Notes for Stanford CS224W
Machine Learning with Graphs

Written by
The Healthy Birds Trio

Lecturer
Jure Leskovec

April 2, 2020

Contents

0 Preliminaries **3**
0.1 Acknowledgement . . . 3
0.2 Necessary Math . . . 3
0.3 Other Relevant Courses . . . 3

1 Introduction **4**
1.1 World Full of Graphs . . . 4
1.2 Real World Application of Graphs . . . 4

2 Structure of a Graph **9**
2.1 Spectral Clustering . . . 9

3 Application of Graphs **10**

4 Graph Representations **11**
4.1 Node Embedding . . . 11
4.2 Graph Embedding . . . 15

5 Neural Model for Graphs **18**
5.1 Components of Graph Convolution . . . 19
5.2 TransE, Translating Embeddings for Modeling Multi-relational Data . . . 20
5.3 TransR, Learning Entity and Relation Embeddings for Knowledge Graph Completion 21
5.4 GraphSAGE, Inductive Representation Learning on Large Graphs . . . 21
5.5 PinSage, Graph Convolutional Neural Networks for Web-Scale Recommender Systems . . . 21

5.6 GAT, Graph Attention Networks 21

0 Preliminaries

0.1 Acknowledgement

0.2 Necessary Math

NEEDS WORK: We should cover

1. Some matrix algebra (matrix multiplication, matrix derivative, eigenvalue, eigenvector, semi-definite)
2. Probabilities (Bayes' rule, conditional independence, union bound)
3. Basics on neural networks

0.3 Other Relevant Courses

Artificial intelligence in theory and in practice are connected to numerous sub-fields in computer science. As you might expect, contents taught in CS224W are also covered in other classes offered at Stanford. For your interest, and to our best knowledge,

CS 265 Randomized Algorithms goes in depth on probabilistic existence of edges, hence strongly related to spread of message (think disease transmission).

CS 261 A Second Course in Algorithms goes in depth on traditional graphs (max-flow min-cut) along with some probabilistic components. With CS261 you'll develop a much better understanding of theoretical graph problems that solve real world problems.

CS 228 Probabilistic Graphical Networks covers exactly what you think, Bayesian inference on graphs. This partially overlaps with CS265 and spends a considerable amount of time on message passing in graph.

CS 229 Machine Learning builds the foundation of machine learning. Though not directly relevant, it forms part of the traditional ML approach vs popular DL approach on data analysis.

CS 230 Deep Learning is a great place to start if you are relatively new to deep learning. CS224W expects you to have decent knowledge in deep learning and all graph neural network techniques build on top of "typical" deep learning approaches.

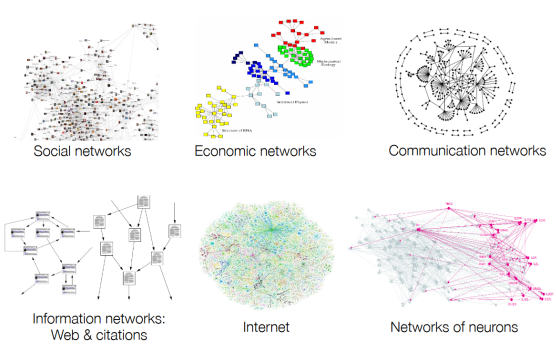
CS 246 Mining Massive Datasets also deals with interconnected data. Page-rank is one of the main topics in CS246 for those interested in the inner-workings of a search engine.

1 Introduction

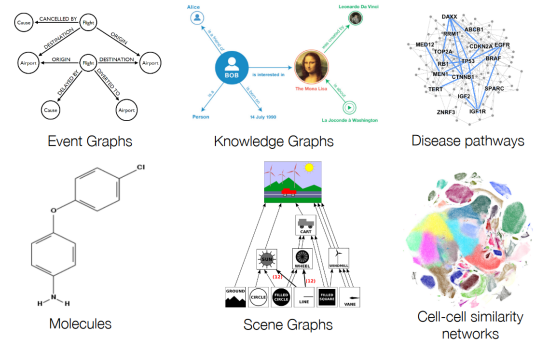
1.1 World Full of Graphs

Graphs are a natural way to describe complex interactions between entities. We use graphs/networks interchangeably in the notes, though graph is a more commonly seen in mathematical settings defined as $G(V, E)$.

Common networks include human society, chemical interactions, connection of neurons, knowledge graphs, etc. You can roughly separate those into (1) naturally defined (2) man-made, but the distinction is often difficult. As we will be discussing in later chapters, network relationships using traditional methods. We use *spectral clustering*?? to extract community association; *pagerank* to trace flow of trust; *message propagation* for probabilistic inference. In addition, we will also introduce the recently booming field of *graph neural networks*, whose effectiveness in understanding rich relational structure have been demonstrated by researchers.



(a) Lecture 1, Page 7



(b) Lecture 1, Page 10

1.2 Real World Application of Graphs

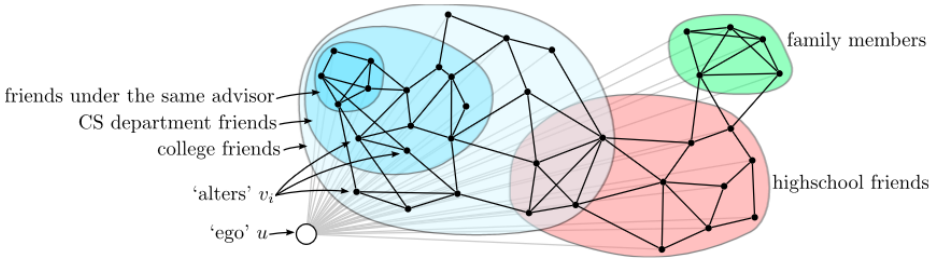
In general, our analysis of a network fall in the following categories:

- *Node classification*: Predict the type of a given node
- *Link prediction*: Predict the interaction (or existence of) between two nodes
- *Community detection*: Identify linked clusters of nodes
- *Network similarity*: Measure similarity among nodes/sub-graphs/whole networks

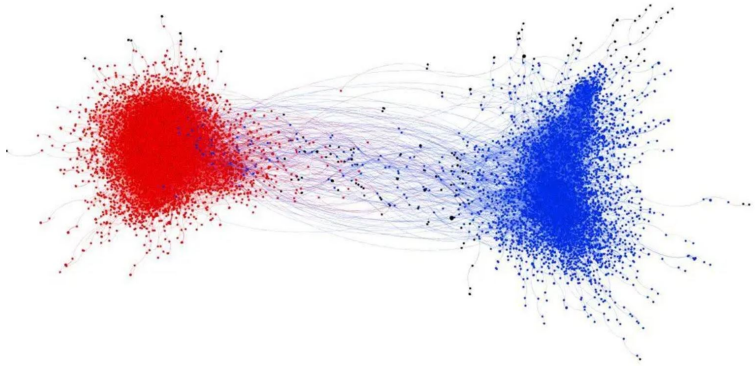
1.2.1 Social Network

We were used to be told there is 6-degree of separation. Researchers found in 2012 [link] that according to social graph built from Facebook data, average distance between people is in fact 3.74, much less than $4.4 - 5.7$ range discovered in 1967 [link] as the famous "The Small World Problem".

With clustering techniques we can also discover social circles. On the right is sample image extracted from a method [link] to identify social circles using network structure and user profiles.

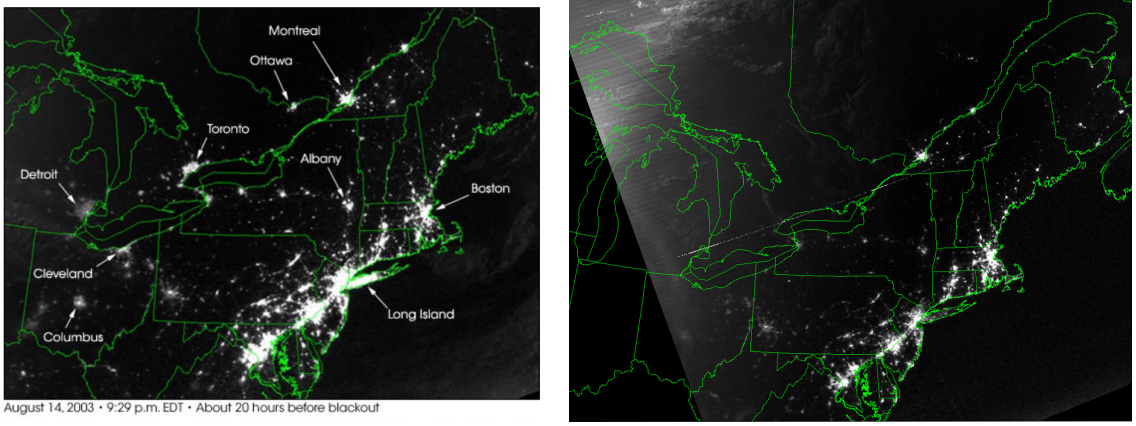


We can separate the re-tweet network along party lines using techniques similar to social circle detection. Explanation for the image below is [here].

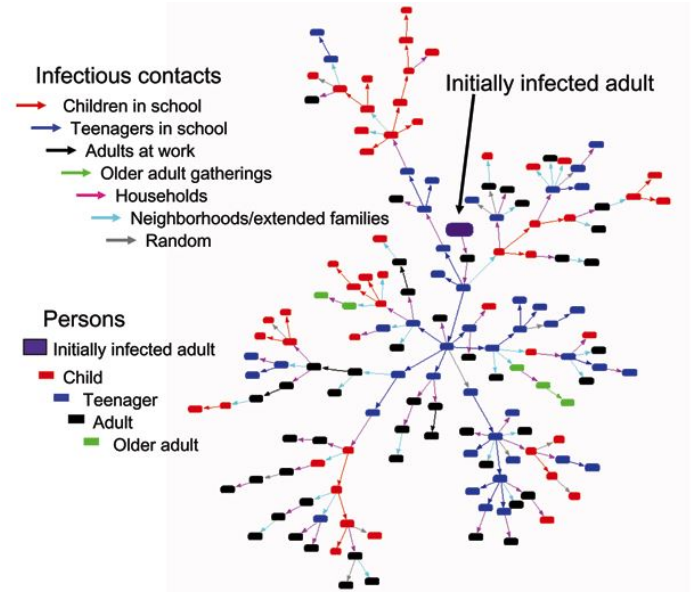


1.2.2 Influence Propagation

Network analysis is also useful for identifying weaknesses in infrastructure network. Below shows a blackout happened on August 15, 2003 (August 14 vs August 15), affecting much of the East Coast, affecting Canadian and American cities alike. Notice how Toronto/Detroid are completely gone and DC - Boston corridor is significantly dimmer. Higher resolution image [here]. With network analysis tools, we can find out which nodes (cities) will be affected, severity of the impact and speed of the spread.

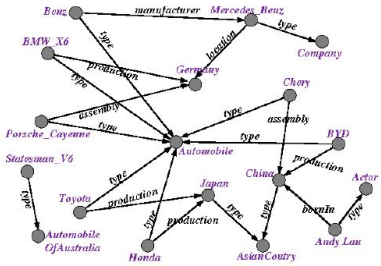


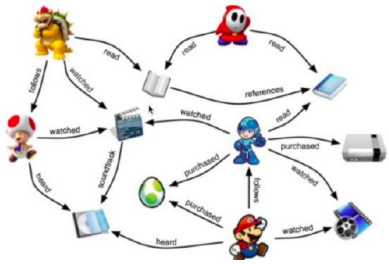
This failure propagation model is also applicable for spread for mis-information and spread of virus. With a virus spread model, you are modeling connectivity (probability of disease spread) between people, community, city and countries. Identifying and cutting off center of spread can localize the effect of disease spread. See CDC's model on pandemic influenza published back in 2006 [here].

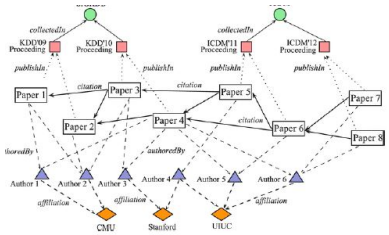


1.2.3 Knowledge Graph

Knowledge graph is a great example of heterogeneous graph, graphs that contain nodes with different meanings. Typically in a knowledge graph, there are item nodes and property/category nodes.



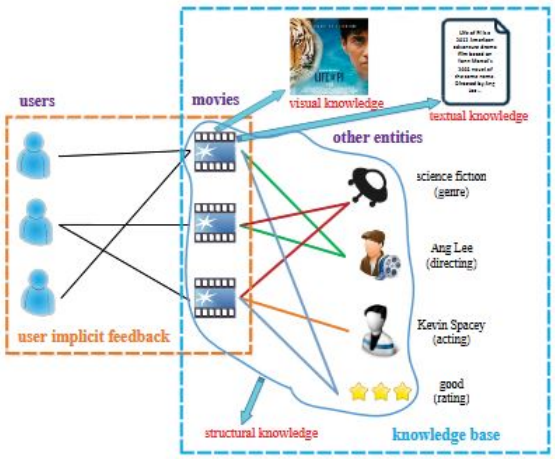




1.2.4 Recommender System

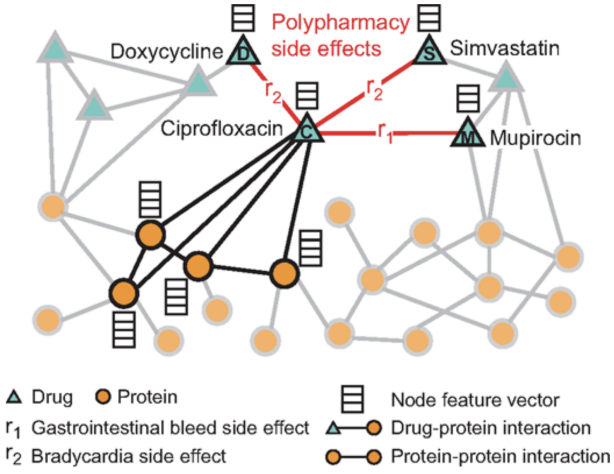
Predicting user preference can be abstracted to predicting existence of edges in a bipartite graph. CS246 covered concepts like SVD (singular value decomposition), but you can also solve this by treating user preference matrix as an adjacency matrix. In class, we showed that Pinterest has its own image-embedding based graph search algorithm, handling 300 million users, more than 4 billion pins and more than 2 billion boards. Notice that Pinterest is building a "tri-partite" graph with user, pins and boards.

It is not un-common to combine recommender system's bi-partite graph with an underlying knowledge graph. An obvious use case is movie recommendation, where movies have attributes like genre, theme, director, actor/actress, etc. This can be treated as an overlay of multiple disjoint graphs, but the fully combined graph has the best predictive power [link].



1.2.5 Biochemical Applications

Biological pathways, "network" derived from atomic structure of drugs and proteins, interaction/effect/side effect of drugs and even the food chain naturally form networks with potentially heterogeneous nodes. Using Graph Convolutional Networks to predict effect of chemicals has gained traction in recent years to compete with point-cloud based, 3D voxel convolution techniques. For example, effect of drug combinations can be modeled using a heterogeneous graph with nodes representing drug and protein [link].



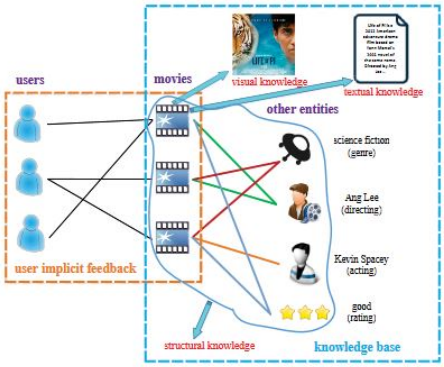
2 Structure of a Graph

2.1 Spectral Clustering

3 Application of Graphs

4 Graph Representations

As we have re-iterated several times, an important goal of studying graph structure is to (1) classify nodes (2) predict existence of links (3) classify (sub)graphs. In this chapter we explore mostly unsupervised ways of performing these tasks. A primary characteristic of graph data is that the data is "simple" but well-structured, compared to typical deep learning problems where each sample is feature rich individually but independent from all other samples. For example, for computer vision tasks, each image itself contains thousands or millions of pixels (e.g. $128 \times 128 = 16,384$ pixels, like ImageNet [link]). In comparison, with a movie recommendation dataset, all we have are user ID, their watched movies and movie properties, which are essentially large tables made of a few columns (like *The Movies Dataset* hosted by Kaggle [link]).

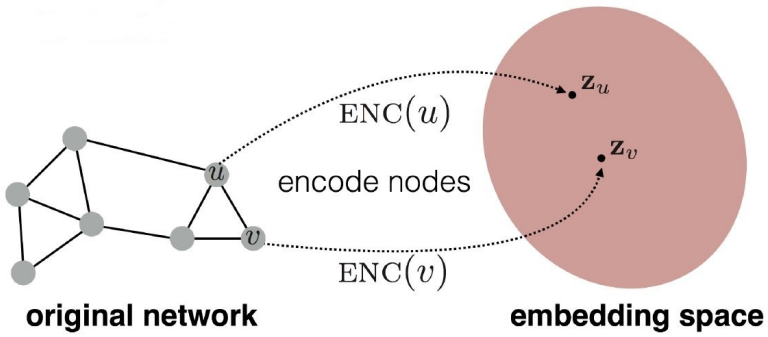


This approach differs from previous influence spread/belief propagation approach. Previous methods are statistical in nature, leveraging heavily knowledge related to Bayesian networks (those that you will learn in CS228). This chapter primarily focuses on, as the section title suggest, learning a graph representation in the form of an embedding that represent graph and its components in the latent space. We will be exploring methods to generate node embedding and graph embedding in the following sections.

4.1 Node Embedding

Generating embedding for nodes is analogus to generating word embeddings as part of a natural language processing task. Node with similar embeddings represent similar objects, much like words with similar embeddings have similar semantic meanings. Researchers evaluate embedding similarities in L1-distance, L2-distance (or Forbenius norm), and cosine-distance. Graph applications often choose cosine distance whereas CV/NLP tasks are less biased on the selection of similarity functions.

Putting it in simple terms, we want to encode nodes so that similarity in the embedding space approximates similarity in the original network. We will be defining what (1) encoding function (2) similarity function (3) neighbor definition (4) optimization function now.



4.1.1 Some Basic Ideas

Embedding lookup is rather simple. For a graph $G(V, E)$, a node can be represented by a one-hot indicator vector $v \in \mathcal{R}^{|V|}$, and d-dimensional embeddings are stored in a lookup matrix $Z \in \mathcal{R}^{d \times |V|}$. Common methods to generate the embedding matrix include DeepWalk, node2vec and TransE, all of which will be covered in later sections of the note.

$$ENC(V_i) = \mathbf{Z}v_i \tag{1}$$

Node similarity can defined in a number of ways, depending on how we want to discover/evaluate node similarity. Here, we use cosine similarity to evaluate node similarity. Recall that

$$\vec{a} \cdot \vec{b} = |a||b|cos(\theta) \tag{2}$$

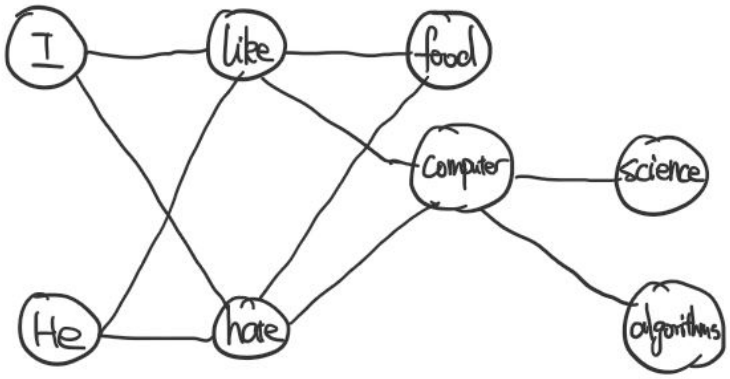
Node similarity in terms of cosine similarity can simply be expressed as the following. And of course, other similarity functions can also be employed, with appropriate change to optimization function that we will be discussing later (mostly in terms of taking derivative to minimize log-likelihood).

$$similarity(u, v) = \mathbf{Z}_u \cdot \mathbf{Z}_v \tag{3}$$
$$= \mathbf{Z}_u^T \mathbf{Z}_v \tag{4}$$

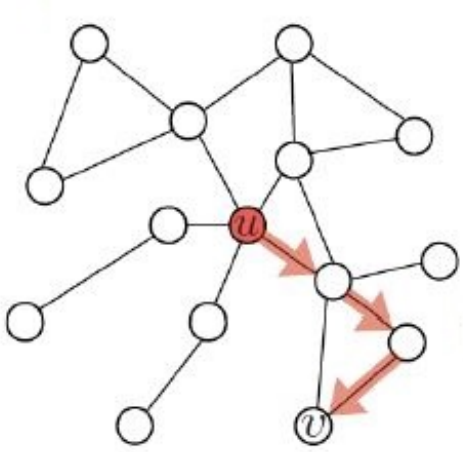
4.1.2 Random Walk

Here we are primarily dealing with one large graph, while the same argument applies to a number of smaller graphs as well with appropriately assigned weights to a series of smaller graphs.

Random walk is the most similar to observing sentences for a language modeling task. Imagine we convert all sentences into a large graph where graph neighbours are defined as words that are immediately next to each other, a walk along these paths should form sentence-like samples (or backwards, but backwards sentences are still useful for bi-directional recurrent models). For these language models, researchers define an *n-gram* (or *Skipgram*, depending on the setup) model that tracks probabilities of word co-occurance. See [here] for Prof. Dan Jurafsky's note on *N-Grams*.



For random walk in a defined graph structure starting from node v, the walk consists of nodes randomly selected from neighbours of the current head. Every time a head is selected, we move to the head then use neighours of this new head as candidates for the next move. You can view this as sampling from all potential "sentences", compared to NLP tasks where you are given many possible sentences. After sampling by random walk, similar to *N-Gram*/*Skipgram*, we want to find node co-occurance probabilities and match them with cosine distances between node pairs through optimizing log-likelihood.



Primary benefits for random walk is the following

- **Expressivity**: Walking distance and similarity function together allow incorporation of local and higher-order neighbourhood information
- **Efficiency**: Do no need to consider all node pairs and random selection is naturally weighted.

More formally, given graph $G = (V, E)$, embedding matrix to be optimized $\mathbf{Z} \in \mathcal{R}^{d|V|}$, neighbor (as defined in lecture) function $N_R(v)$ as set of nodes visited in a random walk following strategy R and probability function P which is really just proportional to the node similarity function. Likelihood and negative log-likelihood can be defined as the following

$$\mathcal{L} =_{u \in V} P(N_R(u)|z_u) \tag{5}$$

$$log\mathcal{L} = \sum_{u \in V} logP(N_R(u)|z_u) \tag{6}$$

$$= \sum_{u \in V} \sum_{v \in N_R(u)} logP(\mathbf{Z}_v|\mathbf{Z}_u) \tag{7}$$

Then for maximum negative log-likelihood

$$\rightarrow \max_{\mathbf{Z}} \sum_{u \in V} \sum_{v \in N_R(u)} -logP(\mathbf{Z}_v|\mathbf{Z}_u) \tag{8}$$

Here, we define $P(\mathbf{Z}_v|\mathbf{Z}_u)$ as

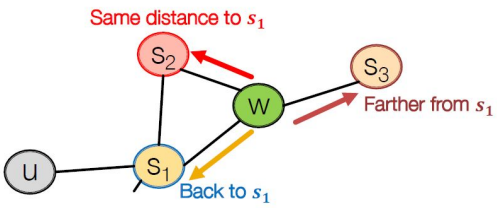
$$P(\mathbf{Z}_v|\mathbf{Z}_u) = \frac{exp(\mathbf{Z}_v^T \mathbf{Z}_u)}{\sum_{n \in V} exp(\mathbf{Z}_n^T \mathbf{Z}_u)} \tag{9}$$

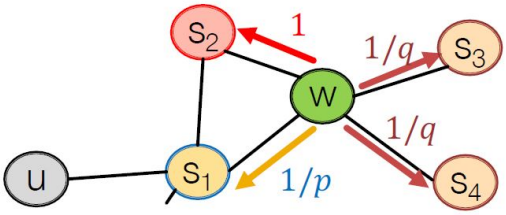
$$log\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} log \frac{exp(\mathbf{Z}_v^T \mathbf{Z}_u)}{\sum_{n \in V} exp(\mathbf{Z}_n^T \mathbf{Z}_u)} \tag{10}$$

We observe that denominator $\sum_{n \in V} exp(\mathbf{Z}_n^T \mathbf{Z}_u)$ have to be computed for all $v \in V$, resulting in $O(n^2)$ complexity. One way to alleviate this problem is simply sub-sample from V when computing the denominator sum. We could use "negative sampling" as used for *word2vec* explained [here]. Here, "positive" means nodes that are in $N_R(u)$ in any of the walks we performed, and "negative" means all other nodes not visited from walks initiated from u. Observe the following expression. We are `(1)` maximizing $exp(\mathbf{Z}_v^T \mathbf{Z}_u)$ and minimizing sum of similarities from negative samples. (exp is used here instead of σ for sigmoid, for simplicity, the same result is achieved regardless). This way, we are minimizing the probability of a random node having high similarity with the source node. Number of negative samples (k) is typically $5 - 20$, depending on the application and degree of vertices in the graph. The random walks are often fixed length without preventing repeats, like *DeepWalk* implementation [link]. This paper contains pseudocode and more formal work on random walk and *skipgram*. For a more intuitive explanation on *hierarchical softmax*, which is not immediately relevant in this context, can be found [here].

$$log\frac{exp(\mathbf{Z}_v^T\mathbf{Z}_u)}{\sum_{n\in V} exp(\mathbf{Z}_n^T\mathbf{Z}_u)} = log(exp(\mathbf{Z}_v^T\mathbf{Z}_u)) - \sum_{n\in neg\_sampling(V,walks)} exp(\mathbf{Z}_n^T\mathbf{Z}_u) \tag{11}$$

Biased random walk is a random walk with a predefined node selection probability. *node2vec* proposes a wal strategy that balances exploration of local neighborhood and going further out. As shown below, S_2 is the same "level" as W, whereas S_3 is further out. We assign different probabilities to S_2 $versus$ S_3, S_4. That is, there is $\propto 2/q$ chance of going further, $\propto 1/p$ chance of returning to S_1 and $\propto c$ chance of going to S_2. (Values shown in labels need to be normalized).





This is only a different strategy of defining random walk preference ($N_R(v)$). The same gradient descent method for maximizing negative log-likelihood still applies.

We will cover *TransE* later in the notes because it falls in the more "neural" approach compared to the above more traditional approach.

4.2 Graph Embedding

Graph(sub-graph) embedding involves transforming embedding of nodes into one embedding for the entire (sub)graph. Typical embedding 'joint" methods include

- Concatenation
- Hadamard (element-wise product)
- Sum/Avg (element-wise sum/average)
- Distance (distance between embeddings with some distance function suitable for 2 or more vectors)

There are clearly more effective methods than the above embedding joining methods that lead to higher quality graph-level embeddings.

4.2.1 Naive Concatenation

As said, we just naively run some node embedding algorithm (neural or traditional), then average all node embeddings [link]. This is the most robust as embeddings will have fixed length and not blow up/vanish in scale if sum or product is taken.

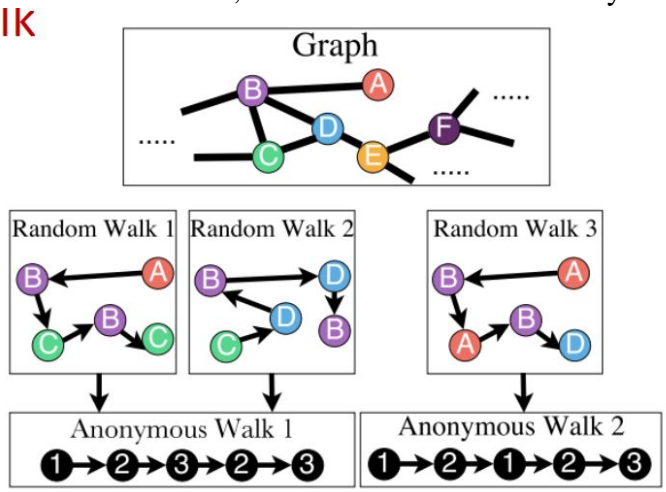
4.2.2 Virtual Node

Instead of averaging embeddings, we create a virtual node that connects to the entire graph (or relevant nodes only, for sub-graph) then compute embedding for the virtual node. Analogy of this is creating a source/sink pair when we are transforming difficult graph problems to max flow min cut problems that have well-known fast-ish solution. See the publication [here].

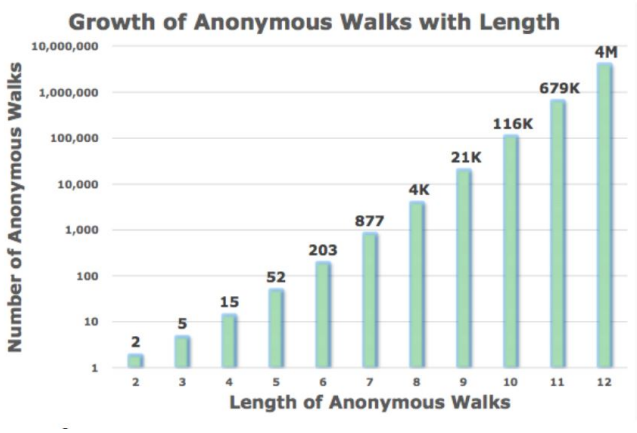
4.2.3 Anonymous Walk

Anonymous walk is a fancy name of saying instead of having a $|V|$ size one-hot vector, we only use a size k one-hot vector for a k-length random walk. That is, we only care about whether a different node has been visited in a walk, rather than the actual identity of the node.

IK



As expected, total number of different k-length grows as k increases. In fact, it should grow proportionally (but faster than) to $\frac{k^k}{p!}$ (think about how many, allowing repeats, different combinations can exist). This estimate is NOT accurate because it over-penalizes low k scenarios. The growth chart below reflect the true number of anonymous walks. The approximation ratio grows (or shrinks, depends on how you define approximation ratio) at approximately 2.



Idea 1, enumerate all walks We define d as number of distinct random walks for length-k walks. Then each $\mathbf{Z}_v^i$ is the probability of obtaining the i-th anonymous walk starting the walk at node v. Without considering weight associated with bias, we can enumerate all possible walk paths then for each node, we compute the probability of obtaining the i-th anonymous walk starting at node v.

Idea 2, sample and deal with sampling error We can obtain the previous probability by sampling a large number of random walks then process for each node. Here we use some statistical trick to argue that to obtain error of no more than ϵ with probability less than δ, we need to sample

$$m = \left[\frac{2}{\epsilon^2}(log(2^\eta - 2) - log(\delta))\right] where\ \eta \text{ is number of different anonymous walks for } k\text{-length walk} \tag{12}$$

NEEDS WORK: I suspect this is derived from multiplicative form of Chernoff bound. Wikipedia page for Chernoff bound is not a bad place to start [link].

Idea 3, use Skipgram again Notice that the above "dumb" methods are what NLP researchers used to create language model. **Idea 2** is not much different for the argument that negative sampling reaches approximately the same error, if you spent the time of digging all the formalities out. Therefore, **Idea 3** is simply treat each anonymous walk as a "token". Suppose from node v we can have anonymous walks $w_1, w_2 \dots w_t$, then as defined for *Skipgram*, we want to maximize $P(w_m | \{w_1, w_2 \dots w_t\} \setminus w_m)$. We are claiming that $N_R(u) = \{w_1, w_2 \dots w_t\}$, re-using the neighborhood notation we user earlier for node embedding. All other expressions for *Skipgram* applies and we want to maximize, as in lecture

$$max \frac{1}{T} \sum_{t=\delta}^{T} logP(w_t|w_{t-\delta}, \dots w_{t-1}) \tag{13}$$

$$P(w_t|w_{t-\delta}, \dots w_{t-1}) = \frac{exp(y(w_t))}{\sum_i^{\eta} exp(y(w_i))} \tag{14}$$

$$y(w_t) = b + U \cdot (\frac{1}{\delta} \sum_{i=1}^{\delta} v_i) \tag{15}$$

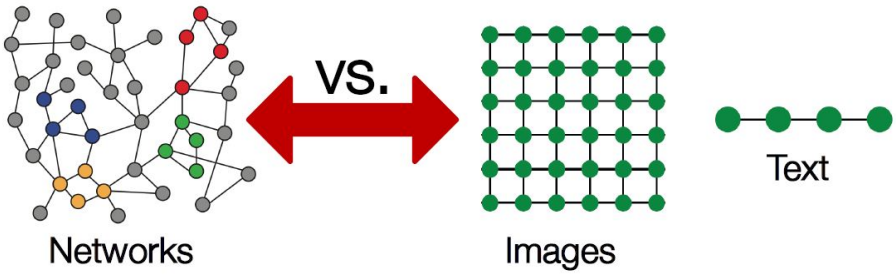
Formal derivation of this part of the notes is discussed in the *Anonymous Walk Embedding* paper [link].

5 Neural Model for Graphs

In the previous section, we discussed traditional methods of generating embeddings for node and graphs. In this section, we move on to neural models that started gaining tracking only in recent years (approximately 2013, when the foundational *TransE* paper was published [link]). This progress, viewing it from the perspective of Stanford's AI course offerings, is from CS229 (machine learning) to CS230 (deep learning). In terms of actual methods used, we are moving from *word2vec* and *Skipgram* model to convolutional and recurrent neural networks. For graph related topics, the following content is the neural "equivalent" for CS228 (probabilistic graphical models).

There are several problems with getting embedding from random walks as shown in Section 4. `(1)` these node embeddings are generated independently (at best locally) from each other (remember that we are only approximating the log-likelihood) `(2)` we cannot generate node embedding for future nodes, which means the graph itself is pre-determined, `(3)` the incorporation of local/macro level structure is questionable despite us playing with biased random walk while leaving structure of the graph itself on the table.

Applying existing models like convolutional neural network and recurrent neural network sounds problematic because they are designed for samples that has internal structures, not samples that are well-structured (but not following a fixed pattern) to connect to other samples. For example, in the case of image convolution, we define a $d_1 \times d_2 \times c$ (dim, dim, channel) kernel that scans the image. This is clearly not possible as we cannot define a fixed size "kernel" for graphs because nodes do not all have the same edge connectivities.



5.1 Components of Graph Convolution

Naturally, convolution is a form of recursion where recursion depth is number of layers we include in the convolution. To fully define this "recursion" we need to figure out `(1)` what do to with embedding from previous layer? `(2)` what message is passed from node in the previous layer to the current node `(3)` how to aggregate messages from previous layer `(4)` how to aggregate/post-process this aggregated message with embedding of the current node. Putting everything together, for h_v^k as embedding for node v after k-th recursion,

$$h_v^k = \phi(AGG_1\Big[op_1\big[(AGG_2(\{MSG(u,v), \forall u \in N(v)\})\big], op_2(h_v^{k-1})\Big]) \tag{16}$$

We define the various "placeholder functions" as the following. I'm sure you can also convert computer vision task to something like this where h_v^k is equivalent to a pixel-level multi-channel vector.

- $\phi \rightarrow$ Activation function for embedding (tanh, softmax, etc.)
- $AGG_1 \rightarrow$ Aggregation function to join embedding from previous layers and embedding of current node from previous step (concatenation, sum/avg, attention, etc.)
- $op_1 \rightarrow$ Weight assigned to messages from previous layer after aggregation (typically just a matrix multiply)
- $AGG_2 \rightarrow$ Aggregation function to join embedding from nodes in the previous layer(concatenation, sum/avg, attention, etc.)
- $AGG_2 \rightarrow$ Message function to compute message from nodes in the previous layer to current node (depends on whether edge embeddings exist or if we are handling non-existing edges differently)
- $op_2 \rightarrow$ Weight assigned to embedding of the current node from previous iteration (typically just a matrix multiply)

- $N \rightarrow$ Neighbour of a node (typically immediate neighbours, but researchers have used other definitions, such as *RippleNet*, [link])

On top of this, of course, there is also embedding initialization methods and loss function for gradient descent. Little generalization can be applied to these functions as they are task dependent.

A comprehensive survey of various graph neural network methods as of 2019 is [here] and [here, a bit older]. For this class, we discuss *TransE* (Sec 5.2), *TransR* (Sec 5.3), *GraphSAGE* (Sec 5.4), *PinSage* (Sec 5.5) and *GAT* (Sec 5.6).

5.2 TransE, Translating Embeddings for Modeling Multi-relational Data

The foundational TransE implementation was proposed in 2013 [link]. We reproduce the algorithm (not a screenshot) below. We define (h, l, t) triplet as (head entity, relation, tail entity), or graphically, $v_h \xrightarrow{l} t_h$.

Algorithm 1 Learning **TransE**

input: Training set $S = \{h, l, t\}$, entities and rel.sets E and L, margin γ, embeddings dim. k.

initialization

$l = uniform(-\frac{6}{\sqrt{k}}, -\frac{6}{\sqrt{k}})$ $\quad \forall l \in L$ $\quad \triangleright$ $\frac{6}{\sqrt{k}}$ is most likely from Xavier initialization

$l = l/||l|| \quad \forall l \in L$

$e = uniform(-\frac{6}{\sqrt{k}}, -\frac{6}{\sqrt{k}})$ $\quad \forall e \in E$

while *Loss not converged* **do**

$e = e/||e|| \quad \forall e \in E$

$S_{batch} = rand\_sample(S, b)$ $\quad \triangleright$ sample b triplets from set S

$T_{batch} = \emptyset$ $\quad \triangleright$ Set initial pairs of triplets to be empty

for $(h, l, t) \in S_batch$ **do**

$(h', l, t') = rand\_sample(S')$ $\quad \triangleright$ Sample an invalid (h', l, t') triplet (exactly one or two of h, l, t have to be different)

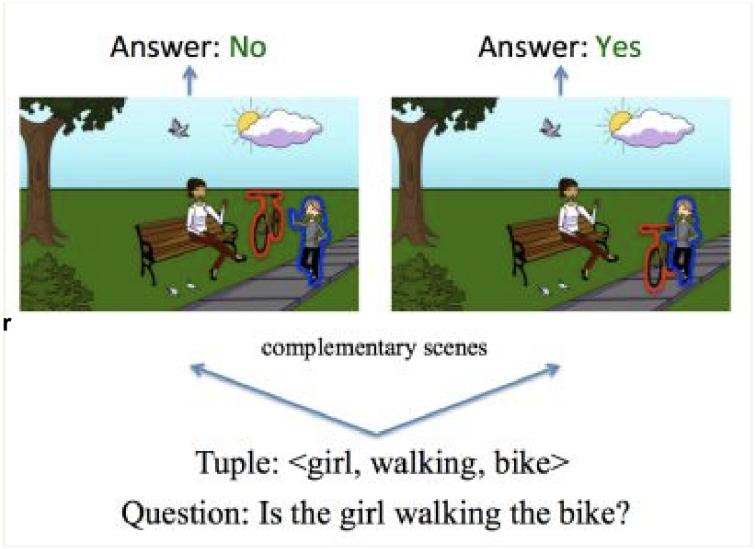
$T_{batch} = T_{batch} \cup \{((h, l, t), (h', l, t'))\}$

end

Gradient descent on $\sum_{((h,l,t),(h',l,t')) \in T_{batch}} \nabla[\gamma + d(h + l, t) - d(h' + l, t')]$

end

To explain what this "sample an invalid triplet" is all about, let's use a visual Q&A example [link]. With this example, you want to make sure that embedding of an image with girl *walking* a `bike` is different from images boy *walking* a `bike`. The analogy is not exactly the same as *TransE*, but you should be able to see the rationale behind setting up the gradient update rule.



5.3 TransR, Learning Entity and Relation Embeddings for Knowledge Graph Completion

5.4 GraphSAGE, Inductive Representation Learning on Large Graphs

5.5 PinSage, Graph Convolutional Neural Networks for Web-Scale Recommender Systems

5.6 GAT, Graph Attention Networks