# Random Walk-based Graph Representation Learning

Prof. O-Joun Lee

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

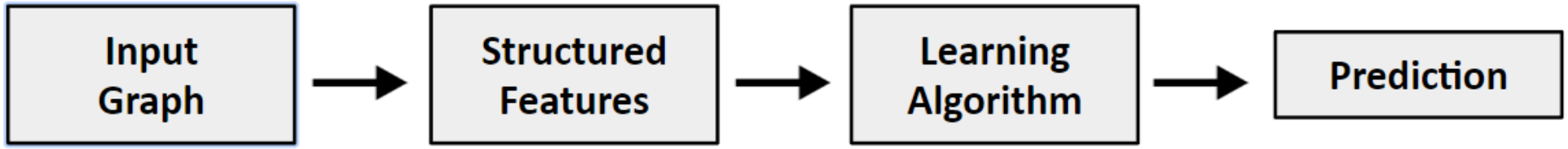
네트워크과학연구실 NETWORK SCIENCE LAB

가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

# Contents

- Graph Representation Learning Introduction
  - Traditional Machine Learning for Graphs
  - Graph Representation Learning
  - Node Embedding and Shallow Encodings
- Random-walk based methods:
  - Deep Walk
  - Node2Vec
  - Div2Vec
  - Node2Vec+
  - WalkLet

- Given a graph, we can extract features (node-level, graph-level) from the graph, then directly put them to a shallow model to map the features to the ground truth



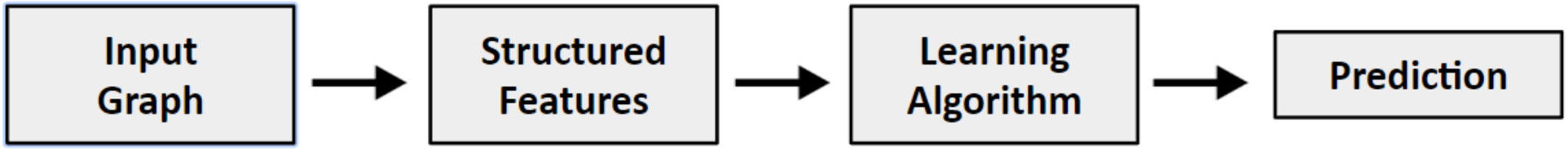

## Feature Engineering

- Node feature
- Edge feature
- Graph feature

- SVM
- Random Forest
- XGBoost
- DNN

- Node-level
- Edge-level
- Graph-level

- GRL aims to generate graph representation vectors that describe graph structures
- We don't need to do feature engineering every single time




**Feature Engineering**

**Representation Learning**

learn the features by itself

- SVM
- Random Forest
- XGBoost
- DNN

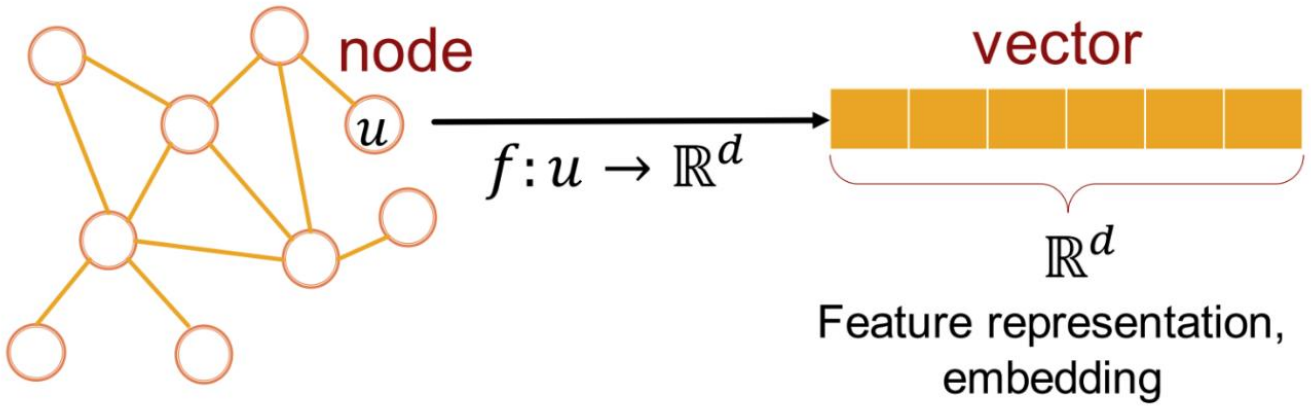
- Node-level
- Edge-level
- Graph-level

# Why Embeddings?

- GRL's goal: Learn efficient task-independent feature for ML with graphs

  → Map nodes into an embedding space

- Requirements: Similarity of embeddings between nodes indicates their similarity in the network
- For simplicity, no node features or extra information is used




**Tasks**

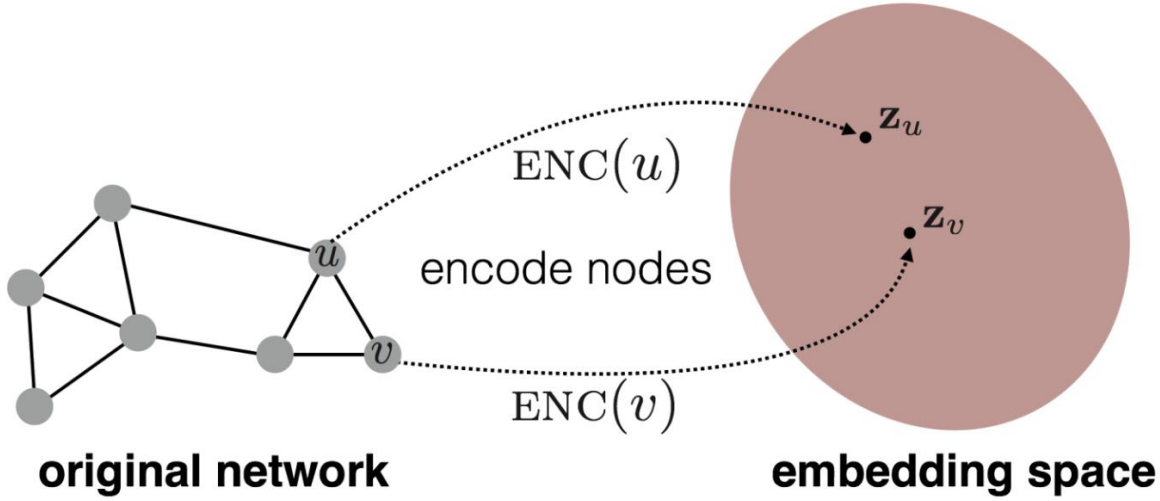
- Node classification/regression
- Edge prediction
- Graph classification/regression
- Graph clustering
- …

# Node Embeddings

- Goal: Mapping nodes from the graph to the embedding space *w.r.t* the similarity between two nodes in the embedding space (via dot product) ≈ similarity between them in the original graph

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

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




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

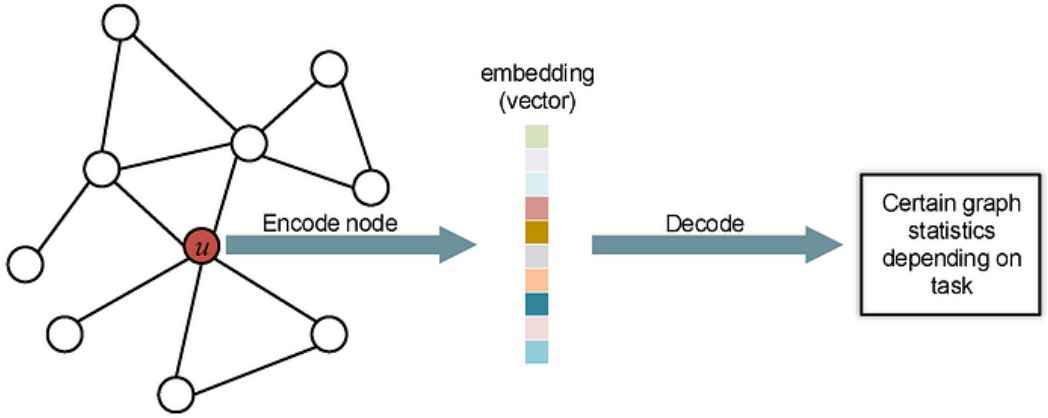
# Learning Node Embeddings

- Encoder: Mapping nodes → embeddings (d-dimensional vector, ENC(v) = $Z_v$)
- Decoder: Mapping embeddings → similarity score (dot product)
- Define a node similarity function:
  - Whether 2 nodes are close in the graph, and how close are they?

Dot product between node embedding

- Optimize → $\underbrace{\text{similarity}(u, v)}_{\text{in the original network}} \approx \underbrace{\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u}_{\text{Similarity of the embedding}}$

  - Maximize $\boldsymbol{z}_v^T \boldsymbol{z}_u$ for node pairs $(u, v)$ that are similar

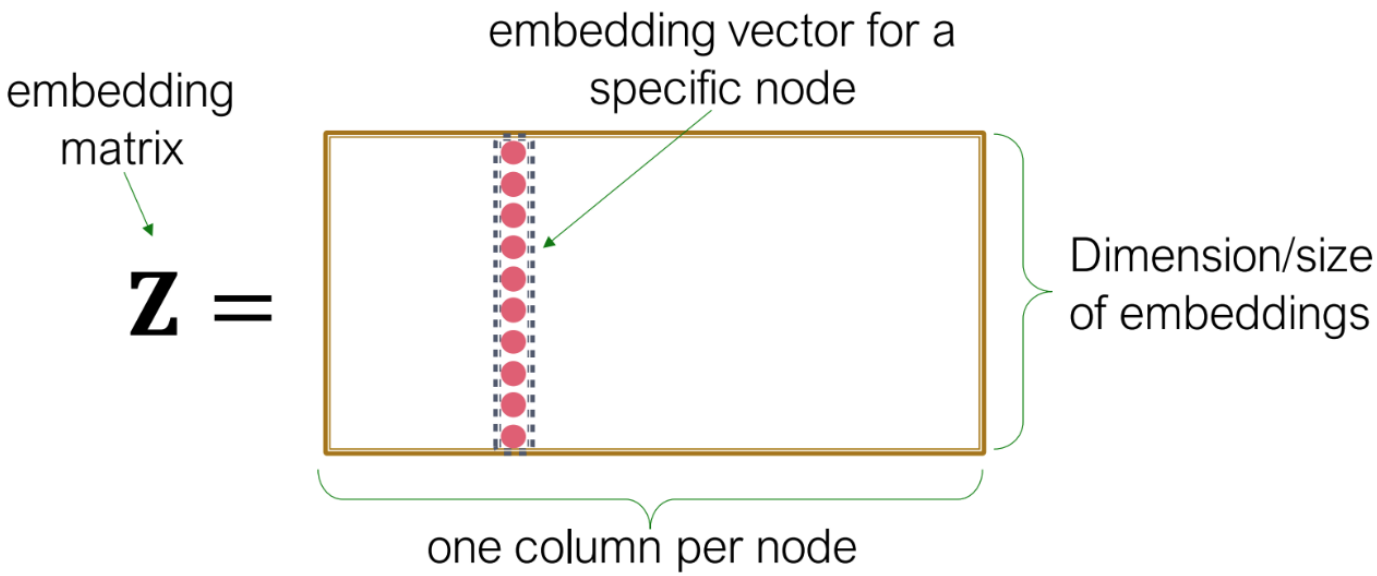



https://medium.com/the-modern-scientist/graph-neural-networks-series-part-3-node-embedding-36613cc967d5

# Shallow Encoding Strategies

- How to learn node embeddings?
- Simplest encoding approach: Build an embedding lookup table

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




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

matrix, each column is a node embedding (this is what we want to learn)

$$v \in \mathbb{I}^{|\mathcal{V}|}$$

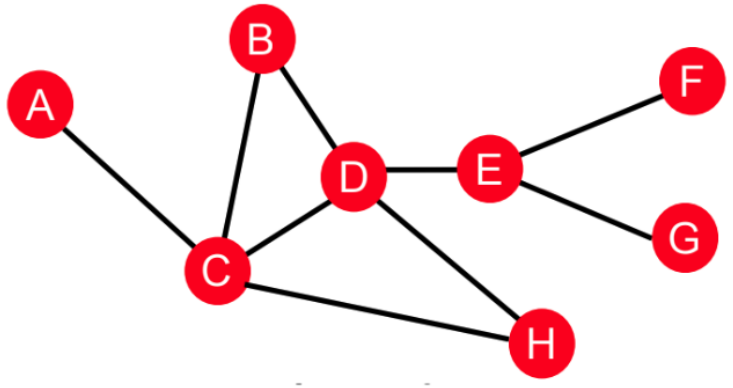
indicator vector, all zeros except a one in column for node v

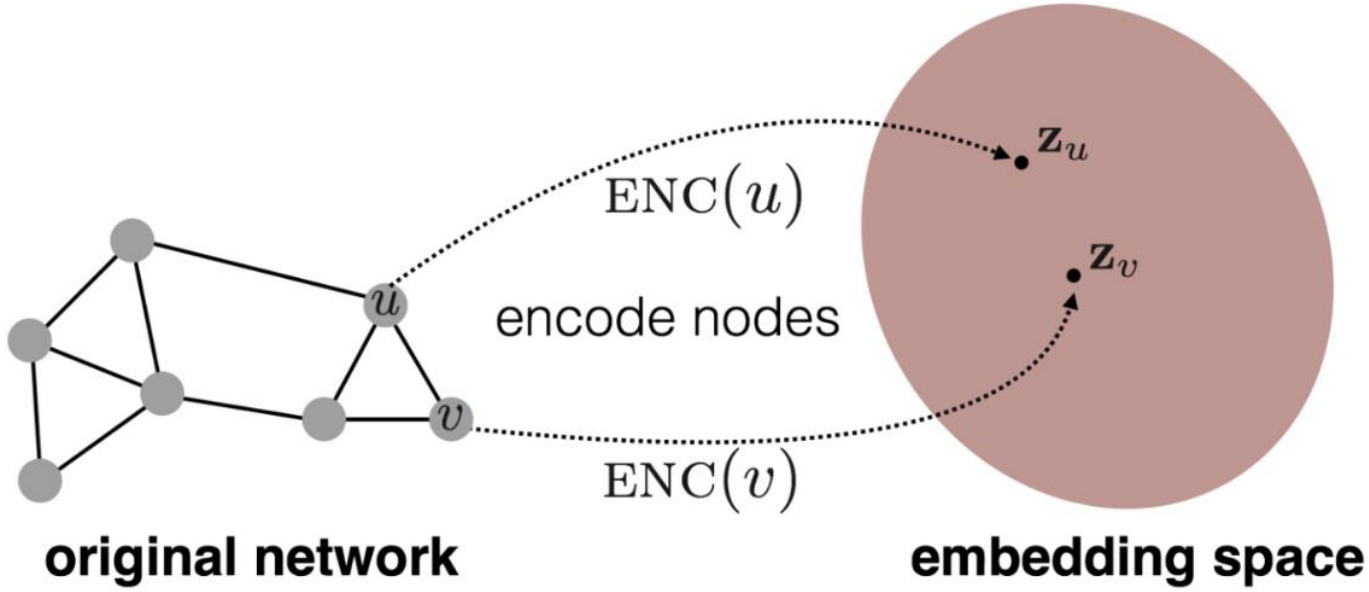
# Shallow Encoding Strategies

- Each node is assigned a unique embedding vector

→ We **directly optimize the embedding of each node**

- Embedding is optimized to maximize $\boldsymbol{z}_v^T \boldsymbol{z}_u$ for each similar node pairs (u, v)
- Key choice: How to define node similarity? Should two nodes have similar embedding if:
  - They are linked?
  - They share neighbours?
  - They have similar structure roles?



# Adjacency-based Approaches

- One of the simplest and most intuitive approaches to define similarity:
  - Adjacency between two nodes v and u
- Similarity: Two nodes are adjacent to one another within the structure of the graph
- Encoding: Find the embedding matrix **Z** that minimizes the loss function **L**

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

loss (what we want to minimize)

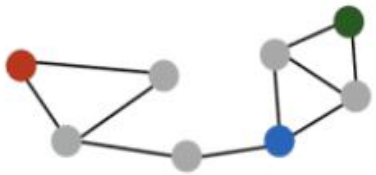
sum over all node pairs

embedding similarity

(weighted) adjacency matrix for the graph

- Limitation: fails to capture the similarity between distant nodes

How to capture the global graph structures?



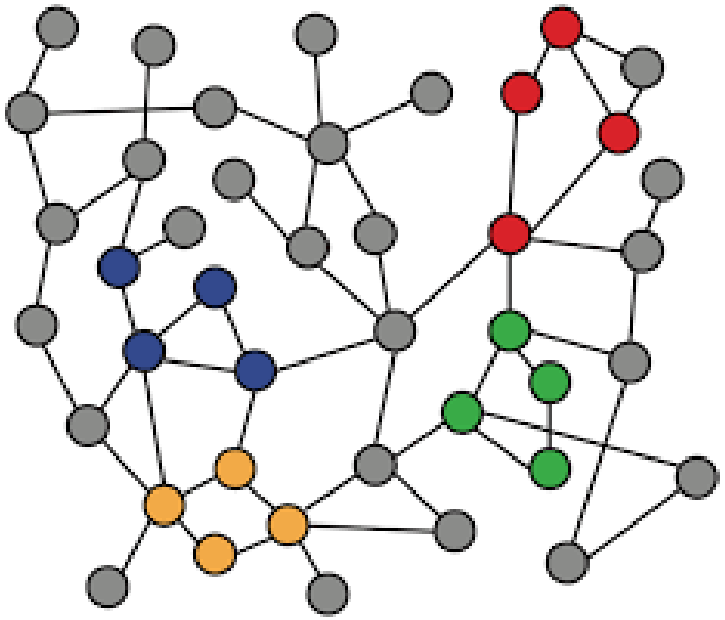
e.g., the blue node is obviously more similar to green compared to red node, despite none having direct connections

https://www.graphable.ai/blog/knowledge-graph-embeddings/

# Random Walk-based Methods for Node Embeddings



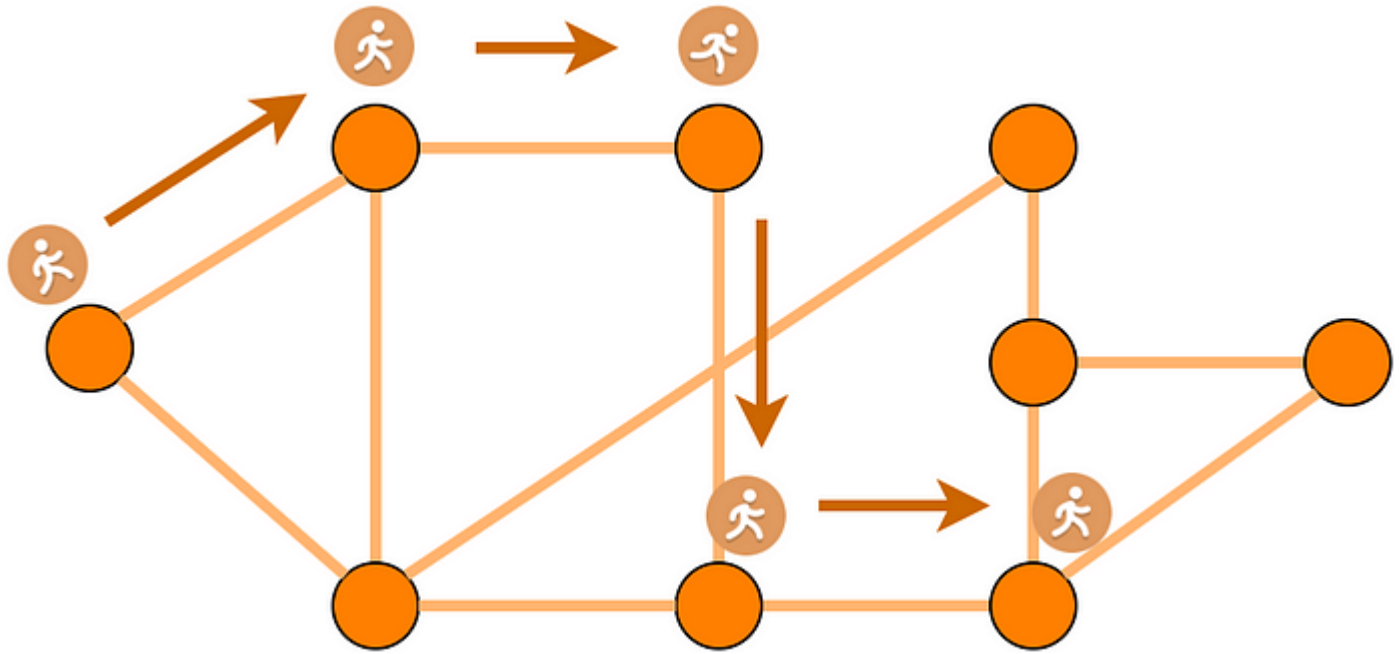
- Random Walk
- Learning Objective & Method
- Representatives:
  - DeepWalk
  - Node2vec
  - Div2Vec
  - Node2vec+
  - WalkLet



# Random Walk



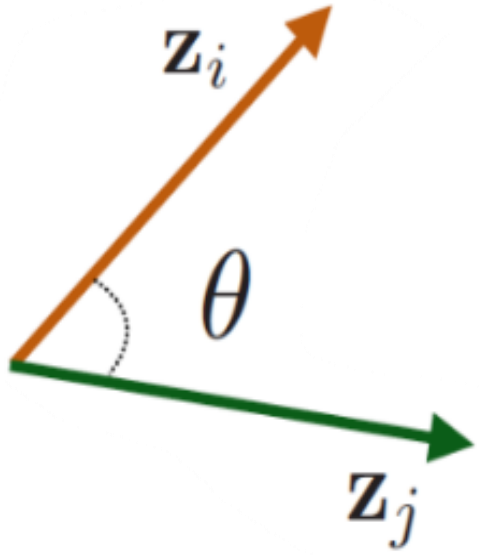
- Given a graph and a starting point, we select a neighbour of it at random, and move to this neighbour
- Then, we select a neighbor of this point at random, and move to it,…
- The random sequence of nodes visited this way is **a random walk on the graph**



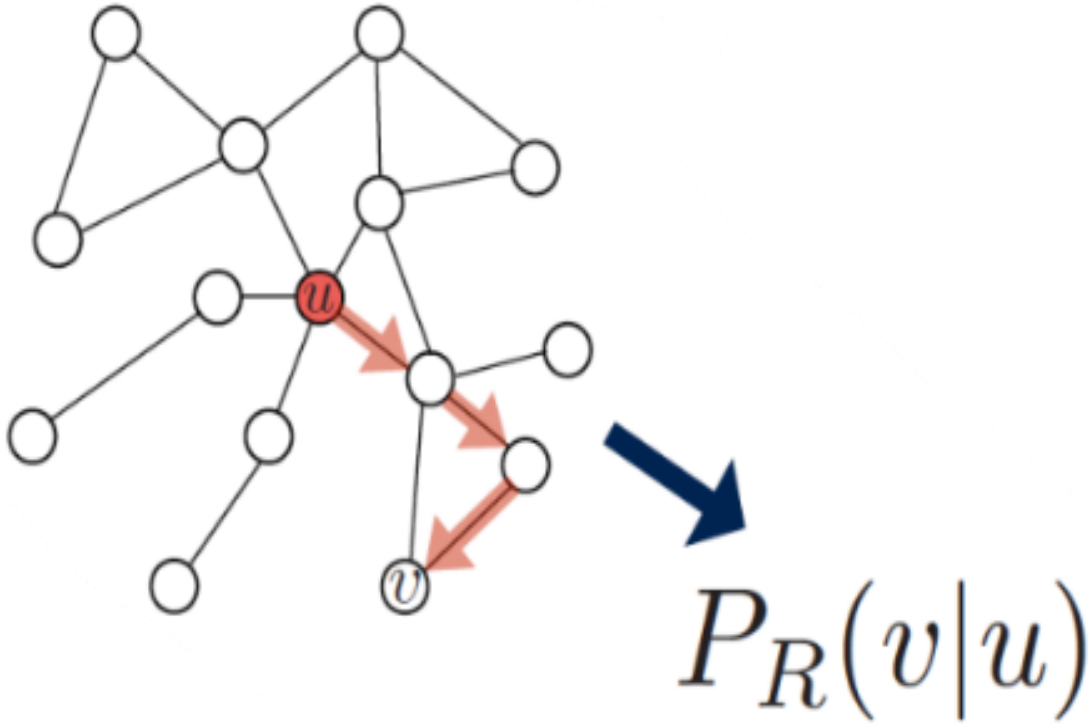
https://medium.com/analytics-vidhya/an-intuitive-explanation-of-deepwalk-84177f7f2b72

# Random Walk Embedding

- Estimate probability of visiting node v on a random walk starting from node u with strategy R: $P(v|\boldsymbol{z}_u)$
- Optimize node embeddings to learn the statistics from random walks




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

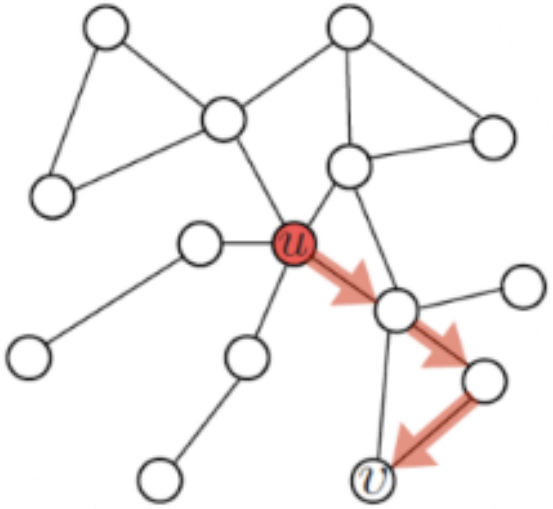


# Why using Random Walks?



- We can simply define nodes are similar if there are connected, why random walks?

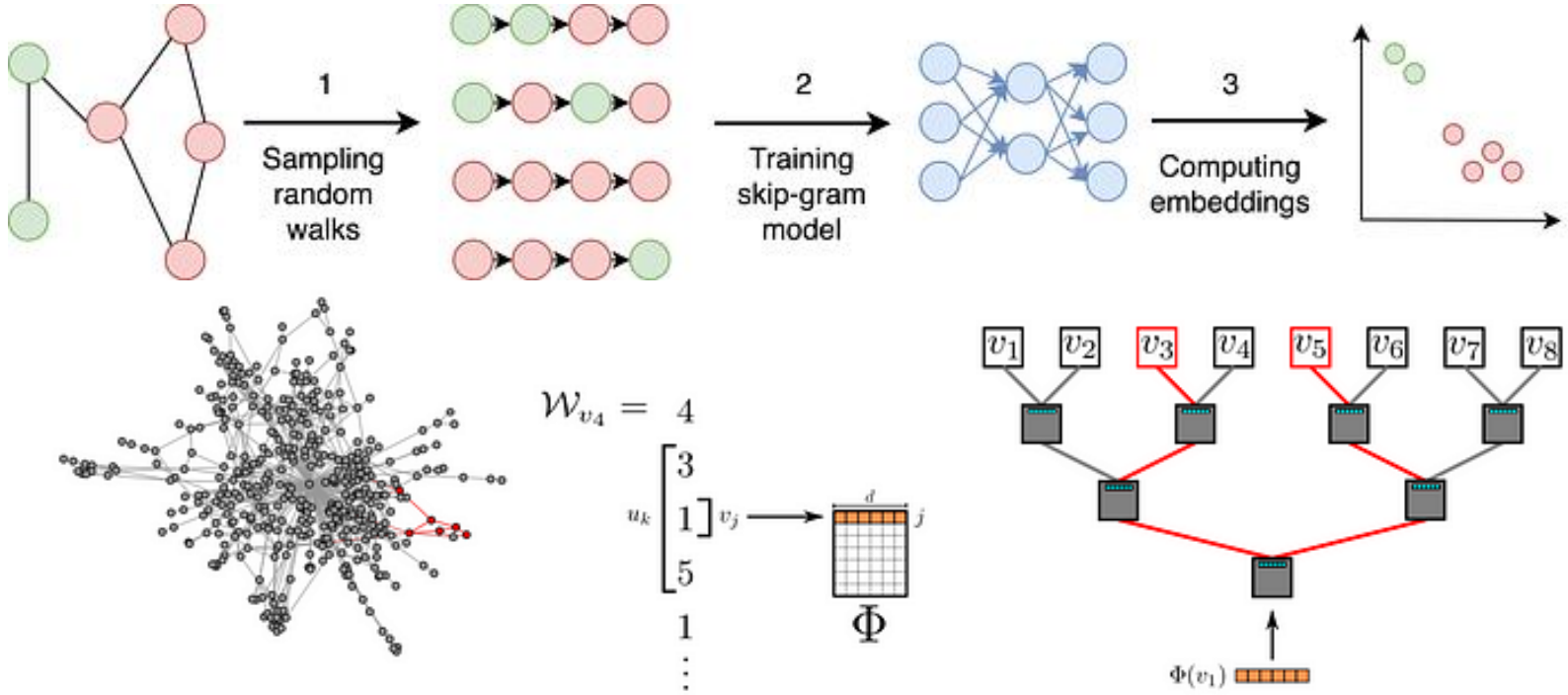
- **Expressivity**:
  - Random walk incorporates both local and higher-order multi-hop neighbourhood information (Global structures)




- **Efficiency**:
  - Don’t need to consider all node pairs at training state; only need to consider node pairs on the random walks

# DeepWalk: Randomly Walk

- What sampling strategies should we use to explore the graph structure?
- Simplest idea:
  - Run fixed-length, unbiased random walks starting from each node (i.e., DeepWalk from Perozzi et al., 2013)
  - Find embeddings $\boldsymbol{z}_u$ that minimizes $\mathcal{L}$




(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

Bryan Perozzi et al: DeepWalk: Online Learning of Social Representations.

# DeepWalk: Feature Learning as Optimization

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

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

$N_R(u)$ is the neighborhood of node u by random walk R

→ learn feature representations that are predictive of the nodes in its random walk neighborhood $N_R(u)$

- Equivalently, loss function:

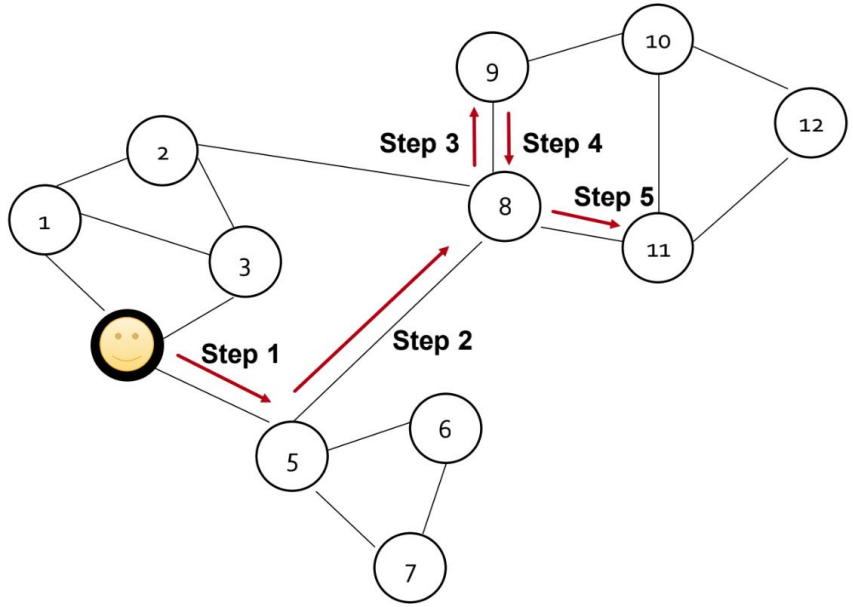
$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

- Parameterize using Softmax:

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

# DeepWalk: Randomly Walk

- Stage 1: Local structure discovery
  - Goal: discover neighborhoods in the network, where assumption is that adjacent nodes are similar and should have similar embeddings
  - Consider a fix-length of each walk $l$
  - Generate a fixed number $k$ of random walks starting at each node
  - When it is finished, we obtain k node sequences of length $l$ or collect the visited node set $N_R(u)$ for each node $u$

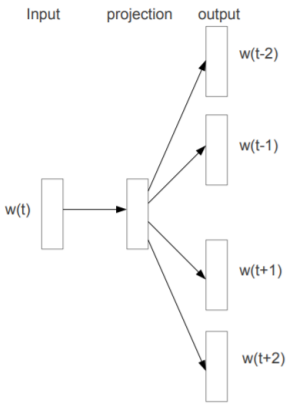



Bryan Perozzi et al: DeepWalk: Online Learning of Social Representations.

# DeepWalk: Skip-Gram Procedure

- Stage 2: Skip Gram
  - Idea: Borrowing idea from NLP
    - Goal: Given a corpus and a window size, SkipGram aims to maximize the similarity of word embeddings of the words that occur in the same window. (window = context in NLP)
    - Assumption: Words that occur in the same context, tend to have close meanings. Therefore, their embeddings should be close to each other as well
  - In Graph: Each walk produced in the previous step as a context or word window in a text
    - Equivalent to node sequences in networks correspond to word sequences in text

- Algorithm:
  - Generate random vectors of dimension d for each node
  - Iterate over the set of random walks and update the node embeddings by gradient descent




Tomas Mikolov et al: Distributed Representations of Words and Phrasesand their Compositionality

# 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:

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

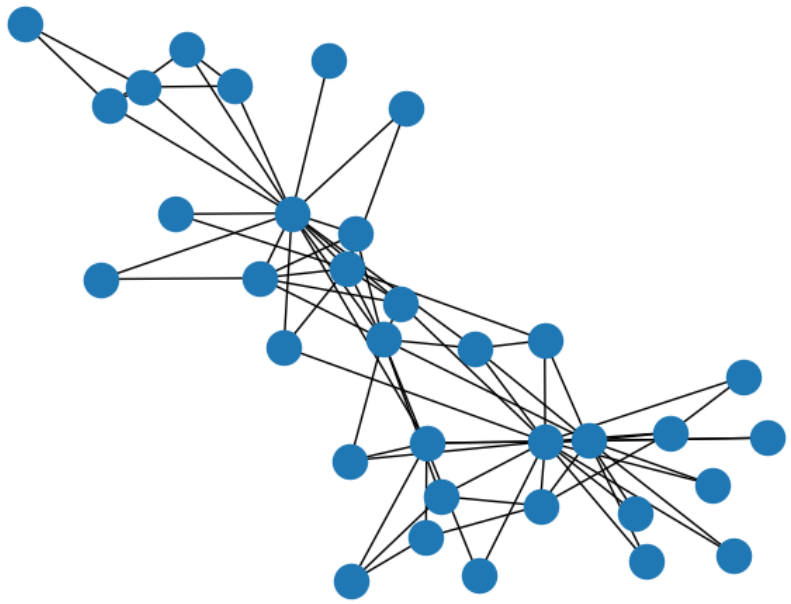
Learning rate

# Random Walk: Sample code with Karate Zachary



```python
import networkx as nx
import random

# Import the Karate Club graph using NetworkX and create an adjacency matrix
karate_graph = nx.karate_club_graph()
adjacency_list = nx.adjacency_matrix(karate_graph, dtype=int)
adjacency_matrix_array = adjacency_list.toarray()

def random_walk(adj_list, node, walk_length):
  walk = [node]        # Walk starts from this node

  for i in range(walk_length-1):
    node = adj_list[node][random.randint(0,len(adj_list[node])-1)]
    walk.append(node)

  return walk

# Perform random walks on the graph
num_walks = 6

for node in karate_graph.nodes():
  print("Node " + str(node) + " :" + str(random_walk(adjacency_matrix_array, node, num_walks)))
```

```
Node 0 :[0, 2, 5, 0, 0, 0]
Node 1 :[1, 5, 0, 0, 0, 0]
Node 2 :[2, 0, 0, 3, 0, 0]
Node 3 :[3, 3, 3, 0, 2, 4]
Node 4 :[4, 0, 0, 0, 2, 5]
Node 5 :[5, 0, 2, 0, 0, 0]
Node 6 :[6, 0, 0, 0, 3, 3]
Node 7 :[7, 0, 2, 5, 0, 0]
Node 8 :[8, 0, 5, 0, 0, 2]
Node 9 :[9, 0, 0, 3, 0, 0]
Node 10 :[10, 0, 0, 2, 0, 3]
Node 11 :[11, 3, 0, 2, 0, 3]
Node 12 :[12, 0, 2, 0, 2, 2]
Node 13 :[13, 0, 0, 0, 2, 0]
Node 14 :[14, 3, 0, 2, 0, 3]
Node 15 :[15, 0, 0, 3, 3, 0]
Node 16 :[16, 0, 3, 3, 0, 2]
```



# Limitation of DeepWalk

- Note:
  - Frequency of co-occurrence in random walks is an indicator of node similarity
  - Effect of $k$ (# walks) and $l$ (length) is important:
    - more $k$ is increased, the more the network is explored
    - $l$ is increased, paths become longer, and more distant nodes are accepted as similar nodes

- **Issue:** Expensive from nested sum over nodes($O(|V|^2)$)

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

sum over all nodes $u$

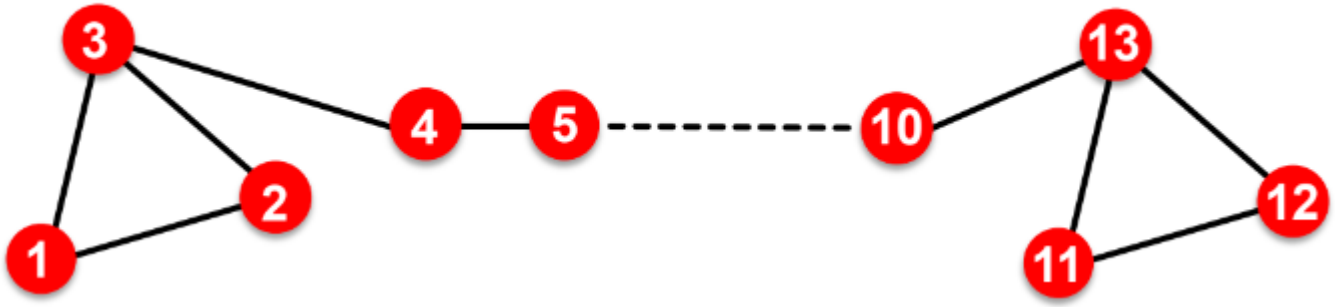
sum over nodes $v$ seen on random walks starting from $u$

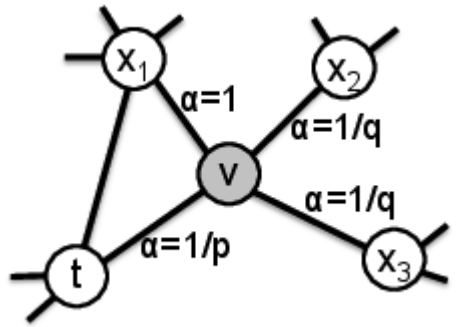
predicted probability of $u$ and $v$ co-occuring on random walk

# Node2Vec: Biased Walks



- **Goal**: Embed nodes with similar network neighbourhoods close in the embedding space and control the next steps in the random walk
- **Key**: Develop different strategies to capture the local and global graph structures, which depends on the specific graphs.
- **Learning method**: Maximum likelihood optimization problem

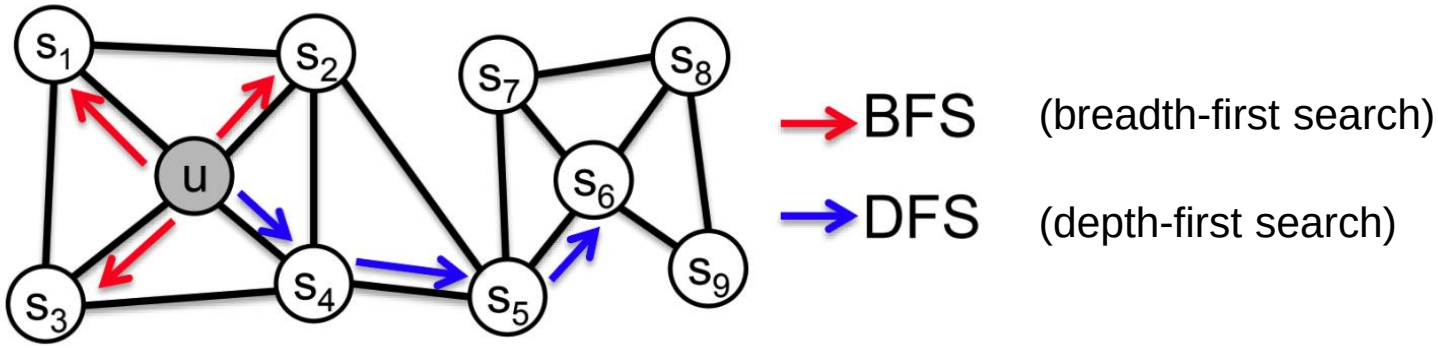



node2vec: Scalable Feature Learning for Networks

# 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)
- Two classic strategies to define a neighborhood $N_R(\boldsymbol{u})$ of a given node u




- Walk of length ($N_R(\boldsymbol{u})$ of size 3):

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

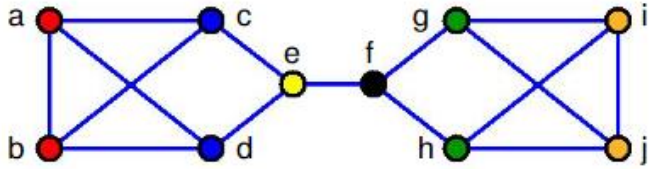
$$N_{DFS}(u) = \{s_4, s_5, s_6\}$$ **Global view**

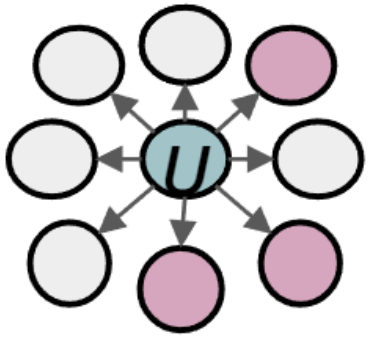
node2vec: Scalable Feature Learning for Networks

# Node2Vec: Equivalence with Interpolating BFS and DFS

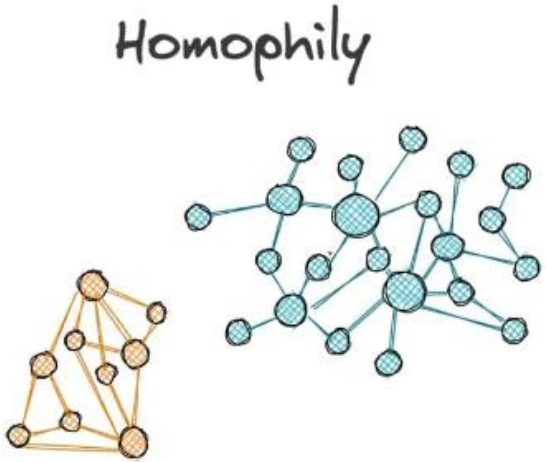


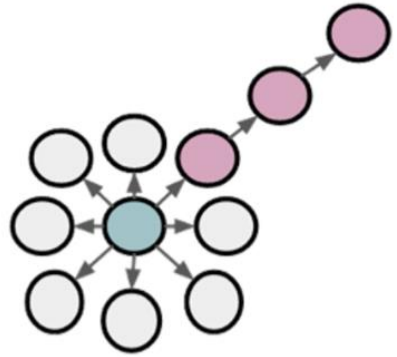
- **Structural Equivalence:** if and only if they have the same structure
- **Homophily:** tendency of individuals to associate and bond with similar others
  - Neighborhoods sampled by **BFS** corresponding to **structural** equivalences
  - **DFS** explore macro-view of the network, which infers communities: **homophily** equivalence




BFS



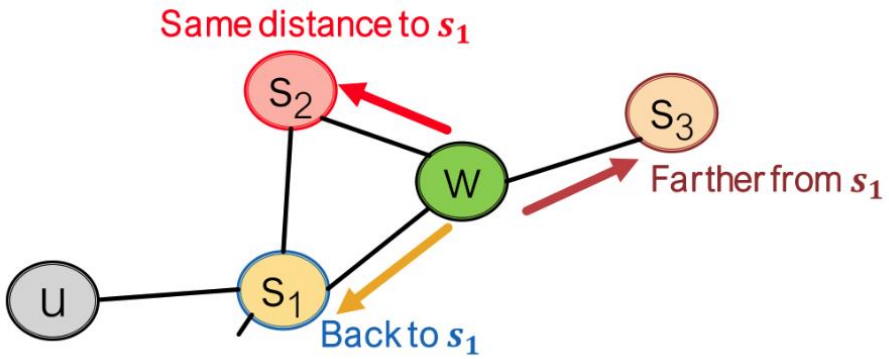
DFS

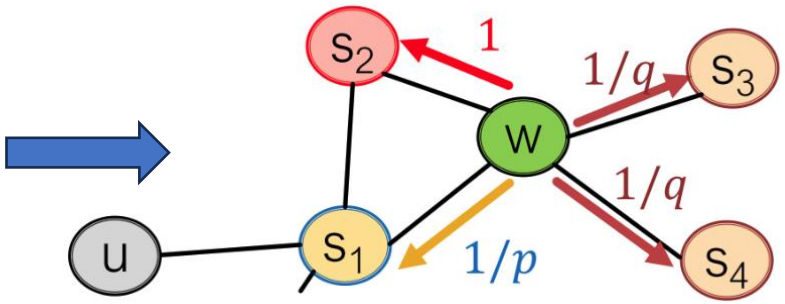
- Biased fixed-length random walk $R$ that given a node $u$ generates neighborhood $N_R(u)$
  - Two parameters of random walks:
    - Return parameter p: Return to the previous node
    - In-out parameter q: Moving outwards (DFS) vs. inwards (BFS) from the previous node

SERGEY IVANOV: GML Newsletter: Homophily, Heterophily, and Oversmoothing for GNNs

# Node2Vec: Biased Random Walk

- Two strategies to explore network neighborhood: BFS and DFS
- Walker just traversed edge $(s_1, w)$ and is at w, now he/she can go using a bias probability:




**1/p, 1, 1/q are unnormalized probs.**
**p: return parameter.**
**q: "walk away" parameter**

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

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

# Node2Vec: Algorithm



- Compute random walk probabilities:
  - For each edge $(s_1, w)$ we compute walk probabilities (based on $p, q$) of $(w, .)$
- Simulate r random walks of length $l$ starting from each node u
- Optimize the node2vec objective using stochastic gradient descent with negative sampling

→ The training process is same as DeepWalk, except the walking strategy

# Node2Vec: Reminding of DeepWalk Optimization

- Problem: Expensive in summing over nodes ($O(|V|^2)$)

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

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

- The normalization term from the softmax is the culprit
- Solution: **Negative Sampling**:
  - normalize against $k$ random "negative samples" $n_i$

# Node2Vec: Optimization with Negative Sampling

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

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

Sigmoid function

random distribution over nodes

- Sample $k$ negative nodes $n_i$ each with probability proportional to its degree
- Higher $k$ gives more robust estimates
- In practice, $k$ from 5 to 20

# Node2Vec: 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:**
  - 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

# DeepWalk and Node2Vec: Limitation on diversity



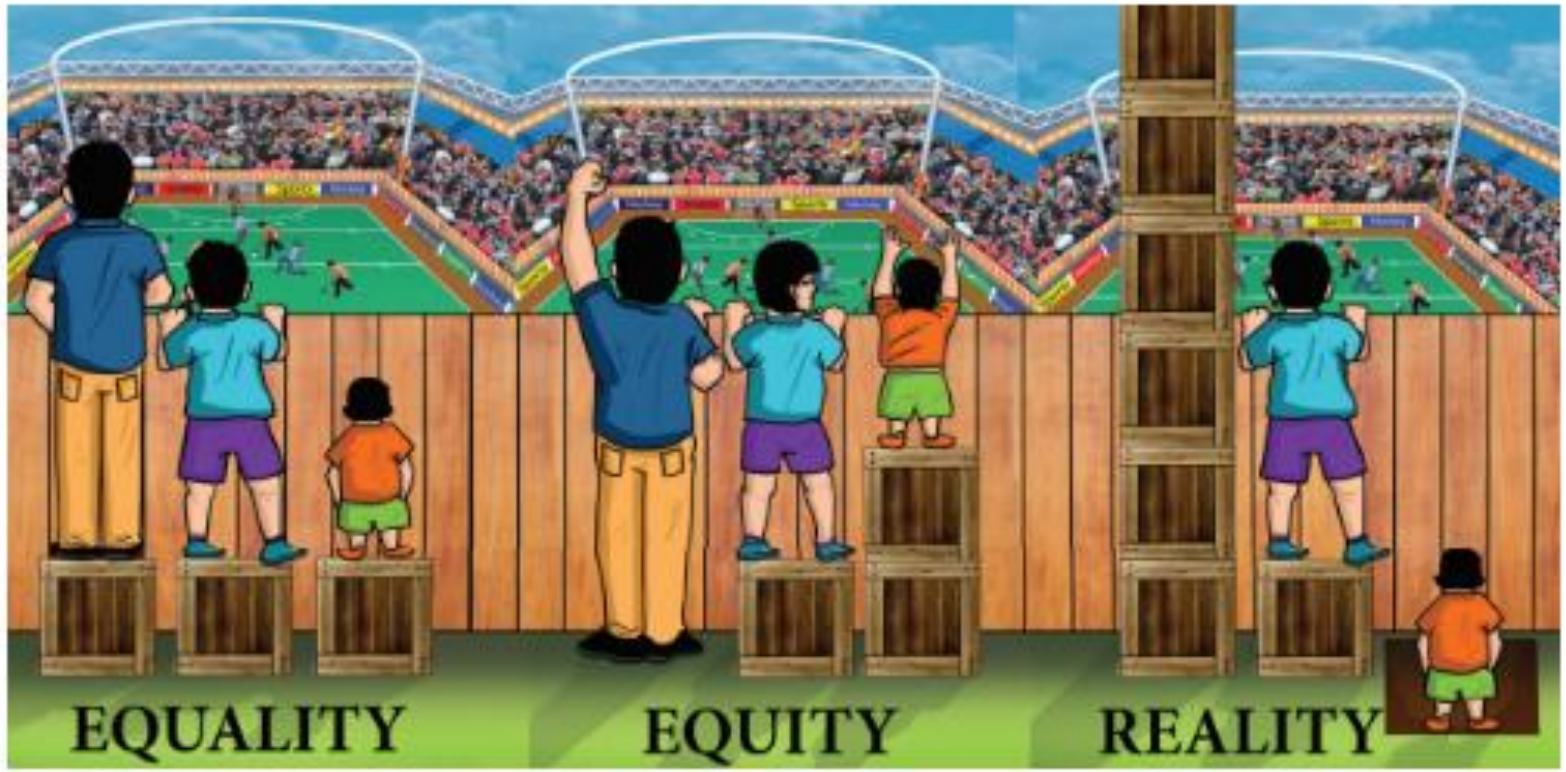
- DeepWalk and Node2Vec do not consider the roles of nodes in the network
  - High degree nodes and node degree are considered equally
  - Problems:
    - For high degree nodes: it may ignore surrounding structures of high degree nodes
    - For low degree nodes: it may cause over-sampling
- Solutions:
  - Sample nodes with high degree more and nodes with low degree less

Jisu Jeong et al: div2vec: Diversity-Emphasized Node Embedding

# Div2Vec: Diversity-Emphasized Node Embedding



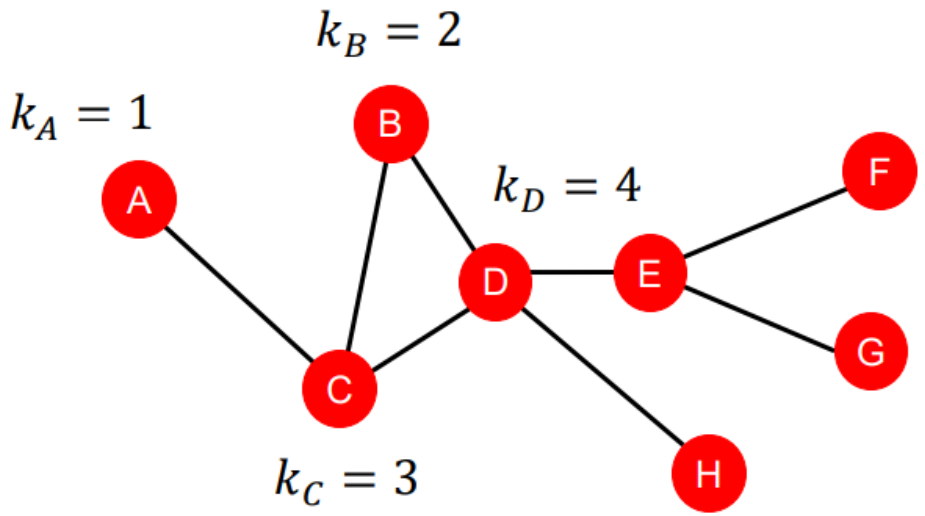
- **Ideas**: degree bias makes it harder for the embeddings to capture the preferences of users for niche or less popular items, limiting diversity and personalization
- **Key**: Instead of sampling nodes proportional to their degree when generating random walks, only samples nodes inversely proportional to their degree




Jisu Jeong et al: div2vec: Diversity-Emphasized Node Embedding

# Div2Vec: Node Degree

- The degree $k_v$ of node $v$ is the number of edges (neighboring nodes) the node has
  - In-degree: Number of incoming edges to the node
  - Out-degree: Number of outgoing edges from the node
  - Total degree: Sum of in-degree and out-degree
- Treats all neighboring nodes equally



# Div2Vec: Methods

- Like Node2Vec's algorithm, the different is how they set the weight for each walk and the choose of next node
- For $x \in N_R(v_i)$, the weight $w(v_i, x)$ is set:

$$w(v_i, x) = \begin{cases} \frac{1}{p} & \text{if } x = v_{i-1}, \\ 1 & \text{if } x \text{ is adjacent to } v_{i-1}, \\ \frac{1}{q} & \text{otherwise.} \end{cases}$$

- The probability chooses next node u $\in N_R(v)$:

$$\frac{f(\deg(u))}{\sum_{w \in N(v)} f(\deg(w))}$$

Where function f is defined: $\frac{1}{x}$ or $\frac{1}{\sqrt{x}}$ if x has two neighbors y and z which y or z is choose based on lower degree

# Limitation of Div2Vec



- Tradeoff between accuracy and diversity
  - $\frac{1}{x}$ : focus on diversity
  - $\frac{1}{\sqrt{x}}$ : focus on accuracy
- The choose should be depending on the dataset and task

# Node2Vec+

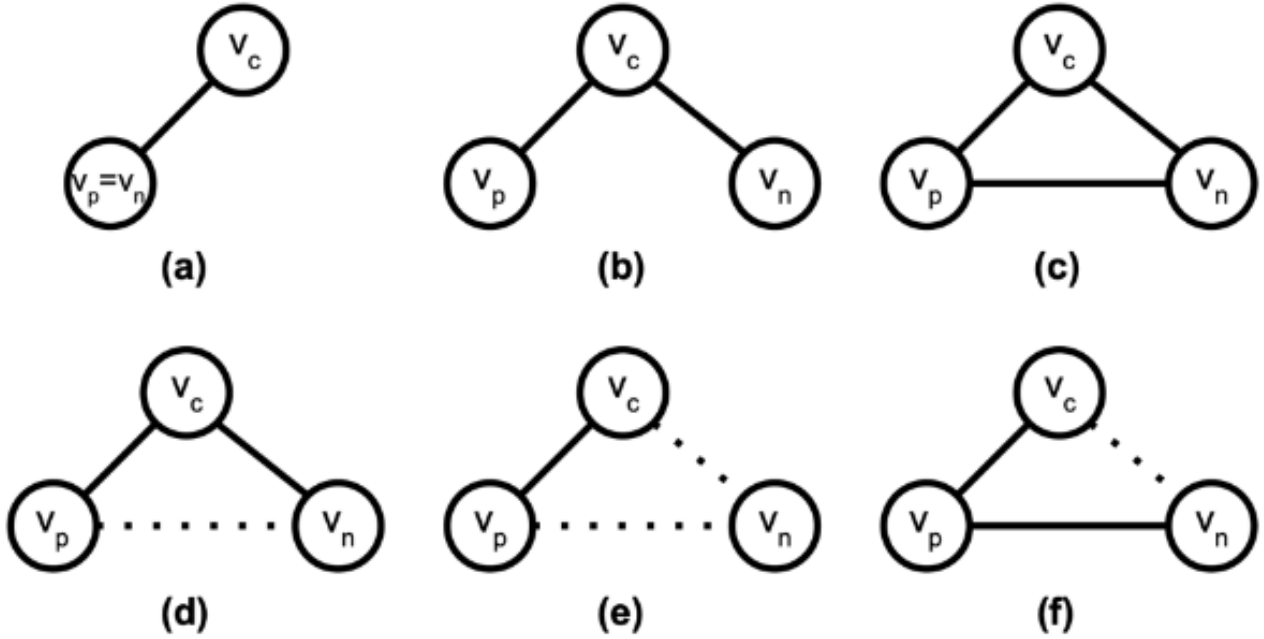


- **Problem**: Biology graph like gene interaction networks, are dense weighted graphs
  - Node2vec can fail in dense graphs:
    - Node2vec cannot effectively utilize the edge weights when generating the biased random walks that underline the embedding process
  - Treats all edges equally and does not differentiate between edges
- **Idea**: Identify potential edge type in weighted graph
- **Solution**: Same as Node2Vec, only different is edge walking strategy

Bryan Perozzi et al: Accurately modeling biased random walks on weighted networks using node2vec+

# Node2Vec+: Edge Types

- There are 3 types of edges:
  - (a): return edge, where the potential next node is the previous node
  - (b): out edge, where the potential next node is not connected to the previous node
  - (c): in edge, where the potential next node is connected to the previous node
- (d-f): Variations of (c) when considering of edge weights:
  - (d), (e), (f): out-edge, but node2Vec will decide in-edge



# Node2Vec+: Edge Weight

- Determine the looseness of $(v_c, v_n) \in E$ based on edge weight statistics for each node $v$

$$\mu(v) = \frac{\sum_{v' \in \mathcal{N}(v)} w(v, v')}{|\mathcal{N}(v)|}$$

$$\sigma(v) = \sqrt{\frac{\sum_{v' \in \mathcal{N}(v)} (w(v, v') - \mu(v))^2}{|\mathcal{N}(v)|}}.$$

$$\tilde{w}_\gamma(v, u) = \frac{w(v, u)}{\max\{\mu(v) + \gamma\sigma(v), \epsilon\}}$$

Normalized of edge weight from the mean $\mu(v)$ and standard deviation $\sigma(v)$ of edge weight connecting $v$

# Node2Vec+: Edge Weight

- Noisy edge: both edge $(v_c, v_n)$ and $(v_n, v_p)$ are loose
- Out edge: $(v_c, v_n)$ is out edge from $v_p$, where $v_n$ are loosely connected to $v_p$
- In edge: $(v_c, v_n)$ is in edge if $(v_n, v_p)$ is tight, regardless w$(v_c, v_n)$
- Overall, biased random walk is defined

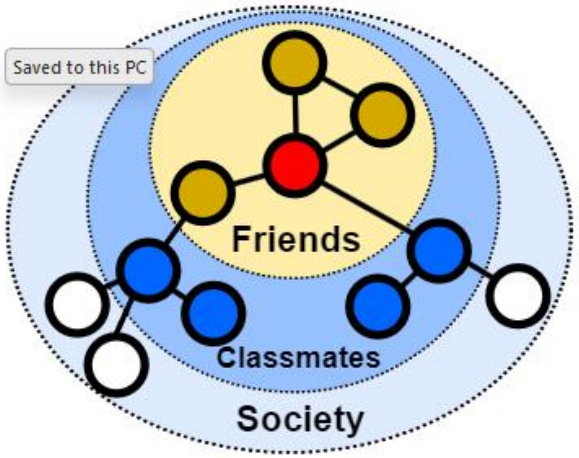
$$\alpha_{\gamma pq}(v_p, v_c, v_n) = \begin{cases} \frac{1}{p} & \text{if } v_p = v_n \\ 1 & \text{if } \tilde{w}_\gamma(v_n, v_p) \geq 1 \\ \min\left\{1, \frac{1}{q}\right\} & \text{if } \tilde{w}_\gamma(v_n, v_p) < 1 \\ & \text{and } \tilde{w}_\gamma(v_c, v_n) < 1 \\ \frac{1}{q} + \left(1 - \frac{1}{q}\right)\tilde{w}_\gamma(v_n, v_p) & \text{if } \tilde{w}_\gamma(v_n, v_p) < 1 \\ & \text{and } \tilde{w}_\gamma(v_c, v_n) \geq 1 \end{cases}.$$

- In edge
- Noisy edge
- Out edge

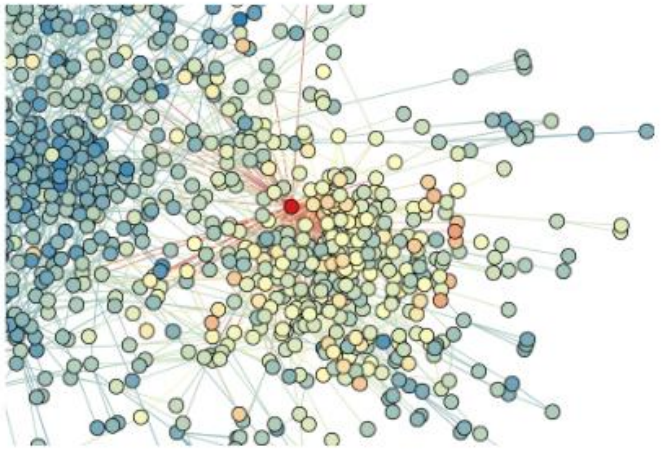
# Don’t Walk – Skip! Online Learning of Multi-scale Network Embeddings



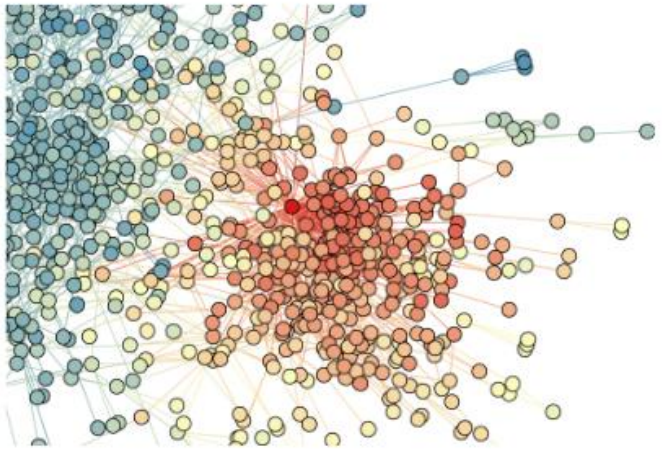
- **Problem**: DeepWalk learn a single global representation that conflates information across all scales
  - Not sufficient in large scale graph
- **Idea**: Multiple scales of random walks at different scales to explicitly deal with large graph
- **Solutions**: Propose WalkLets algorithm with skipping mechanism




(a) A student (in red) is a member of several increasing larger social communities.



(b) WALKLETS Fine Representation

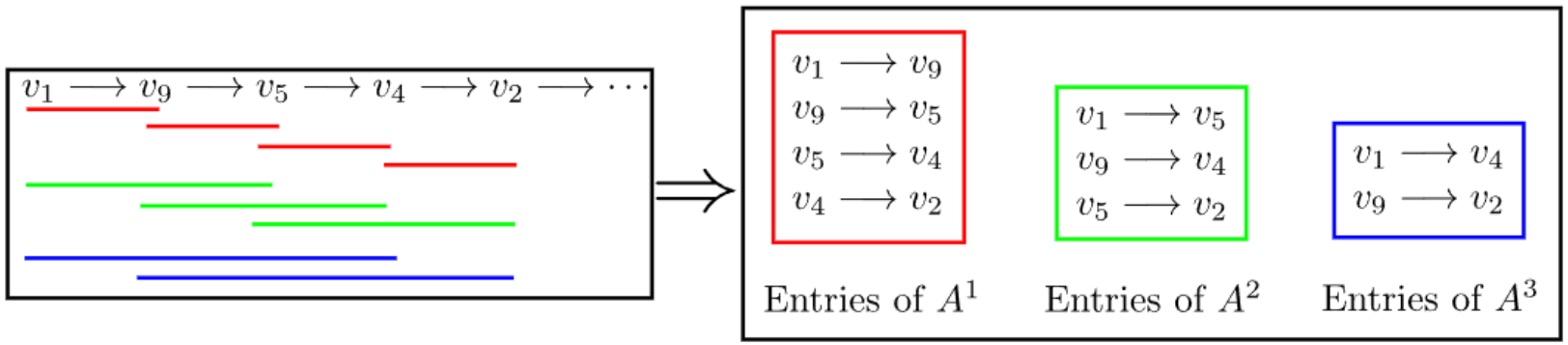


(c) WALKLETS Coarse Representation

# WalkLets: Algorithm

- Generates truncated random walks on the graph, but "skips" over nodes at different scales
  - Results in multiple "corpora" of node co-occurrences at different scales
- Using skip-gram models to learns separate embeddings from each of these corpora
  - Learns multiple embeddings capturing relationships at different scales explicitly
- Optimization by stochastic gradient descent

→ Same as DeepWalk but add skipping nodes at different scales




Random walk, rooted at $v_1$

WALKLETS corpus of skipped random walks

# Random Walks-based Strategies: Summary



- **What are good embeddings?**

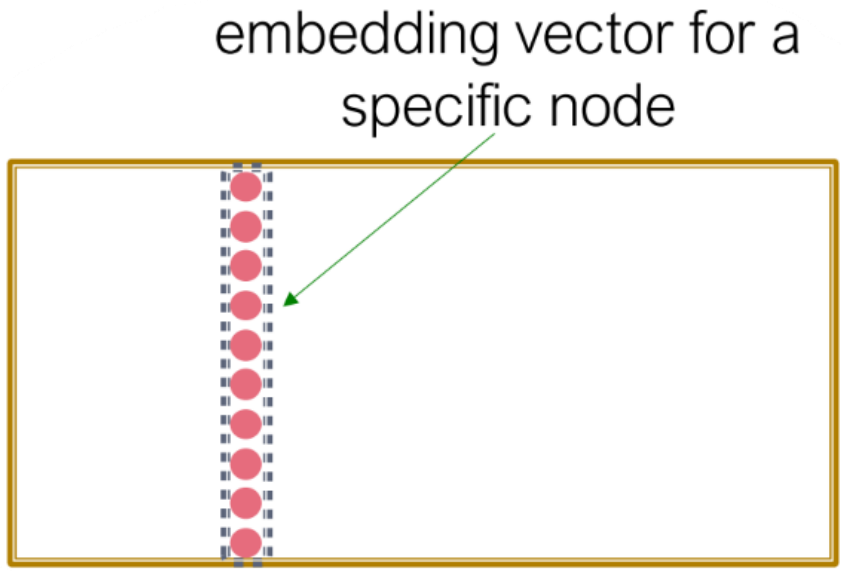
→ We can get good node embeddings that distances between them in embedding space reflect their similarities in the original graph network

- **Two different methods:**
  - Naive: similar if 2 nodes are connected
  - Random walk approaches
- **Which method should we use?**
  - No one method wins in all cases
  - Choose proper methods depending on specific tasks

**Random Walks for Shallow Encoding**:

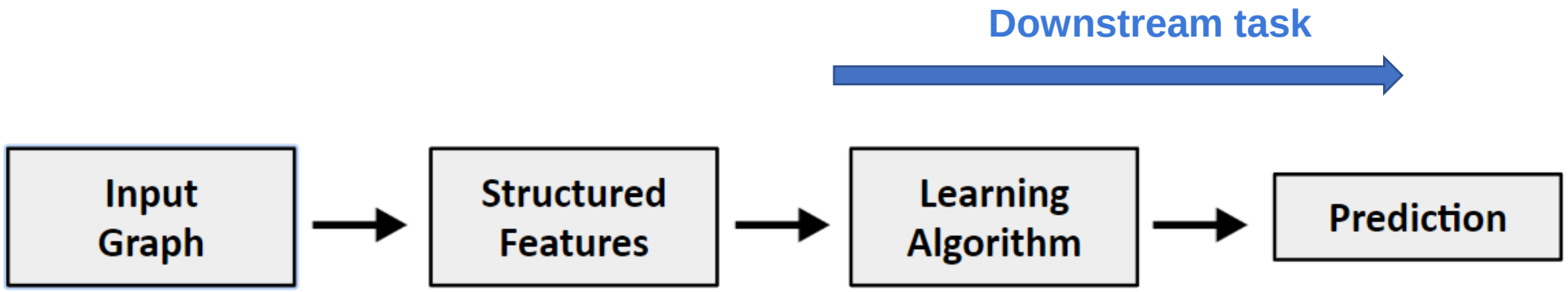
- **Goal**: Generate a lookup table for node embeddings
- Step-by-step:
  1. Run random walks for each node
  2. Collect the set of visited nodes for each node on the walks
  3. Optimize the embeddings $Z_u$ using stochastic gradient descent

$\mathbf{Z} =$



# What after Node Embeddings?

- Once we have node embeddings (independent to task), we can continue to the downstream prediction




**Shallow Encoding**

- SVM
- Random Forest
- XGBoost
- DNN

- Node-level
- Edge-level
- Graph-level

# How to Use these Node Embeddings?



- **Given a node $v_i$ in a graph, we have its embedding $\boldsymbol{Z_i}$, we can do:**
  1. Clustering/Community Detection: Cluster $\boldsymbol{Z_i}$
  2. Node Classification: Predict label of node $v_i$ based on $\boldsymbol{Z_i}$
  3. Link Prediction: Predict edge $(v_i, v_j)$ based on $(\boldsymbol{Z_i}, \boldsymbol{Z_j})$
     - Concatenate: f($\boldsymbol{Z_i}$, $\boldsymbol{Z_j}$) = g([$\boldsymbol{Z_i}$, $\boldsymbol{Z_j}$])
     - Hadamard: f($\boldsymbol{Z_i}$, $\boldsymbol{Z_j}$) = g($\boldsymbol{Z_i} * \boldsymbol{Z_j}$) (per position product)
     - Sum/Average: f($\boldsymbol{Z_i}$, $\boldsymbol{Z_j}$) = g($\boldsymbol{Z_i} + \boldsymbol{Z_j}$)
     - Distance: f($\boldsymbol{Z_i}$, $\boldsymbol{Z_j}$) = g($\|\boldsymbol{Z_i} - \boldsymbol{Z_j}\|_2$)
  4. Graph classification: Aggregate node embeddings to form graph embedding $\boldsymbol{Z_G}$. Predict graph label based on graph embedding $\boldsymbol{Z_G}$


네트워크 과학 연구실
NETWORK SCIENCE LAB

가톨릭대학교
THE CATHOLIC UNIVERSITY OF KOREA