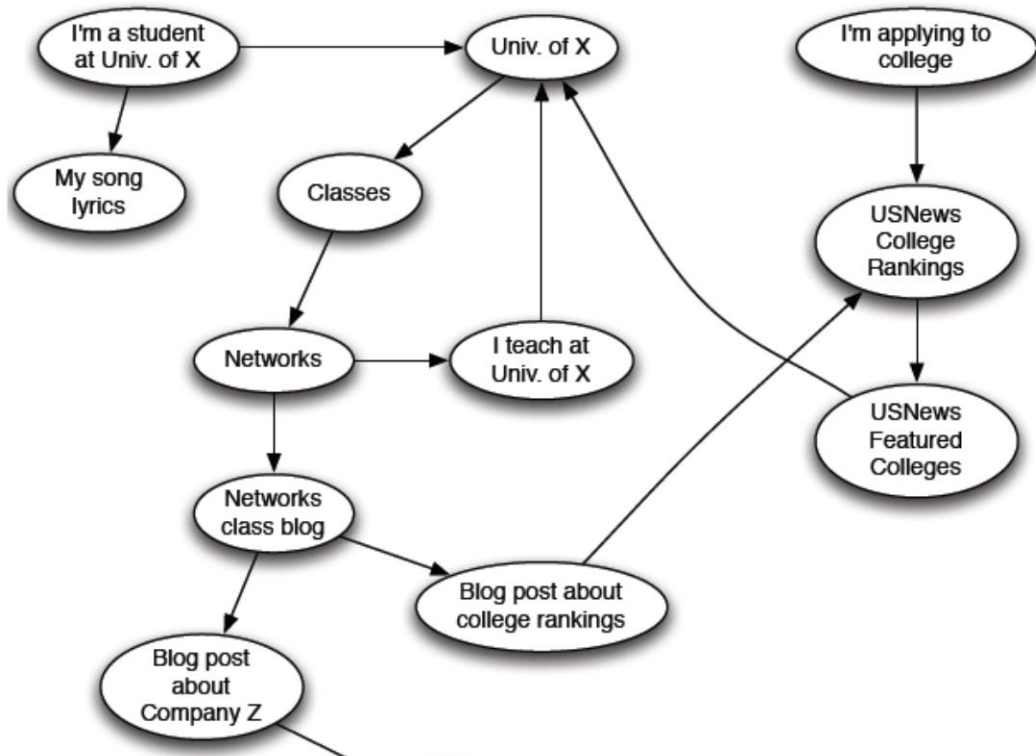
Stanford
University

The Web as a Graph



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

The Web as a Directed Graph

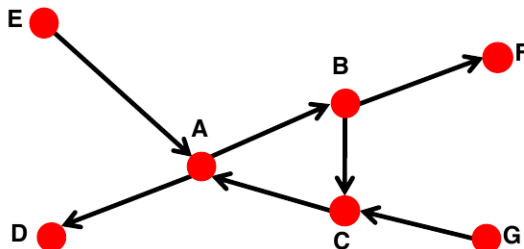


What Does the Web Look Like?

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

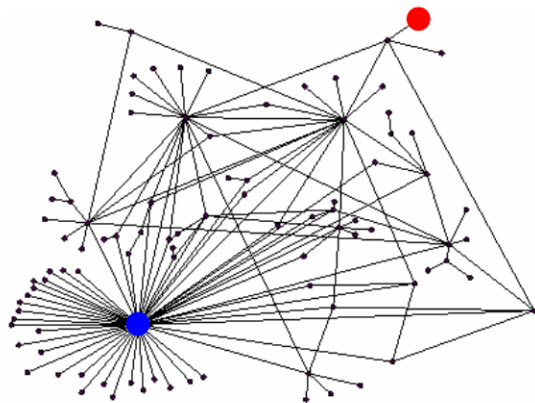
Web as a directed graph [Broder et al. 2000]:

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



Ranking Nodes on the Graph

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



Link Analysis Algorithms

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

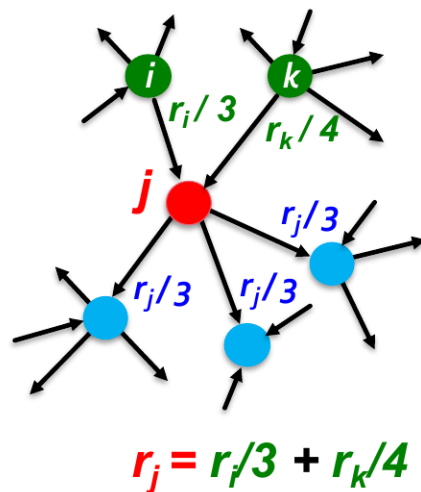
Links as Votes

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

PageRank: The “Flow” Model

- A “vote” from an important page is worth more:

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



PageRank: The “Flow” Model

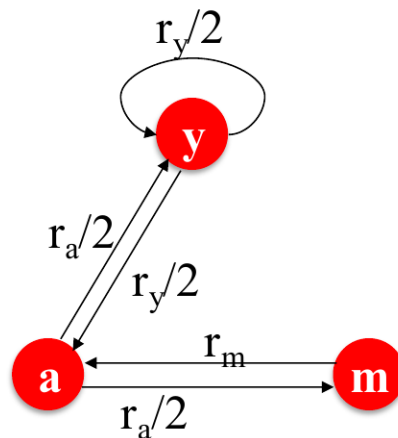
- A page is important if it is pointed to by other important pages
- Define “rank” r_j for node j

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

d_i ... out-degree of node i

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

The web in 1839



“Flow” equations:

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

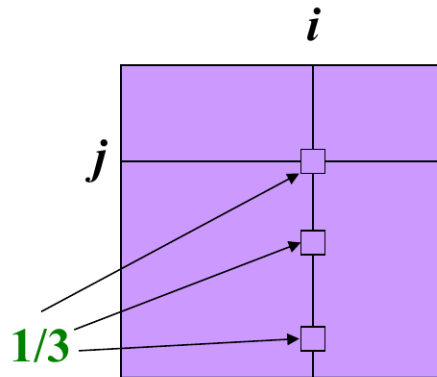
$$r_a = r_y/2 + r_m$$

$$r_m = r_a/2$$

PageRank: Matrix Formulation

- **Stochastic adjacency matrix M**

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



M

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

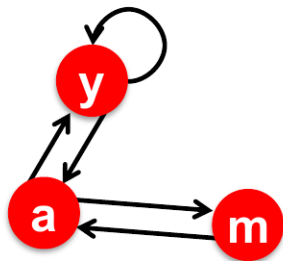
- r_i is the importance score of page i
- $\sum_i r_i = 1$

- **The flow equations can be written**

$$r = M \cdot r$$

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

Example: Flow Equations & M



	r_y	r_a	r_m
r_y	$\frac{1}{2}$	$\frac{1}{2}$	0
r_a	$\frac{1}{2}$	0	1
r_m	0	$\frac{1}{2}$	0

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

$$r_a = r_y/2 + r_m$$

$$r_m = r_a/2$$

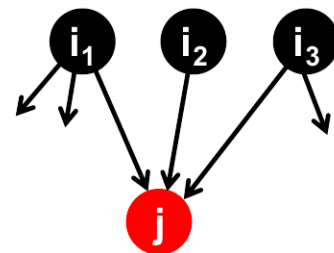
$$\begin{matrix} r_y \\ r_a \\ r_m \end{matrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{bmatrix} \begin{matrix} r_y \\ r_a \\ r_m \end{matrix}$$

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

Connection to Random Walk

- **Imagine a random web surfer:**

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



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

- **Let:**

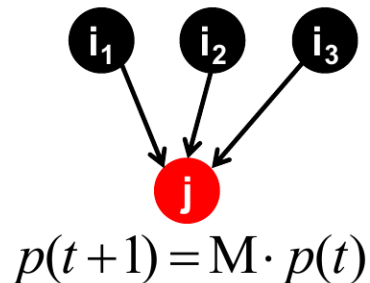
- $\mathbf{p}(t)$... vector whose i^{th} coordinate is the prob. that the surfer is at page i at time t
- So, $\mathbf{p}(t)$ is a probability distribution over pages

The Stationary Distribution

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

- Follow a link uniformly at random

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



- Suppose the random walk reaches a state

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

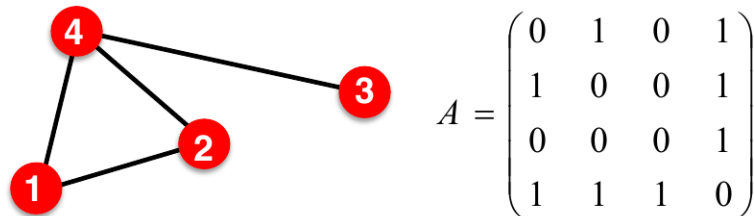
then $p(t)$ is **stationary distribution** of a random walk

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

- So, r is a stationary distribution for the random walk

Recall Eigenvector of A Matrix

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



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

Eigenvector Formulation

- The flow equation:

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

$$\begin{array}{|c|} \hline r_y \\ \hline r_a \\ \hline r_m \\ \hline \end{array} = \begin{array}{|ccc|} \hline \frac{1}{2} & \frac{1}{2} & 0 \\ \hline \frac{1}{2} & 0 & 1 \\ \hline 0 & \frac{1}{2} & 0 \\ \hline \end{array} \begin{array}{|c|} \hline r_y \\ \hline r_a \\ \hline r_m \\ \hline \end{array}$$

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

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

PageRank: How to solve?

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

- Assign each node an initial page rank
- Repeat until convergence ($\sum_i |r_i^{t+1} - r_i^t| < \epsilon$)
 - Calculate the page rank of each node

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

d_i out-degree of node i

Power Iteration Method

- Given a web graph with N nodes, where the nodes are pages and edges are hyperlinks
- **Power iteration:** a simple iterative scheme
 - Initialize: $\mathbf{r}^{(0)} = [1/N, \dots, 1/N]^T$
 - Iterate: $\mathbf{r}^{(t+1)} = \mathbf{M} \cdot \mathbf{r}^{(t)}$
 - Stop when $|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}|_1 < \varepsilon$

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

d_i out-degree of node i

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

About 50 iterations is sufficient to estimate the limiting solution.

PageRank: How to solve?

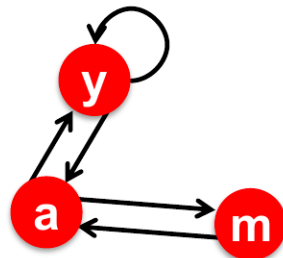
■ Power Iteration:

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

■ Example:

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

Iteration 0, 1, 2, ...



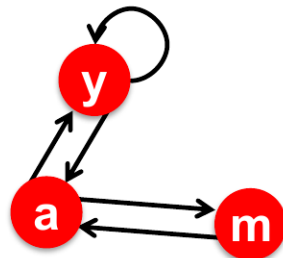
	y	a	m
y	1/2	1/2	0
a	1/2	0	1
m	0	1/2	0

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

PageRank: How to solve?

■ Power Iteration:

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



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

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

■ Example:

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

Iteration 0, 1, 2, ...

PageRank: Three Questions

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

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

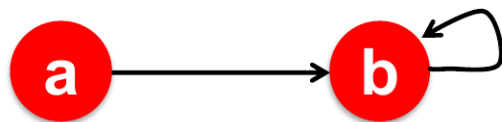
PageRank: Problems

Two problems:

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

Does this converge?

- The “Spider trap” problem:



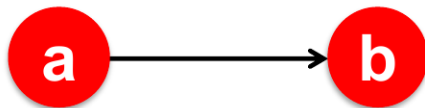
$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

- Example:

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

Does it converge to what we want?

- The “Dead end” problem:



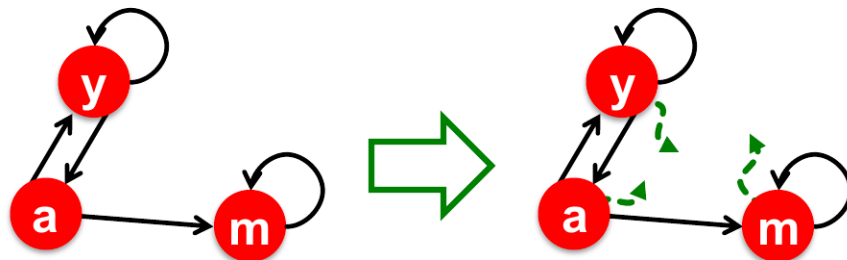
$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

- Example:

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

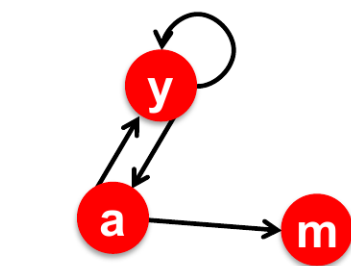
Solution to Spider Traps

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

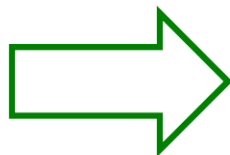
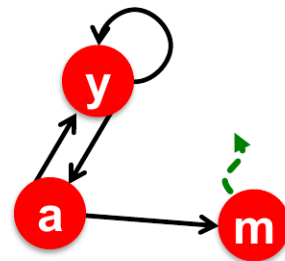


Solution to Dead Ends

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



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



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

Why Teleports Solve the Problem?

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

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

Solution: Random Teleports

- Google's solution that does it all:

At each step, random surfer has two options:

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

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

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

$d_i \dots$ out-degree of node i

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

The Google Matrix

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

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

- **The Google Matrix G :**

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

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

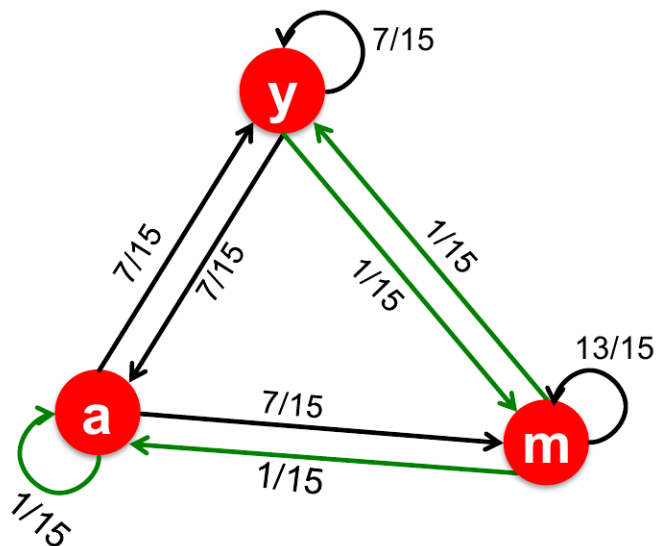
- **We have a recursive problem: $\mathbf{r} = G \cdot \mathbf{r}$**

And the Power method still works!

- **What is β ?**

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

Random Teleports ($\beta = 0.8$)



$$\begin{matrix}
 & \mathbf{M} & & \mathbf{[1/N]_{N \times N}} \\
 0.8 & \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} & + 0.2 & \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} \\
 & \mathbf{G} & & \\
 & \begin{matrix} y \\ a \\ m \end{matrix} \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix} & &
 \end{matrix}$$

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

PageRank Example

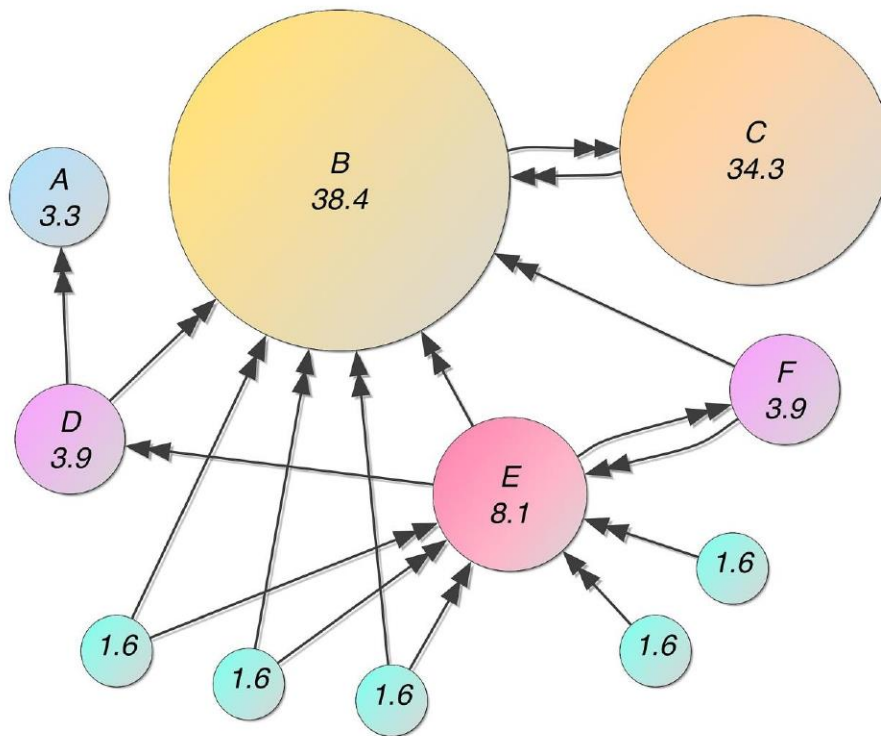


Image credit: [Wikipedia](#)

Solving PageRank: Summary

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

Stanford CS224W: Random Walk with Restarts and Personalized PageRank

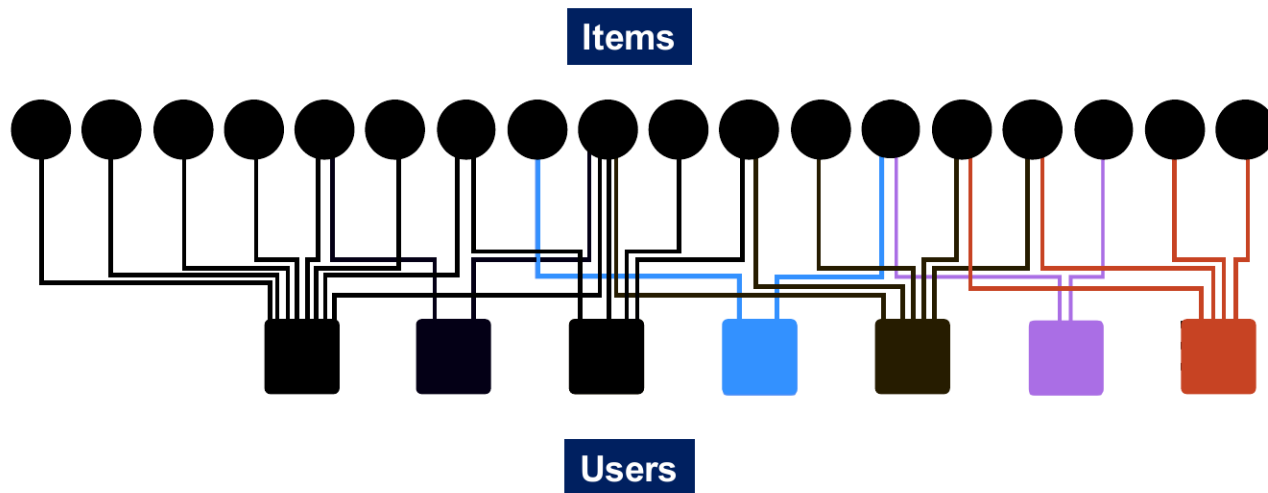
CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
<http://cs224w.stanford.edu>



Example: Recommendation

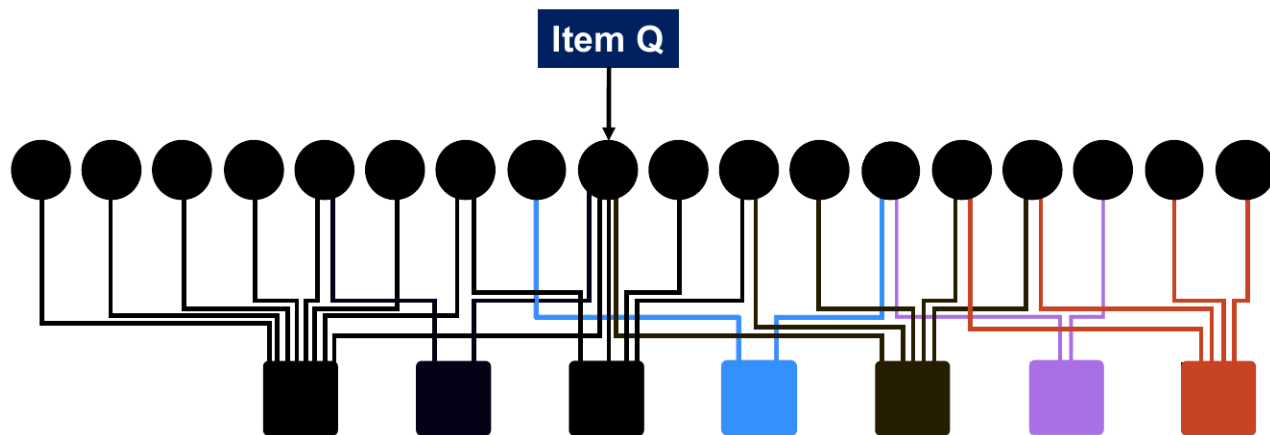
- **Given:**

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



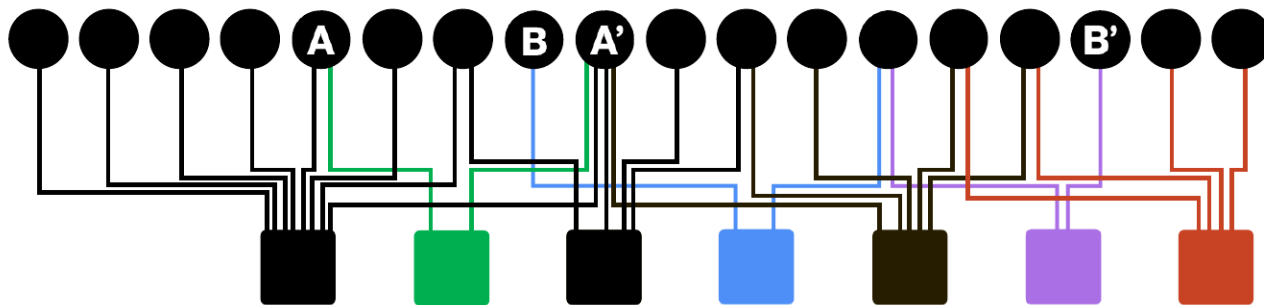
Bipartite User-Item Graph

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



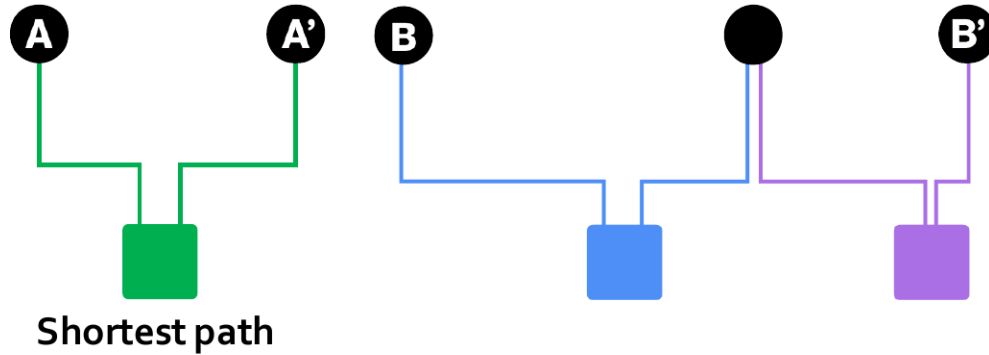
Bipartite User-to-Item Graph

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



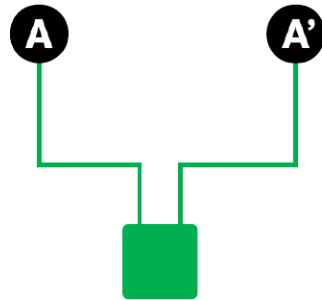
Node proximity Measurements

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

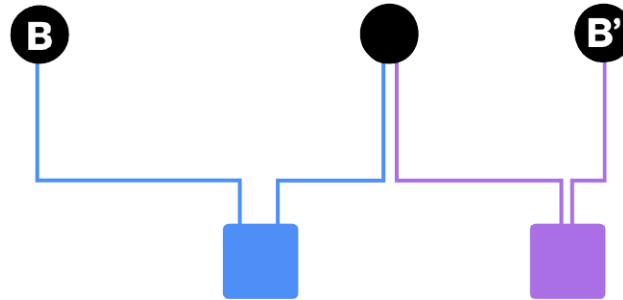
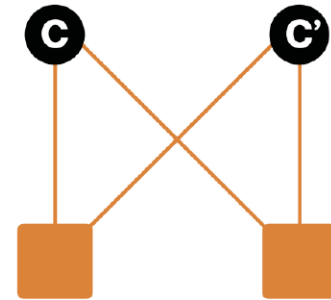


Node proximity Measurements

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



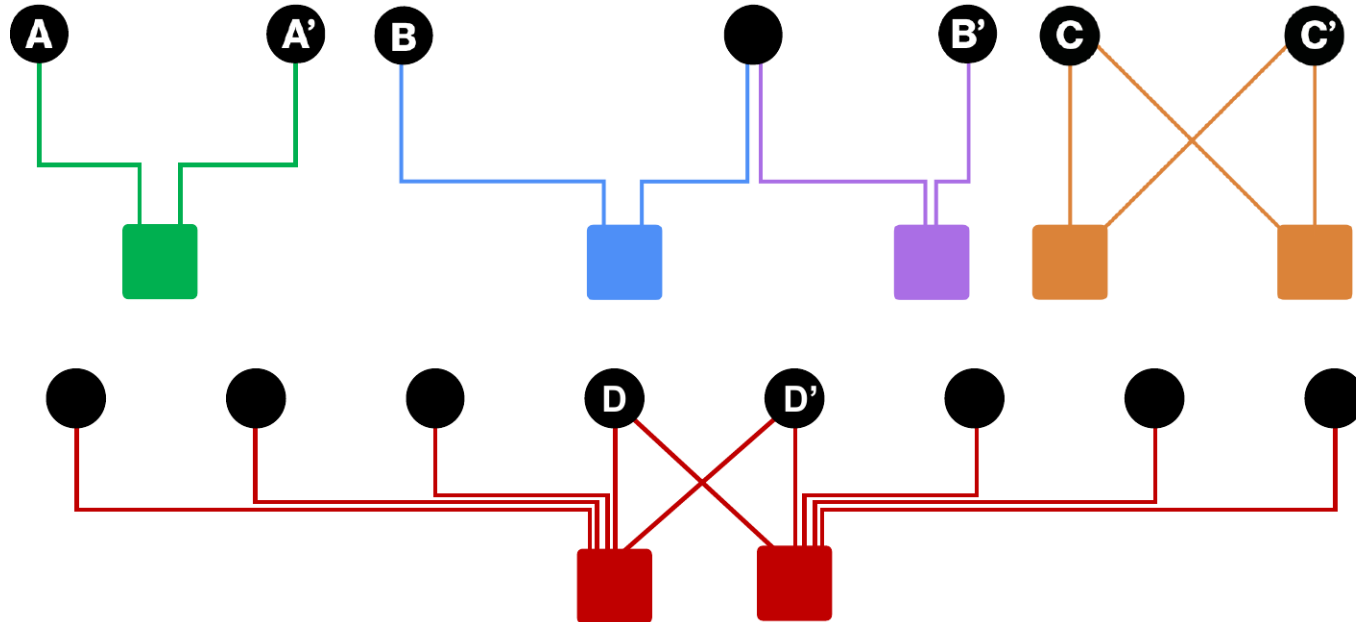
Shortest path



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 S

- **Proximity on graphs:**

- **Q:** What is most related item to **Item Q**?
- **Random Walks with Restarts**
 - Teleport back to the starting node: $S = \{Q\}$

Idea: Random Walks

- **Idea**

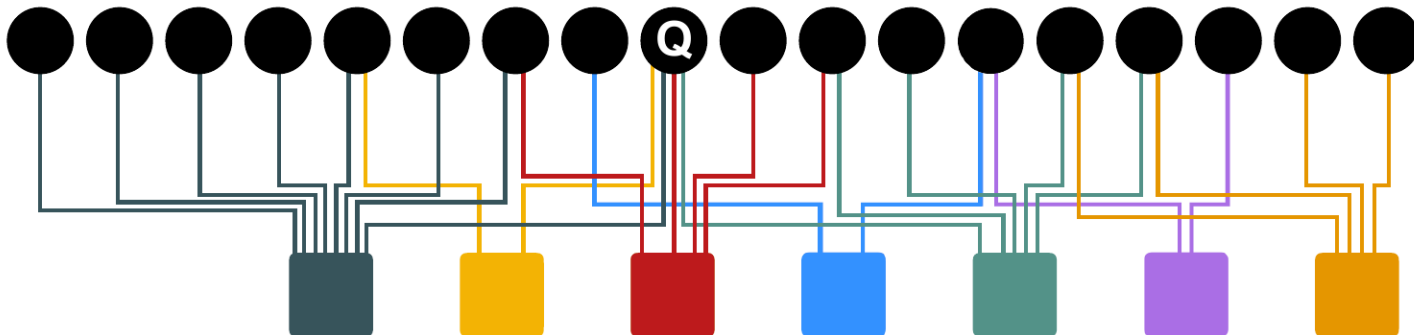
- Every node has some importance
- Importance gets evenly split among all edges and pushed to the neighbors:

- **Given a set of QUERY_NODES, we simulate a random walk:**

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

Random Walks

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

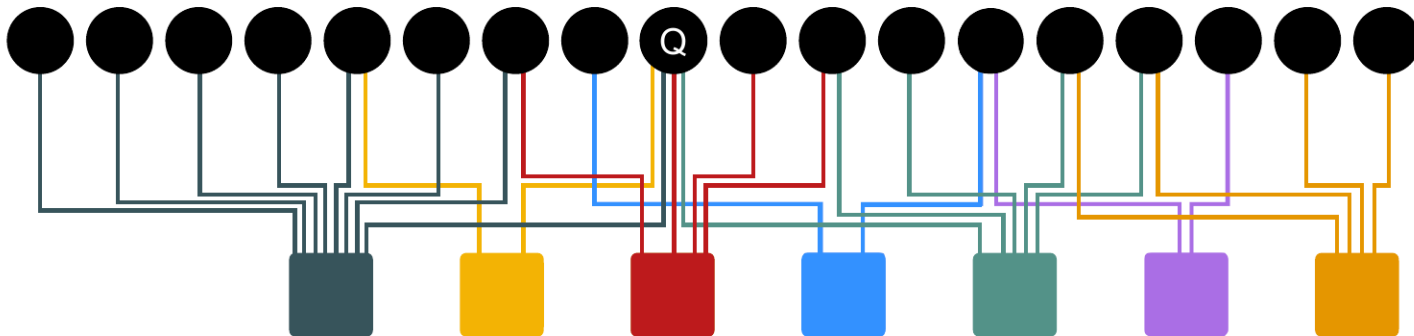


Random Walk Algorithm

■ Proximity to query node(s) Q :

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

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



Random Walk Algorithm

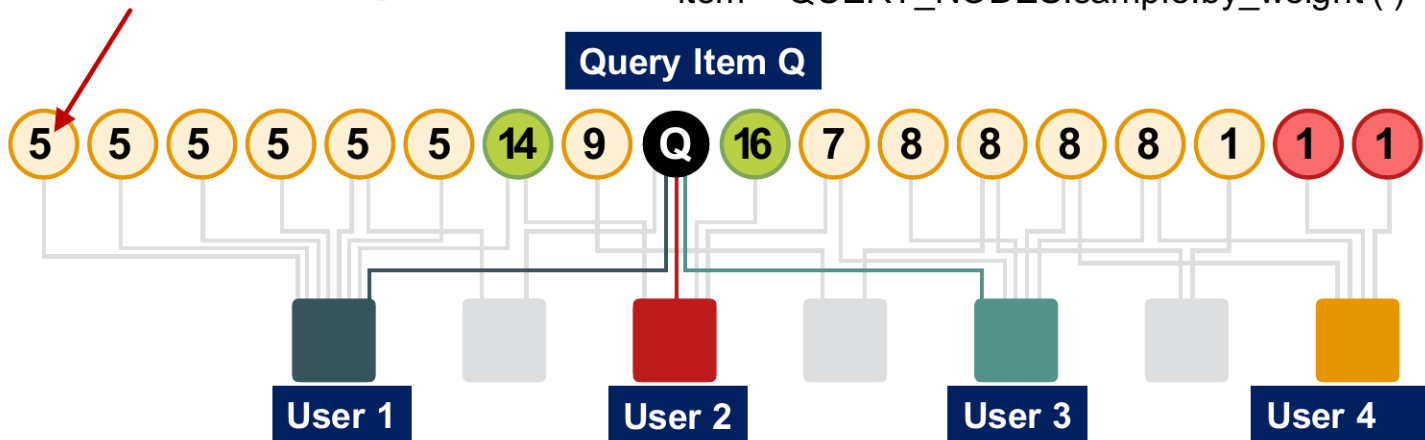
■ Proximity to query node(s) Q :

$\text{ALPHA} = 0.5$

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

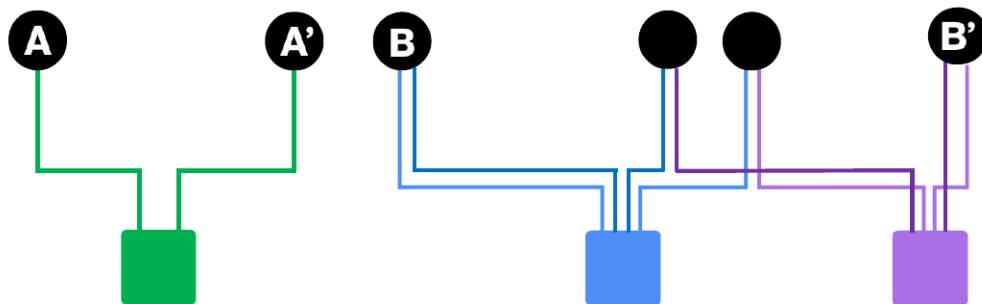
```
item = QUERY_NODES.sample_by_weight( )  
for i in range( N_STEPS ):  
    user = item.get_random_neighbor( )  
    item = user.get_random_neighbor( )  
    item.visit_count += 1  
    if random( ) < ALPHA:  
        item = QUERY_NODES.sample_by_weight( )
```

Number of visits by
random walks starting at Q



Benefits

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



Summary: Page Rank Variants

■ PageRank:

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

■ Topic-Specific PageRank aka Personalized PageRank:

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

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

■ Random Walk with Restarts:

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

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

Summary

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

Summary

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