Graph Neural Networks

Lecture 3

Danila Biktimirov
Applied Computer Science
Neapolis University Pafos

21 February 2025

Overview

Previously on...

Feature Engineering

Graphs are complex objects, so we need to figure out how to create features.
As well as for tasks, we examined features (feature engineering) for:

- vertices
- edges
- graphs

Features for Vertices

- vertex degree
- centrality
- clustering coefficient
- graphlets

Features for Edges

- distance
- local intersection
- global intersection

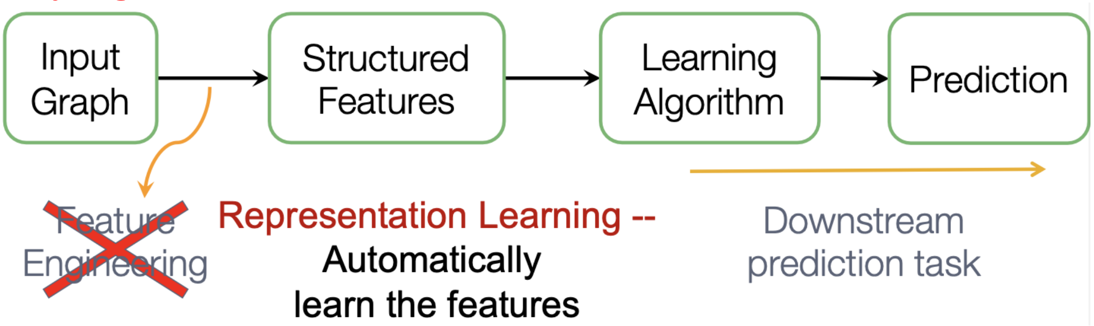
Features for Graphs

- graphlet kernels
- WL kernels

Embeddings

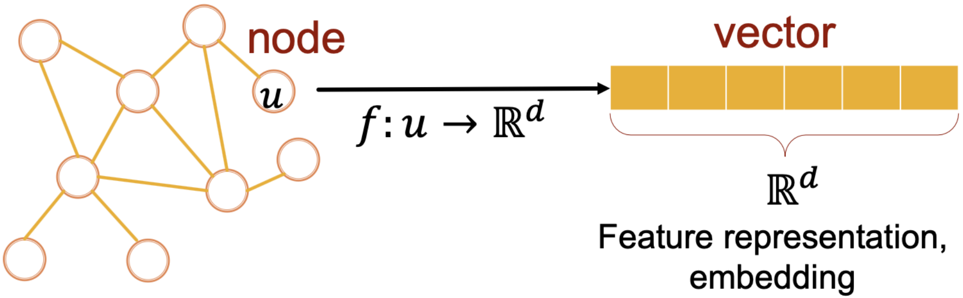
Graph Representation Learning

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


Input Graph
Structured Features
Learning Algorithm
Prediction
~~Feature Engineering~~
Representation Learning --
Automatically learn the features
Downstream prediction task

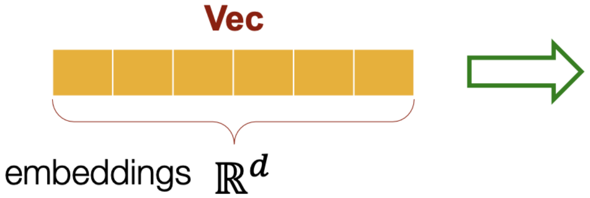
Graph Representation Learning

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



Why Embedding?

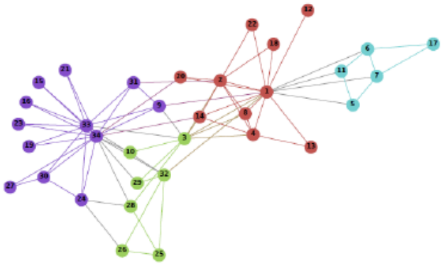
- **Task: Map nodes into an embedding space**
  - Similarity of embeddings between nodes indicates their similarity in the network. For example:
    - Both nodes are close to each other (connected by an edge)
  - Encode network information
  - Potentially used for many downstream predictions



Tasks
- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
- ....

Example Node Embedding

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


Input

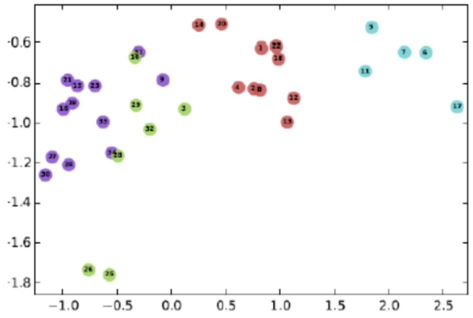
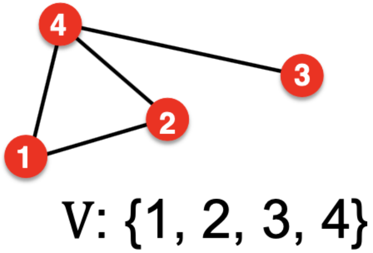

Output

Image from: Perozzi et al. DeepWalk: Online Learning of Social Representations. *KDD 2014*.

Setup

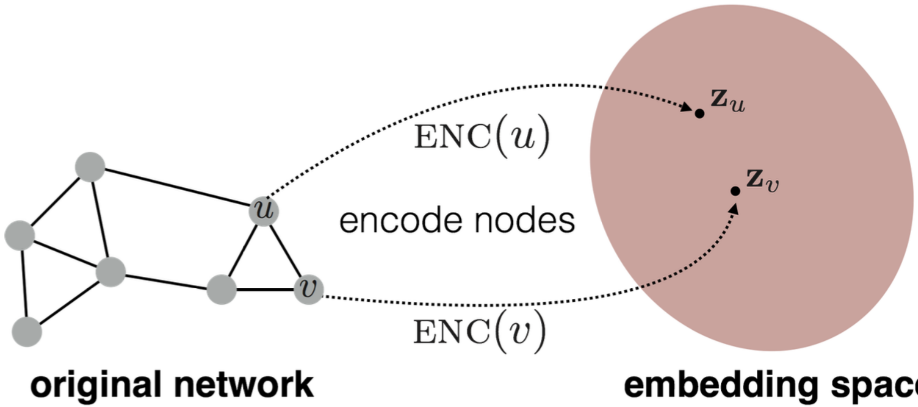
- **Assume we have an (undirected) graph G:**
  - V is the vertex set.
  - A is the adjacency matrix (assume binary).
  - **For simplicity: No node features or extra information is used**



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

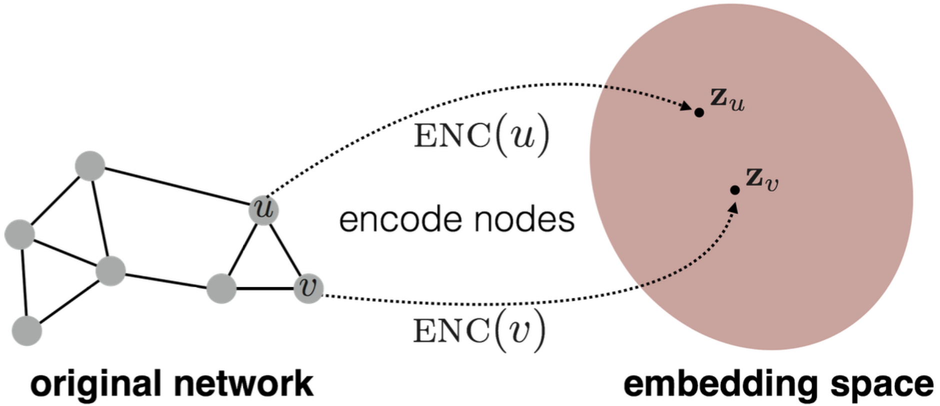
Embedding Nodes

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



Embedding Nodes

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



Learning Node Embeddings

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

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

Two Key Components

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

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

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

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

"Shallow" Encoding

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

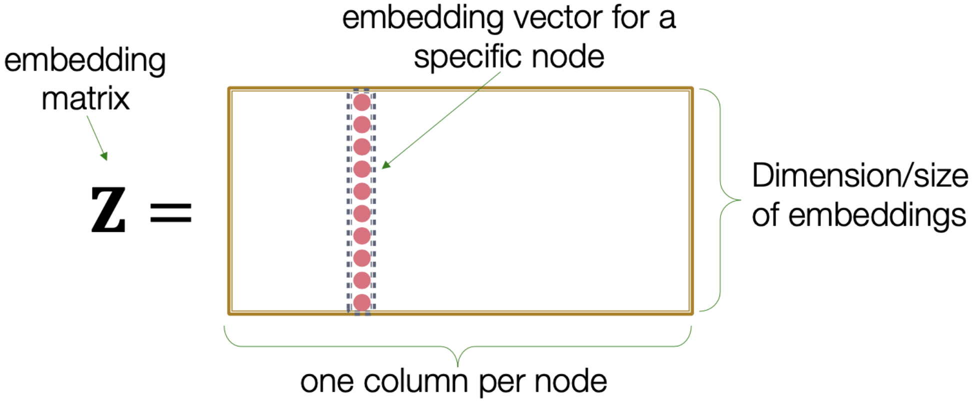
$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

$\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ matrix, each column is a node embedding [what we learn / optimize]

$v \in \mathbb{I}^{|\mathcal{V}|}$ indicator vector, all zeroes except a one in column indicating node v

"Shallow" Encoding

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



embedding matrix
$\mathbf{Z} =$
embedding vector for a specific node
Dimension/size of embeddings
one column per node

"Shallow" Encoding

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

Each node is assigned a unique embedding vector

(i.e., we directly optimize
the embedding of each node)

Many methods: DeepWalk, node2vec

Framework Summary

- **Encoder + Decoder Framework**
    - Shallow encoder: Embedding lookup
    - Parameters to optimize: $\mathbf{Z}$ which contains node embeddings $\mathbf{z}_u$ for all nodes $u \in V$
    - We will cover deep encoders in the GNNs

- **Decoder:** based on node similarity.
- **Objective:** maximize $\mathbf{z}_v^T \mathbf{z}_u$ for node pairs (u, v) that are similar.

How to Define Node Similarity?

- Key choice of methods is **how they define node similarity**.

- Should two nodes have a similar embedding if they...
  - are linked?
  - share neighbors?
  - have similar "structural roles"?

- We will now learn node similarity definition that uses **random walks**, and how to optimize embeddings for such a similarity measure.

Note on Node Embeddings

- This is **unsupervised/self-supervised** way of learning node embeddings.
  - We are **not** utilizing node labels.
  - We are **not** utilizing node features.
  - The goal is to directly estimate a set of coordinates (i.e., the embedding) of a node so that some aspect of the network structure (captured by DEC) is preserved.
- These embeddings are **task independent**:
  - They are not trained for a specific task but can be used for any task.

Notation

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

Non-linear functions used to produce predicted probabilities

- **Softmax function**:
  - Turns vector of K real values (model predictions) into K probabilities that sum to 1:

$$S(\mathbf{z})[i] = \frac{e^{z[i]}}{\sum_{j=1}^{K} e^{z[j]}}$$

- **Sigmoid function**:
  - S-shaped function that turns real values into the range of $(0, 1)$. Written as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Random Walk

random_walk_diagram.png

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.*

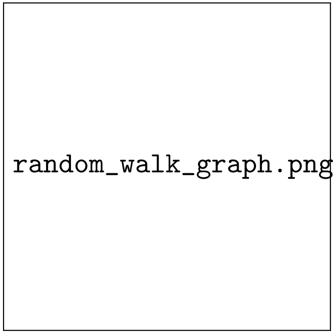
The (random) sequence of points visited this way is a random walk on the graph.

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.



$$P_R(v \mid u)$$

2. **Optimize embeddings to encode these random walk statistics:**

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 $f : u \to \mathbb{R}^d$:

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

- Log-likelihood objective:

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

- $N_R(u)$ is the neighborhood of node u by strategy R.

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

Random Walk Optimization

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

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

$\Longrightarrow$ Maximum likelihood objective

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

Random Walk Optimization

Equivalently,

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

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

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

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

Why softmax? We want node v to be most similar to node u (out of all nodes n).
Intuition: $\sum_i \exp(x_i) \approx \max \exp(x_i)$

Random Walk Optimization

Putting it all together:

$$\mathcal{L} = \underbrace{\sum_{u \in V}}_{\text{sum over all nodes } u} \underbrace{\sum_{v \in N_R(u)}}_{\text{sum over nodes } v} -\log\left(\underbrace{\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}}_{\text{predicted probability of } u \text{ and } v}\right)$$

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!

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

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

Solution: Negative Sampling

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

Sigmoid function (makes each term a "probability" between 0 and 1)

Random distribution over nodes

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

Negative sampling allows for quick likelihood calculation.

Why is the approximation valid? Technically, this is a different objective. But Negative Sampling is a form of Noise Contrastive Estimation (NCE) which approx. maximizes the log probability of softmax.
New formulation corresponds to using a logistic regression (sigmoid func.) to distinguish the target node v from nodes n_i sampled from background distribution P_v.
More at https://arxiv.org/pdf/1402.3722.pdf.

Negative Sampling

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

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 $k = 5\text{-}20$.

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?

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

Gradient Descent: a simple way to minimize $\mathcal{L}$:

- Initialize $\mathbf{z}_u$ at some randomized value for all nodes u.
- Iterate until convergence:
  - For all u, compute the derivative $\frac{\partial \mathcal{L}}{\partial \mathbf{z}_u}$.
  - For all u, make a step in reverse direction of derivative:

$$\mathbf{z}_u \leftarrow \mathbf{z}_u - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{z}_u}$$

η: learning rate

Stochastic Gradient Descent

Stochastic Gradient Descent: Instead of evaluating gradients over all examples, evaluate it for each **individual** training example.

- Initialize $\mathbf{z}_u$ at some randomized value for all nodes u.
- Iterate until convergence:
    - $\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v \mid \mathbf{z}_u))$
    - Sample a node u, for all v calculate the gradient $\frac{\partial \mathcal{L}^{(u)}}{\partial \mathbf{z}_v}$.
    - For all v, update:

$$\mathbf{z}_v \leftarrow \mathbf{z}_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial \mathbf{z}_v}$$

Random Walks: Summary

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

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

We can efficiently approximate this using negative sampling!

How should we randomly walk?

- So far we have described how to optimize embeddings given a random walk strategy R.

What strategies should we use to run these random walks?

- Simplest idea: **Just run fixed-length, unbiased random walks** starting from each node (i.e., DeepWalk from Perozzi et al.)
- The issue is that such notion of similarity is too constrained.

How can we generalize this?

Reference: Perozzi et al. DeepWalk: Online Learning of Social Representations. *KDD*.

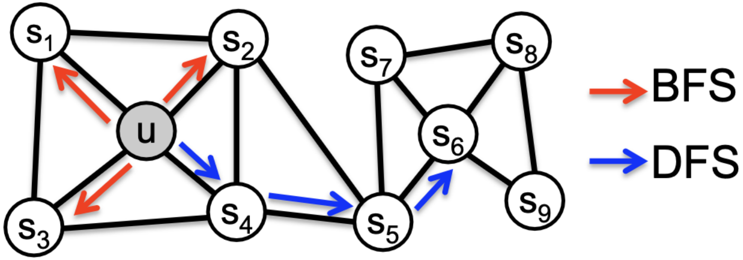
Overview of node2vec

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

Reference: Grover et al. node2vec: Scalable Feature Learning for Networks. *KDD*.

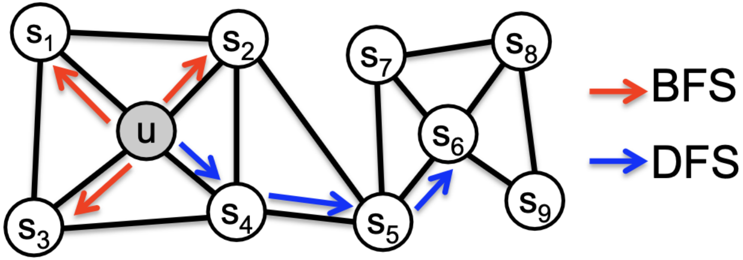
node2vec: Biased Walks

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



node2vec: Biased Walks

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

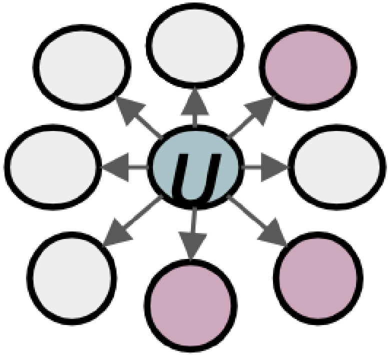


Walk of length 3 ($N_R(u)$ of size 3):

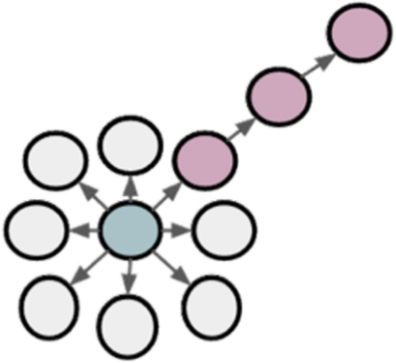
$N_{\text{BFS}}(u) = \{s_1, s_2, s_3\}$ Local microscopic view

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

BFS vs. DFS



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



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

Interpolating BFS and DFS

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

Random walk has two parameters:

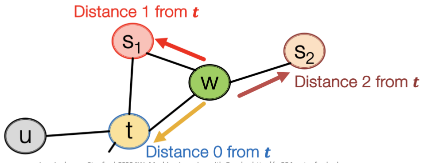
- **Return parameter p:**
  - Return back to the previous node.
- **In-out parameter q:**
  - Moving outwards (DFS) vs. inwards (BFS) from the previous node.
  - Intuitively, q is the "ratio" of BFS vs. DFS.

Next, we specify how a single step of biased random walk is performed.
Random walk is then just a sequence of these steps.

One Step of the Biased Random Walk

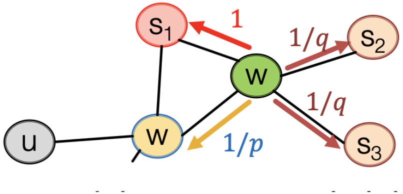
Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node w:

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



One Step of the Biased Random Walk

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

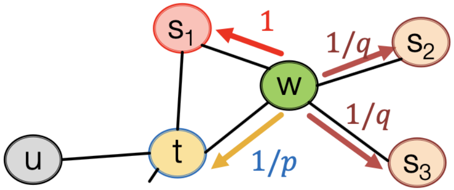


- **p, q model transition probabilities**
  - p … return parameter
  - q … "walk away" parameter

One Step of the Biased Random Walk

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

How to set edge transition probabilities?



$w \rightarrow$

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

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

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

node2vec algorithm

1. **Compute edge transition probabilities:**
    - For each edge (t, w) we compute edge walk probabilities (based on p, q) of edges $(w, \cdot)$.
2. **Simulate r random walks of length l starting from each node u.**
3. **Optimize the node2vec objective using Stochastic Gradient Descent.**

Linear-time complexity

All 3 steps are individually parallelizable

Other Random Walk Ideas

Different kinds of biased random walks:

- Based on node attributes (Dong et al.).
- Based on learned weights (Abu-El-Haija et al.).

Alternative optimization schemes:

- Directly optimize based on 1-hop and 2-hop random walk probabilities (as in LINE from Tang et al.).

Network preprocessing techniques:

- Run random walks on modified versions of the original network (e.g., Ribeiro et al. struct2vec, Chen et al. HARP).

Summary so far

Core idea: Embed nodes so that distances in embedding space reflect node similarities in the original network.

Different notions of node similarity:

- **Naive:** Similar if two nodes are connected.
- **Random walk approaches:** (covered today).

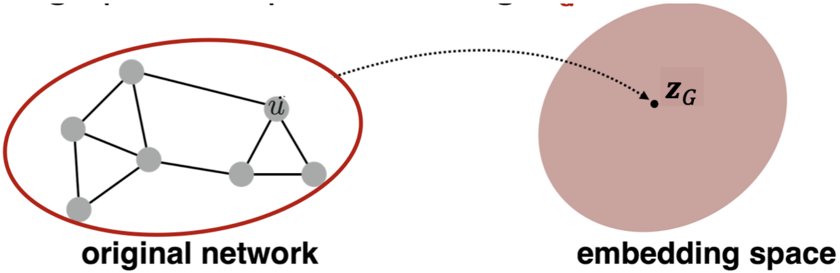
Summary so far

So, what method should I use..?

- No one method wins in all cases....
  - E.g., node2vec performs better on node classification while alternative methods perform better on link prediction (Goyal and Ferrara, survey).
- Random walk approaches are generally more efficient.
- **In general:** Must choose definition of node similarity that matches your application.

Embedding Entire Graphs

Goal: Want to embed a subgraph or an entire graph G. Graph embedding: z_G.



Tasks:

- Classifying toxic vs. non-toxic molecules
- Identifying anomalous graphs

Approach 1

Simple (but effective) approach 1:

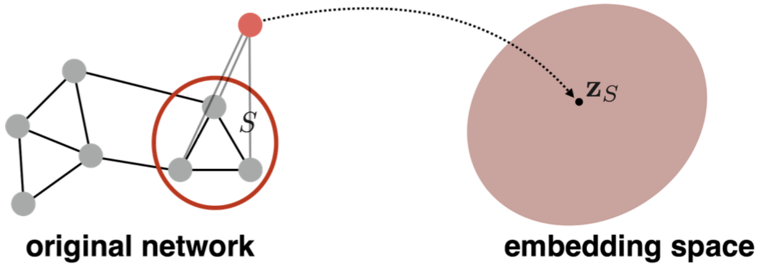
- Run a standard graph embedding technique *on* the (sub)graph G.
- Then just sum (or average) the node embeddings in the (sub)graph G.

$$z_G = \sum_{v \in G} z_v$$

Used by Duvenaud et al. to classify molecules based on their graph structure.

Approach 2

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



Proposed by Li et al. as a general technique for subgraph embedding.

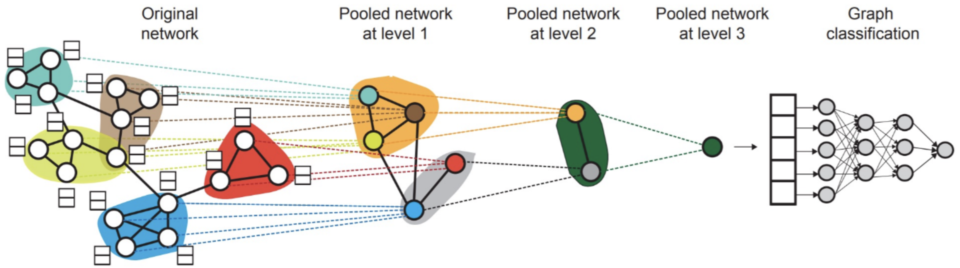
Summary

We discussed 2 ideas to graph embeddings:

- **Approach 1:** Embed nodes and sum/avg them
- **Approach 2:** Create super-node that spans the (sub)graph and then embed that node.

Preview: Hierarchical Embeddings

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


Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3
Graph classification

The End?