Proximity and Role-based Graph Representation Learning

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

네트워크 과학 연구실 NETWORK SCIENCE LAB

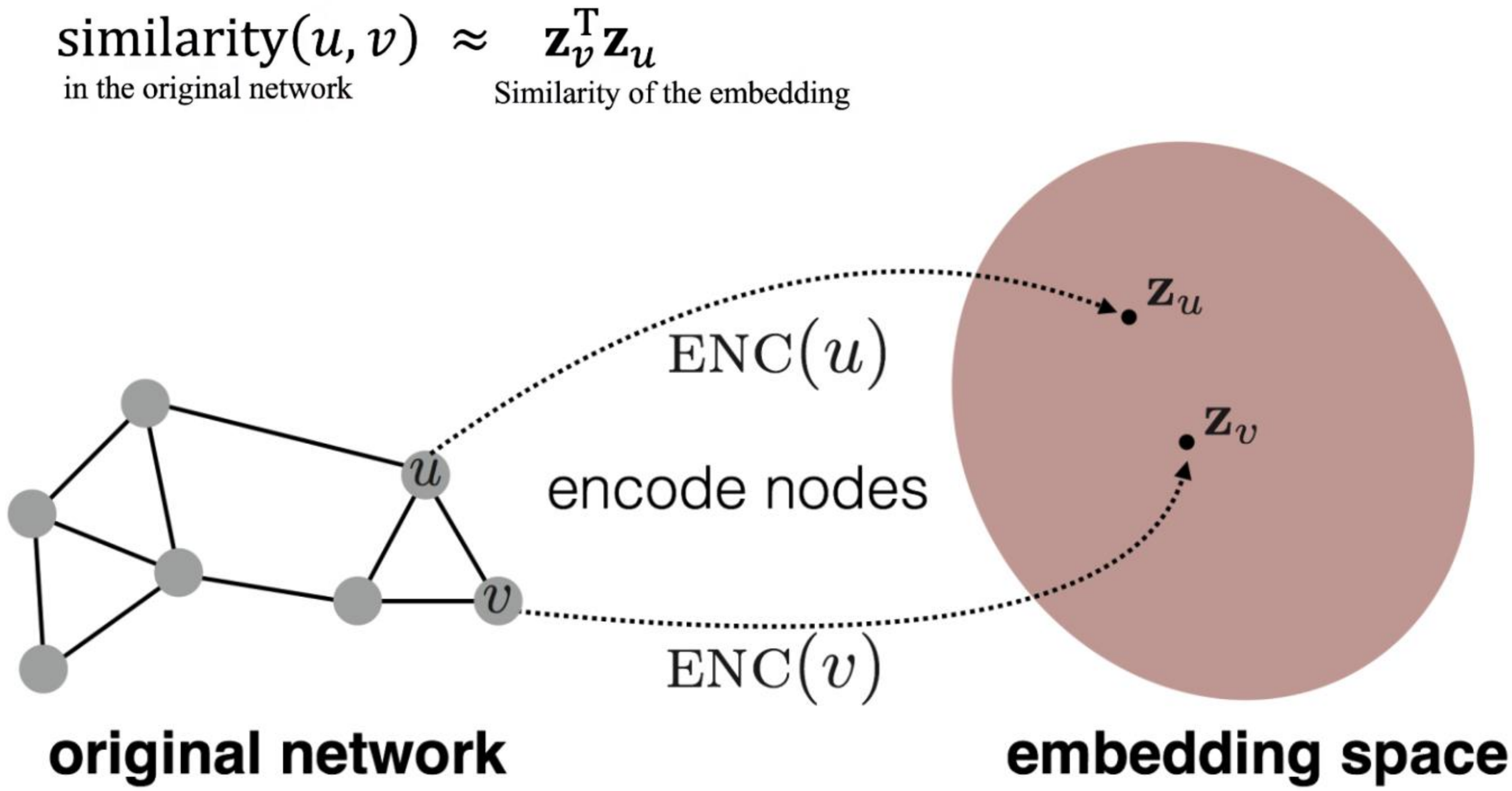
가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

Contents

- From Random Walk to Proximity
- Proximity-based methods:
  - LINE: Large-scale Information Network Embedding
  - Asymmetric Transitivity Preserving Graph Embedding
  - APP: Scalable Graph Embedding for Asymmetric Proximity
- Role-based methods:
  - Struc2vec: Learning Node Representations from Structural Identity
  - Role-Based Graph Embeddings

Remind the Node Embeddings

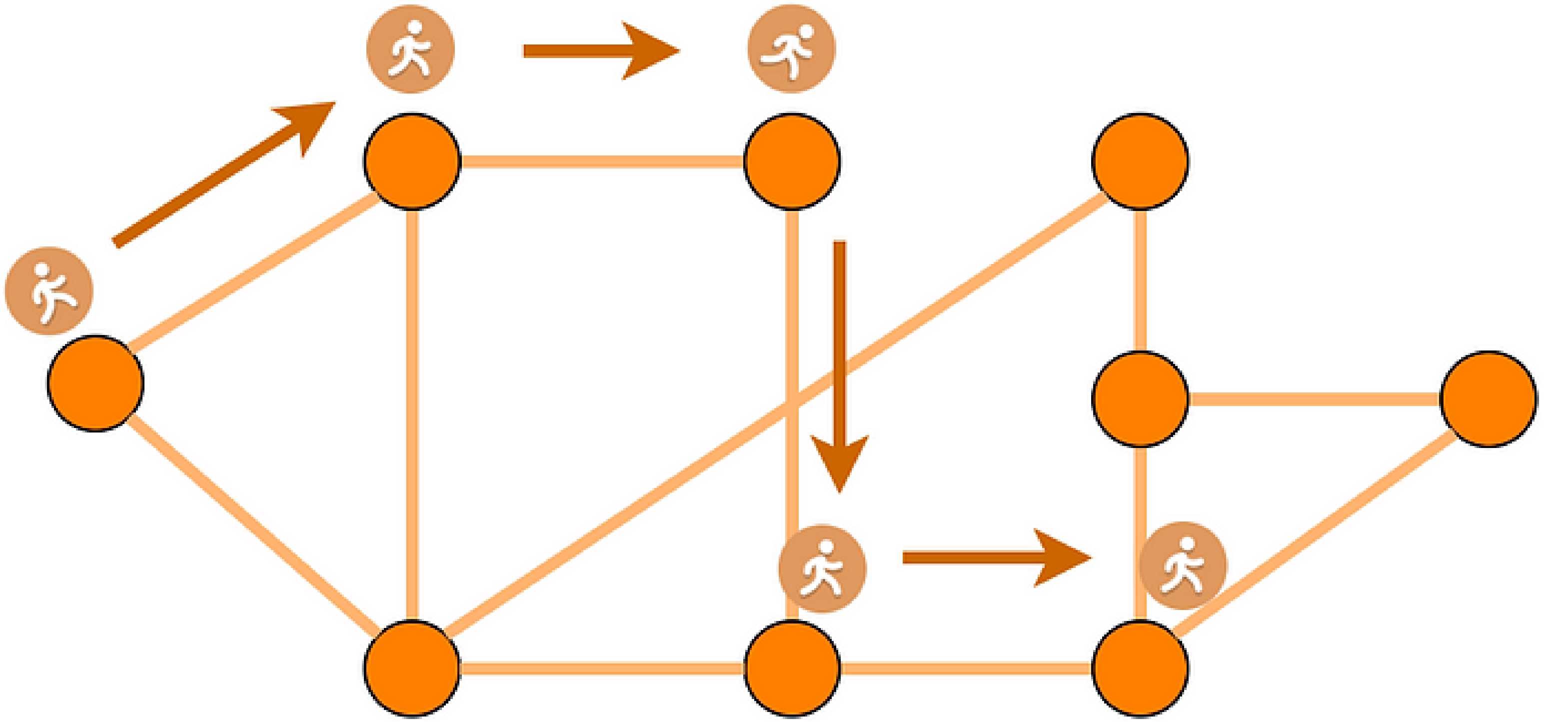
- Goal: Encode nodes → similarity in embedding space (dot product) ≈ similarity in the original graph



We need to define:
ENC(u)
Similarity(u, v)

https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/node-representation-learning

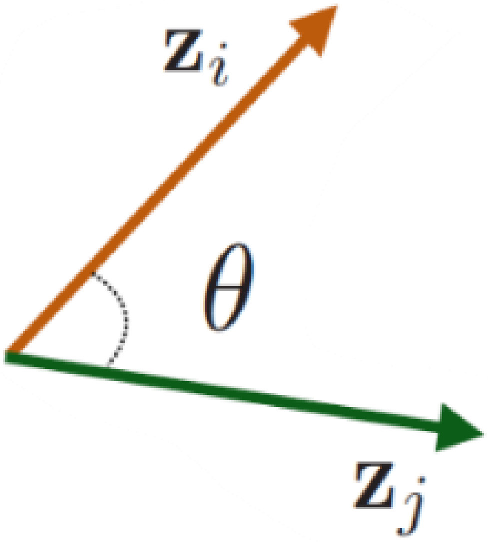
- Given a graph and a starting point, we select a neighbour of it at random, and move to this neighbour
- Then, we select a neighbor of this point at random, and move to it,…
- The random sequence of nodes visited this way is **a random walk on the graph**



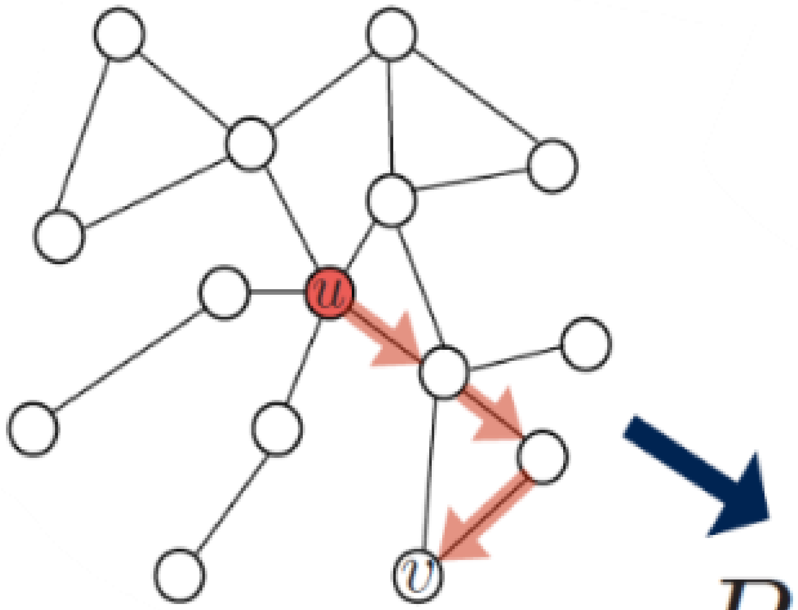
https://medium.com/analytics-vidhya/an-intuitive-explanation-of-deepwalk-84177f7f2b72

Random Walk Embeddings

- Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R
- Optimize embeddings to encode these random walk statistics



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

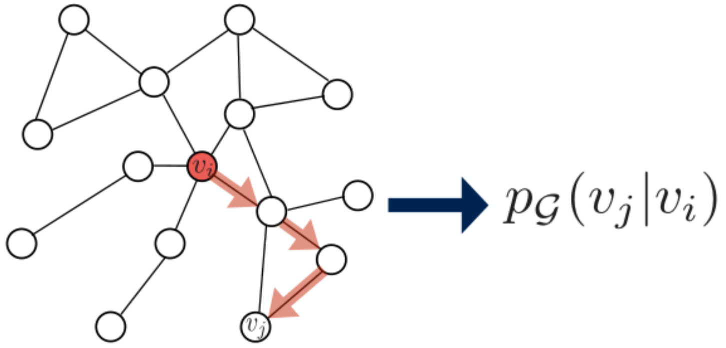


$$P_R(v|u)$$

DeepWalk

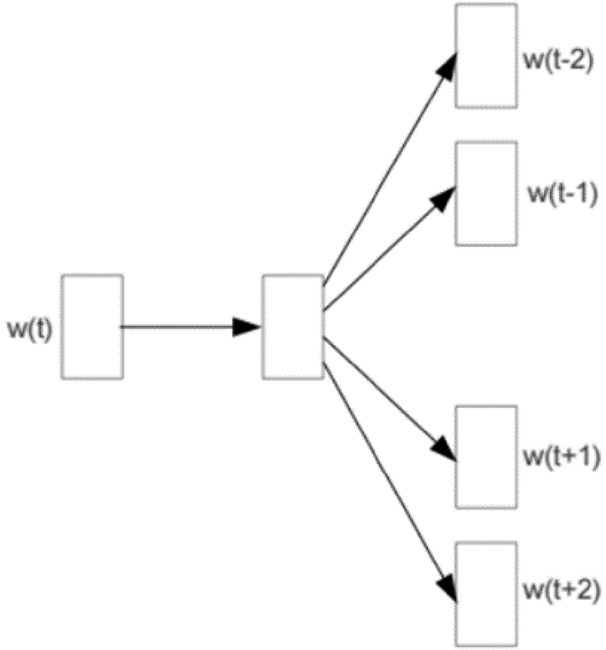
- Employ random walks on the graph to discover the structure

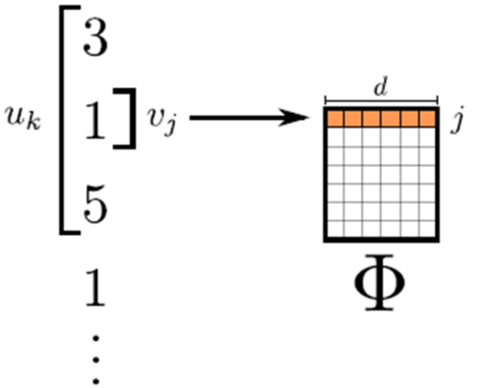
$$v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_1 \rightarrow v_{46}$$



1. Run random walks to obtain co-occurrence statistics.

Random walks in Network
=
Sentences in NLP





Node sequence as the input of word2vec models

$$\Phi : v \in V \mapsto \mathbb{R}^{|V| \times d}$$

DeepWalk: Optimization using Stochastic Gradient Descent

- **Goal**: Optimize

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

- (Stochastic) Gradient Descent: a simple way to minimize L

- **SGD Algorithm:** evaluate it for each individual training example
  - Initialize z_u at some randomized value for all nodes u
  - Iterative until L converges:
    - Sample a node u, for all v calculate the derivative $\frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$
    - For all v, update:

$$z_v \leftarrow z_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$$

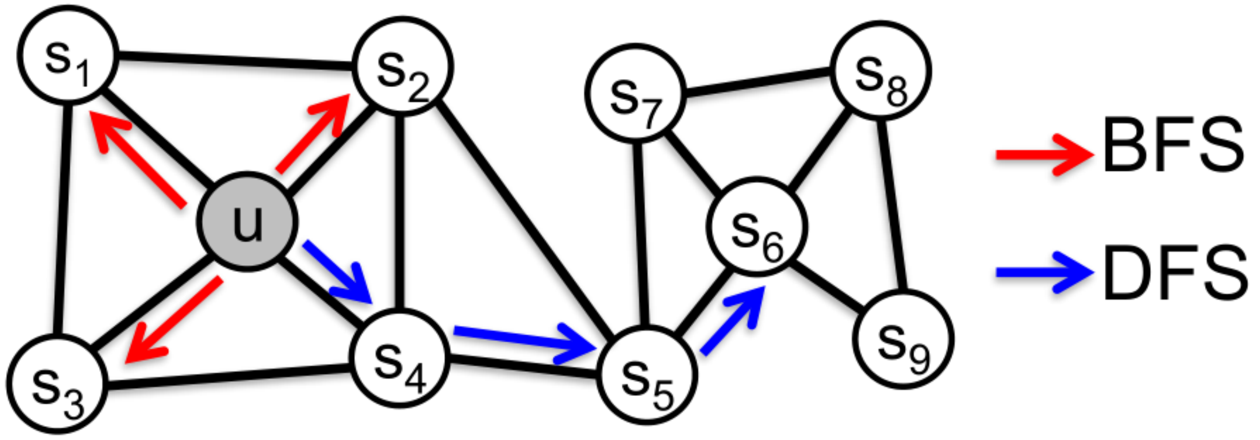
Learning rate

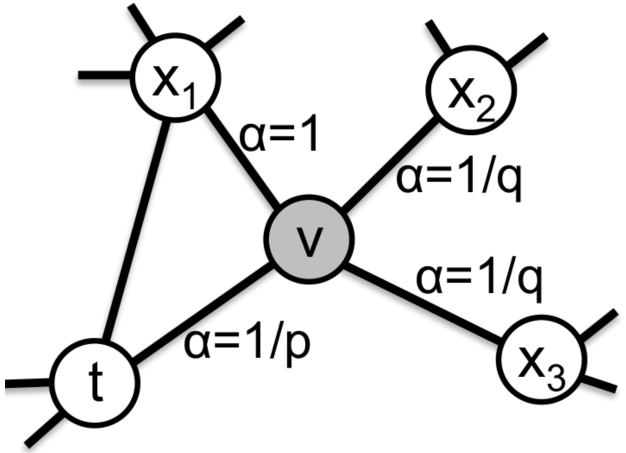
Node2Vec

Biased random walks

p: controls the walk revisiting a node

q: controls the walk revisiting anode's one-hop neighborhood





$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

***p* increase ⇒ DFS** ***q* increase ⇒ BFS**

node2vec: Scalable Feature Learning for Networks

Node2Vec: Negative Sampling in SGD

- Problem with DeepWalk Random Walk Optimization: Expensive in summing over nodes ($O(|V|^2)$)

$$\mathcal{L} = \underbrace{\sum_{u \in V}}_{\text{sum over all nodes } u} \underbrace{\sum_{v \in N_R(u)}}_{\text{sum over nodes } v \text{ seen on random walks starting from } u} - \log\Big(\underbrace{\frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}}_{\text{predicted probability of } u \text{ and } v \text{ co-occuring on random walk}}\Big)$$

- **Solution:** Negative Sampling (Softmax → Sigmoid)

$$\log\Big(\frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}\Big) \approx \log\big(\sigma(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)\big) - \sum_{i=1}^{k} \log\big(\sigma(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_{n_i})\big), n_i \sim P_V$$

Sigmoid function

Random distribution over nodes

- Sample k negative nodes n_i each with probability proportional to its degree

- Motivation:
  - Preserving network proximities: LINE seeks to preserve both the **first-order proximity** (direct connections between nodes) and **second-order proximity** (shared neighborhood structures) in the learned embeddings
  - Scalable objective functions:
    - Consider first-order and second-order proximities separately
    - Different network types: directed, undirected, weighted or unweighted
  - An edge-sampling algorithm is proposed to help stochastic gradient descent on weighted edges

- First-order proximity: local pairwise proximity between two connected nodes.
- For each node pair (v_i, v_j)
  - If $(v_i, v_j) \in E$, the first-order proximity between v_i and v_j is w_{ij}
  - Otherwise, the first-order proximity between v_i and v_j is 0
- Given an undirected edge (v_i, v_j), the joint probability of v_i and v_j:

$$p_1(v_i, v_j) = \frac{1}{1 + \exp(-\vec{u}_i^T \cdot \vec{u}_j)}$$

$\vec{u}_i$:Embedding of node v_i

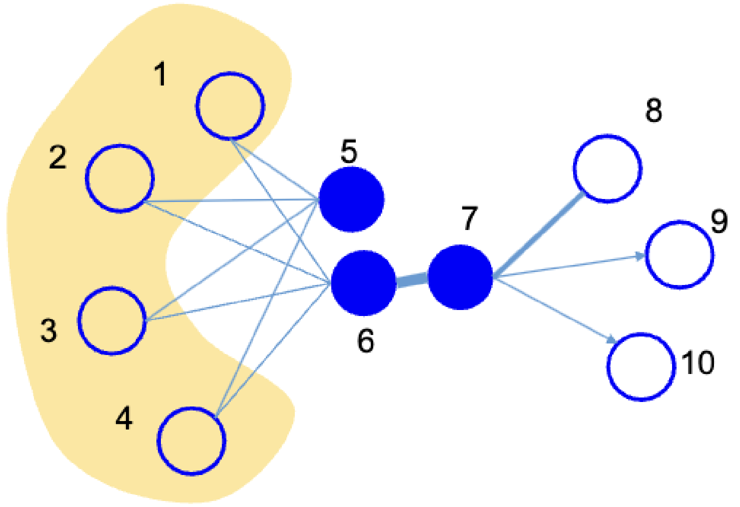
$$\hat{p}_1(v_i, v_j) = \frac{w_{ij}}{\sum_{(i',j')} w_{i'j'}}$$

- Objective:

KL-divergence

$$O_1 = d(\hat{p}_1(\cdot,\cdot), p_1(\cdot,\cdot))$$

$$\propto -\sum_{(i,j)\in E} w_{ij} \log p_1(v_i, v_j)$$



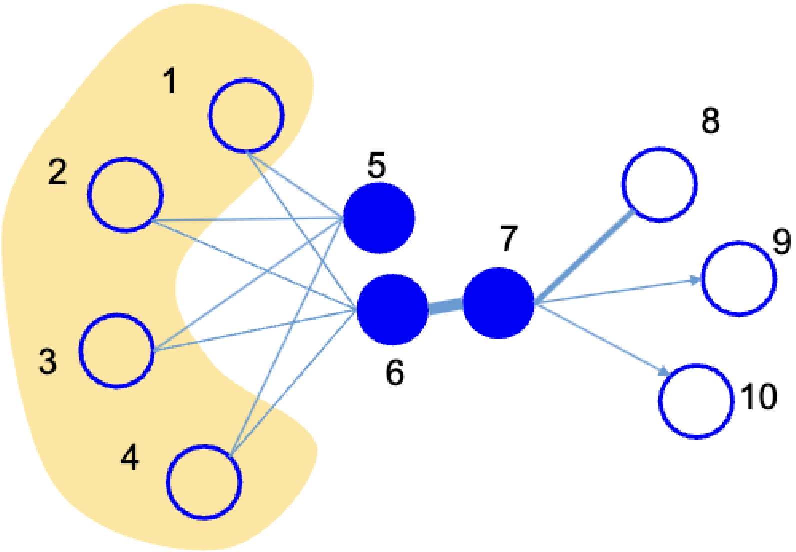
First-order: node 6 and 7

LINE: Second-order Proximity

- Second-order proximity captures the 2-step relations between each pair of nodes
- For each node pair (v_i, v_j)
  - determining by the number of common neighbours shared by the two nodes

$$\hat{p}_u = (w_{u1}, w_{u2}, \dots, w_{u|V|})$$

$$\hat{p}_v = (w_{v1}, w_{v2}, \dots, w_{v|V|})$$



Second-order: node 5 and 6

$$\hat{p}_5 = (1,1,1,1,0,0,0,0,0,0)$$

$$\hat{p}_6 = (1,1,1,1,0,0,5,0,0,0)$$

- Given an undirected edge (v_i, v_j), the joint probability of v_i and v_j:

$$p_2(v_j|v_i) = \frac{\exp(\vec{u}_j'^T \cdot \vec{u}_i)}{\sum_{k=1}^{|V|} \exp(\vec{u}_k'^T \cdot \vec{u}_i)},$$

$\vec{u}_i$:Embedding of node v_i when i is a source node
$\vec{u_i'}$:Embedding of node v_i when i is a target node

$$\hat{p}_2(v_j|v_i) = \frac{w_{ij}}{\sum_{k \in \mathrm{V}} w_{ik}}$$

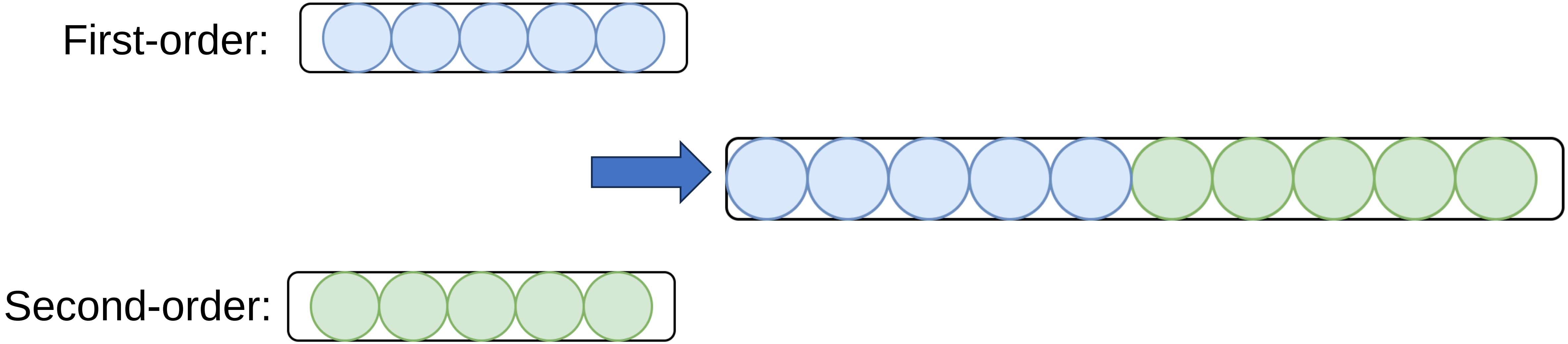
- Objective:

λ_i: Prestige of node in the network $\lambda_i = \sum_j w_{ij}$

$$O_2 = \sum_{i \in V} \lambda_i d(\hat{p}_2(\cdot|v_i), p_2(\cdot|v_i)),$$

$$\propto - \sum_{(i,j) \in E} w_{ij} \log p_2(v_j|v_i).$$

- Concatenate the embeddings individually learned by the two proximity:



LINE: Optimization

- Stochastic Gradient Descent + Negative Sampling
  - Randomly sample an edge and multiple negative edges
- The gradient w.r.t the embedding with edge (i,j)

$$\frac{\partial O_2}{\partial \vec{u}_i} = w_{ij} \cdot \frac{\partial \log p_2(v_j|v_i)}{\partial \vec{u}_i}$$

Multiplied by the weight of the edge w_{ij}

- Problematic when the weights of the edges diverge
  - The scale of the gradients with different edges diverges
- **Solution:** Edge sampling
  - Sample the edges according to their weights and treat the edges as binary
- Complexity: $\Theta(dK|E|)$
  - Linear to the dimension d, number of negative samples K, and number of edges E

LINE: Algorithm

- **Initialization**: Initialize the embedding vectors for all nodes randomly
- **Edge Sampling**: Sample edges from the network based on their weights
- **Gradient Descent**: For each sampled edge, update the embedding vectors using SGD
- **Negative Sampling**: For each positive edge, sample negative edges and update the embedding vectors to maximize the difference between positive and negative samples
- **Iteration**: Repeat the edge sampling and gradient descent steps until convergence

Same as DeepWalk, Node2Vec, and Node2Vec+ but consider the preservation of proximity and directed edge sampling

Jian Tang et al.: Asymmetric Transitivity Preserving Graph Embedding

Sample code: Edge Sampling

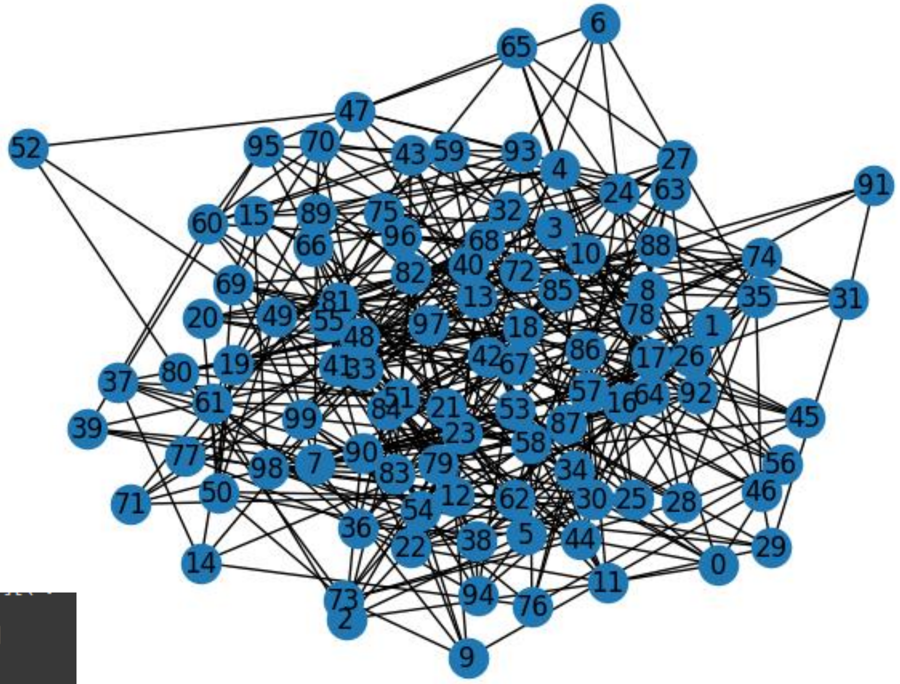

```
# Import library
import networkx as nx
import numpy as np
from vose_sampler import VoseAlias
import matplotlib.pyplot as plt
import collections
from tqdm import tqdm
from tqdm import trange
```

```
# Create an Erdős-Rényi graph with 100 nodes and probability 0.1
G_erdos = nx.erdos_renyi_graph(100, 0.1)

# Get the edge list of the graph
edge_list = G_erdos.edges

# Create a dictionary to store the edge list with weights and total weightsum
edgedistdict = collections.defaultdict(int)
weightsum = 0

# For each edge in the edge list, assign a random weight between 1 and 100.
for edge in edge_list:
    weight = np.random.uniform(1, 100)
    edgedistdict[(edge[0], edge[1])] = weight
    weightsum += weight

for edge, weight in edgedistdict.items():
    edgedistdict[edge] = weight / weightsum

# Print the normalized edge list with weights
print(edgedistdict)
```

```
defaultdict(<class 'int'>, {(0, 12): 0.0022917282387984626, (0, 25): 0.001658375
```

```
# Edge sampling using alias table
edgesaliassampler = VoseAlias(edgedistdict)
batchrange = int(len(edgedistdict) / 5)

for b in trange(batchrange):
  print(edgesaliassampler.sample_n(size=5))
```

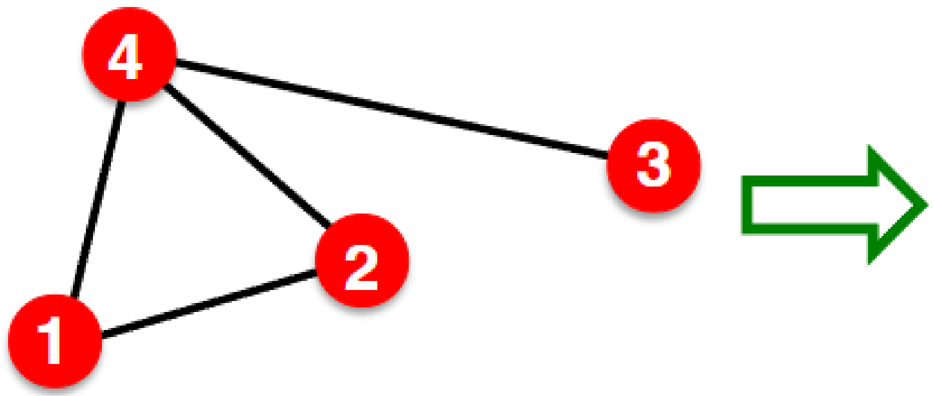
```
[(3, 8), (37, 93), (21, 63), (67, 94), (42, 79)]
[(22, 31), (50, 57), (12, 71), (44, 66), (27, 70)]
[(45, 71), (50, 55), (10, 74), (2, 22), (15, 71)]
[(4, 26), (21, 53), (22, 23), (5, 41), (11, 24)]
[(29, 64), (17, 51), (31, 69), (12, 22), (35, 63)]
[(17, 18), (59, 75), (15, 95), (43, 76), (0, 66)]
[(3, 22), (45, 71), (75, 93), (7, 75), (21, 98)]
[(11, 80), (1, 62), (31, 61), (0, 68), (12, 19)]
[(90, 95), (0, 60), (17, 18), (12, 20), (52, 99)]
[(38, 46), (27, 88), (22, 73), (24, 54), (0, 93)]
[(52, 74), (52, 99), (13, 70), (11, 47), (20, 45)]
[(15, 71), (12, 22), (0, 66), (5, 11), (64, 89)]
[(10, 74), (7, 97), (69, 74), (58, 90), (48, 89)]
[(38, 44), (4, 39), (50, 57), (7, 75), (14, 97)]
[(7, 15), (13, 18), (25, 50), (25, 54), (38, 44)]
[(0, 81), (90, 95), (15, 64), (32, 55), (21, 98)]
[(11, 18), (2, 47), (22, 83), (13, 26), (9, 78)]
[(22, 73), (62, 63), (38, 44), (35, 81), (6, 27)]
[(47, 75), (21, 53), (7, 15), (51, 92), (13, 84)]
[(11, 47), (2, 96), (43, 45), (29, 30), (18, 78)]
[(94, 99), (33, 61), (45, 85), (41, 46), (15, 95)]
[(69, 82), (79, 82), (9, 56), (21, 63), (18, 40)]
[(23, 94), (69, 80), (69, 82), (33, 66), (69, 74)]
[(59, 62), (20, 27), (3, 84), (79, 81), (40, 63)]
```



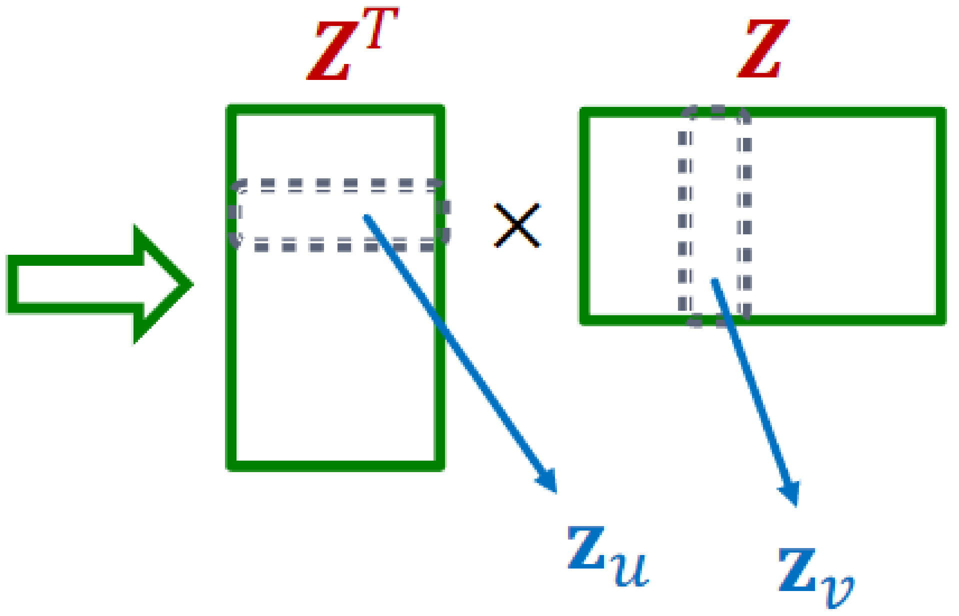
Remind the Adjacency-based Approaches

- One of the simplest and most intuitive approaches to defining similarity:
  - Adjacency between two nodes v and u
- Similarity: Two nodes are adjacent to one another within the structure of the graph

$$\mathbf{z}_v^T \mathbf{z}_u = A_{u,v} \ or \ \mathbf{Z}^T \mathbf{Z} = A$$



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



➢ Encoding: Find the embedding matrix **Z** that minimizes the loss function **L**

$$\boxed{\mathcal{L}} = \boxed{\sum_{(u,v)\in V\times V}} \|\boxed{\mathbf{z}_u^\top \mathbf{z}_v} - \boxed{\mathbf{A}}_{u,v}\|^2$$

loss (what we want to minimize)

sum over all node pairs

embedding similarity

(weighted) adjacency matrix for the graph

➢ Solution:

- ➢ Option 1: Use stochastic gradient descent (SGD) as a general optimization method
- ➢ Option 2: Solve matrix decomposition solvers (e.g., SVD or QR decomposition routines)
    - ➢ Matrix Factorization-based

- Let A be an m × n adjacency matrix. The factorization of A takes the form

$$A = USV^T$$

where U is a $m \times m$ orthogonal matrix, V^T is a $n \times n$ orthogonal matrix and S is a $m \times n$ diagonal matrix

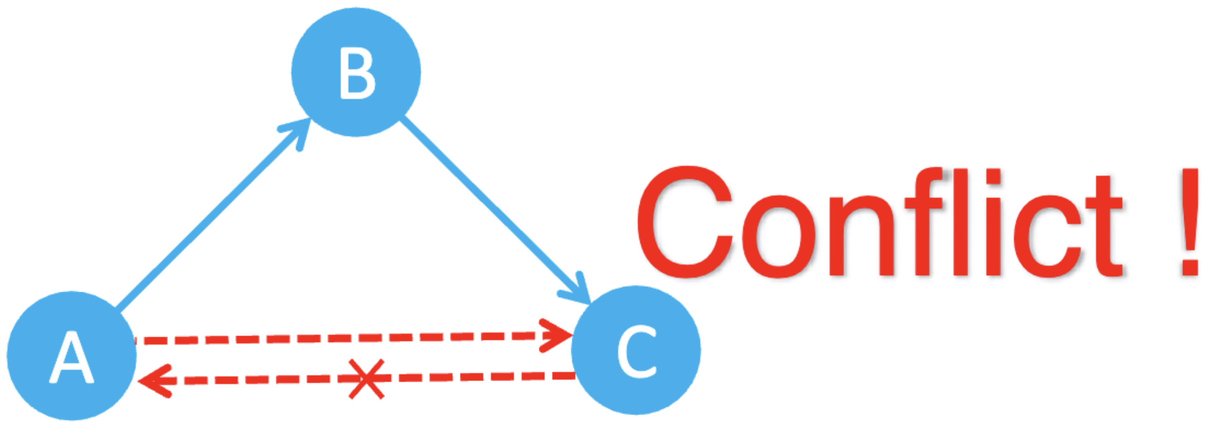
- Factorize to low-rank approximations based on minimizing the sum-squared distance using **Singular Value Decomposition**
- To decompose:
  - Evaluate the n eigenvectors v_i and eigenvalues λ_i of $A^T A$
  - Make a matrix V from the normalized vectors v_i
  - Make a diagonal matrix S from the square roots of the eigenvalues

$$\sigma_i = \sqrt{\lambda_i} \quad \text{and} \quad \sigma_1 \geq \sigma_2 \geq \sigma_3 \ldots$$

  - Find U: $A = USV^T \Rightarrow US = AV \Rightarrow U = AVS^{-1}$

Asymmetric Transitivity Preserving Graph Embedding

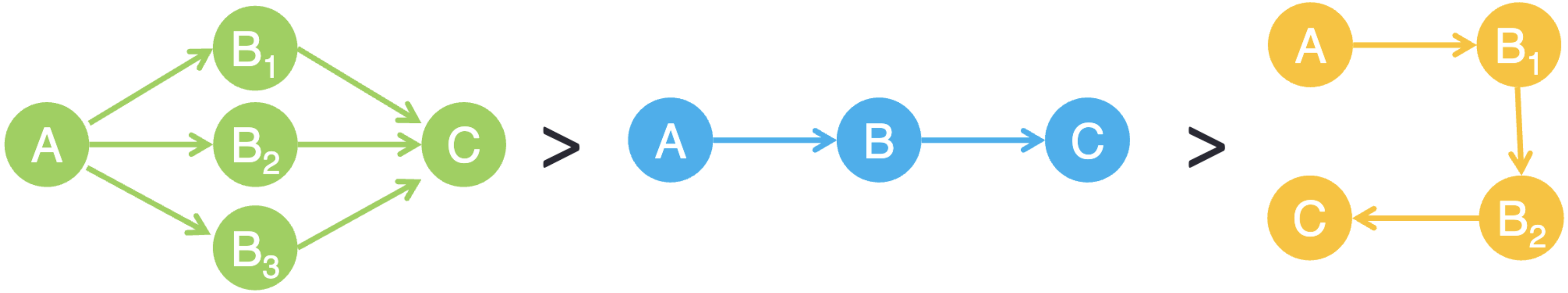
- Idea: Critical property in directed graph - Asymmetric Transitivity
- Transitivity is Asymmetric in directed graph
- Challenge: Incorporate asymmetric transitivity in graph embedding
- Problem: Metric space is symmetric



Asymmetric transitivity

Matrix space

- High-order Proximity with asymmetric transitivity:
  - Asymmetry: not symmetric in directed graph
  - Transitivity:
    - More directed paths, larger similarity
    - Shorter paths, larger similarity



- Compare A -> C similarity: Katz Index

$$\mathbf{S}^{Katz} = \sum_{l=1}^{+\infty} (\beta \cdot A)^l$$

- Preserve high-order proximity embedding:
  - Katz Index modified:

$$\mathbf{S}^{Katz} = \sum_{l=1}^{+\infty}(\beta \cdot A)^l = (I - \beta \cdot A)^{-1} \cdot (\beta \cdot A)$$

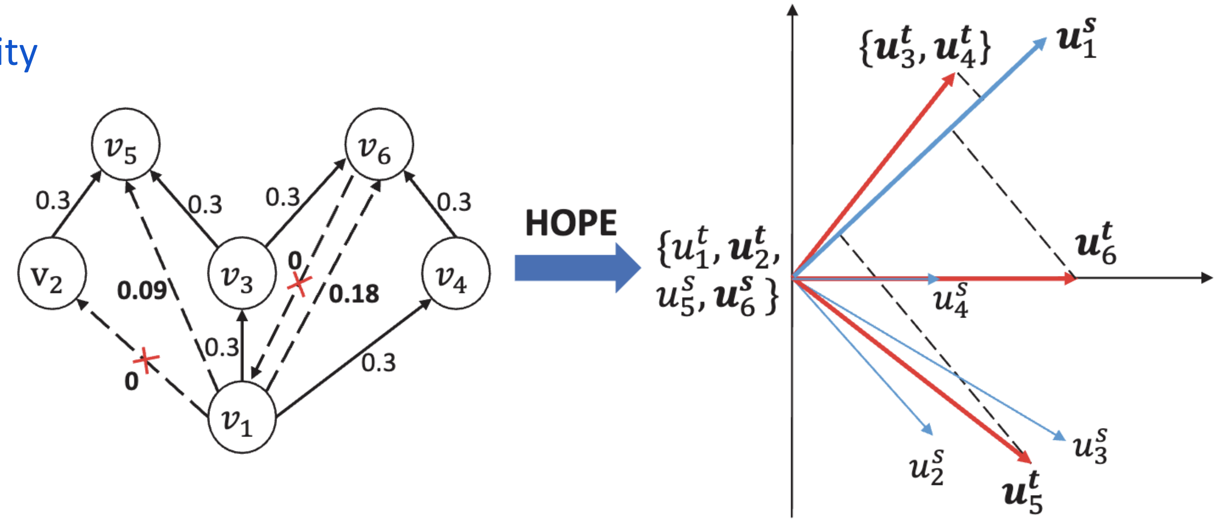
  - Objective:

Embedding similarity

$$\min_{U_s, U_t} \|\boldsymbol{S} - U_s \cdot U_t^T\|_F^2$$

$$\boldsymbol{S} = M_g^{-1} \cdot M_l$$

where M_g, M_l are polynomial of adjacency matrix or proximity measurements: Katz, Adamic Adar, PageRank, Common Neighbors



- Solving by **Generalized Singular Value Decomposition**: decompose $\boldsymbol{S}$ without calculating it
  - Linear complexity w.r.t. the volume of data (i.e. edge number)

Generalized Singular Value Decomposition

- Objective:

$$\min_{U_s,U_t} \|\boldsymbol{S} - U_s \cdot U_t^T\|_F^2$$
$$\boldsymbol{S} = M_g^{-1} \cdot M_l$$

- If we have the singular value decomposition of the general formulation:

$$\mathbf{M}_g^{-1} \cdot \mathbf{M}_l = \mathbf{V}^s \Sigma \mathbf{V}^{t\top}$$

V^t and V^s: orthogonal matrices.

$$\Sigma = diag(\sigma_1, \sigma_2, \cdots, \sigma_N)$$

Non-negative diagonal matrix.

- There exists a nonsingular matrix X and two diagonal matrices, i.e. Σ^l and Σ^g, satisfying

$$\mathbf{V}^{t\top}\mathbf{M}_l^\top\mathbf{X} = \Sigma^l$$
$$\mathbf{V}^{s\top}\mathbf{M}_g^\top\mathbf{X} = \Sigma^g$$

where
$$\Sigma^l = diag(\sigma_1^l, \sigma_2^l, \cdots, \sigma_N^l)$$
$$\Sigma^g = diag(\sigma_1^g, \sigma_2^g, \cdots, \sigma_N^g)$$
$$\sigma_1^l \geq \sigma_2^l \geq \cdots \geq \sigma_K^l \geq 0$$
$$0 \leq \sigma_1^g \leq \sigma_2^g \leq \cdots \leq \sigma_K^g$$
$$\forall i \qquad {\sigma_i^l}^2 + {\sigma_i^g}^2 = 1$$
$$\sigma_i = \frac{\sigma_i^l}{\sigma_i^g}$$

- Linear complexity w.r.t. the volume of data (i.e. edge number)

$$O(K^2 L \cdot m)$$

Embedding Dimension (constant) — Iteration (constant) — Edge number

HOPE Algorithm

Algorithm 1 High-order Proximity preserved Embedding

Require: adjacency matrix $\mathbf{A}$, embedding dimension K, parameters of high-order proximity measurementθ.

Ensure: embedding source vectors $\mathbf{U}^s$ and target vectors $\mathbf{U}^t$.

1: calculate $\mathbf{M}_g$ and $\mathbf{M}_l$.

2: perform JDGSVD with $\mathbf{M}_g$ and $\mathbf{M}_l$, and obtain the generalized singular values $\{\sigma_1^l, \cdots, \sigma_K^l\}$ and $\{\sigma_1^g, \cdots, \sigma_K^g\}$, and the corresponding singular vectors, $\{\mathbf{v}_1^s, \cdots, \mathbf{v}_K^s\}$ and $\{\mathbf{v}_1^t, \cdots, \mathbf{v}_K^t\}$.

3: calculate singular values $\{\sigma_1, \cdots, \sigma_K\}$ according to Equation (21).

4: calculate embedding matrices $\mathbf{U}^s$ and $\mathbf{U}^t$ according to Equation (19) and (20).

JDGSVD: Iteration of Jacobi-Davidson Generalized Singular Value Decomposition

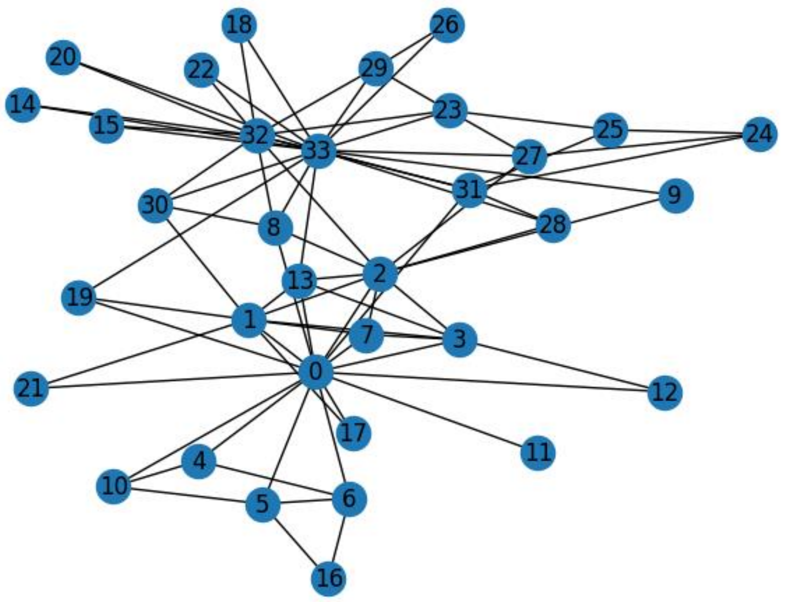
No need Stochastic Gradient Descent optimization

Focus on solving matrix factorization

Sample code: SVD on Katz index of Karate Graph

```python
import networkx as nx
import numpy as np
from scipy.sparse.linalg import svds
from scipy.sparse import csr_matrix

# Load the karate club network
G = nx.karate_club_graph()

# Define the Katz index similarity
def katz_index(G, alpha=0.5, max_iter=100, tol=1e-3):
    n = len(G.nodes())
    A = nx.adjacency_matrix(G).todense()
    I = np.eye(n)
    X = I
    converged = False
    for i in range(max_iter):
        X_prev = X
        X = alpha * A.dot(X) + (1 - alpha) * I
        if np.linalg.norm(X - X_prev) < tol:
            converged = True
            break
    return X

# Compute the Katz index similarity matrix
X = katz_index(G)

# Perform singular value decomposition on the adjacency matrix
U, s, Vh = svds(X, k=2)

# Print the singular values
print(s)

# Print the left singular vectors
print(U)

# Print the right singular vectors
print(Vh)
```

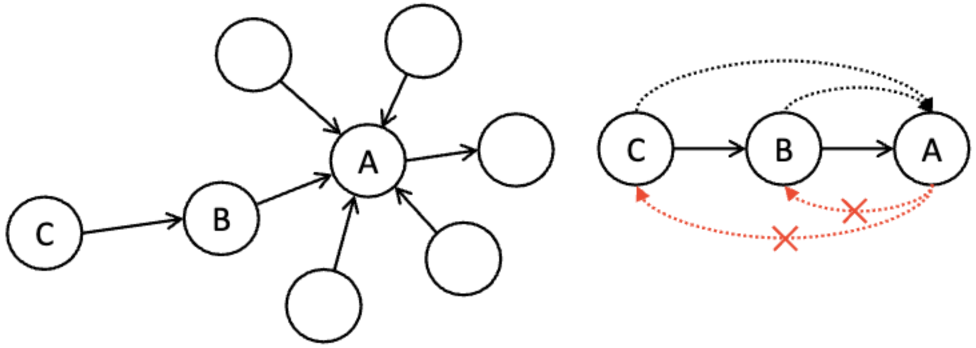
```
[2.14015007e+092 3.46416835e+103]
[[ 0.33091154 -0.31235123]
 [ 0.33411655 -0.30175117]
 [ 0.27459589 -0.36058862]
 [ 0.24818157 -0.19856719]
 [ 0.0830955  -0.05567475]
 [ 0.1082727  -0.06742808]
 [ 0.10598536 -0.06644822]
 [ 0.22455028 -0.17843234]
 [-0.03319846 -0.24695058]
 [-0.02345542 -0.05020307]
 [ 0.07224995 -0.04583322]
 [ 0.05803347 -0.04320695]
 [ 0.06286895 -0.04186974]
 [ 0.18811321 -0.24048654]
 [-0.0978487  -0.07953765]
 [-0.13735606 -0.1131142 ]
 [ 0.03757528 -0.01851886]
 [ 0.05822106 -0.04271819]
 [-0.05895455 -0.04889692]
 [ 0.05799926 -0.07342003]
 [-0.07809517 -0.06274937]
 [ 0.07775279 -0.05663175]
 [-0.09815528 -0.08100556]
 [-0.28970329 -0.21804371]
 [-0.06604098 -0.05104891]
 [-0.17460707 -0.12258229]
 [-0.08337386 -0.05862926]
 [-0.12623386 -0.14768307]
 [-0.03089142 -0.08614539]
 [-0.18759871 -0.13583302]
 [-0.08436143 -0.15831325]
 [-0.20089774 -0.20945644]
 [-0.33267    -0.33226141]
 [-0.33791409 -0.36409688]]
[[ 0.18797699 -0.05775017 -0.05189329  0.20018482 -0.05997669 -0.03472371
   0.07575835 -0.01988671 -0.12292961  0.04664735 -0.37814548 -0.05137177
   0.20501379  0.33504652  0.31253281  0.14576309  0.03485069  0.02443477
  -0.1227842  -0.13729933  0.03702881 -0.28451102  0.00194154 -0.01871236
  -0.31041159  0.16264699  0.07591105  0.05179984  0.02405201  0.27974509
  -0.17606471  0.00428549 -0.05424557 -0.33020121]
```



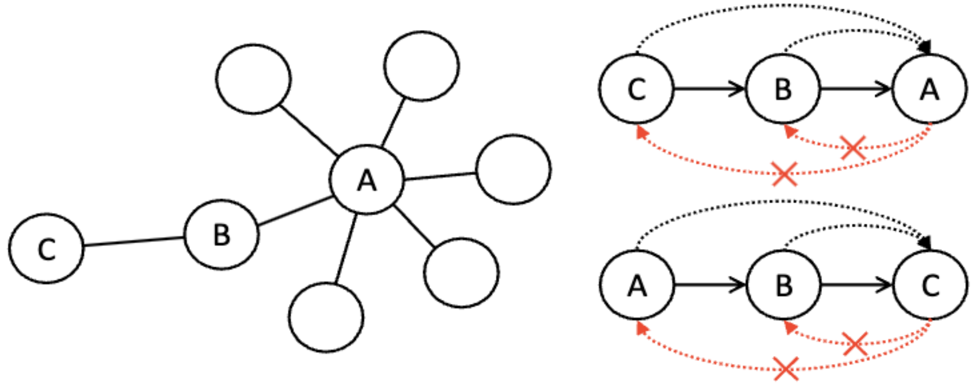
Mingdong Ou et al.: Asymmetric Transitivity Preserving Graph Embedding

APP: Scalable Graph Embedding for Asymmetric Proximity

- Motivation: Conventional methods, i.e., DeepWalk, LINE, Node2Vec can only preserve symmetric proximities between nodes
  - Insufficient for many applications where asymmetric relationships exist, even in undirected graphs
- Solution: treats each path as a directed sequence and only observes positive node pairs along the forward direction of the path
  - Conventional methods have utilized random walk to sample path as an undirected sequence



(a) Directed



(b) Undirected

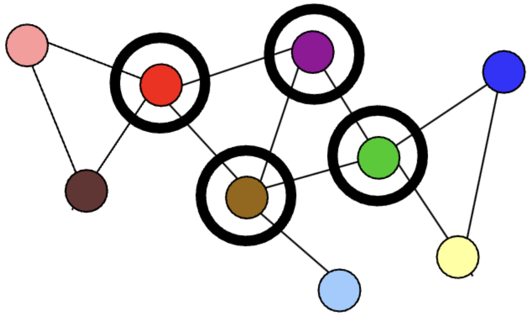
APP: Algorithm

- **Random Walk with Restart Sampling Algorithm**: Simulate the asymmetric transition probabilities between nodes
  - Perform a random walk starting from node v, with a restart probability α at each step
  - The walk terminates when a restart occurs, and the last visited node u is returned as the endpoint
  - This sampling procedure approximates the **Rooted PageRank** score between v and u
- **Stochastic Gradient Descent Algorithm**: For each sampled path p = v→ u
  - Treat this as an observed positive pair (v, u) in the forward direction only
  - Optimize the following Skip-gram style objective with negative sampling

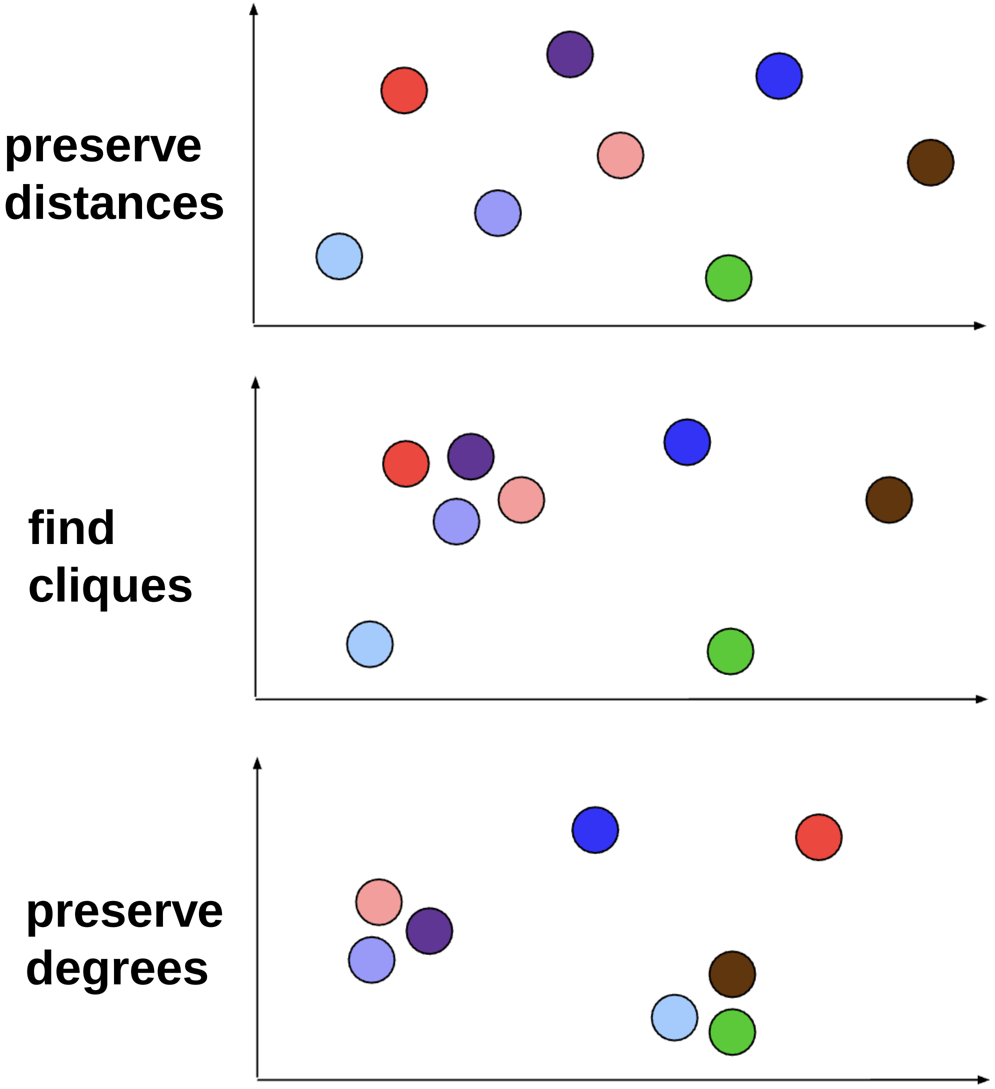
APP: Algorithm

- **Path Sharing Optimization**: Propose extracting multiple node pairs from a single sampled path, by treating all suffix sub-paths or prefix sub-paths as valid sampled paths as well
- Combine Skip-gram and SGD from DeepWalk with Negative sampling from Node2vec
- Different is proposing random walk with restart sampling algorithm and treating as directed path sequence

- Network embedding: map network nodes into Euclidean space
- Structural Identity:
  - Nodes in networks have specific roles
    - E.g., individuals, web pages, proteins, etc
  - Structural identity: identification of nodes based on network structure (no other attribute)
    - often related to role played by node
  - **Automorphism**: strong structural equivalence



Red, Green: Automorphism
Purple, Brown: Structurally similar



Leonardo F. R. Ribeiro et al: struc2vec: Learning Node Representations from Structural Identity

struc2vec: Learning Node Representations from Structural Identity

- Idea**:** Based on structural identity
- Structural similarity does not depend on hop distance
  - Neighbour nodes can be different, far away nodes can be similar
- Structural identity as a hierarchical concept
  - Depth of similarity varies
- Flexible four step procedure
  - Operational aspect of steps are flexible

struc2vec: Step 1 - Structural Similarity

- $g(D_1,D_2)$: distance between two ordered sequences
  - Cost of pairwise alignment: max(a, b) / min(a, b) - 1
  - Optimal alignment by Dynamic Time Warping (DTW) in our framework

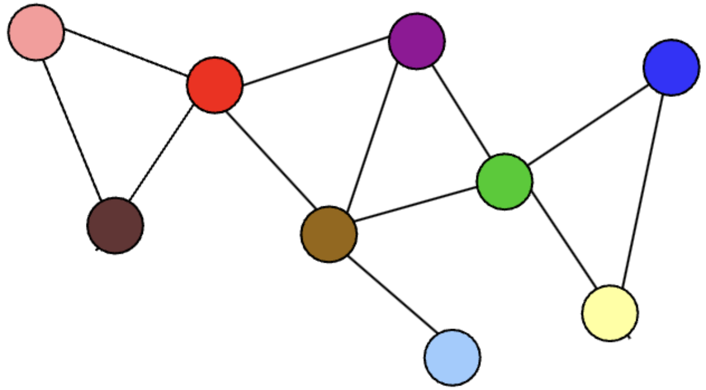
$s(R_0(u)) = 4$	$s(R_1(u)) = 1,3,4,4$	$s(R_2(u)) = 2,2,2,2$
$s(R_0(v)) = 3$	$s(R_1(v)) = 4,4,4$	$s(R_2(v)) = 1,2,2,2,2$
$g(. , .) = 0.33$	$g(. , .) = 3.33$	$g(. , .) = 1$

- $f_k(u,v)$: Structural distance between nodes u and v considering first k rings
  - $f_k(u,v) = f_{k-1}(u,v) + g(s(R_k(u)), s(R_k(v)))$

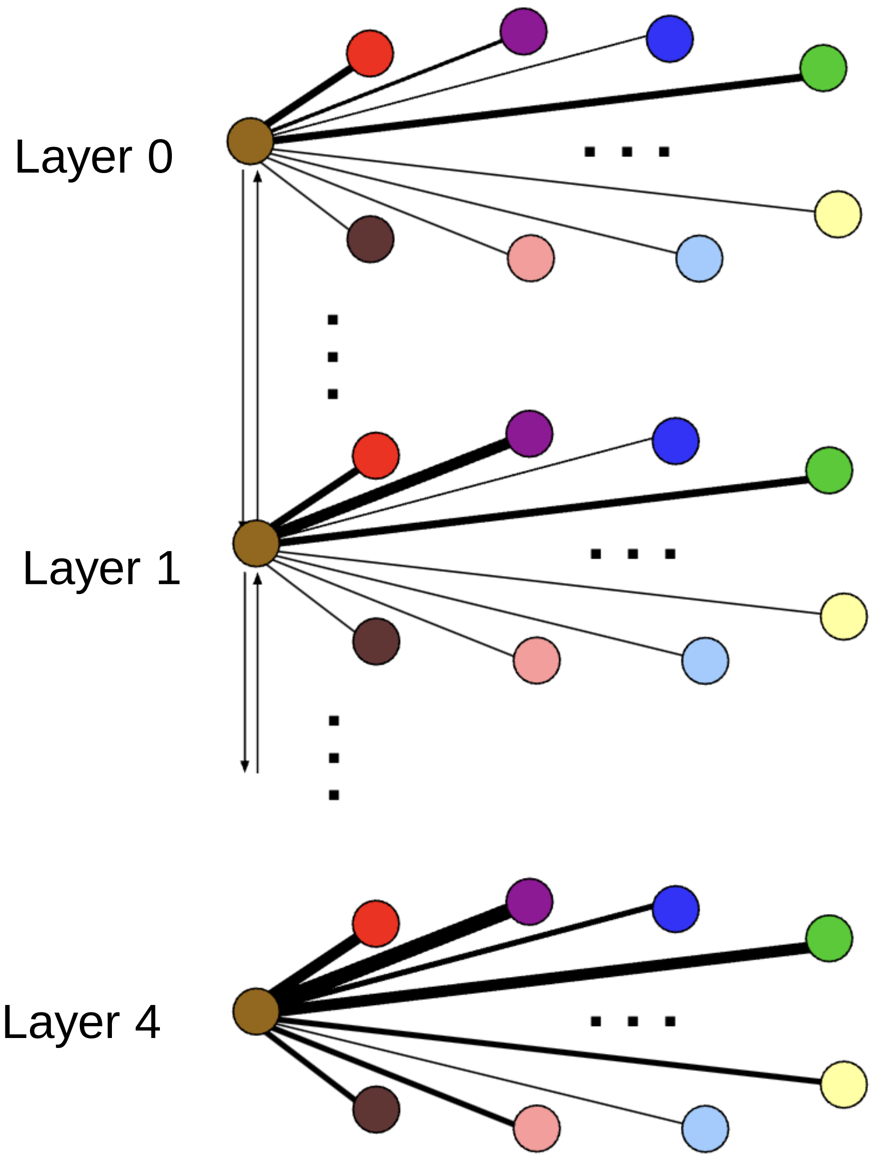
$f_0(u,v) = 0.33$ $\quad$ $f_1(u,v) = 3.66$ $\quad$ $f_2(u,v) = 4.66$

struc2vec: Step 2 - Multi-layer Graph

- Encodes structural similarity between all node pairs



- Each layer is weighted complete graph
  - Correspond to similarity hierarchies
- Edge weights in layer k
  - $w_k(u, v) = \exp\{-f_k(u, v)\}$
- Connect corresponding nodes in adjacent layers



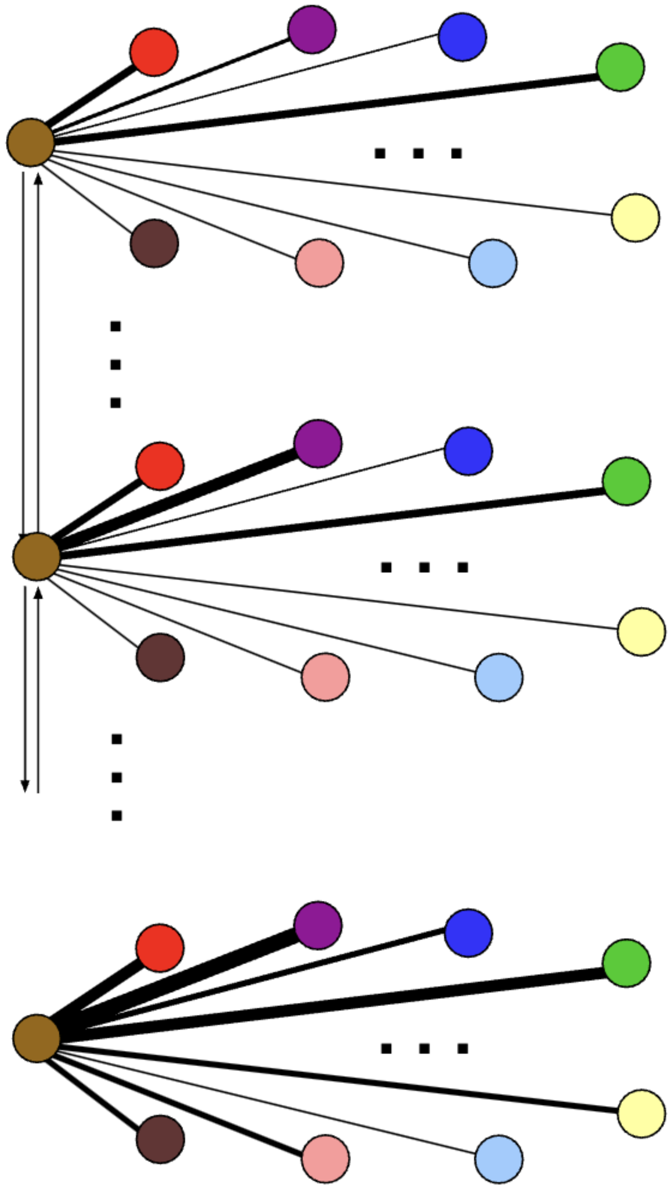
struc2vec: Step 3 - Generate Context

- Context generated by biased random walk (same as Node2vec)
  - Walk on multi-layer graph
- Walk in current layer with probability p
  - Choose neighbour according to edge weight
  - RW prefers more similar nodes
- Change layer with probability 1-p
  - Choose up/down according to edge weight
  - RW prefer layer with less similar neighbours

struc2vec: Step 4 – Learn Representation

- For each node, generate set of independent and relative short random walks
  - Context for node; sentences of a language

- Train a neural network to learn latent representation for nodes
  - Maximize probability of nodes within context
  - Skip-gram (Hierarchical Softmax) adopted

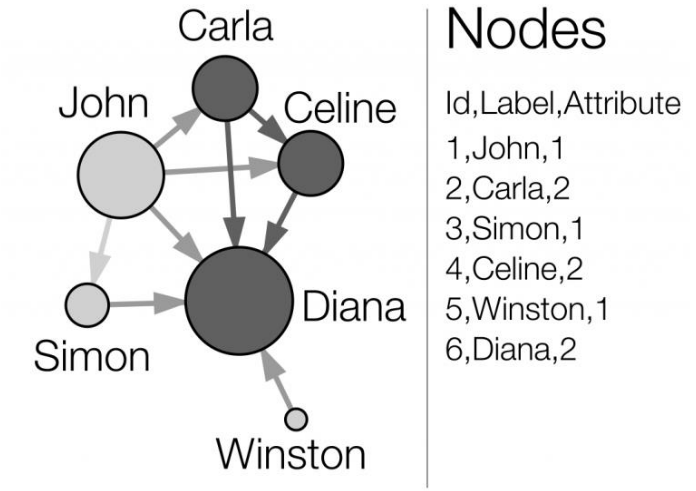


struc2vec: Optimization

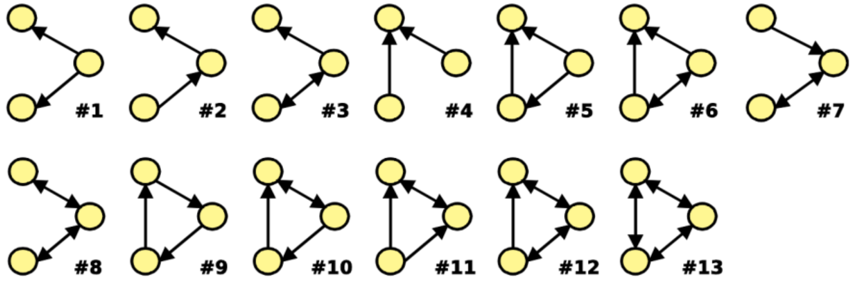
- Reduce time to generate/store multi-layer graph and context for nodes:
  - Option 1: Reduce length of degree sequences
    - Use pairs (degree, number of occurrences)
  - Option 2: Reduce number of edges in multi-layer graph
    - Only log n neighbours per node
  - Option 3: Reduce number of layers in multi-layer graph
    - Fixed (small) number of layers
  - Scales quasi-linearly
    - Over 1 million nodes

role2vec: Learning Role-based Graph Embeddings

- Ideas: struc2vec and conventional methods (DeepWalk, Node2vec) learn in node representation
  - Role2Vec learns embeddings for "node types/roles" determined by node attributes/features
- Solution: "attributed random walks"
  - Map nodes to node types/roles using node attributes like motif counts, degrees, etc
  - Generate attributed random walks over the node types instead of node IDs
  - Finally, it learns embeddings for the node types rather than individual nodes



- Given a network, most of the time, some subgraphs are "overrepresented"
- A connected graph that has many occurrences in a network is called a motif of the network
- Assume set of occurrences G' in G is $occ_G(H)$
  - Cardinality of $occ_G(H)$ in G is frequent
  - How to know if G' is frequent in G?



➡ Compute the probability that $occ_N(G') \geq occ_G(G')$ for a random network N

G' is said to be frequent in G if this probability is small enough

To compute this probability, we need to have a distribution over networks

role2vec: Algorithms

- **Input**: Graph G, node attribute matrix X, embedding dimension D, number of walks per node R, walk length L, context window size ω
- If X is not available, extract structural features like motif counts from the graph structure itself and use those as node attributes in X.
- Apply logarithmic binning to the node attribute values in X. Map nodes to node types/roles using a function $\phi(x)$ that takes the node attribute vector x as input
  - Two types of ϕ functions:
    - Simple functions like concatenation of attribute values
    - Low-rank matrix factorization of X
- Precompute random walk transition probabilities π. Generate R attributed random walks of length L for each node, using the node type mapping ϕ instead of node IDs
- Learn embeddings using stochastic gradient descent on the attributed random walks, optimizing the probability of observing the context node types in the walks
- **Output**: Learned embeddings for each node type w ∈ W, where W is the set of node types found by ϕ

role2vec: Different from Previous Methods

- Mostly same as Node2Vec, except their work are considered in terms of node attribute, not node representation

- There are many advantage from node attributed embedding:
  - Embeddings generalize to new nodes/graphs (inductive learning)
  - Better captures structural node roles
  - Space-efficient as it learns fewer embeddings for node types instead of all nodes
  - Supports attributed graphs

Nesreen K. Ahmed et al: Learning Role-based Graph Embeddings

네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA