

Machine Learning in Graphs

Node Embeddings

Acknowledgement: Jure Leskovec, Stanford University

Overview

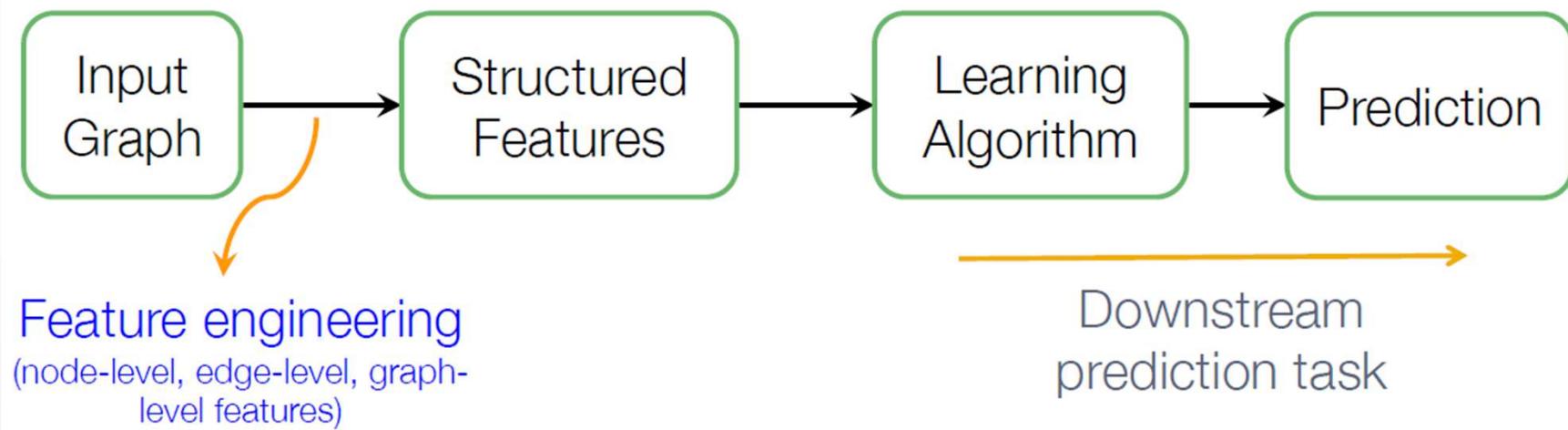
- Introduction
- Node Embeddings: Encoders and Decoders
- Random Walk Approaches for Node Embeddings

Overview

- Introduction
- Node Embeddings: Encoders and Decoders
- Random Walk Approaches for Node Embeddings

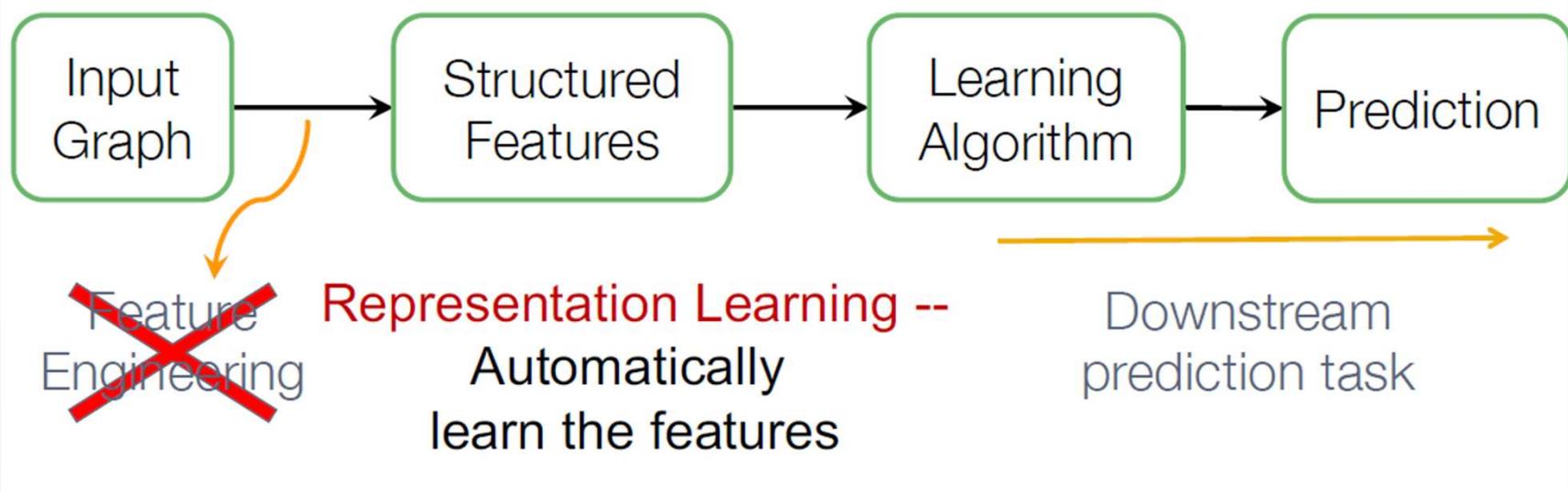
Traditional ML Pipeline for Graphs

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



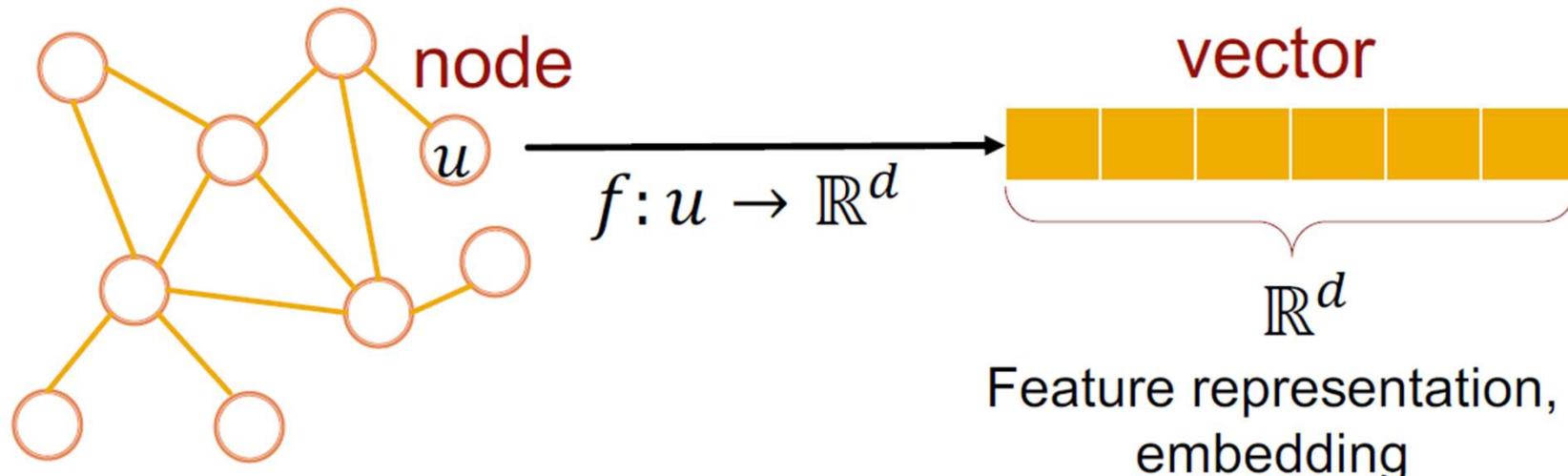
Graph Representation Learning

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



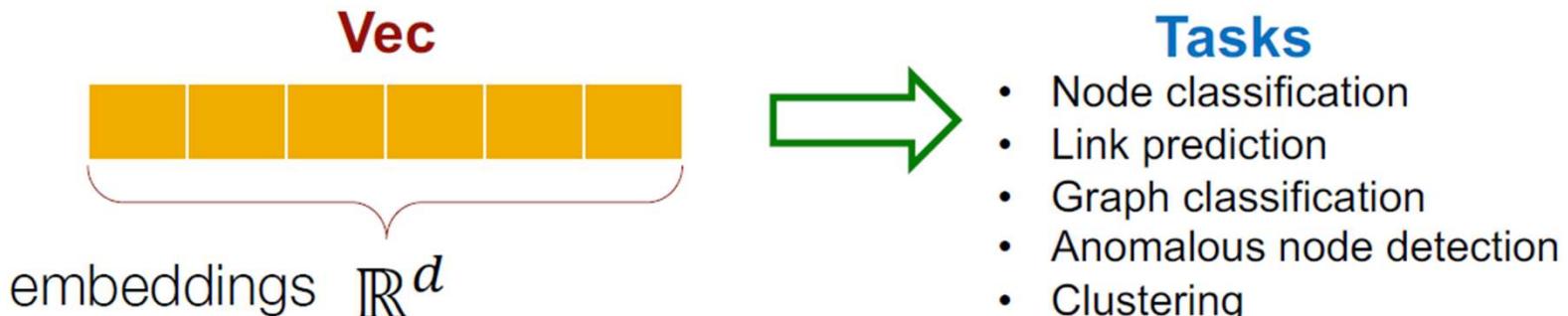
Graph Representation Learning

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



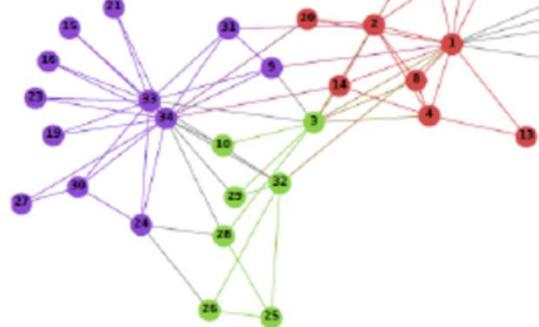
Why Embedding?

- **Task: map nodes into an embedding space**
 - Similarity of embeddings between nodes indicates their similarity in the network. For example:
 - Both nodes are close to each other (connected by an edge)
 - Encode network information
 - Potentially used for many downstream predictions

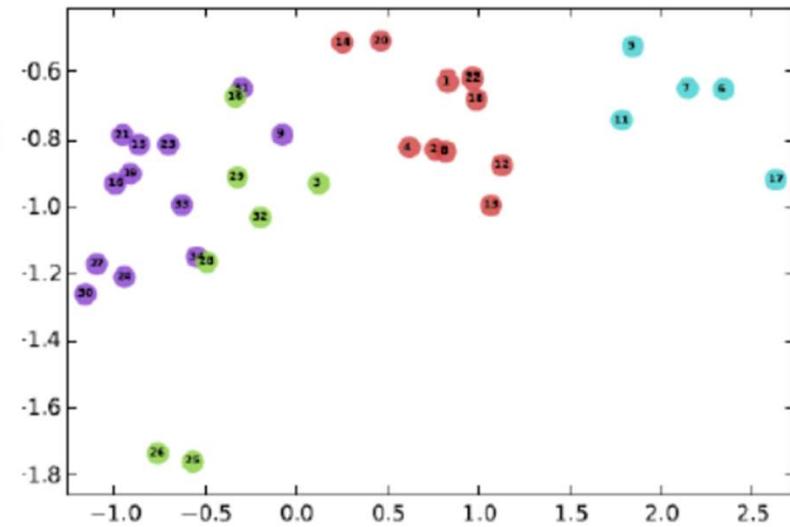


Example: Node Embedding

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



Input



Output

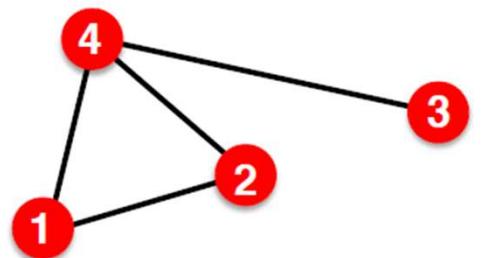
Image from: [Perozzi et al.](#). DeepWalk: Online Learning of Social Representations. *KDD 2014*.

Overview

- Introduction
- Node Embeddings: Encoders and Decoders
- Random Walk Approaches for Node Embeddings

Setup

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

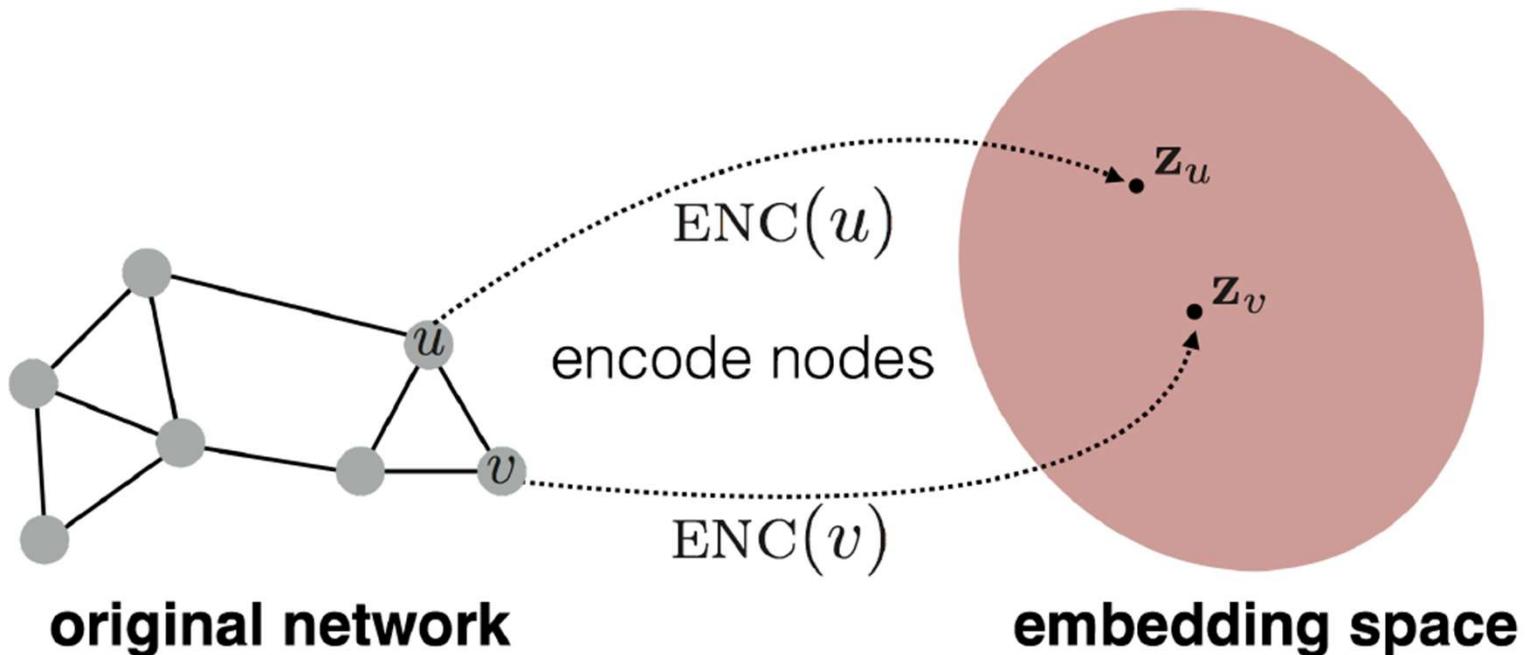


$V: \{1, 2, 3, 4\}$

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

Embedding Nodes

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



Learning Embedding Nodes

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:

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

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

in the original network

Similarity of the embedding

Two Key Components

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

$\text{ENC}(v) = \boxed{\mathbf{z}_v}$ *d*-dimensional embedding
node in the input graph

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

$\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$ **Decoder**
Similarity of u and v in the original network dot product between node embeddings

“Shallow” Encodings

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

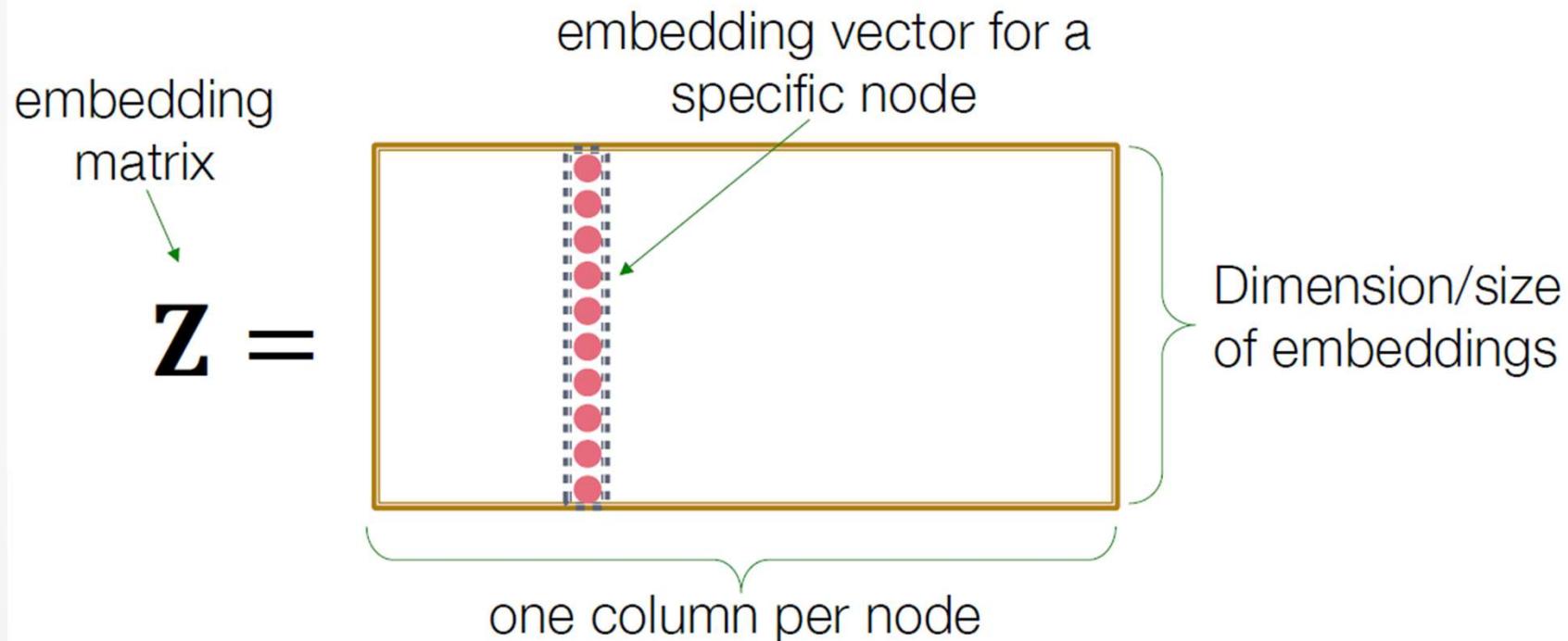
$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

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

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



For every node we have to determine d parameters.
So the method is not scalable.

“Shallow” Encodings

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}' for all nodes $u \in V$
- We will cover deep encoders (GNNs) later
- **Decoder:** based on node similarity
- **Objective:** maximize $\mathbf{z}_v^T \mathbf{z}_u$ for node pairs (u, v) that are **similar**

Node Embeddings Learning

- 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.

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.

Overview

- Introduction
- Node Embeddings: Encoders and Decoders
- Random Walk Approaches for Node Embeddings

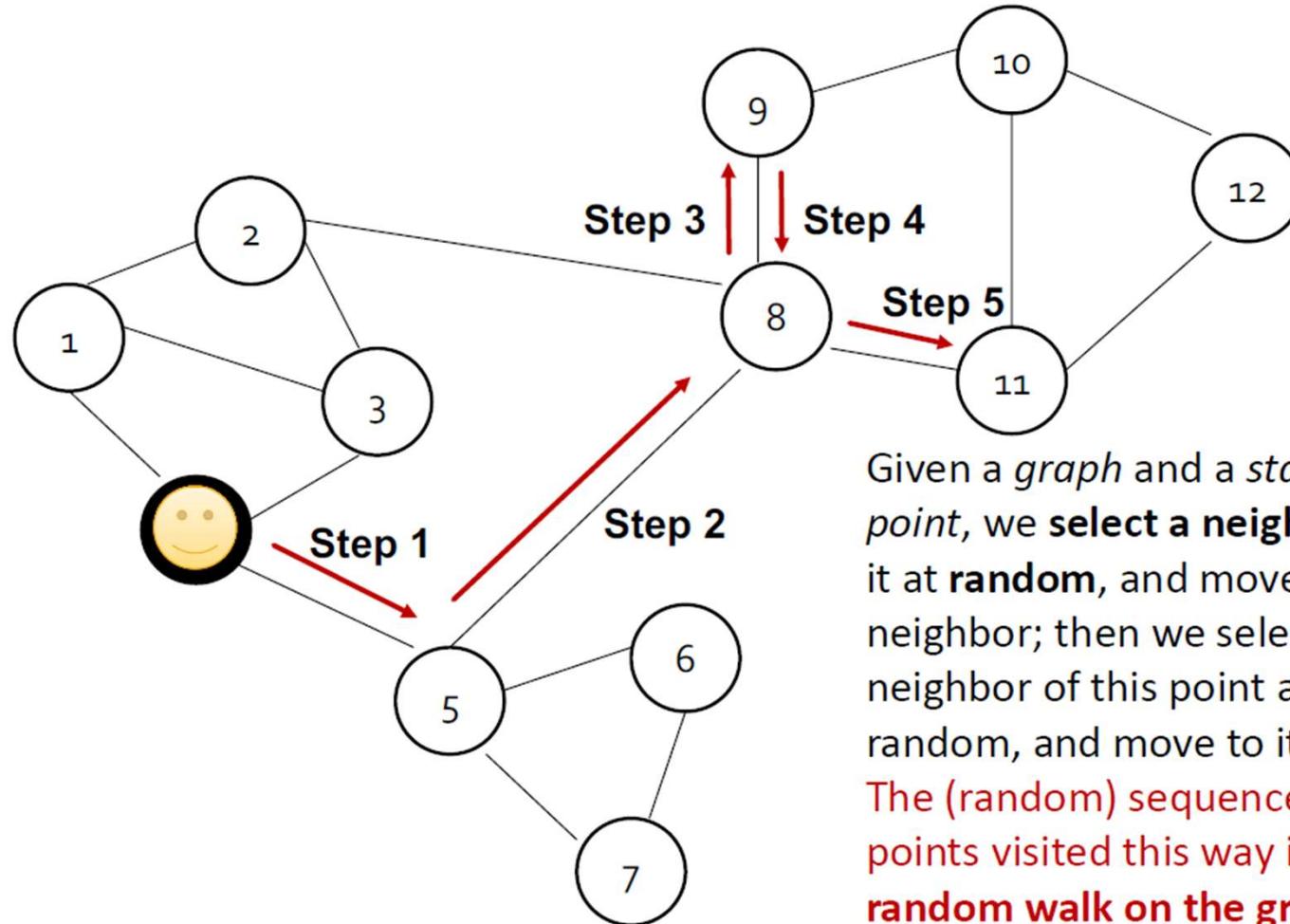
Notation

- **Vector \mathbf{z}_u :**
 - The embedding of node u (what we aim to find).
 - **Probability $P(v | \mathbf{z}_u)$** :  Our model prediction based on \mathbf{z}_u
 - The **(predicted) probability** of visiting node v on random walks starting from node u .
-

Non-linear functions used to produce predicted **probabilities**

- **Softmax** function
 - Turns vector of K real values (model predictions) into K probabilities that sum to 1: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$.
- **Sigmoid** function:
 - S-shaped function that turns real values into the range of $(0, 1)$. Written as $S(x) = \frac{1}{1+e^{-x}}$.

Random Walk



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

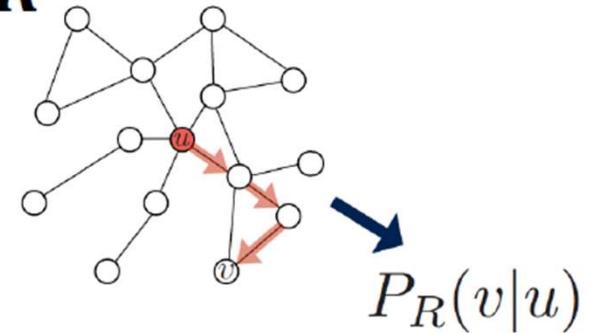
Random Walk Embeddings

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

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

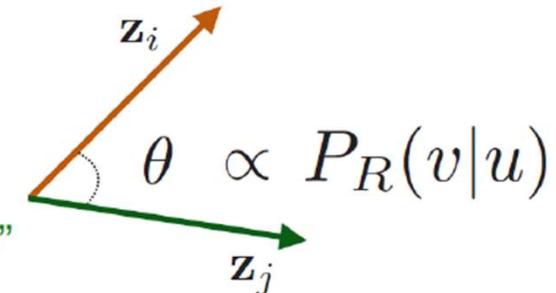
Random Walk Embeddings

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:

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

- $N_R(u)$ 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 $N_R(u)$**

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

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

Random Walk Optimization

Equivalently,

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

- **Intuition:** Optimize embeddings \mathbf{z}_u to maximize the likelihood of random walk co-occurrences
- **Parameterize** $P(v|\mathbf{z}_u)$ **using softmax:**

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

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

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>

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

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 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 eventsIn practice $k = 5-20$

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_i at some randomized value for all i .

- Iterate until convergence.

- For all i , compute the derivative $\frac{\partial \mathcal{L}}{\partial z_i}$.

η : learning rate

- For all i , make a step towards the direction of derivative: $z_i \leftarrow z_i - \eta \frac{\partial \mathcal{L}}{\partial z_i}$.

Stochastic Gradient Descent

■ Problem with gradient descent:

- Exact gradient requires computing $\nabla_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}))$, where \mathbf{x} is the **entire** dataset!
 - This means summing gradient contributions over all the points in the dataset
 - Modern datasets often contain billions of data points
 - Extremely expensive for every gradient descent step

■ Solution: Stochastic gradient descent (SGD)

- At every step, pick a different **minibatch** \mathcal{B} containing a subset of the dataset, use it as input \mathbf{x}

Stochastic Gradient Descent

- **Stochastic Gradient Descent:** Instead of evaluating gradients over all examples, evaluate it for each **individual** training example.
 - Initialize z_i at some randomized value for all i .
 - Iterate until convergence:
$$\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$
 - Sample a node i , for all j calculate the derivative $\frac{\partial \mathcal{L}^{(i)}}{\partial z_j}$.
 - For all j , update: $z_j \leftarrow z_j - \eta \frac{\partial \mathcal{L}^{(i)}}{\partial z_j}$.

Random Walks: Summary

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

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

We can efficiently approximate this using negative sampling!

How should we randomly walk?

- So far we have described how to optimize embeddings given a random walk strategy R
- **What strategies should we use to run these random walks?**
 - Simplest idea: Just run fixed-length, unbiased random walks starting from each node (i.e., [DeepWalk from Perozzi et al., 2013](#))
 - The issue is that such notion of similarity is too constrained
- **How can we generalize this?**

Reference: Perozzi et al. 2014. [DeepWalk: Online Learning of Social Representations](#). *KDD*.

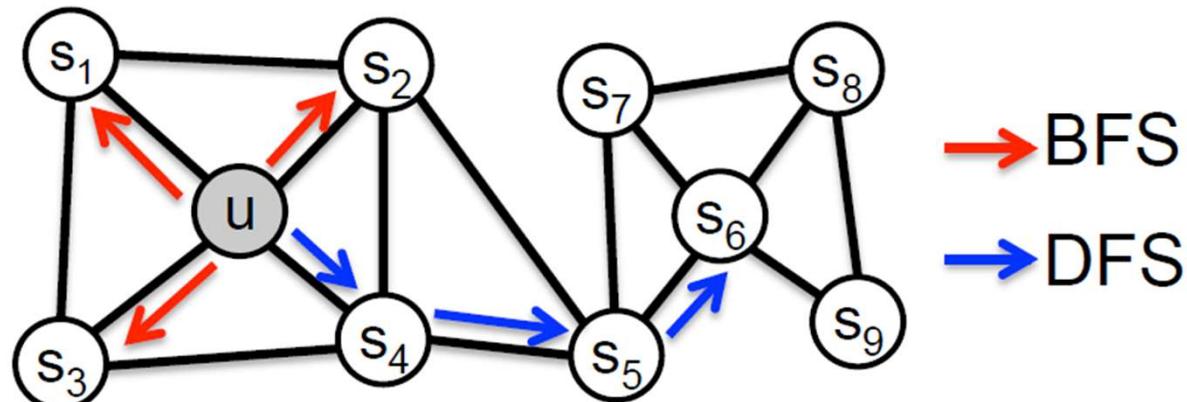
Overview of node2vec

- **Goal:** Embed nodes with similar network neighborhoods close in the feature space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- **Key observation:** Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 2nd order random walk R to generate network neighborhood $N_R(u)$ of node u

Reference: Grover et al. 2016. [node2vec: Scalable Feature Learning for Networks](#). KDD.

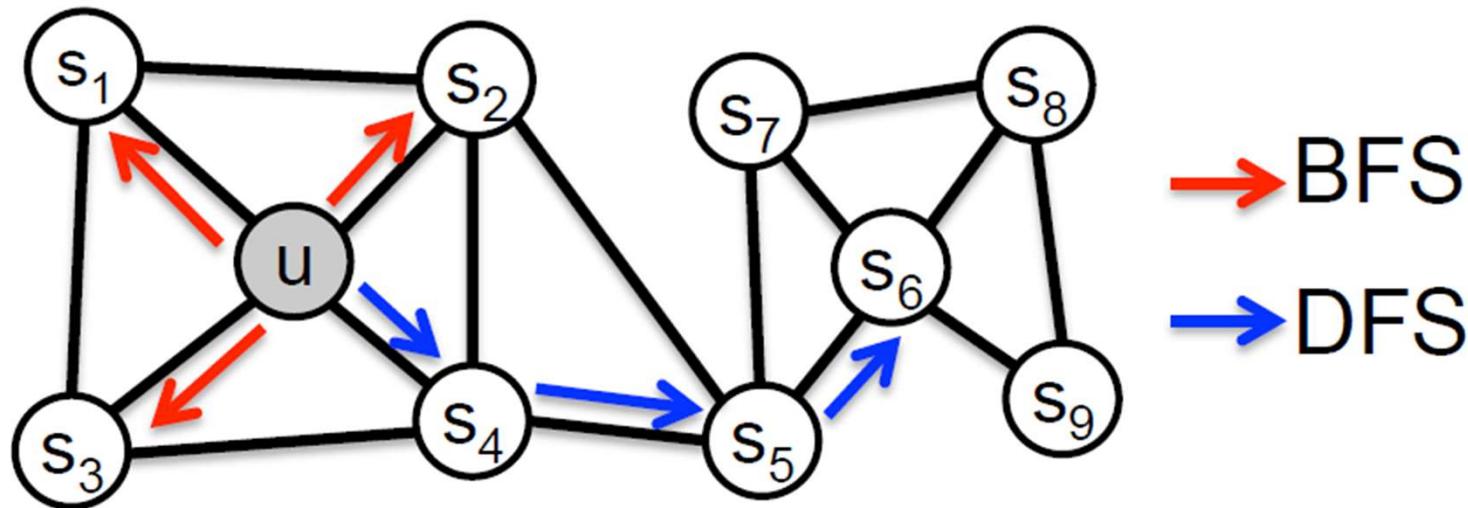
node2vec: Biased Walks

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



node2vec: Biased Walks

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

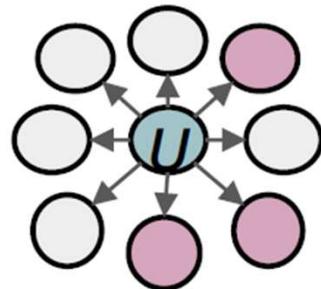


Walk of length 3 ($N_R(u)$ of size 3):

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

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

BFS vs DFS



BFS:

Micro-view of
neighbourhood



DFS:

Macro-view of
neighbourhood

Interpolating BFS and DFS

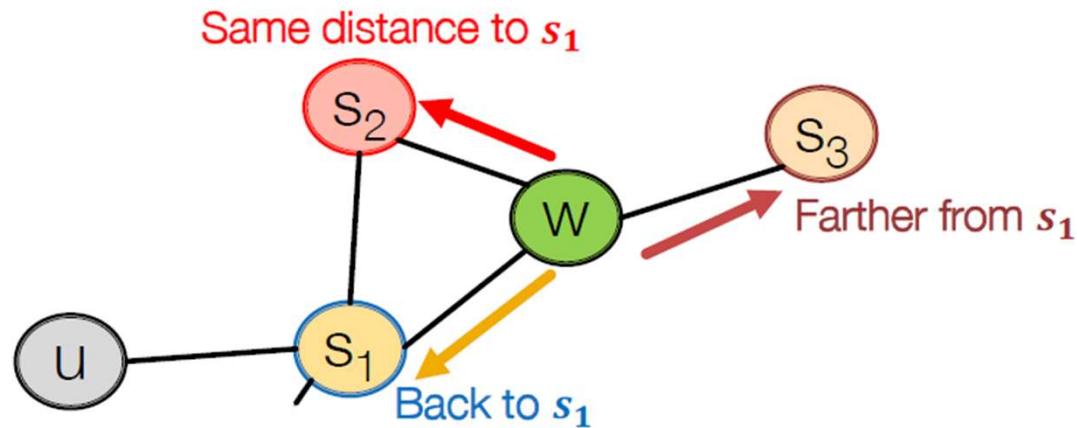
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- Two parameters:
 - **Return parameter p :**
 - Return back to the previous node
 - **In-out parameter q :**
 - Moving outwards (DFS) vs. inwards (BFS)
 - Intuitively, q is the “ratio” of BFS vs. DFS

Biased Random Walks

Biased 2nd-order random walks explore network neighborhoods:

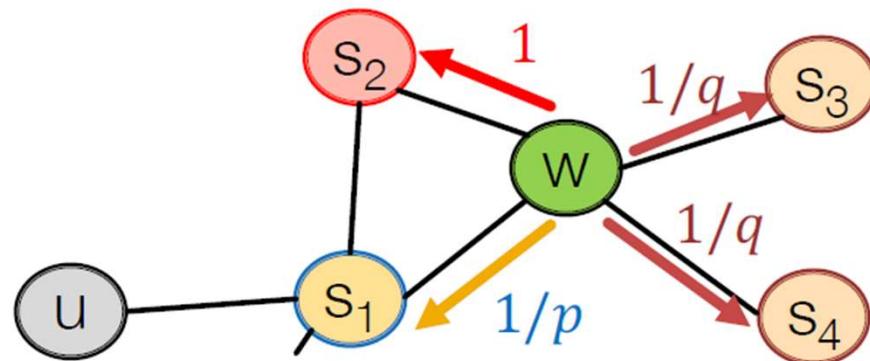
- Rnd. walk just traversed edge (s_1, w) and is now at w
- **Insight:** Neighbors of w can only be:



Idea: Remember where the walk came from

Biased Random Walks

- Walker came over edge (s_1, w) and is at **w**. Where to go next?

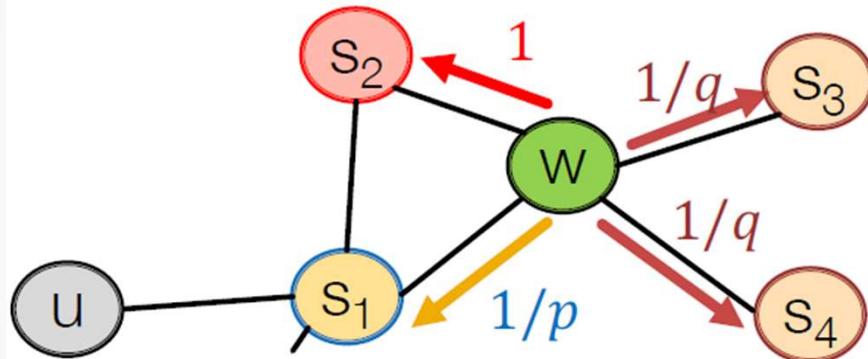


$1/p, 1/q, 1$ are unnormalized probabilities

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

Biased Random Walks

- Walker came over edge (s_1, w) and is at w .
Where to go next?



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

Unnormalized
transition prob.
segmented based
on distance from s_1

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

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

node2vec algorithm

- 1) Compute random walk probabilities
- 2) Simulate r random walks of length l starting from each node u
- 3) Optimize the node2vec objective using Stochastic Gradient Descent

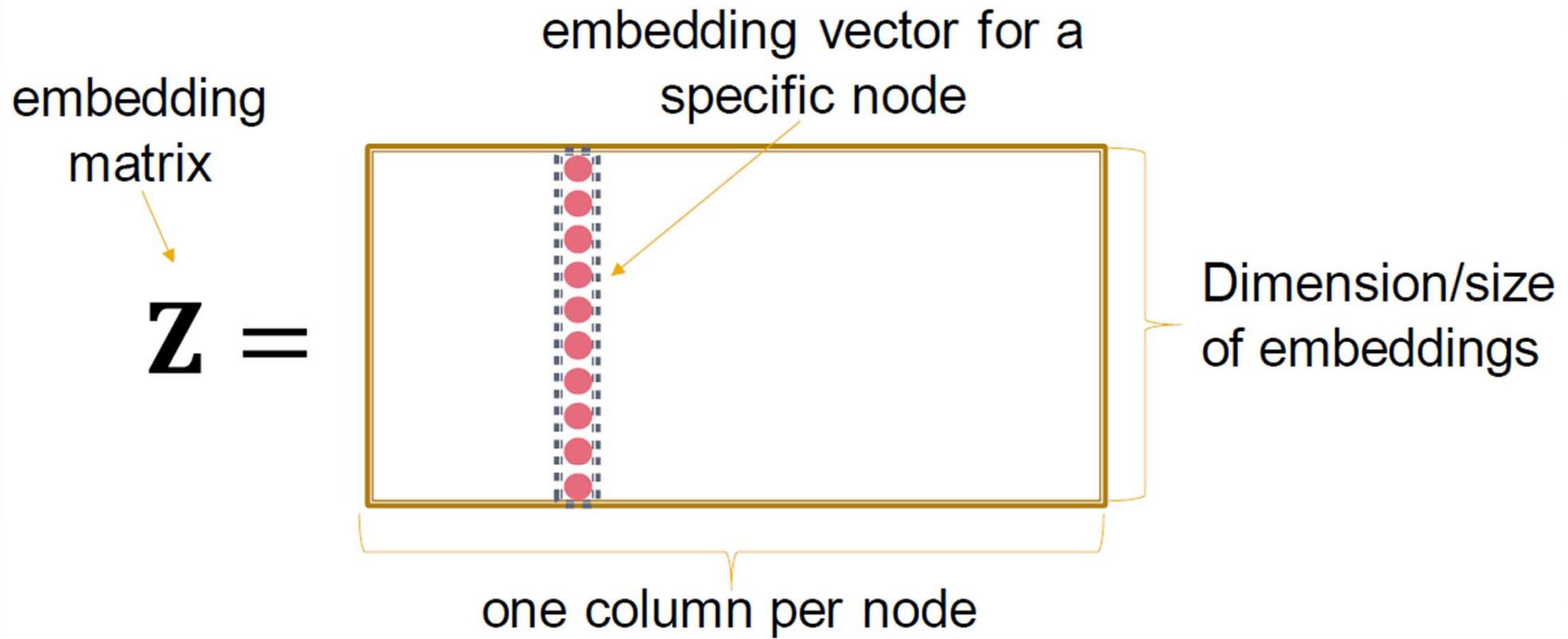
- **Linear-time complexity**
- All 3 steps are **individually parallelizable**

Other Random Walk Ideas

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

Embeddings & Matrix Factorization

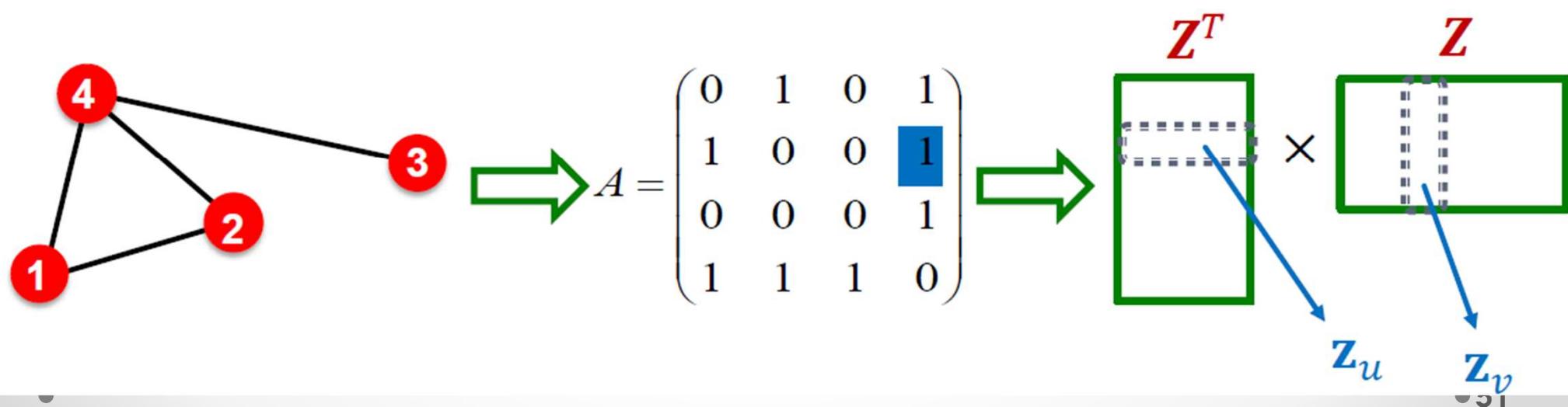
- Recall: encoder as an embedding lookup



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 A .**

Random-walk Based Similarity

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

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

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

How to use Node Embeddings

- **How to use embeddings 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 $(\mathbf{z}_i, \mathbf{z}_j)$

Summary

- **Core idea:** Embed nodes so that distances in embedding space reflect node similarities in the original network.
- **Different notions of node similarity:**
 - Naïve: similar if 2 nodes are connected
 - Neighborhood overlap
 - Random walk approaches
- **So what method should to use..?**
- No one method wins in all cases....
 - E.g., node2vec performs better on node classification while alternative methods perform better on link prediction ([Goyal and Ferrara, 2017 survey](#))
- Random walk approaches are generally more efficient
- **In general:** Must choose definition of node similarity that matches your application!

References

- Social Network Analysis Tanmoy Chakraborty. Chapter 9