WEB INTELLIGENCE

Lecture 12:
Graph Embeddings

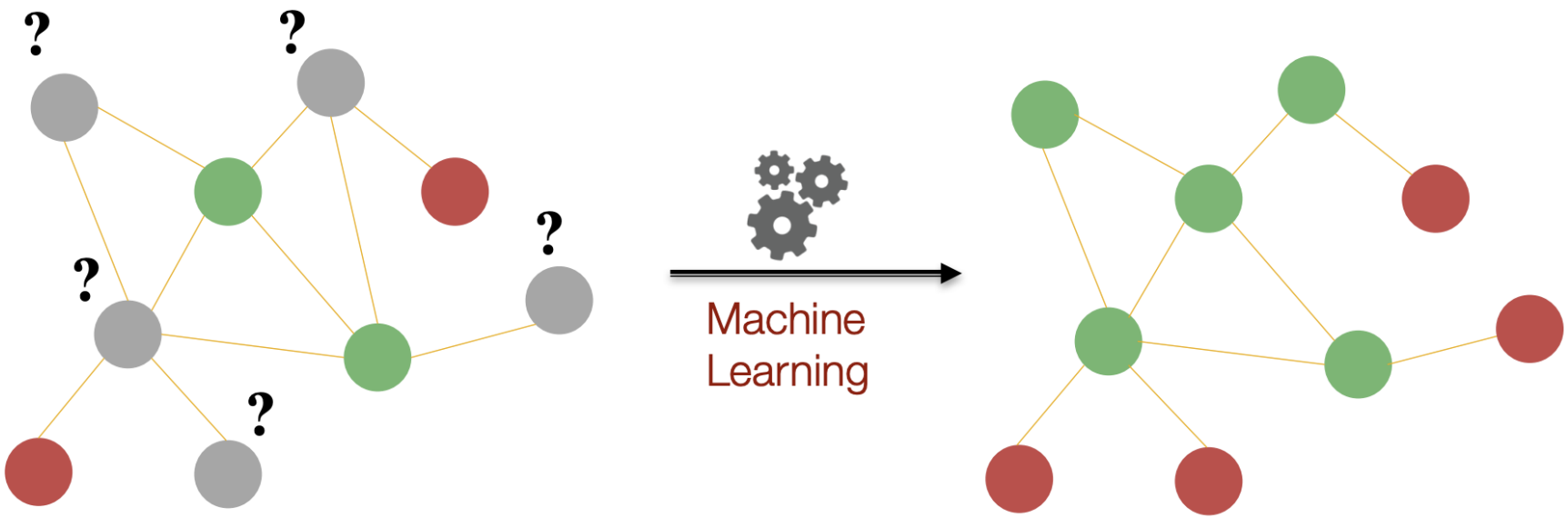
Russa Biswas

November 25, 2025

AALBORG UNIVERSITY

Based on Slides from Jure Leskovec

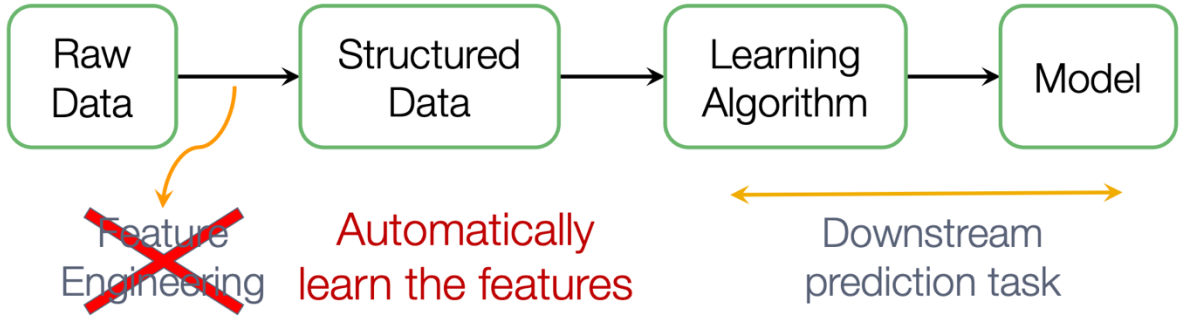
Node Classification



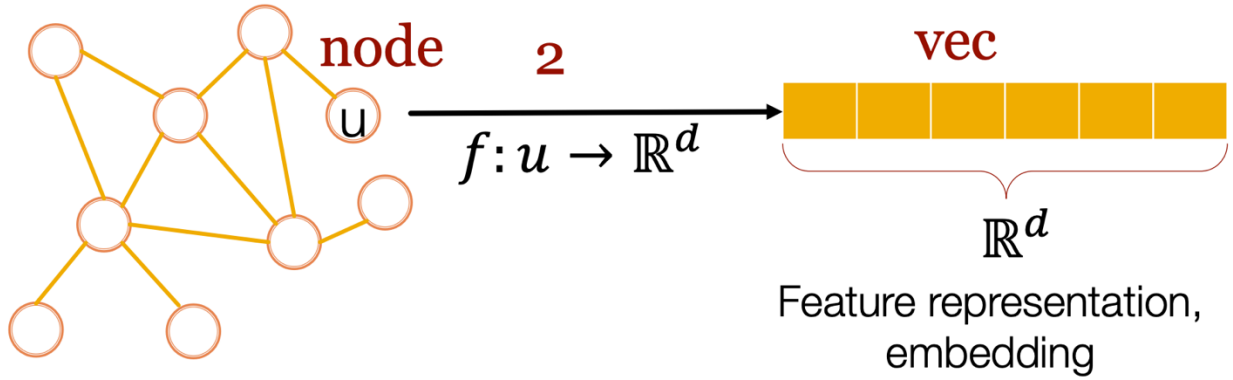
Node classification

Machine Learning in Graphs

- **(Supervised) Machine Learning Lifecycle: This feature, that feature. Every single time!**

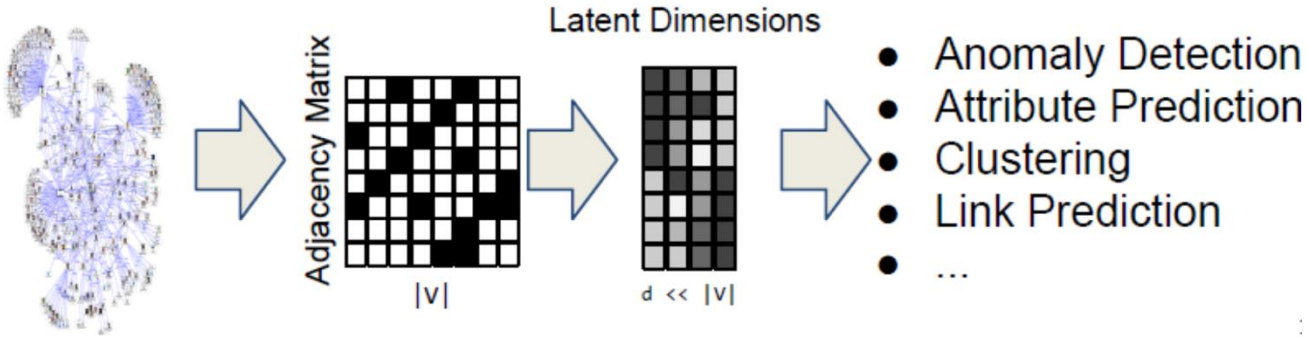


Goal: Efficient task-independent feature learning for machine learning in networks!

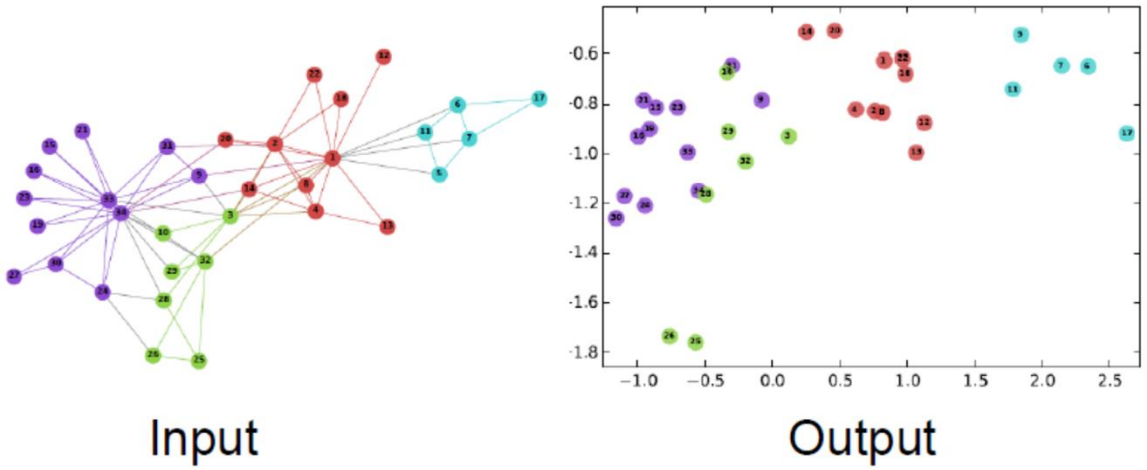


Machine Learning in Graphs

- We map each node in a network into a low-dimensional space
  - Distributed representation for nodes
  - Similarity between nodes indicates link strength
  - Encode network information and generate node representation



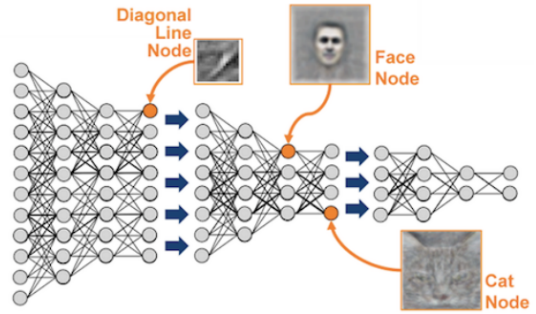
- **Zachary's Karate Club network:**

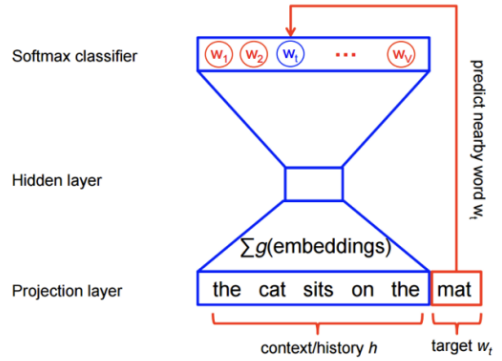


Machine Learning in Graphs

Graph representation learning is hard:

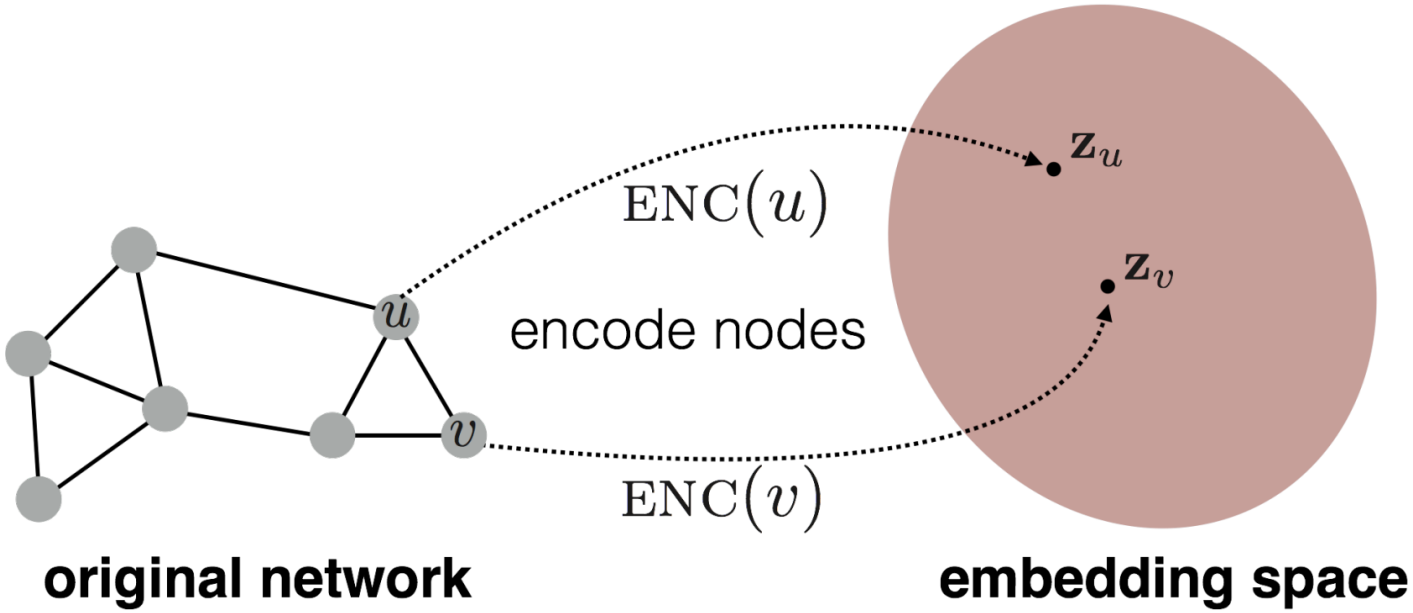
- Images are fixed size
  - Convolutions (CNNs)
- Text is linear
  - Sliding window (word2vec)
- **Graphs are neither of these!**
  - Node numbering is arbitrary (node isomorphism problem)
  - Much more complicated structure





Node Embeddings

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



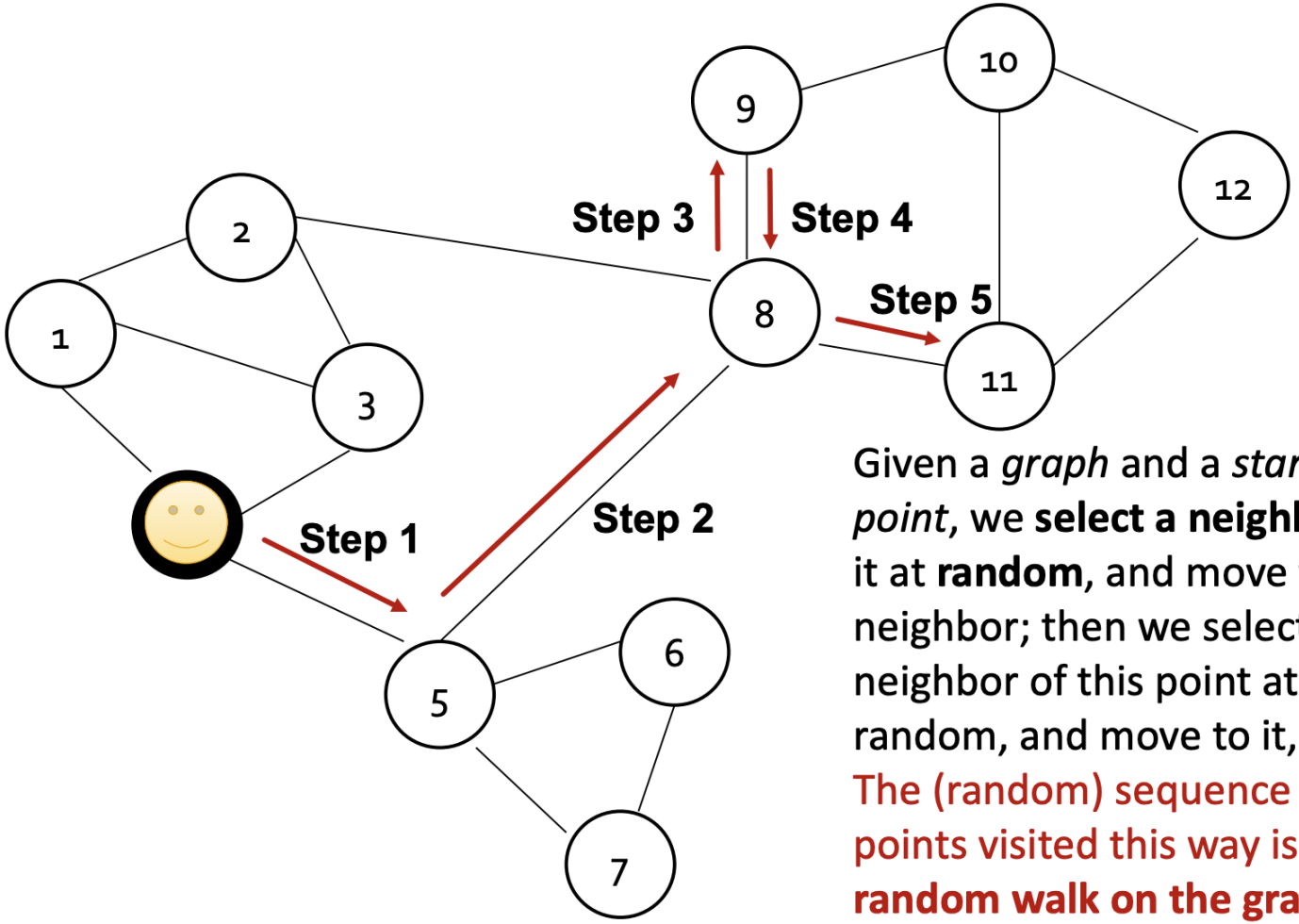
Node Embeddings

- **Vector $\mathbf{z}_u$:**
  - The embedding of node u (what we aim to find).
- **Probability $P(v \mid \mathbf{z}_u)$:** ⟸ 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: $S(\boldsymbol{z})[i] = \frac{e^{\boldsymbol{z}[i]}}{\sum_{j=1}^{K} e^{\boldsymbol{z}[j]}}$
- **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 Walks



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**.

Feature Learning

- Intuition: Find embedding of nodes to d-dimensions that preserves similarity
- Idea: Learn node embedding such that nearby nodes are close together
- Given a node *u*, how do we define nearby nodes?
  - $N_S(u)$ … neighbourhood of *u* obtained by some strategy *S*

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

- Log-likelihood objective:
  $\max_f \sum_{u \in V} \log \Pr(N_S(u) | f(u))$
  - where $N_S(u)$ is neighborhood of node u.

- Given node u, we want to learn feature representations predictive of nodes in its neighborhood $N_S(u)$.

Random Walk Optimisation

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 $N_R(u)$.

$$\arg\max_z \sum_{u \in V} \log \mathrm{P}(N_R(u) | \mathbf{z}_u) \Longrightarrow$$ Maximum likelihood objective

Equivalently,

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

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

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

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:

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

- 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

Optimizing random walk embeddings =

Finding embeddings $\mathbf{z}_u$ that minimize $\mathcal{L}$

But doing this naively is too expensive!

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

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

Negative Sampling

- **Solution:** Negative sampling

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

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

sigmoid function (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 allows for quick likelihood calculation.

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

random distribution over nodes

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

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

η: learning rate

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:

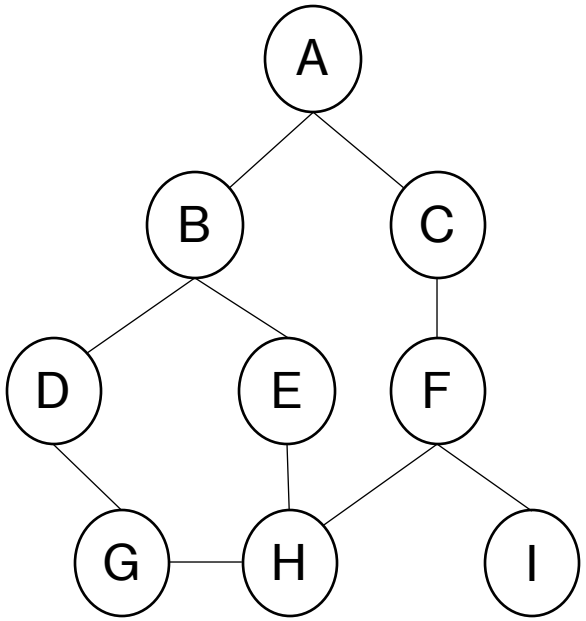
$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

We can efficiently approximate this using negative sampling!

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

Breadth First Search



A, B, C, D, E, F, G, H, I

Queue

A									

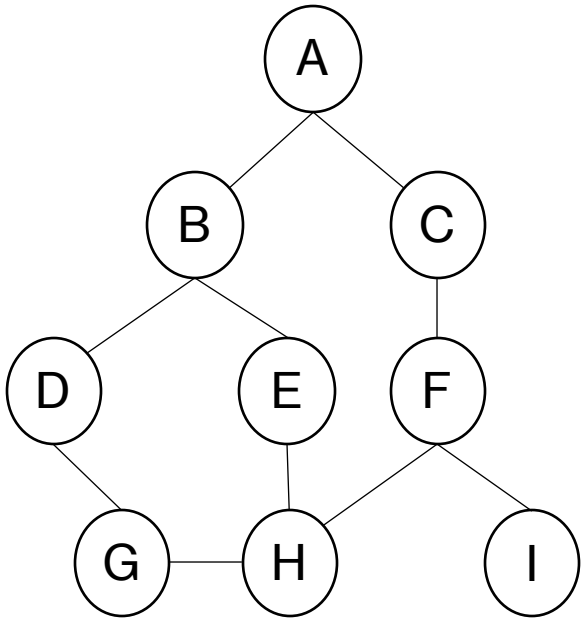
Visited = {}

~~A~~	B	C							

Visited = {A}

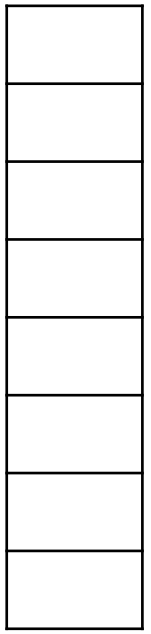
- Dequeued: B, Queue: ['C', 'D', 'E'] Visited = {A, B}
- Dequeued: C, Queue: ['D', 'E', 'F'] Visited = {A, B, C}
- Dequeued: D, Queue: ['E', 'F', 'G'] Visited = {A, B, C, D}
- Dequeued: E, Queue: ['F', 'G', 'H'] Visited = {A, B, C, D, E}
- Dequeued: F, Queue: ['G', 'H', 'I'] Visited = {A, B, C, D, E, F}
- Dequeued: G, Queue: ['H', 'I'] Visited = {A, B, C, D, E, F, G}
- Dequeued: H, Queue: ['I'] Visited = {A, B, C, D, E, F, G, H}
- Dequeued: I, Queue: [] Visited = {A, B, C, D, E, F, G, H, I}

Depth First Search



Start at A:
Stack: ['A']
Visited: {}
Order So Far: []

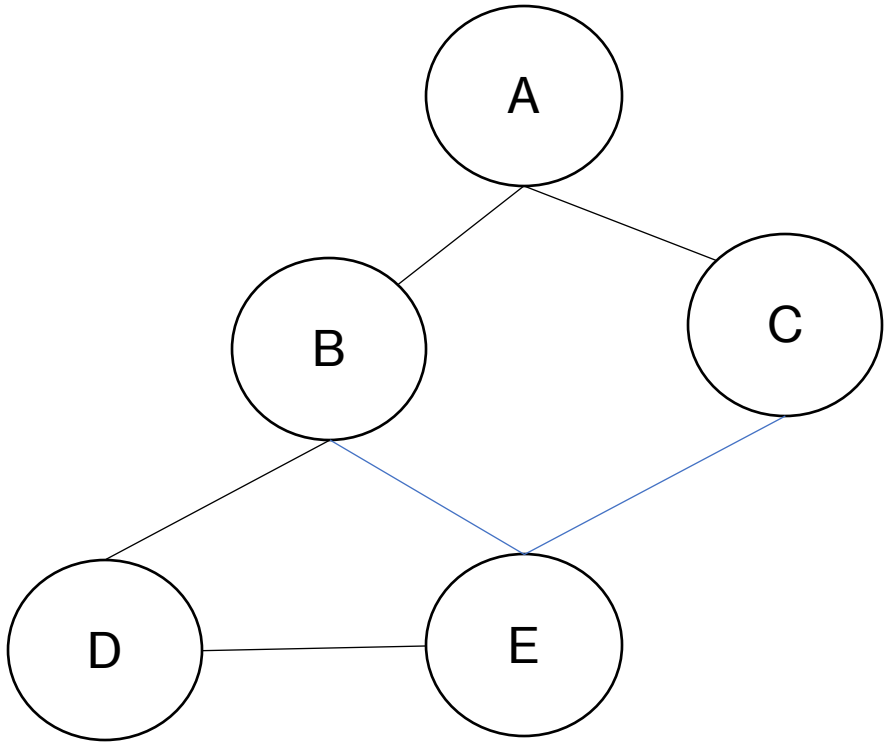
- Popped: A, Stack: ['C', 'B’] Visited: A
- Popped: B, Stack: ['C', 'E', 'D’] Visited: A, B
- Popped: D, Stack: ['C', 'E', 'G’] Visited: A, B, D
- Popped: G, Stack: ['C', 'E', 'H’] Visited: A, B, D, G
- Popped: H, Stack: ['C', 'E', 'F’] Visited: A, B, D, G, H
- Popped: F, Stack: ['C', 'E', 'I’] Visited: A, B, D, G, H, F
- Popped: I, Stack: ['C', 'E’] Visited: A, B, D, G, H, F, I
- Popped: E, Stack: ['C’] Visited: A, B, D, G, H, F, I, E
- Popped: C, Stack: [] Visited: A, B, D, G, H, F, I, E, C



Stack

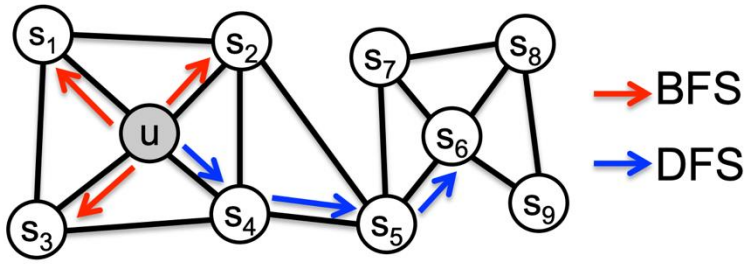
DFS: ['A', 'B', 'D', 'G', 'H', 'F', 'I', 'E', 'C']

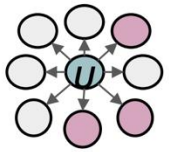
Compute BFS and DFS



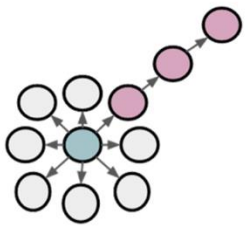
Biased Walks

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



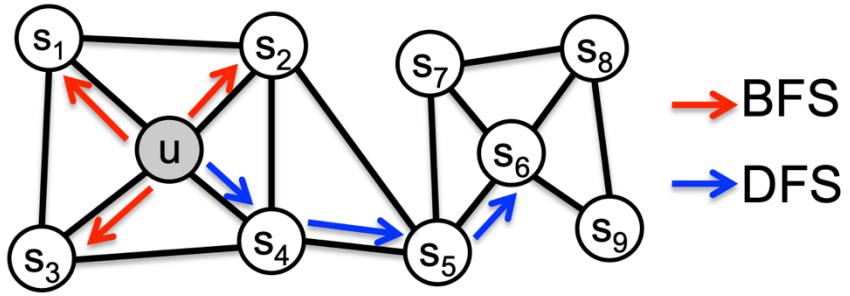


BFS:
$N_R(\cdot)$ will provide a micro-view of neighbourhood



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

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):

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

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

Biased Random Walks

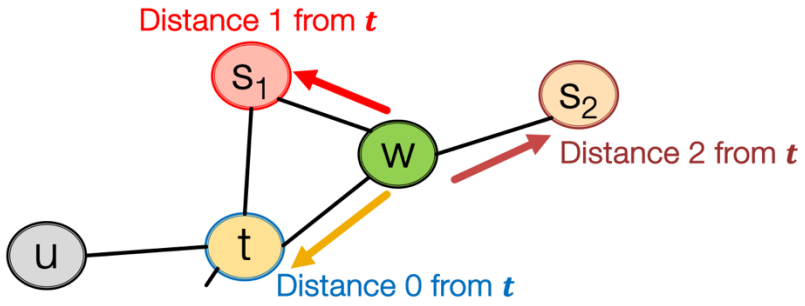
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- 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.

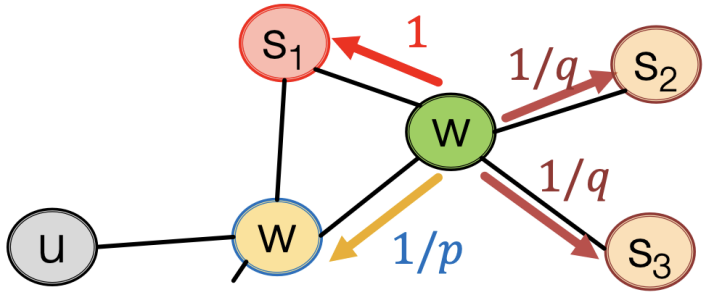
Biased Random Walk

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:



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



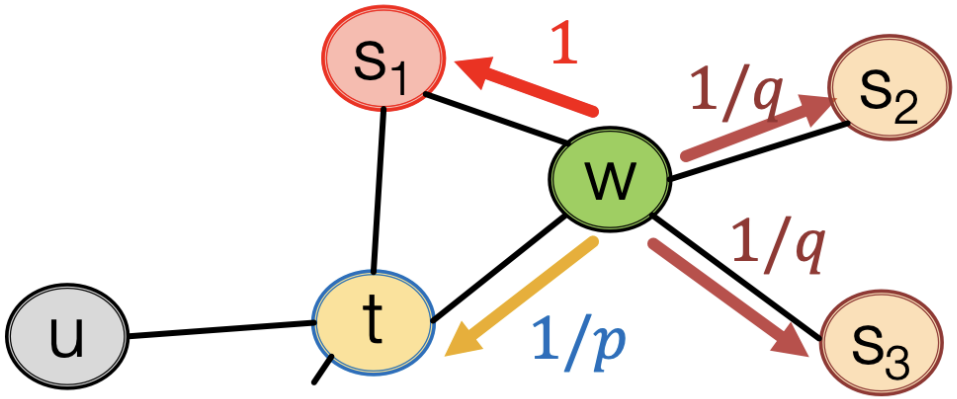
$1/p, 1/q, 1$ are unnormalized probabilities

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

Biased Random Walk – 1 step

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

How to set edge transition probabilities?



$w \rightarrow$

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

Unnormalized transition prob. segmented based on distance from t

- **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 overall

- 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

- Get the biased paths
- Consider each paths as sentences
- Train word2vec model (skip-gram)

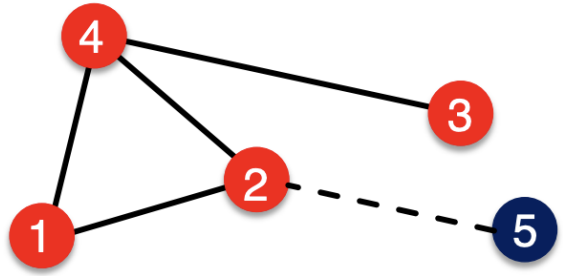
Node2vec: https://arxiv.org/abs/1607.00653

Node Embeddings

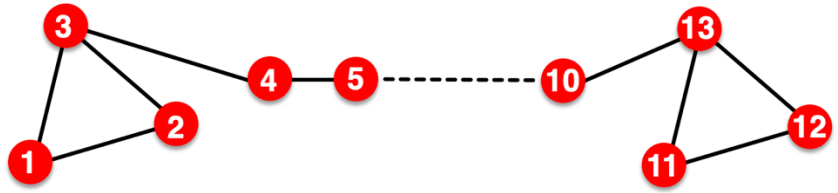
- **How to use embeddings z_i of nodes:**
  - **Clustering/community detection:** Cluster points $\boldsymbol{z_i}$
  - **Node classification:** Predict label of node i based on $\boldsymbol{z_i}$
  - **Link prediction:** Predict edge (i, j) based on $(\boldsymbol{z_i}, \boldsymbol{z_j})$
    - Where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

Limitations

- Transductive (not inductive) method:
  - Cannot obtain embeddings for nodes not in the training set
  - Cannot apply to new graphs

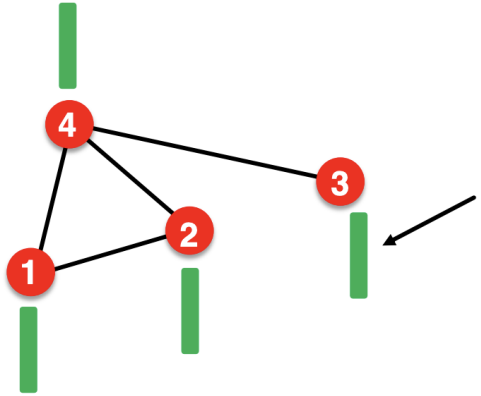


- Cannot capture **structural similarity**:



  - 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.

- Cannot utilize node, edge and graph features



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

DeepWalk / node2vec embeddings do not incorporate such node features

Solution to these limitations: Deep Representation Learning and Graph Neural Networks

Graph Embeddings

Pros

- Improves performance for downstream machine learning tasks
- Flexible and applicable to various types of graphs
- Captures semantic and relational information

Cons

- Training can be computationally expensive
- Scalability on large Networks/Graphs
- Difficulty with Complex Relations
- Bias in Embeddings

Literature

- https://archives.leni.sh/stanford/CS224w.pdf