Random Walk Approach for Node Embedding

# Notation

Vector zu --- The embedding of node u (what we aim to find).

Probability P(v|zu) --- Our model prediction based on zu

* The (predicted) probability of visiting node v on random walks starting from node u

Non-linear functions used to product predicted probabilities:

* Softmax function --- Turns vector of K real values (model predictions) into K probabilities that sum to 1:
* Sigmoid function --- S-shaped function that turns real values into the range of (0, 1) Written as:

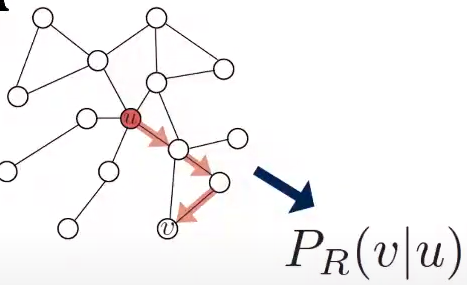
# Random Walk

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 graph.

zuTzv = 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. **A graph of mathematical equations

   Description automatically generated**Optimize embeddings to encode these random walk statistics:

**Similarity in embedding space (dot product) encodes random walk “similarity”**

# Why Random Walks?

1. **Expressive** --- 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)

1. **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 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?

* NR(u) --- neighborhood 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->Rd: f(u) = z(u)

Log-likelihood objective:

* NR(u) is the neighborhood of node u by strategy R

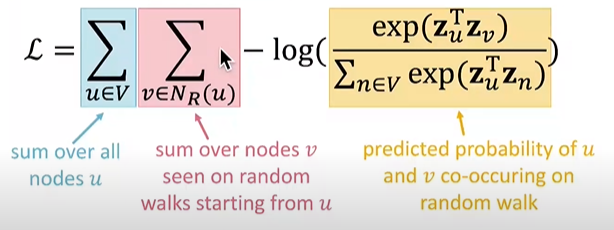
Give node u, we want to learn feature representations that are predictive of the nodes in its random walk neighborhood NR(u)

# 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 NR(u), the multiset\* of nodes visited on random walks starting from u.
3. Optimize embeddings according to: Given node u, predict its neighbors NR(u), we can change from Log-likelihood. Equivalently,

* Intuition --- Optimize embeddings zu to maximize the likelihood of random walk co-occurrences.
* Parameterize P(v|zu) using softmax:

Note: we use softmax function, because we want node v to be most similar to node u (out of all nodes n)

Putting it all together:

Optimizing random walk embeddings = Finding embeddings zu that minimize L

Note: But doing this natively is too expensive! Nested sum over nodes gives O(|V|2) complexity!

# Negative Sampling

To solve the above problem, we come up with negative sampling:

similar to:

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

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)

# Stochastic Gradient Descent

After we obtained the objective function, how do we optimize (minimize) it?

Gradient Descent --- A simple way to minimize L:

* Initialize zi at some randomized value for all i
* Iterate until convergence:
* For all i, compute the derivative
* For all i, make a step towards the direction of derivatize
* : learning rate

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

* Initialize zi at some randomized value for all i
* Iterate until convergence:
* Sample a node i, for all j calculate the derivative
* For all j, update:

# Random Walks Summary

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

# Overview of node2vec

**Goal** --- Embed nodes with similar network neighborhoods close in the feature space.

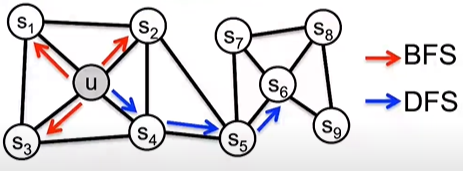
We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.

**Key observation** --- Flexible notion of network neighborhood NR(u) of node u leads to rich node embeddings.

Develop biased 2nd order random walk R to generate network neighborhood NR(u) of node u

# Node2vec --- Biased Walks

Idea --- use flexible, biased random walks that can trade off between local and global view of the network.

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

Walk of length 3 (NR(u) of size 3):

NBFS(u) = {S1, S2, S3} Local microscopic view

NDFS(u) = {S4, S5, S6} Global macroscopic view

**Interpolating BFS and DFS**

Biased fixed length random walk R that given a node u generates neighborhood NR(u)

Two parameters:

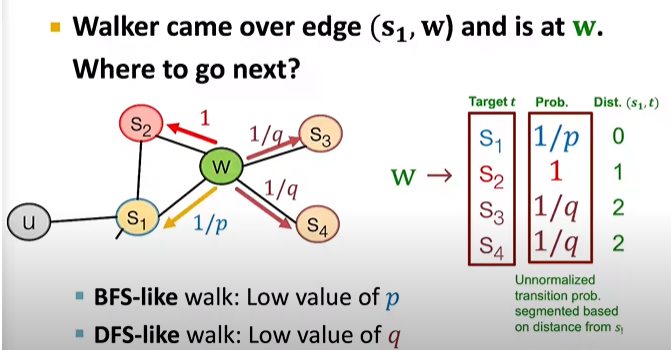
**Return parameter p** --- Return back to the previous node

**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 traverse edge (s1, w) and is now at w
* Insight --- Neighbors of w can only be:

# 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.

# Conclusion

No one method wins in all cases --- node2vec performs better on node classification while alternative methods perform better on link prediction.

Random walk approaches are generally more efficient.

In general: you must choose definition of node similarity that matches your application!