Notes for Stanford CS224W
Machine Learning with Graphs

Written by
The Healthy Birds Trio

Lecturer
Jure Leskovec

April 19, 2020

Contents

0 Preliminaries

0.1 Acknowledgement

We thank Professor Jure Leskovec for a great quarter in Fall 2019. It was an inspiring experience to learn methods for analyzing graphs and explore the frontier of neural methods for graph. CS224W is definitely a great course on networks, find the most up to date course website [here].

We would also like to acknowledge the effort of TAs and students who compiled this collection of class notes. We hope this note serve as an extension to the existing notes.

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.

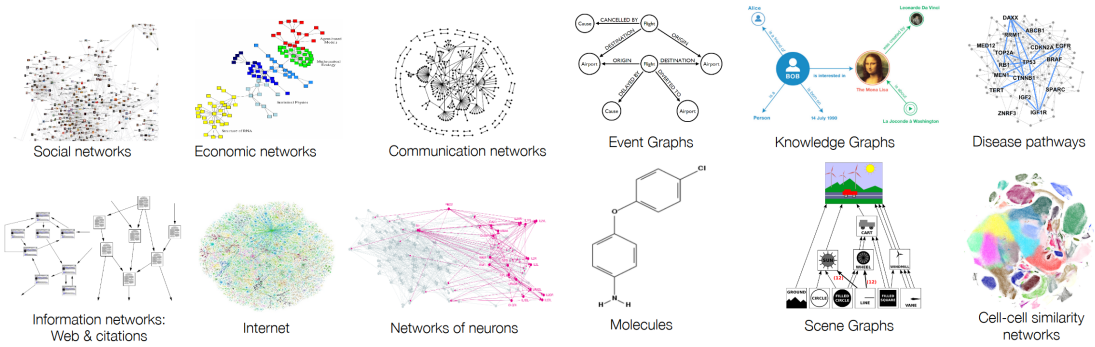
CS 246 Mining Massive Datasets also deals with interconnected data. Page-rank is one of the main topics in CS246 for those interested in the inner-workings of a search engine.

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.



1.2 Real World Application of Graphs

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

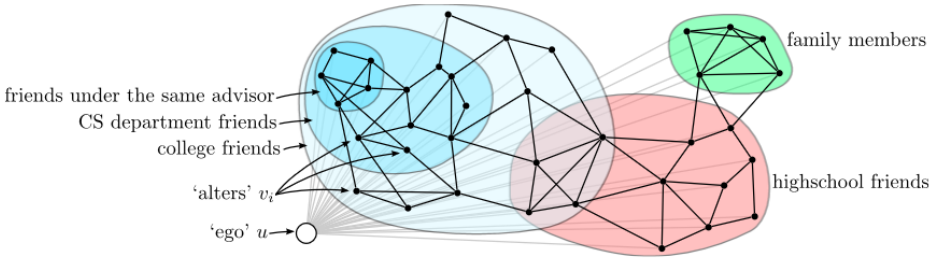
- *Node classification*: Predict the type of a given node
- *Link prediction*: Predict the interaction (or existence of) between two nodes
- *Community detection*: Identify linked clusters of nodes
- *Network similarity*: Measure similarity among nodes/sub-graphs/whole networks

1.2.1 Social Network

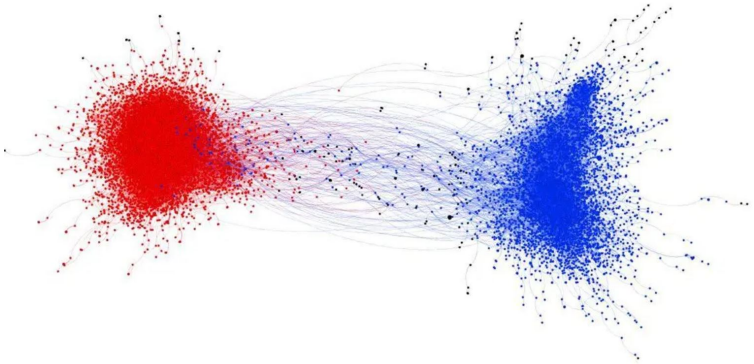
We were used to be told there is 6-degree of separation. Researchers found in 2012 [link] that according to social graph built from Facebook data, average distance between people is in fact

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.

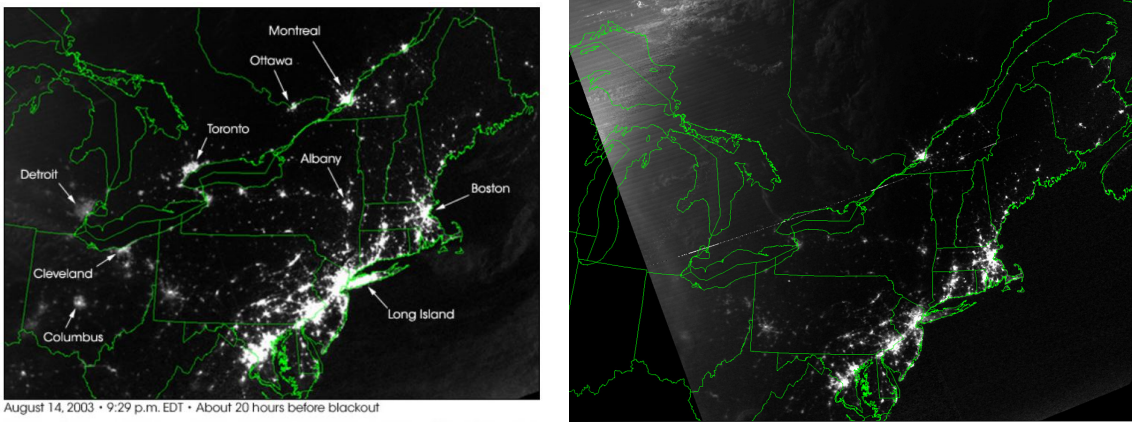


We can separate the re-tweet network along party lines using techniques similar to social circle detection. Explanation for the image below is [here].

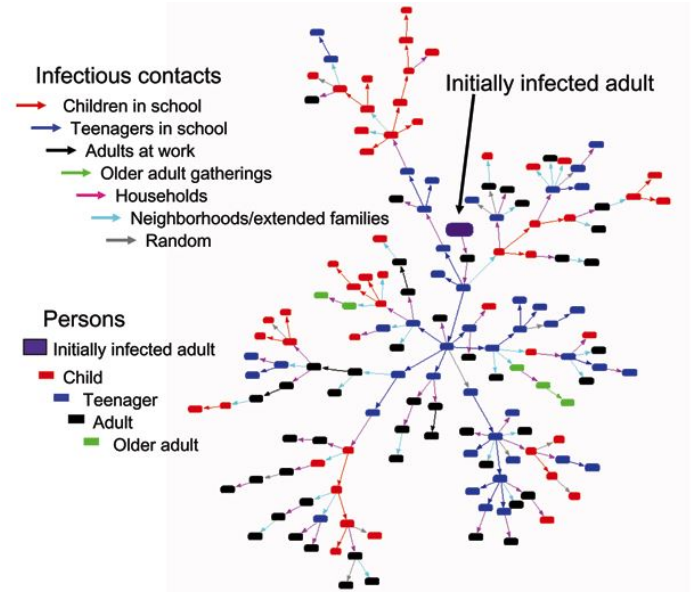


1.2.2 Influence Propagation

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.

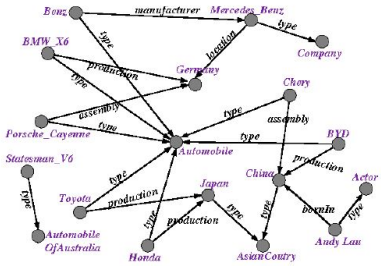


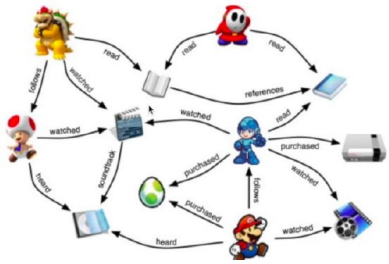
This failure propagation model is also applicable for spread for mis-information and spread of virus. With a virus spread model, you are modeling connectivity (probability of disease spread) between people, community, city and countries. Identifying and cutting off center of spread can localize the effect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

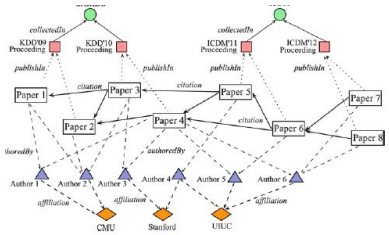


1.2.3 Knowledge Graph

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.



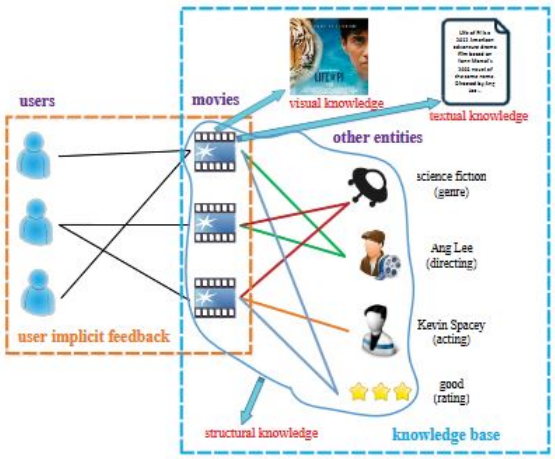




1.2.4 Recommender System

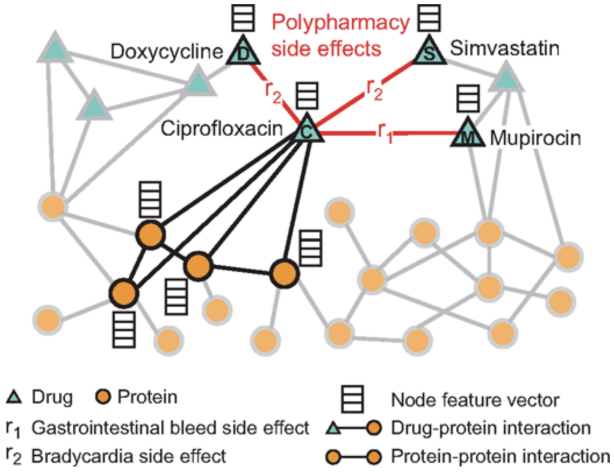
Predicting user preference can be abstracted to predicting existence of edges in a bipartite graph. CS246 covered concepts like SVD (singular value decomposition), but you can also solve this by treating user preference matrix as an adjacency matrix. In class, we showed that Pinterest has its own image-embedding 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 Network Properties

2.1.1 Degree Distribution

For undirected graphs with N nodes, let N_k be the number of nodes with degree k. Plot normalized ($P(k) = \frac{N_k}{N}$) histogram of degree. It is common to plot with log-log scale.
For directed graphs, plot in-degree and out-degree distributions separately.

2.1.2 Path Length

Path:

- sequence of nodes in which each node is linked to the next one
- can intersect itself and pass through the same edge multiple times
- follow the direction of arrows in directed graphs

Distance: number of edges along the shortest path between two nodes
Diameter: the maximum distance between any pair of nodes in a graph
Average Path Length: for a connected undirected graph (strongly connected directed graph):

- Definition: $\bar{h} = \frac{1}{2E_{max}} \sum_{i,j \neq i} h_{ij}$
- If the graph is not connected, we compute average only over the connected pairs (ignoring infinite length paths)
- We can also apply the measure to a (strongly) connected component of a graph

2.1.3 Clustering Coefficient (for undirected graph)

Clustering Coefficient for a node i:

- measures how connected are i's neighbors to each other.
- i has degree k_i
- e_i: number of edges are between the neighbors of node i
- Possible edges among k_i neighbors of i's: $\binom{k_i}{2}$
- Define clustering coefficient of i: $C_i = \frac{e_i}{\binom{k_i}{2}} = \frac{2e_i}{k_i(k_i-1)}$.
- $C_i \in [0, 1]$

Average Clustering Coefficient: Take the average over all of the N nodes of the graph $C = \frac{1}{N} \sum_{i=1}^{N} C_i$

2.1.4 Connected Component

Size of the largest connected component: largest set where any two nodes can be connected via a path.
To compute the size of largest connected component, we need to find connected components following the below steps via BFS:

- Start BFS from a random node
- Label the nodes BFS visited
- If all nodes are visited, the network is connected
- Otherwise, find an unvisited node and BFS from the node

2.2 Random Graph Generation Models

2.2.1 Erdos-Renyi Model

There are two variants of Erdos-Renyi Model: For an undirected graph of n nodes

- *variant 1* G_{np}: each edge (u, v) appears *i.i.d.* with probability p (Similar to a Binomial distribution $Binomial(n-1, p)$) ($n-1$ since for every node, it is possible to have at most $n-1$ edges with other nodes).
- *variant 2* G_{nm}: m edges picked uniformly at random

The network properties (section 2.1) for a variant 1 G_{np} Erdos-Renyi Model is as following:

- Degree distribution: by properties of $Binomial(n-1, p)$
  - $P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$
  - By Law of Large Number, the distribution of average $P(k)$ becomes increasingly narrow as n increases. We become increasingly confident that the degree of a node is in the vicinity of k as n grows.
- Clustering Coefficient:
  - expected number of edges connecting to node i: $E(e_i) = \mathcal{P}$(a pair of neighbors of node i is connected $\times$ number of distinct pairs of neighbors of node $i = p \times \frac{k_i(k_i-1)}{2}$
  - expected clustering coefficient of node i $E(C_i) = \frac{p \times k_i(k_i-1)}{k_i(k_i-1)} = p = \frac{\bar{k}}{n-1}$.
- Path length: O(log(n))
  - First we define **Expansion**: TODO

  - Expansion on random graphs TODO
  - Average shortest path: If we keep np constant, even when G_{np} grows very large, the nodes will still be less than 20 hops (if we consider shortest path) apart on average.

- Connected Components:

2.2.2 Small World Model

2.2.3 Kronecker Model

2.3 Subgraphs: Motifs and Graphlets

2.3.1 Definition

2.3.2 How to find motifs and graphlets

2.4 Structural Roles

2.4.1 How to find structural roles: RoIX

2.4.2 Application of structural roles

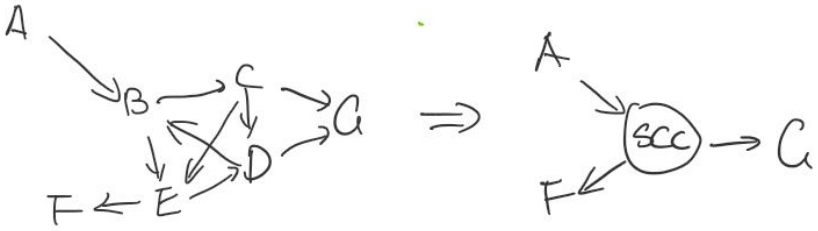
2.5 Spectral Clustering

3 Application of Graphs

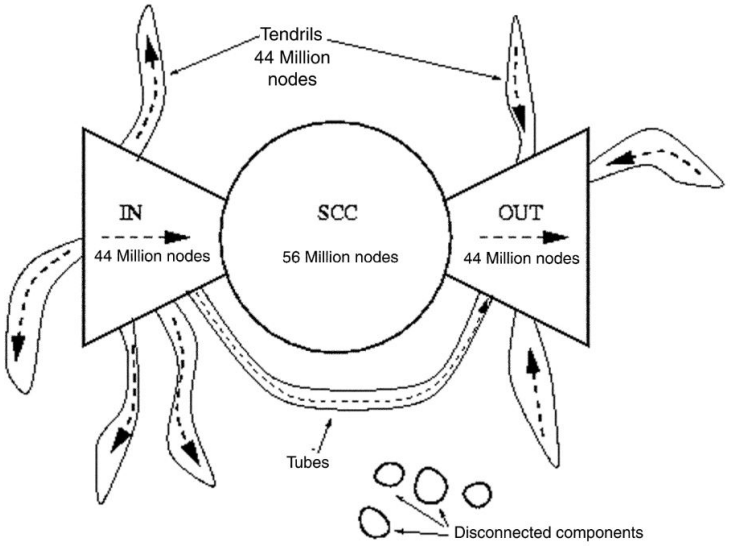
3.1 Structure of the Web

The web can be treated as a directed graph where each link (url) is an edge pointing to another web page. Any directed graph can be broken down into 2 components `(1)` *Strongly connected component* where any node can reach any other node via a directed path `(2)` *Directed acyclic graph, DAG* where if node u can reach v, then v cannot reach u.

Treating a strongly connected component (SCC) as one node, any graph can be "distilled" into a directed acyclic graph (DAG), as shown below. Nodes B, C, D, E form an SCC. Compressing these nodes into one SCC node, we turned a "'mixed" graph into a DAG. Notice that there cannot be intercepting SCCs because any such intersection will force the 2 SCCs to become a larger one due to definition of strongly connected component.

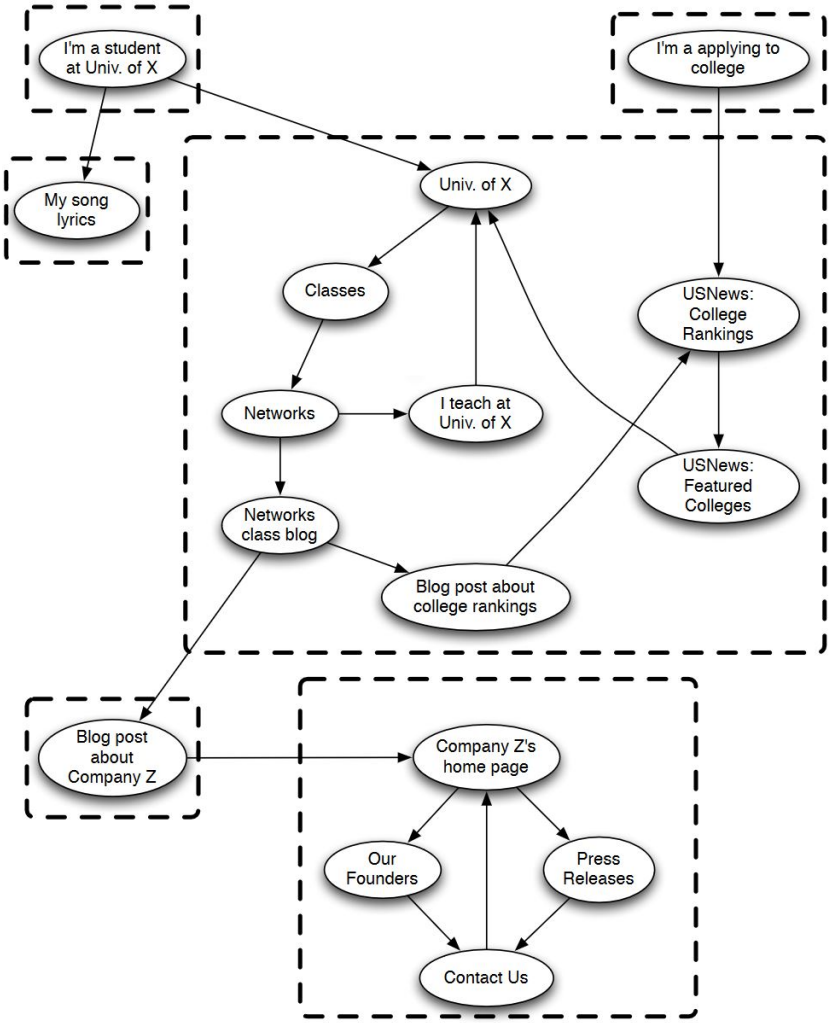


As shown below, the famous bow-tie analogy for the structure of the web was proposed in 1999 [link]. The graph was obtained after crawling through 203 million URLs (web pages) and 1.466 billion links. `IN` is a set of nodes that can reach *SCC* whereas `OUT` are nodes that can be reached by nodes in *SCC*. Other components are defined following similar convention.



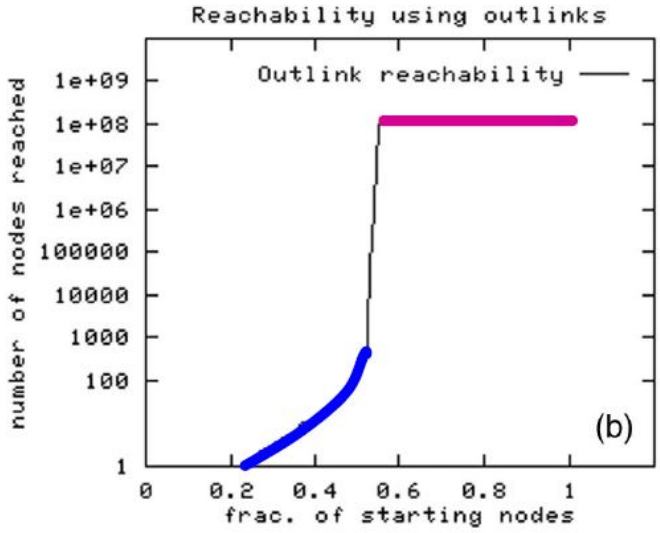
At a glance, you might be wondering how does one URL connect to a whole bunch of all other URLs? Well, the next image extracted from *Networks, Crowds, and Markets: Reasoning about a*

Highly Connected World [link] illustrates how such connection can happen for college admission. BTW, the craw was supported by `AltaVista` [link], world's first modern search engine and has since been absorbed into Yahoo! search (which then is no longer popular, `Yahoo.com` is only 10-th website on Alexa ranking).



To draw such bow-tie, we need to be able to classify each URL into `IN`, `OUT`, `SCC`, `Tendrils`, `Tubes` and `Disconnected components`. Using traditional search/traversal algorithms such as BFS and DFS, we can easily identify nodes reachable by node v and nodes that reach v. Essentially, nodes reachable from v from a tree rooted at v, let's call this $Out(v)$. Similarly, we can construct a "reversed" tree for nodes that lead to v, let's call this $In(v)$. Clearly, an *SCC* containing node v is simply $Out(v) \cap In(v)$. With *SCC*, `IN` consists of nodes "reverse-reachable" by `SCC`. Tendrils originating from `IN` are nodes reachable by `IN` but not in `SCC` and not in `OUT`. Depending on the structure of this network, we might observe several and potentially sizable `Disconnected components`.

Following only out-links, we can plot the following diagram. Most of the blue portion here represents `IN` while the pink portion represents `SCC` and `OUT`. Overlaying this with nodes reached through in-links ("reversed tree"), we can again see the proportion of nodes in `IN`, `SCC` and `OUT`.



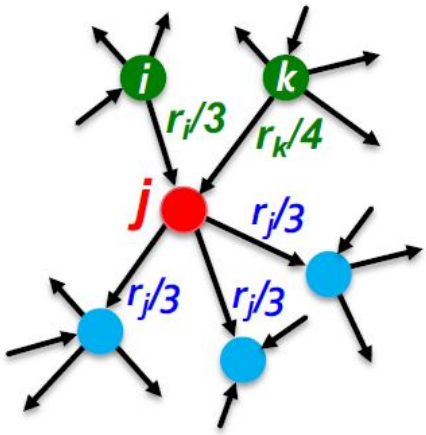
Web is a bow-tie analogy is basesd on result obtained pre-2000. The `WWW` has evolved substantially since then (think how, as long as you point to Google, you point to the world). A report in the 2019 offering of *CS 341* [link 1][link 2] shows that the current web looks more like a tie, where `IN` and `SCC` components have not changed substantially but a huge `OUT` portion has developed and is continuing to grow according to web crawls conducted in 2003, 2004 2007 and 2010. It might surprise you that `adobe.com` is a highly popular destination domain due to sites that tell you to download *Adobe Reader*. `youtube.com` and `facebook.com` have moved up the rank to be come the most popular destination sites. We strongly suggest you to to read the report.

3.2 PageRank

Famously Google's success is based on its use of PageRank. As the name suggests, PageRank is a method of ranking importance of web pages. The basic idea behind PageRank is that pages with lots of in-links (those pointing to the page) is more important than one with few. For example stanford.edu has $23,400$ in-links while joe-schmoe.com has 1 in-link (WARNING! This site is not for the faint-hearted). Moreover, not all links carry equal importance. A site like `joe-schmoe.com` can setup lots of self-refernces or have a swarm of other sites point to it (link farm). Modern search engine optimization (SEO) techniques, including simple PageRank operations that we will discuss later, can filter out many forms of link farm.

PageRank follows a "flow" model. That is, rank of nodes flow to children nodes. Therefore PageRank is an iterative process. As shown below, let r_v represent the *importance* of a node. Then node i has 3 out-links and node j has 4 out-links, therefore node j carries $r_j = \frac{r_i}{3} + \frac{r_k}{4}$ important.

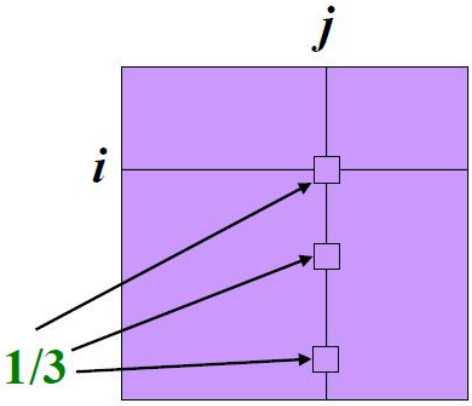
It subsequently distributes $\frac{r_j}{3}$ to its own out-links.



More mathematically, for node j and its parents P_j, and d_v as number of out-links of node v, we write

$$r_j = \sum_{i \in P_j} \frac{r_i}{d_i} \tag{1}$$

In matrix form, where M is stochastic adjacency matrix. (Typical adjacency matrix that represent connectivity has $rowcol$, meaning that number of out-links are sum of entries of a row. The setup for PageRank is to avoid having to write M^T all the time). Stochastic adjacency matrix stores $\frac{1}{d_j}$ for corresponding locations along its columns. That is, if node j has 3 out-links, the stochastic adjacency matrix should look like the following.



With rank vector $r \in \mathcal{R}^{|N|}$ that represent rank for all nodes. For people familiar with Markov Chain, you should immediately recognize M as state transition probability matrix where as r is the state vector. See this note for basic note to Markov process [link]. See CS168 notes [link] for

a slightly deeper explanation Markov Chain Monte Carlo and stationary distribution. See CS265 notes [link] [link] for more advanced topics.

$$r^{new} = M \cdot r^{old} \tag{2}$$

$$r^t = M^t \cdot r^{old} \qquad \text{superscript } t \text{ for matrix power} \tag{3}$$

$$\forall r^t, \sum_i r_i^t = 1 \tag{4}$$

Suppose that we have reached a stationary distribution, and r_{final} represents such stationary distribution, that is $M^t u \to r_{final}$, where $t \to \infty$ and u is any starting vector that represents a valid distribution ($\sum_i u_i = 1$). What we are getting at is, eventually we will have $r_{final} = M \cdot r_{final}$, therefore r_{final} is an eigenvector of M with corresponding eigenvalue of 1.

Given the above, it is not hard to see the rationale behind the **power iteration method**, where

- Initialize $r_{init} = [1/N, 1/N, \dots 1/N]^T$
- Iteratively compute $r_{(t+1)} = M \cdot r^t$
- Stop when $|r^{(t+1)} - r^t| < \epsilon$ for some pre-determined convergence threshold ϵ

On stationary/limiting distribution and some formalities While convergence is guaranteed for well-formed transition matrices (you should definitely read the CS265 note [link] for mixing time of Markov chains, also of course, you can treat this transition matrix as a random walk, which is the underlying Markov process. This Caltech note [link] seems to be thorough as well with minor change in nomenclature), structures such as *dead ends* and *spider traps* can cause issues. Eventually these structures can siphon significant amount of importance from the overall network.

- Dead ends. These are nodes with no out-links, so "importance" accumulates here. In other words, importance given to the "useful" part of the network decreases.
- Spider traps. These are self-referencing groups where out-links are only pointing to nodes within the group.

Let's formalize convergence of the *power iteration method*. As said, we view transition matrix M as transition matrix of a random walk performed on a *Markov chain*. A Markov chain will reach a limiting distribution, that is, S as all states (each state is a one-hot vector that can be represented by a single number, here it can simply be node i), n as number of iterations, X_i as the "walker" being observed in state i and π_i as probability of the "walker" being at node i,

$$\pi_j = \lim_{n \to \infty} P(X_n = j) \qquad \forall j \in S \tag{5}$$

What the above means is that after a higher number of iterations, probability of landing at any node is the same as value in the limiting distribution π_j. Notice that $n \to \infty$ is necessary because the above is clearly not true for small n. With *dead end*, limiting distribution will be one where the dead ends share some distribution while the remaining "normal" nodes sharing no probability at all. With *spider traps* the limiting distribution will not be unique.

We then define stationary distribution, where the stationary distribution is solved rather than computed iteratively like the limiting distribution. For a Markov chain to have a unique stationary distribution, it must be

- Aperiodic. For all paths that the random walker can take to return to its starting position, lowest common divisor of length of these paths must be 1. Typically, as soon as you see a sel-loop, the chain is aperiodic.
- Irreducible. All nodes can reach any other nodes. Simply, no disconnected components, no "one way" path to any component.

If the above are true then for transition matrix M, we are guaranteed to find $\pi = \pi M$ (or the transposed form, depending on definition of M) and $\sum_{j \in S} \pi_j = 1$. We claim that such stationary distribution IS the limiting distribution. Note that we do NOT know the number of steps needed to approach such distribution, we still have to bound the error by ϵ.

Google's solution to spider-trap is through teleportation. Some background first, the paper was written by Sergei Brin and Larry Page in 1998 [link]. The proposed PageRank solution was tested on a 322 million link database with 75 million unique URLs (Stanford Web Project [link]) and converged approximately 52 iterations. By the way, 1998 was when Pentium II was running at less than *half* of a gigahertz and your personal computer was running with $256MB$ of memory [ebay link]. These computers are worse than the original Apple Watch [link]. Also, back then, "Download Netscape Software" page had the highest importance (in case you don't know what Netscape is, see [link]).

Paraphrasing the paper, PageRank models a random surfer who randomly clicks through web pages. However, an actual web surfer can get bored, so the surfer might arbitrarily click to an unrelated page to restart this random walk, hence the "teleportation". Suppose that at any time, the surfer has $1 - \beta$ probability of teleporting away. We can re-write the PageRank equation as

$$r_j = \sum_{i \in P_j} \beta \cdot \frac{r_i}{d_i} + (1-\beta)\frac{1}{N} \tag{6}$$

Then in matrix form

$$r = \beta \cdot M \cdot r + \frac{1-\beta}{N} I \tag{7}$$

$$r = Ar \qquad \text{where } A = \beta M + \frac{1-\beta}{N} I \tag{8}$$

You might be wondering that there is a missing r in the PageRank form. This is from the fact that $\sum_i r_i = 1$, so eventually the whole thing becomes adding a simple constant.

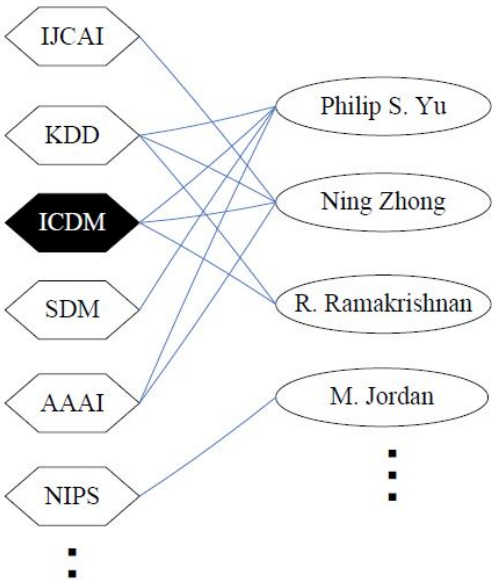
Dead end "leaks" PageRank because the rank goes to nowhere from the end node, causing sum of rows of M or A to be less than 1. Therefore, $r^{new} < A_{dead\ end} r^{old}$. We can fill in the bank by either filling it up or by adding a shared constant. Due to potential size of the matrix, adding a shared constant is the more common approach, thus

$$r^{new} = Ar^{old} \tag{9}$$

$$r^{new'} = r^{new} + \frac{1 - \sum_i r_i^{new}}{N} \qquad \text{add a simple constant} \tag{10}$$

3.3 Topic-Specific PageRank

A potential problem that can be solved using graphs is, given a bipartite graph representing a list of conferences and a list of authors, we want to know conferences that are the most similar to ICDM. Similar questions can be raised on, given a list of images and a list of boards (Pinterest), what are images that are the most similar to a select image. In the example below we want to use "topic-specific" PageRank to find conferences that are the most similar to ICDM.



Topic-specific PageRank is achieved by changing the teleportation part of PageRank. Instead of teleporting randomly to all nodes, only relevant nodes are allowed, called the teleport set. For this case, the teleport set is $\{ICDM\}$. Suppose we want to know things that are common to ICDM and KDD, the teleport set would then be $\{ICDM, KDD\}$. Recall the random walk equivalent for PageRank's transition matrix, with a pre-defined teleport set, topic-specific PageRank is simply a generalized form of random walk with restarts. Note that the proper "random walk with restart" has teleport set size of exactly 1 whereas a "topic" can be represented by a larger teleport set.

The generalized topic-specific PageRank follows the same setup as PageRank but uses α as the probability for restart.

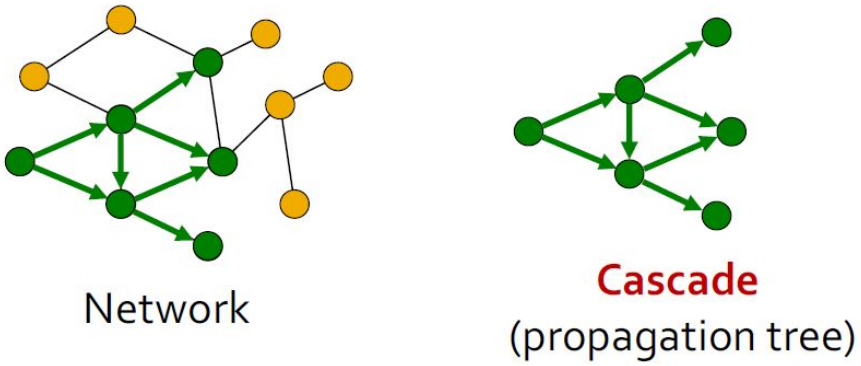
- **Vanilla PageRank** teleport uniformly at random to any node. All nodes have the same probability of being the target for teleportation.
- **Random Walk with Restarts** teleport to *exactly* 1 node. That 1 node has probability of 1 (teleportation vector being one-hot).
- **Topic-Specific PageRank** teleport to multiple nodes. However, these nodes can have different probabilities depending on definition of "topic".

Before we end here, it is worth noting that finding similar items through random walk and weighted random walk is one of the foundational tools for machine learning on graphs. See Section 4 for more details.

3.4 Cascades

Cascade is a fancy way of saying propagation. Originating from one node, we can build a directed propagation tree from a directed or an undirected network. Here, we use *contagion* for the thing

that is spreading (virus, fake news); *infectious event* for the act of spread (adoption, infection, activation); *main player* as the infected/active nodes. There are 2 ways to model network cascade `(1)` deterministic, based directly on properties of nodes/edge `(2)` probabilistic, activation/infection is probabilistic.



3.4.1 Decision based models

Decision based models are simple to implement and of course, deterministic in nature.

Software adoption A simple examples is, in a friends network, people are adopting different video conferencing software. Suppose each person can only install either Skype (S) or Zoom (Z). We can define the payoff matrix as follows, where $a, b > 0$ for 2 friends adopting the same software.

		Friend 2	
		Z	S
Friend 1	Z	a	0
	S	0	b

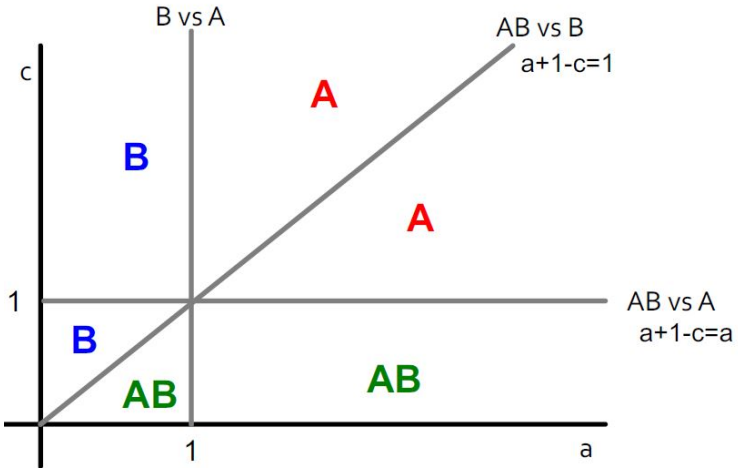
Then for an undecided friend, we can compute the payoff of adopting $S \to r_S$ also $Z \to r_Z$. The friend adopts S according to a threshold t, that is $\frac{r_S}{r_S + r_Z} > t$. Setting $t = 0.5$ simply means $r_S > r_Z$.

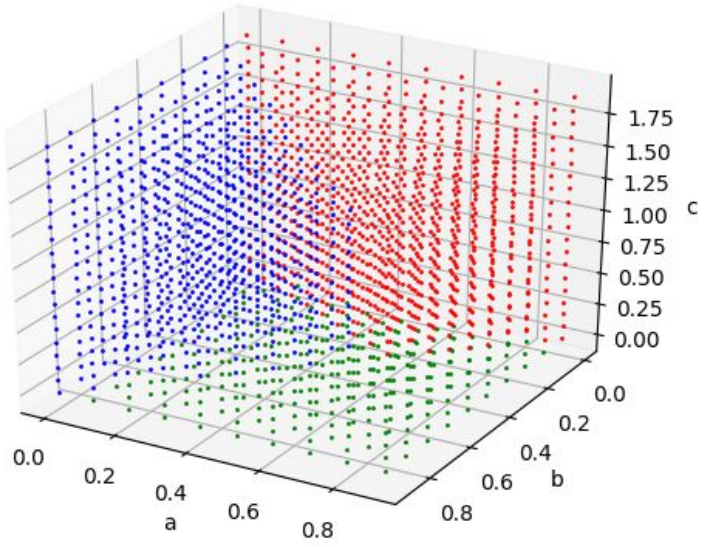
For new software like Zoom, it is fair to assume that the entire network is initially populated with Skype users. Then marketing/financial incentive lets a number of early adopters flip to the Zoom camp, from which we can start iteratively decide whether Zoom will take over the network. Note that there can be an equilibrium state separated by some adoption boundaries.

Similarly, suppose we allow adoption of both software. In the case where benefit of using both is larger than a, b, it is easy to see that more of both will be adopted. In contrast, if sharing has limited but non-zero benefit, only those along the adoption boundaries will be adopting both. Using example presented in class, we have the following payoff matrix, where a, c are positive real number. The undecided friend is W in $A - W - B$.

		Friend 2	
		A	B
Friend 1	A	a	a+1-c
	B	a+1-c	1

Then adoption by W follows the following graph, essentially only adopting both when the harm $c < a + 1$ when a is low and $c < 1$ then a is high. This might be a little confusing because we have scaled b to 1. We'd have to plot a 3D graph by letting b be arbitrary positive value. See 3D screenshot below (same color scheme), you can picture a 3-way division similar to the 2D image above it.



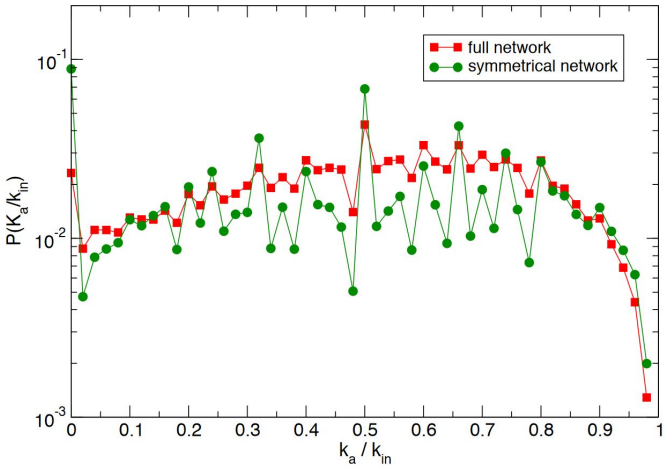


Protest and recruitment on social network Example shown in class is about the anti-austerity 15-M protest in Spain (May 15-22, 2011 [link]). This is part of a series of "Occupy" movements

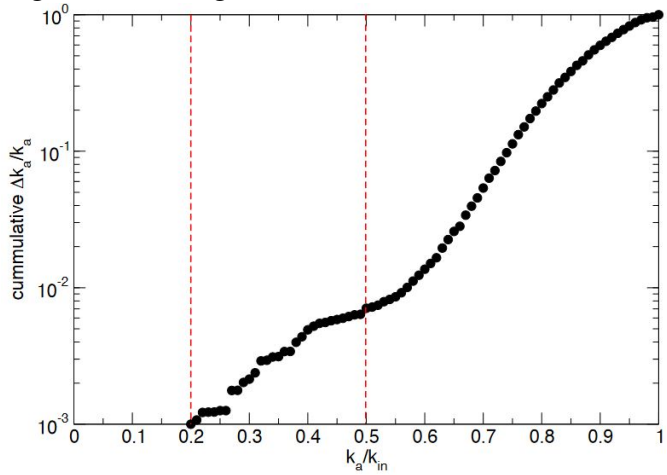
that include the famous 2011 Occupy Wall Street movement [link].

In a research after the protest [link] researchers identified 70 hashtags used by the protesters by crawling Twitter over a 1 month period, yielding $581,750$ tweets from $87,569$ accounts. From here, researchers constructed `(1)` Full network with all follow as links `(2)` Symmetric network where only $a \leftrightarrow b$ following count as edges. Activation threshold is defined as $\frac{k_a}{k_{in}}$ where k_a (number of active neighbours) and k_in (total number of neighbours). This activation threshold is clearly dynamic and can be framed as "social pressure".

In the image below, there are 2 peaks. `(1)` At low $\frac{k_a}{k_{in}}$, representing a large number of "self-activating" users `(2)` At $\frac{k_a}{k_{in}} = 0.5$, a large portion of users joined the movement when more than half of the neighbours are active.

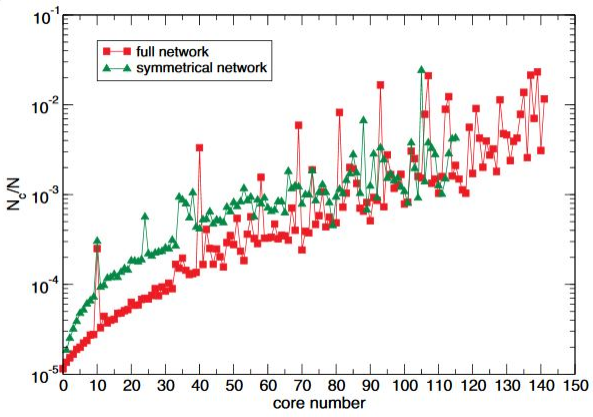


The second image depends on $\Delta\frac{k_a}{k_{in}}$, that is, the speed at which their neighbours become active. Low threshold users become active regardless of the speed of change, but large threshold users only became active after a large burst of neighbourhood activation (2nd red dotted line).

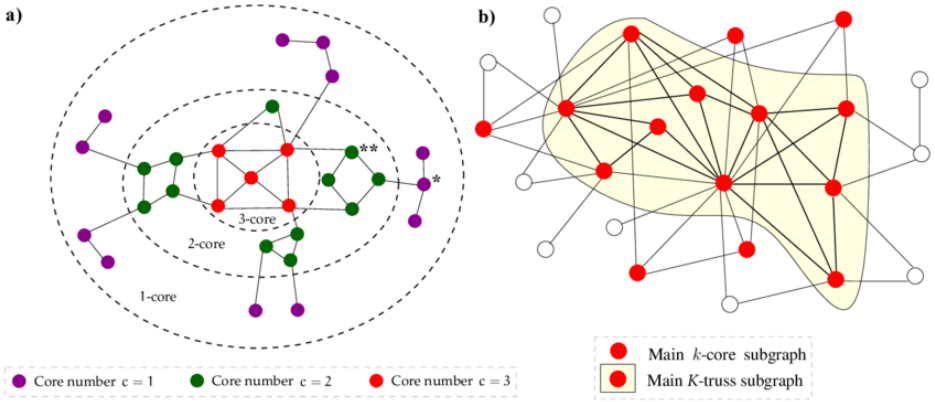


The third image here compares size of a spreading "core" against cascade size. What this is showing is that a larger initial core user will cause a more complete spread. Therefore to start a

more influential campaign a large number of connected users need to be mobilized (for example, having game reviewing websites/popular streamers preview video game before launch then have them all tweet about this new game at the beginning).



The following is extracted from [link], where $k-core$ represents a connected subgraph where each node has at least degree k. Clearly, a cluster with high k number is more central to the graph. We can obtain $k-core$ decomposition by repeatedly remove nodes with lower number of edges. Each layer removed represent one group of nodes with the same number of edges.



To summarize, if you want to start a campain on Twitter, you should be aware that

- Activation threshold for users is uniform, with exception of 2 peaks at around 0 (self-motivated) and around 0.5 (more than half of neighbours)
- Most cascades are short (because you get out of the core area quickly on Twitter, remember the 3.7 hops for average connection on Facebook)
- Successful cascades are started by central nodes (so you should consider arranging for a concerted tweet storm started by influencers/celebrities/review websites)

3.4.2 Probabilistic models

Clearly decision based models are not suitable for many real world applications. Simple rules are not suitable for complex scenarios such as spread of disease (depends on wearing protective equipment, method of contact, etc.) adoption of view (depends on your background, life event, etc.) and even installation of software (depends on cost, learning curve, capability of computer, etc.). Often it is much easier to model these complex properties with probability with frequent calibrations.

The simplest representation of a probabilistic spreading model is a random tree. Starting at an active root, we want to know the number of people eventually activated, given that each edge has activation probability q and each node is connected to d leaves. With this, at each node, the probability of passing on the infection is

$$p_{pass} = 1 - (1-q)^d \qquad p_{not\ activated} = 1 - q, \text{ then } d \text{ leaves} \tag{11}$$

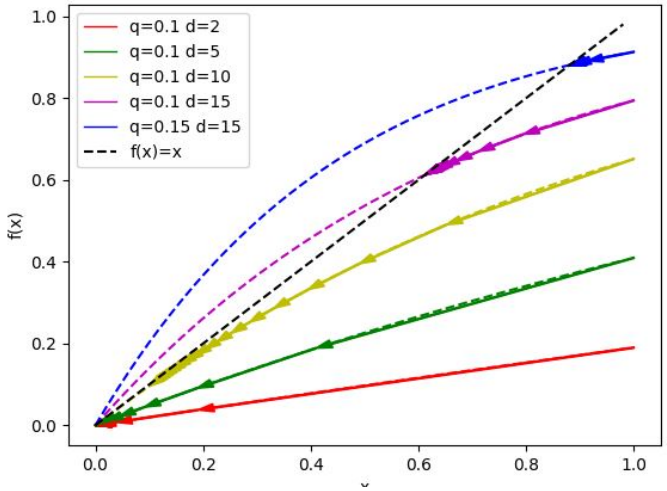
We want to build a recurrent relationship into this equation, so instead of viewing the root of the $1 \to d$ nodes branch as having an active node, we let it having an $p^{(h)}$ chance of being active. Therefore, we can express

$$p_{pass}^{(h+1)} = 1 - (1 - q \cdot p_{pass}^{(h)})^d \tag{12}$$

This interpretation does not care about the total number of nodes at each level, but instead view the parent as being a recurrent probabilistic parent, thus solving the problem of having to account for increasing number of nodes where each d of such nodes are correlated to the parent (properly handling a growing Bayes's network is not fun). The the recurrent relationship is simply as shown.

$$f(x) = 1 - (1 - q \cdot x)^d \tag{13}$$
$$f(x=0) = 0 \tag{14}$$
$$f(x=1) = 1 - (1-q)^d < 1 \tag{15}$$
$$\tag{16}$$
$$f'^{(x)} = q \cdot d(1 - q \cdot x)^{(d-1)} \tag{17}$$
$$f'^{(x=0)} = q \cdot d \tag{18}$$

Since all of $f'(x)$ are above 0, $f(x)$ is monotonic and *increasing* on $[0, 1]$. With some observation, it is not hard to tell that $f'(x)$ is monotonic (if you take the $2nd$ derivative) and *decreasing* on $[0, 1]$. Let's plot a few $f(x)$ curve using a few (q, d) tuples. Since the function starts from $f(x = 1)$, observe direction of the arrows. All of these values approach the equilibrium value of $f(x) = x$. For the epidemic to die out, we must have $q \cdot d < 1$, then $lim_{h\to\infty} p^{(h)} = 0$. This number, as you have heard on the news is called *Basic reproduction number*, where $R_0 = q \cdot d$. For your reference, the 2020 COVID19 has estimated R_0 of $1.4 - 5.7$. Extracted from Wikipedia [link], here is a few more data points.



Disease	Transmission	R0
Measles	Airborne	12–18
Chickenpox (varicella)	Airborne	10–12
Polio	Fecal–oral route	5–7
Rubella	Airborne droplet	5–7
Mumps	Airborne droplet	10–12
Pertussis	Airborne droplet	5.5
Smallpox	Airborne droplet	3.5–6
COVID-19	Airborne droplet	1.4–5.7
HIV/AIDS	Body fluids	2–5
SARS	Airborne droplet	2–5
Common cold	Airborne droplet	2–3
Diphtheria	Saliva	1.7–4.3
Influenza(1918 pandemic strain)	Airborne droplet	1.4–2.8
Ebola(2014 Ebola outbreak)	Body fluids	1.5–2.5
Influenza(2009 pandemic strain)	Airborne droplet	1.4–1.6
Influenza(seasonal strains)	Airborne droplet	0.9–2.1
MERS	Airborne droplet	0.3–0.8

Clearly, as social distancing measures and protective equipment become widely adopted, q, d

changes, which is why disease control teams need to continuously sample the population to dynamically obtain accurate estimate of q, d and change their models. The goal for disease control agency is to be sure that R_0 is below 1 and remain at this level by enacting appropriate measures and encouraging disease limiting behaviour. The measures of course depend on disease transmission path

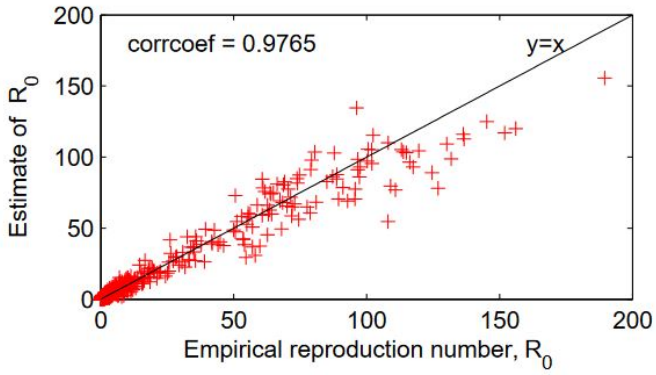
- Droplets/particles in the air (flu, influenza): wearing mask/protective goggle, avoid direct contact, social distancing, etc.
- Contact and smear (herpes simplex, smallpox): avoid direct contact, especially around wound and/or mucous membranes
- Blood and tissue, direct and indirect via blood-sucking pests (AIDS, hepatitis, black death): avoid direct contact, hygiene of living environment, eradication of pests
- Contaminated water and food (salmonellosis, cholera): proper water sanitation and thoroughly cooking food/water

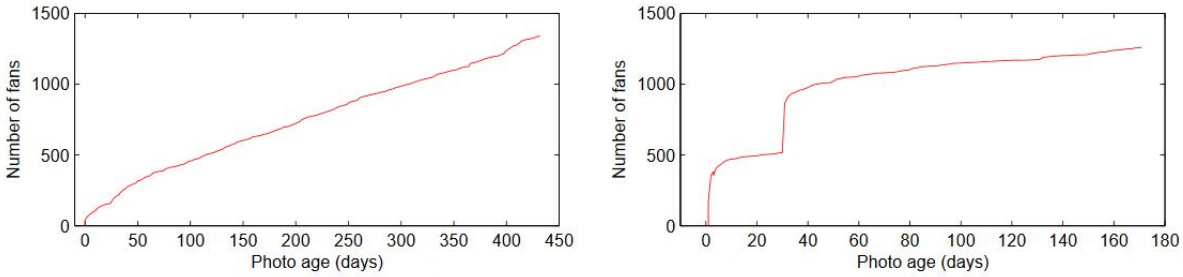
Flickr photo like propagation The dataset consists of 100 days of photo likes for 2 million users, accumulating to $34,734,221$ likes on $11,267,320$ photos.

The issue with estimating R_0 is that number of neighbours d is not uniform across all users, so we estimate R_0 as the following (d_i as degree of node i, d as average node degree, q as average proportion of neighbours infected). Then empirically, we can measure R_0 as proportion of directly infected nodes for the root noed of a cascade.

$$R_0 = q \cdot d \cdot \frac{avg(d_i)^2}{(avg\ d_i)^2} \tag{19}$$

As shown, the measured R_0 has high correlation with empirical R_0. The value is between 1 and 190, much higher than that of most contagious diseases. Moreover, in the next image, 2 growth methods are compared. Photo A is fueled by "organic" growth from social network itself, whereas photo B has external contributions at around day 40, potentially due to the image being shared on an external website.





(a) Growth of fans, photo A

(b) Growth of fans, photo B

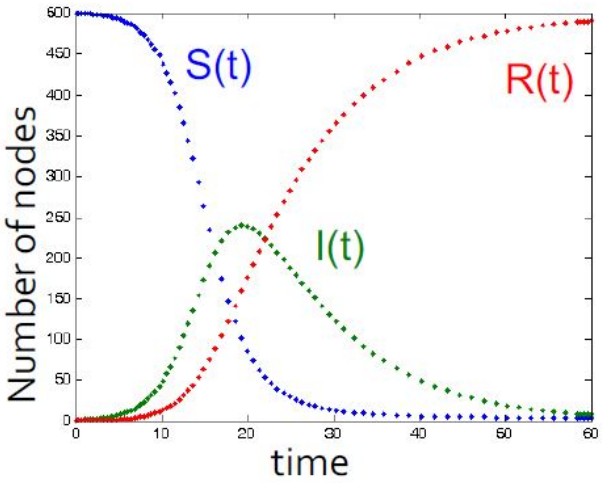
3.4.3 Math driven epidemic model

Diseases can also be modelled using a probabilistic model. However, the difficulty lies in correctly identifying social connections and modeling probability of spread. At the beginning of an epidemic, it is possible to trace those in contact by a pathogen carrier. As the epidemic develops, comprehensive tracking of contacts become near impossible, though modern tools such as GPS-capable smartphones and transparent yet privacy protecting data sharing software are utilized in the 2019-2020 COVID-19 outbreak. Epidemiologists resort to mathematical tools to model the spread, death and recovery of diseases.

Side note For all of the following mathematical expressions, S, I, R, E, Z represent proportions of population. The sum of these proportions should be 1. On the Internet, you will see some versions including N, total population, thus S, I, R, E, Z are number of individuals. Moreover, constants such as β, δ, γ can be a function of time as social distancing and advanced medical treatments start to take effect.

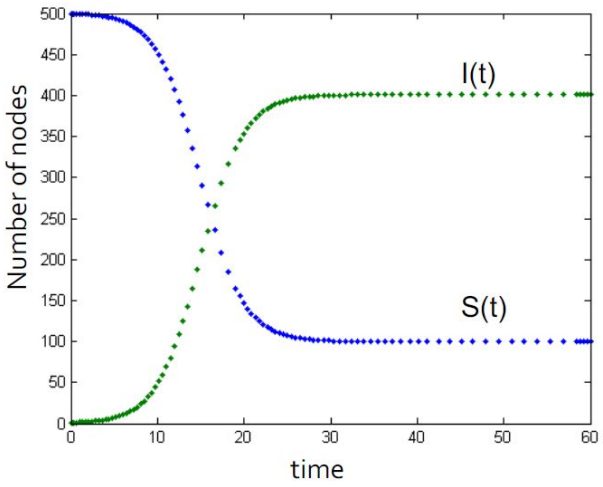
SIR (susceptible, infected, recovered) model This model is used for diseases which you become immune to after infection and death is rare, such as chickenpox. This model, as with all "naive" math-based models, assumes perfect mixing. From a network perspective, the network is fully connected. The model can be expressed as

$$Susceptible(S) \rightarrow_{\beta} Infected(I) \tag{20}$$
$$Infected(I) \rightarrow_{\delta} Recovered(R) \tag{21}$$
$$\frac{dS}{dt} = -\beta SI \tag{22}$$
$$\frac{dR}{dt} = \delta I \tag{23}$$
$$\frac{dI}{dt} = \beta SI - \delta I \tag{24}$$



SIS (susceptible, infected, susceptible) model This model is used for diseases which after cureved, the infected become susceptible again, such as fake news (you are convinced again) and e-coli.

$$Susceptible(S) \rightarrow_{\beta} Infected(I) \tag{25}$$
$$Infected(I) \rightarrow_{\delta} Susceptible(S) \tag{26}$$
$$\frac{dS}{dt} = -\beta SI + \delta I \tag{27}$$
$$\frac{dI}{dt} = \beta SI - \delta I \tag{28}$$

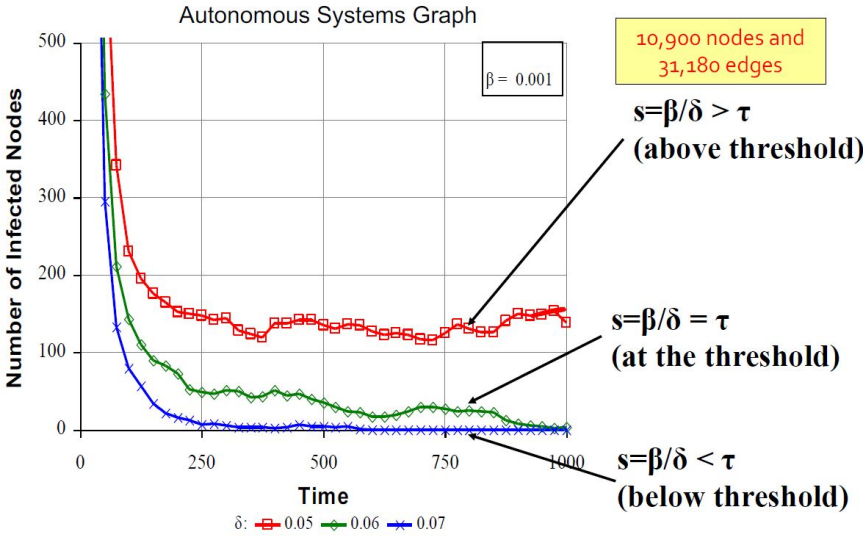


For this model we call $s = \frac{\beta}{\delta}$ as "stregnth" of the spread. There certianly exists a threshold where if $s = \frac{\beta}{\delta} < \tau$ the disease will die out. We can show that if

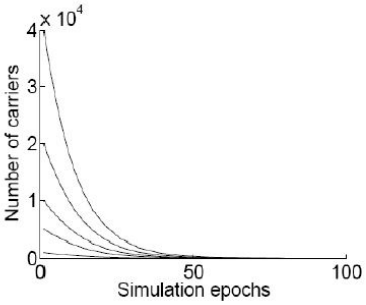
$$\frac{\beta}{\delta} < \tau \tag{29}$$

$$= \frac{1}{\lambda_{1,A}} \tag{30}$$

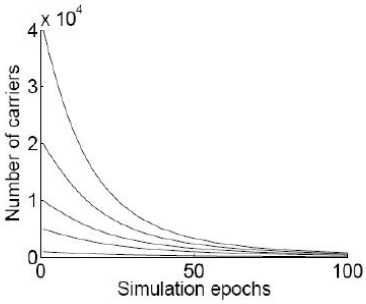
that is the threshold being the largest eigenvalue of the adjacency matrix, then the spread will die off. Recall that the largest eigenvalue $d_{avg} < \lambda_{1,A} < d_{max}$ (between average and largest node degrees). See more interpretations of eigenvalues of adjacency matrix [here] and [here]. In the graph below, "autonomous system" simply means a system modeled by ordinary differential equations.



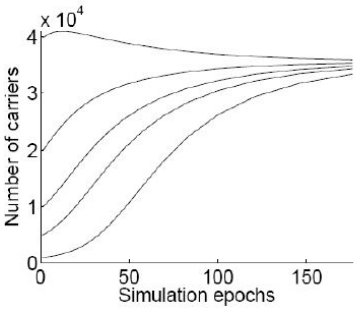
Through simulation, we observe that (1) Below threshold, number of carriers decrease rapidly (2) Near threshold, number of carriers converge to 0 (3) Above threshold, number of carrier stabilizes at above 0.



(a) Below the threshold, s=0.912



(b) At the threshold, s=1.003



(c) Above the threshold, s=1.1

SEIR (susceptible, exposed, infected, recovered) model This model adds another stage to the *SIR* model, where the *exposed* stage can be seen as an incubation period. Worth noting that this is a period where the individual ins *infected* but *NOT infectious*, which is different from the incubation period of COVID-19, during which the individual is infectious. The following formulation is for SEIR, and R_0 for this formulation is still $\frac{\beta}{\delta}$, but can be more complex if latent period and duration of infectious period are accounted for [link].

$$Susceptible(S) \rightarrow_{\beta} Exposed(E) \tag{31}$$

$$Exposed(E) \rightarrow_{\alpha} Infected(I) \tag{32}$$

$$Infected(I) \rightarrow_{\delta} Recovered(R) \tag{33}$$

$$\frac{dS}{dt} = -\beta SI \tag{34}$$

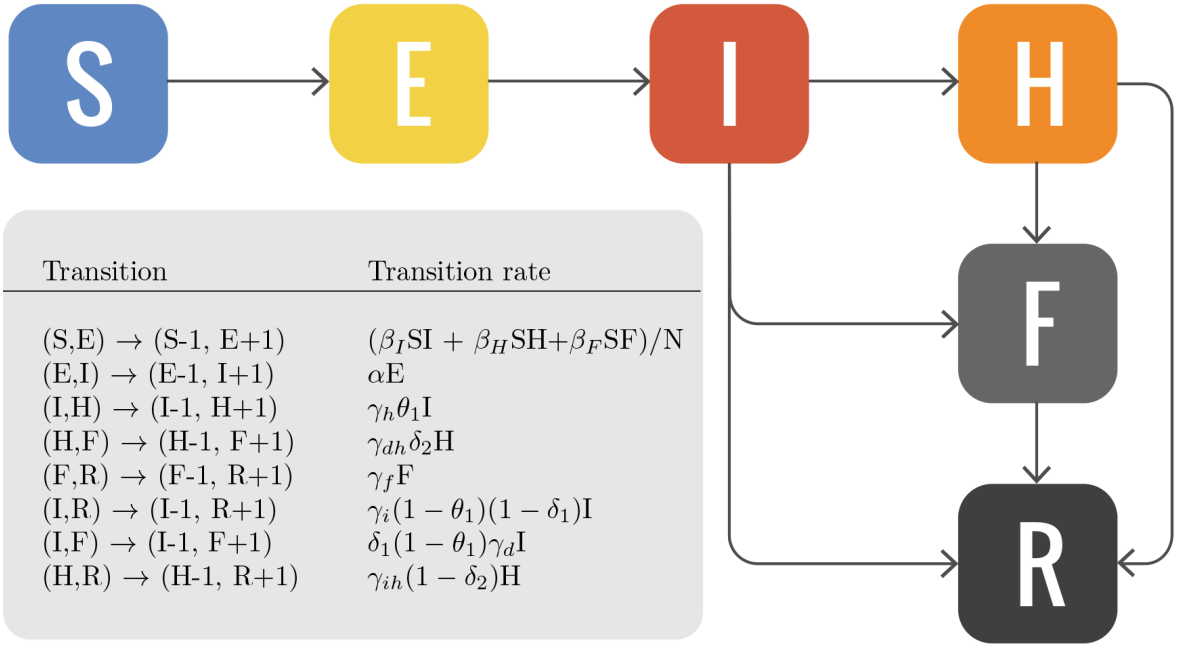
$$\frac{dE}{dt} = \beta SI - \alpha E \tag{35}$$

$$\frac{dI}{dt} = \alpha E - \delta I \tag{36}$$

$$\tag{37}$$

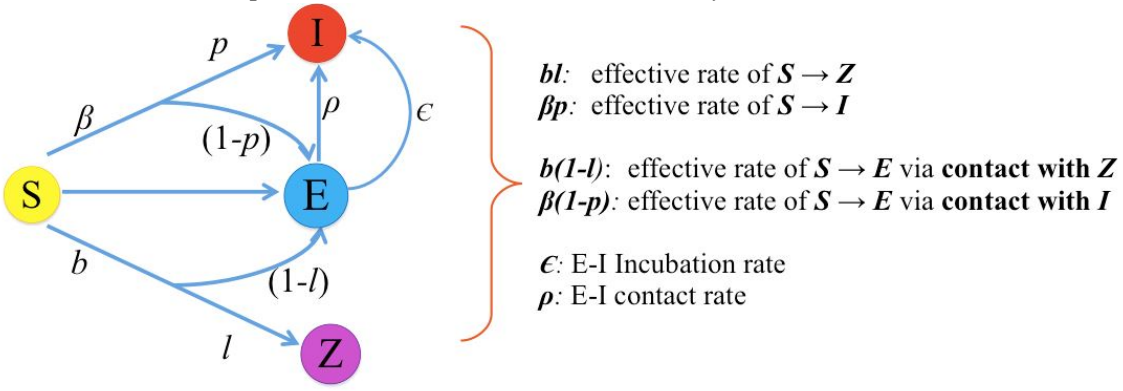
Models that describe diseases in terms of stages are called compartmentalized models. Stages can have complex interactions with each other and each interaction leads to an additional ordinary differential equation. For the 2014 Ebola outbreak **??**[link], a more sophisticated version of *SEIR model* was utilized, as shown below. Here, H represents hospitalized cases, F represents those

who are dead but not yet buried; various γ^{-1} represents duration of respective stage and α^{-1} being the incubation duration. This setup is to account for those not hospitalized and the fact that Ebola often transmitted through burial ceremonies and inadequate hospital quarantine measure.



Transition	Transition rate
(S,E) → (S-1, E+1)	$(\beta_I \mathrm{SI} + \beta_H \mathrm{SH} + \beta_F \mathrm{SF})/\mathrm{N}$
(E,I) → (E-1, I+1)	$\alpha \mathrm{E}$
(I,H) → (I-1, H+1)	$\gamma_h \theta_1 \mathrm{I}$
(H,F) → (H-1, F+1)	$\gamma_{dh} \delta_2 \mathrm{H}$
(F,R) → (F-1, R+1)	$\gamma_f \mathrm{F}$
(I,R) → (I-1, R+1)	$\gamma_i (1-\theta_1)(1-\delta_1) \mathrm{I}$
(I,F) → (I-1, F+1)	$\delta_1 (1-\theta_1) \gamma_d \mathrm{I}$
(H,R) → (H-1, R+1)	$\gamma_{ih} (1-\delta_2) \mathrm{H}$

SEIZ (susceptible, exposed, infected, skeptical) model This model [link] claims to better represent the life cycle of rumor by replacing *recovered* stage with *skeptical* stage, meaning that the person become immune to being infected by fake news. The model has 2 additional implicit transitory stages `(1)` upon receiving fake news and believing it, decides whether to re-post immediately or wait a few days `(2)` upon receiving fake news, decides to become skeptical or repost in a few days. *Personally, since E represents a stage of contemplation, we believe it is missing a $E \to Z$ transition that allows for a period of fact-checking instead of simply delaying the re-post. This additional link does complicate the ODE, which could be why the authors chose to leave it out.*

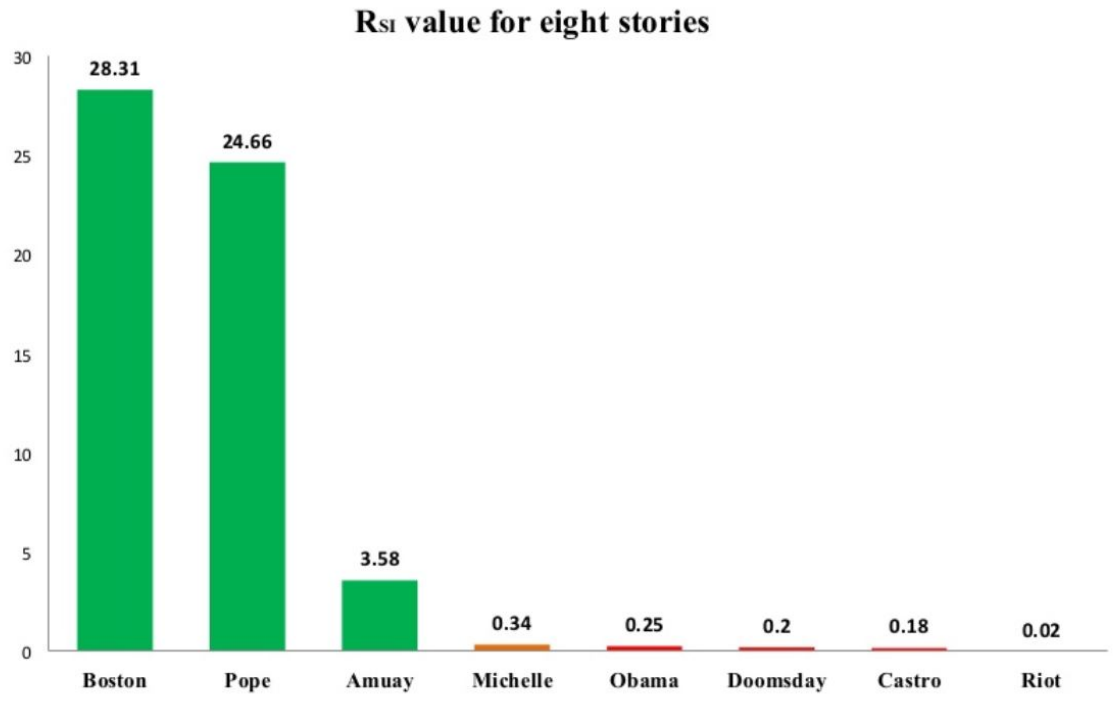


Mathematically, we have the following (leaving out all division by N to be consistent with out

previous expressions).

$$\frac{dS}{dt} = -\beta SI - bSZ \tag{38}$$
$$\frac{dE}{dt} = (1-p)\beta SI + (1-l)bSZ - \rho EI - \epsilon E \tag{39}$$
$$\frac{dI}{dt} = p\beta SI + \rho EI + \epsilon E \tag{40}$$
$$\frac{dZ}{dt} = lbSZ \tag{41}$$
$$\tag{42}$$

The observable part of this model is number of tweets sent by users, corresponding to an increase in number of infected people. Therefore, we want to optimize parameters to minimize $|I(t) - tweets(t)|$. The auhtors also defined a new parameter $R_{SI} = \frac{(1-p)\beta+(1-l)b}{\rho+\epsilon}$ to characterize spread of information on Twitter. The image below compares R_{SI} for various events. Here, high R_{SI} value represent "convincing" information because R_{SI} partially represents conversion rate, in contrast, rumors have much lower R_{SI} due to their dubious nature that get people question their validity.



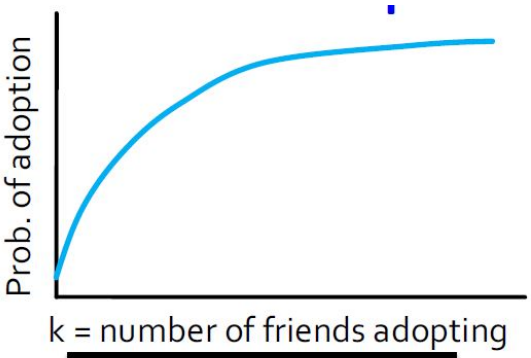
Other compartmental models We will not go in details for other models. However, compartments, or stages are used for more complex models.

- *Carrier (C)*. A stage in which a person carries the disease but suffer little to no symptom. In this stage, the person is infectious, unlike the *exposed* stage in *SEIR*. Models that include this stage is useful for diseases with length incubation AND infectious period.
- *Passive immunity (M)*. A stage in which a person has passive immunity. During this stage, the person carries insufficient amount (through breast milk or umbilical cord) or has yet to synthesize (immediately after vaccination) antibody. As you might have guesses, this stage is found in models for AIDS.
- *Temporary immunity (also R)*. A stage in which a person is temporarily immune to the disease, typical use case can be the seasonal flu.
- *Deceased (D)*. A stage in which a person has died from the disease. Complimentary to this stage is typically a recovered stage. Essentially the 2 outcomes for such disease is either death or permanent immunity. This is often used for particularly deadly diseases such as smallpox.

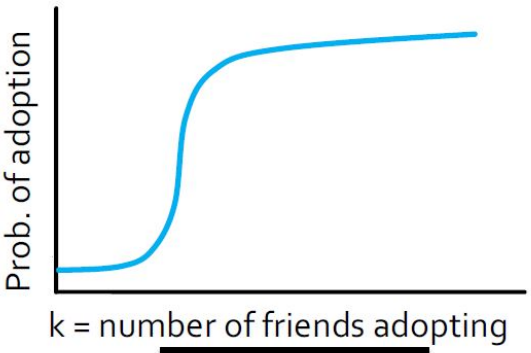
3.4.4 Independent cascade model

An independent cascade model is one that has full control over all edges of the network, where each edge represents an activation probability p_{uv}. As said before, a fully connected network in which all edges have the same activation probability is the same as the mathematical models we discussed just now. The difficulty is that modeling a network full, potentially $O(n^2)$ number of parameters is quite difficult and susceptible to overfitting to limited data (Hopefully outbreak for the same disease happens once in a life time. Also, our society changes rapidly, we might have a functioning world government next time).

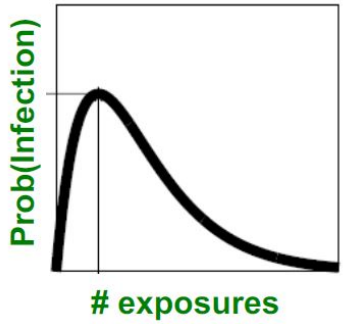
To alleviate this issue, we return to the decision based model (Section 3.4.1). We use the proportion of friends who are activated as basis for activation, then apply an activation function. This is called an exposure curve. Total number of adoptions can then form an adoption curve.

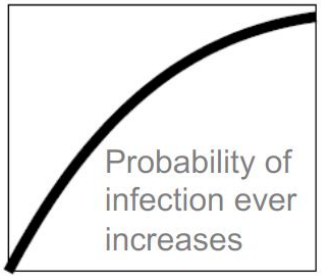


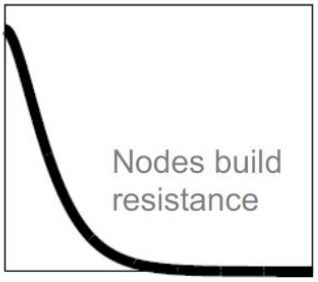
"Probabilistic" spreading: Viruses, Information



Critical mass: Decision making

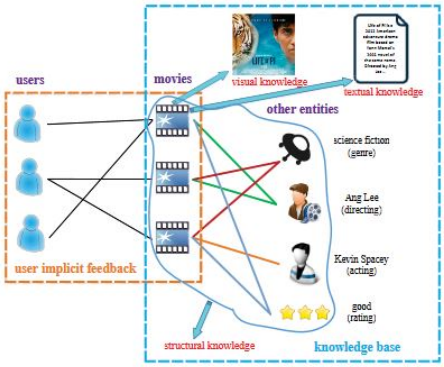






4 Graph Representations

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]).

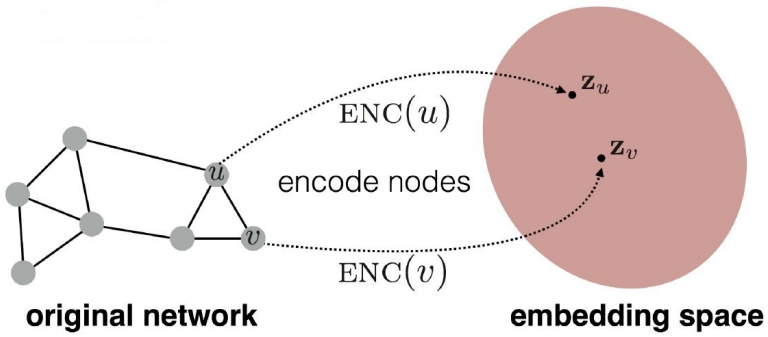


This approach differs from previous influence spread/belief propagation approach. Previous methods are statistical in nature, leveraging heavily knowledge related to Bayesian networks (those that you will learn in CS228). This chapter primarily focuses on, as the section title suggest, learning a graph 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.

$$ENC(V_i) = \mathbf{Z}v_i \tag{43}$$

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

$$\vec{a} \cdot \vec{b} = |a||b|cos(\theta) \tag{44}$$

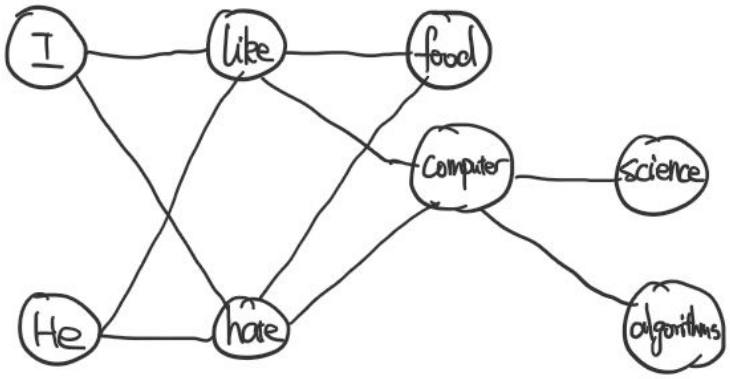
Node similarity in terms of cosine similarity can simply be expressed as the following. And of course, other similarity functions can also be employed, with appropriate change to optimization function that we will be discussing later (mostly in terms of taking derivative to minimize log-likelihood).

$$similarity(u, v) = \mathbf{Z}_u \cdot \mathbf{Z}_v \tag{45}$$
$$= \mathbf{Z}_u^T \mathbf{Z}_v \tag{46}$$

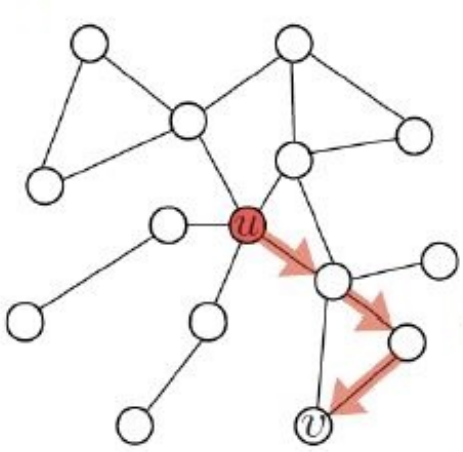
4.1.2 Random Walk

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

$$\mathcal{L} =_{u \in V} P(N_R(u)|z_u) \tag{47}$$

$$log\mathcal{L} = \sum_{u \in V} logP(N_R(u)|z_u) \tag{48}$$

$$= \sum_{u \in V} \sum_{v \in N_R(u)} logP(\mathbf{Z}_v|\mathbf{Z}_u) \tag{49}$$

Then for maximum negative log-likelihood

$$\rightarrow \max_{\mathbf{Z}} \sum_{u \in V} \sum_{v \in N_R(u)} -logP(\mathbf{Z}_v|\mathbf{Z}_u) \tag{50}$$

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

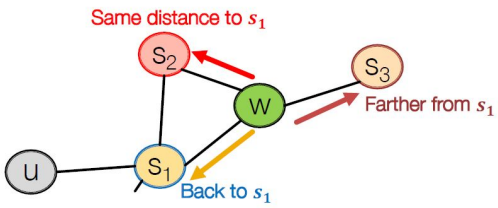
$$P(\mathbf{Z}_v|\mathbf{Z}_u) = \frac{exp(\mathbf{Z}_v^T\mathbf{Z}_u)}{\sum_{n \in V} exp(\mathbf{Z}_n^T\mathbf{Z}_u)} \tag{51}$$

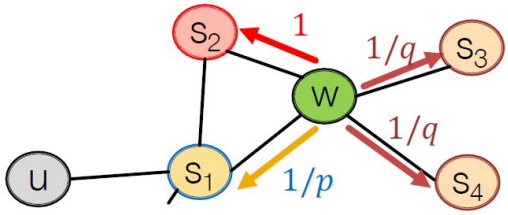
$$log\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} log\frac{exp(\mathbf{Z}_v^T\mathbf{Z}_u)}{\sum_{n \in V} exp(\mathbf{Z}_n^T\mathbf{Z}_u)} \tag{52}$$

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].

$$log\frac{exp(\mathbf{Z}_v^T\mathbf{Z}_u)}{\sum_{n\in V} exp(\mathbf{Z}_n^T\mathbf{Z}_u)} = log(exp(\mathbf{Z}_v^T\mathbf{Z}_u)) - \sum_{n\in neg\_sampling(V,walks)} exp(\mathbf{Z}_n^T\mathbf{Z}_u) \tag{53}$$

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.

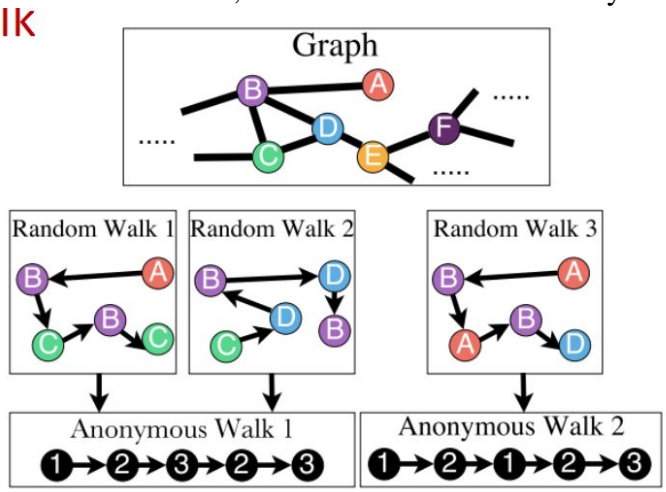
4.2.2 Virtual Node

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].

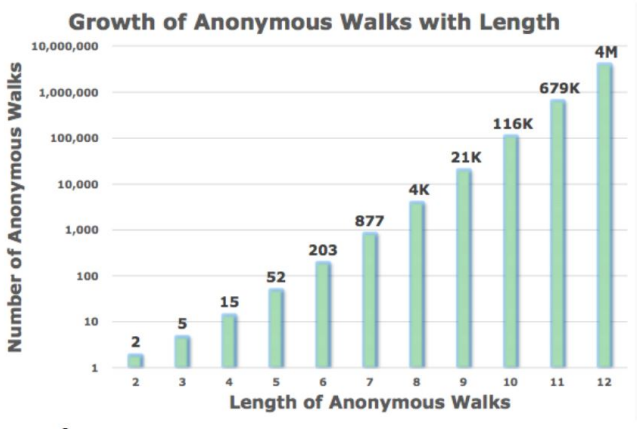
4.2.3 Anonymous Walk

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

$$m = \left[\frac{2}{\epsilon^2}(log(2^\eta - 2) - log(\delta))\right] where\ \eta \text{ is number of different anonymous walks for } k\text{-length walk} \tag{54}$$

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

$$max \frac{1}{T} \sum_{t=\delta}^{T} logP(w_t|w_{t-\delta}, \dots w_{t-1}) \tag{55}$$

$$P(w_t|w_{t-\delta}, \dots w_{t-1}) = \frac{exp(y(w_t))}{\sum_i^{\eta} exp(y(w_i))} \tag{56}$$

$$y(w_t) = b + U \cdot (\frac{1}{\delta} \sum_{i=1}^{\delta} v_i) \tag{57}$$

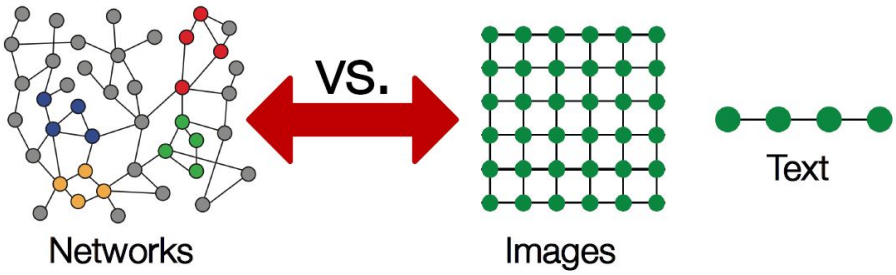
Formal derivation of this part of the notes is discussed in the *Anonymous Walk Embedding* paper [link].

5 Convolutional 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,

$$h_v^k = \phi(AGG_1\Big[op_1\big[(AGG_2(\{MSG(u,v), \forall u \in N(v)\})\big], op_2(h_v^{k-1})\Big]) \tag{58}$$

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), *Trans[OTHERS]* (Sec 5.3), *GraphSAGE* (Sec 5.5), *PinSage* (Sec 5.6) and *GAT* (Sec 5.7). We could argue that the various Trans[X] models are not exactly neural. However, we can think of them as neural methods that have nothing to do with the graph structure, but only contain a minor manipulation (a simple layer) on node/edge embedding then a loss function. Note that neural methods can still share the same loss function as those defined as these Trans[X] methods.

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, $v_h \xrightarrow{l} t_h$. For distance function, the authors defined $d(h + l, t) = h + l - t$.

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)

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

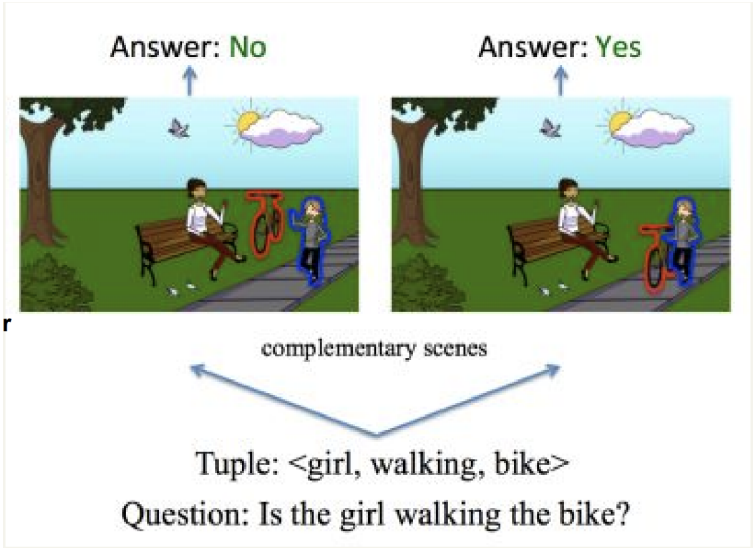
end

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*

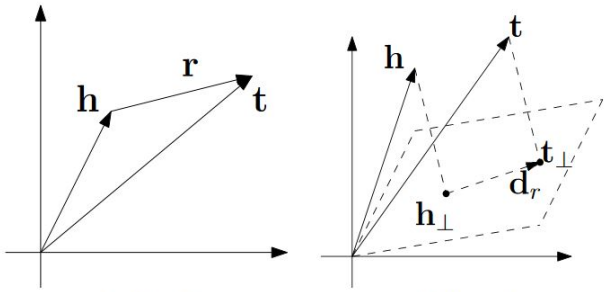
a bike is different from images boy *walking a bike*. So, h is *boy/girl*, l is *walking a bike*, t is the image. We want to make sure that translating *boy/girl* with operator *walking a bike* do NOT land in the same target image. Hopefully this analogy help you understand the rationale behind setting up the gradient update rule. Search for Siamese network [link] for similar setups (commonly used for facial identification).



5.3 Many other Trans[X] models

Numerous Trans[X] have been proposed following the success of TransE. A collection of these models and cmparison of performance is presented [here].

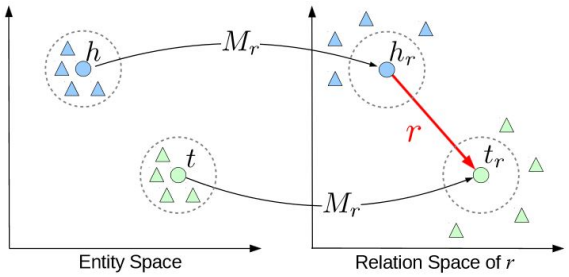
TransH Knowledge Graph Embedding by Translating on Hyperplanes (AAAI 2014) [link]. This model claims to better handle reflexive (think of "invertible"), many-to-one and one-to-many relationships by projecting h, t onto a hyperplane first before training using TransE metrics. That is, for each relationship r, we derive a hyperplane represented by w_r. Then the distance function becomes $d(h+r,t) = ||(h-w_r^T h w_r)+r-(t-w_r^T t w_r)||_2$. Notice that the additional vector means all relationships now require one additional $\mathcal{R}^k$ vector, increasing model complexity. Image below shows TransH projecting both h and r onto the same hyperplane first before computing distance measure.



(a) TransE (b) TransH

TransR Learning Entity and Relation Embeddings for Knowledge Graph Completion (AAAI 2015) [link]. The main argument for TransR is that node embeddings can be in distinct entity space for different relationships. TransE and TransH assumed that all embeddings exist in one shared latent space. As you might have guessed, the new distance function is simply $d(h + r, t) = ||hM_r + r - tM_r||_2$ with translation matrix M_r for each relation r. Notice how this is the same thing as applying a $1 - layer$ perceptron to node embeddings (and remember this fact if you want to implement this method).

In lecture you saw this image as part of TransE, but it should be part of TransR.



TransD Knowledge Graph Embedding via Dynamic Mapping Matrix (ACL 2015) [link]. This work goes one step further than TranR, arguing that all entities and relations require individual projections, therefore, we have to create projection vector v_p (turns into h_p, t_p for equation) and r_p. We then have

$$M_{rh} = r_p h_p^T + I^{m \times n} \tag{59}$$
$$M_{rt} = r_p t_p^T + I^{m \times n} \tag{60}$$
$$h_\perp = M_{rh} h \tag{61}$$
$$t_\perp = M_{rt} t \tag{62}$$
$$d(h + r, t) = ||h_\perp + r - t_\perp||_2 \tag{63}$$

Notice that M_{rh}, M_{rt} look suspiciously like a neuron with bias. Again, these Trans[X] methods only deal with embedding, but we can look at them from a neural lens and stuff them into frameworks like Tensorflow and Pytorch.

Each of the papers we referenced here have a well-written (though the same content) background section and performance/model compelxity comparison against previous models. Read the original papers to compare number of parameters needed for each of these models.

5.4 Vanilla Graph Convolution

We have developed a very general version of what all graph convolution methods should look like. A vanilla graph convolutional neural network defines node embedding as

$$h_v^k = \sigma\Big(W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|} + B_k h_v^{k-1}\Big) \tag{64}$$

Here, σ is an activation function like *ReLU* or *Tanh*. $W_k \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|}$ gives a transformed average embedding from neighbours, $B_k h_v^{k-1}$ gives a transformed embedding from previous iteration. Hence aggregation is just taking average, and the *op* is simple matrix multiplication.

In fully vectorized form, for embedding matrix $H = [h_1, h_2, \dots h_{|v|}]$, A as adjacency matrix, D as degree matrix, and $\hat{A} = D^{-1/2} A D^{-1/2}$

$$H^{(l+1)} = \sigma(H^{(l)} W_0^{(l)} + \hat{A} H^{(l)} W_1^{(l)}) \tag{65}$$

After obtaining the embeddings, we can `(1)` Use one of the Trans[X] methods to further manipulate the embeddings and leverage its loss function `(2)` Directly use it for prediction by passing it through 1 additional neural layer for n-class classification `(3)` Dot-product (or any distance metric) with nodes that should be in the same class with appropriate loss aggregation. All of these lead to an aggregate loss from which gradient descent method can be used to optimize embeddings. Most graph neural networks follow the same setup.

5.5 GraphSAGE, Inductive Representation Learning on Large Graphs

GraphSAGE (NIPS 2017) [link] does `(1)` replaces taking average with a generic aggregation function `(2)` replaces embedding addition with embedding concatenation. Its expression can be written as

$$h_v^k = \sigma\Big(CONCAT[AGG(h_u^{k-1}, \forall u \in N(v)), B_k h_v^{k-1}]\Big) \tag{66}$$

Here, the generalize aggregation function can be average, max pool, min pool, RNN or any method that converts a collection of neighbour embeddings into one aggregate embedding.

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

Pinsage (KDD 2018) [link] is an improvement on GraphSAGE, making it suitable for processing large graphs. Remember that Pinterest has billions of images and many more interconnecting edges. It is impossible to perform matrix multiplications that we came up with just now. Instead of aggregating on all neighbours, Pinsage samples a select set of neighbours then perform aggregation on these neighbours only.

The Pinsage formulation is a bit more involved, so we break it down to a few steps. The first is convolution step, Q, W are weight matrices, q, w are bias, α is a scaling factor. Notice that we are multiplying all neighbours by a shared weight then scaled by neighbour importance function α. Then the normal GraphSAGE-like aggregation with bias is applied. Finally new embedding is normalized to ensure its magnitute does not change over several iterations.

$$n_u = \sigma\Big(\alpha[ReLU(Qh_v + q), \forall vinN(u)]\Big) \tag{67}$$

$$h_u^{new} = ReLU(W \cdot CONCAT(h_u, n_u) + w) \tag{68}$$

$$h_u^{new} = \frac{h_u^{new}}{||h_u^{new}||_2} \tag{69}$$

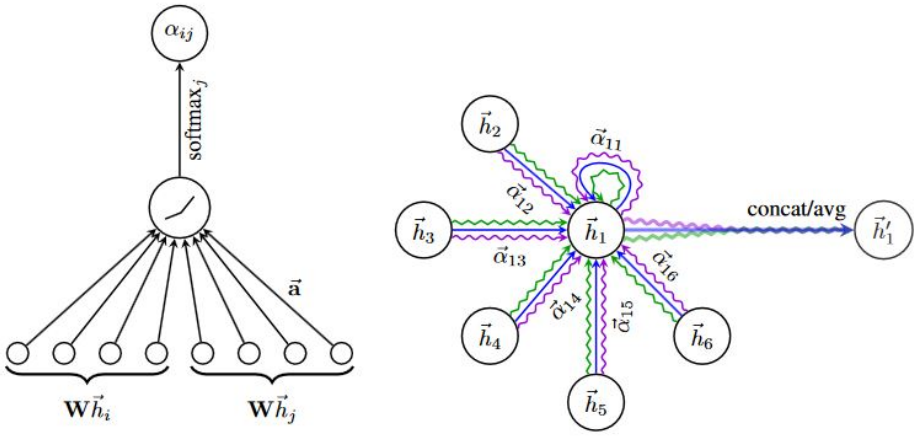
Neighbourhood definition is fairly simple. Again we have neighbourhood function $N(u)$ for vertex u. At layer 1, neighbours are simply $N(u)$, but in the second layer, we have to perform convolution on $N(u) \cup u$, the third layer is consequently $N(N(u)) \cup N(u) \cup u$. You can think of this as a wave propagating outward where the center is still generating new waves. The same analogy is applicable to vanilla GCN.

From the paper, these neighbourhoods are sub-sampled based on visit frequency (by tracking set union with counts) to reduce total number of neighbours involved in the convolution. Also, node mini-batches with positive and negative samples are selected such that only a small portion of the graph need to be loaded at a time. The graph is re-indexed to so that a generic graph neural network algorithm can still process the adjacency matrix. This concept is seen in numerous other

GNN implementations. In fact, extracting a sub-graph is relatively easy for Pinterest because as described before, the Pinterest graph only has 3 components: images, boards and users.

5.7 GAT, Graph Attention Networks

In PinSAGE we introduced a neighbour importance parameter α. This parameter originates from Graph Attention Network (ICLR 2018) [link]. The effect of graph attention is similar to attention for NLP tasks, where the model learns to ignore certain neighbours that have low priority, like skipping *and, a, the*, etc. for developing semantic understanding. We can also have multi-head attention the same way as NLP ones, which we won't be discussing in this note. The image below represents how `(1)` attention weight is computed between 2 nodes `(2)` how attentions are used together to re-weight neighbour contributions.



Define α as a function that describes such importance assignment function, then attention coefficients $e_{v\to u}$ can be written as

$$e_{vu} = \alpha(Wh_u, W_{hv}) \tag{70}$$

Consequently, we can write α_{vu}

$$\alpha_{vu} = softmax_i(e_{vu}) \tag{71}$$

$$= \frac{e_{vu}}{\sum_{q\in N(u)} e_{qu}} \tag{72}$$

In the original GAT implementation, attention mechanism α was implemented as a 1-layer neural network with LeakyReLU activation scaled by $\hat{\alpha}$ (for a 1-layer neural network, its parameters should be a matrix, but we only have a scaler output, thus $\hat{\alpha}$ is sufficient). We can re-write the

above e_{vu} as

$$e_{vu} = \hat{\alpha} \cdot CONCAT[Wh_u, Wh_v] \tag{73}$$

Then finally we can express node embedding h_u as

$$h_u^{new} = \sigma\Big(\sum_{v \in N(u)} a_{vu} W h_v \Big) \tag{74}$$

5.8 List of other notable papers

See a list of notable papers [here].

6 Recurrent Model for Graphs

The previous section built the foundation for applying neural methods for graph problems. We covered a number of Trans[X] models and convolutional models. Here, we continue on to recurrent model for graphs. We do not make the distinction between LSTM and GRU, as suitable recurrent models can be easily swapped, as long as dependencies (for generative models) are respected.

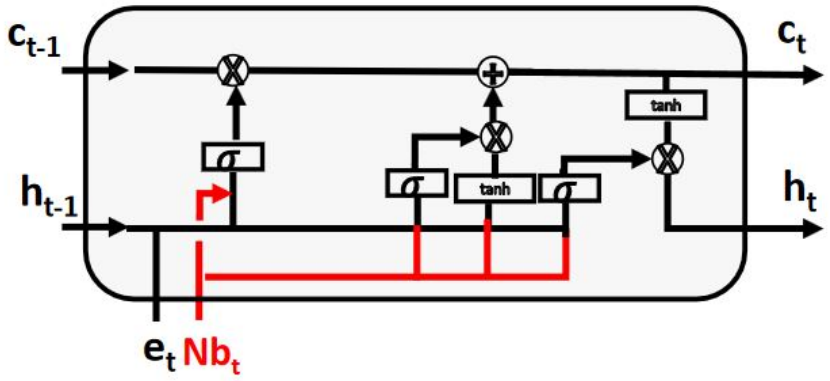
See class note of CS224N on general recurrent structures [link].

6.1 Recurrent Model for Graph Embedding

Recall that in Section and Section we discussed how to use random walk and similarity function to represent transition probabilities of such random walk. Instead of computing $P(x_t|x_{t-1} \dots x_1)$ for co-occurance probability, we pass the corresponding embeddings $h_1 \dots h_t$ to LSTM model. One such implementation is presented in Learning Graph-Level Representations withRecurrent Neural Networks [link], presented as below. Notice that the model has additional input channel Nb_t, meaning that it takes both `(1)` embedding of node on the walk e_t AND `(2)` aggregated node information from $N(t) \rightarrow Nb(t)$.

Author of this paper acknowledge the issue of random walk with graph isomorphism. With sufficient number of walks sampled and having an order-invariant aggregation function (sum, average, max-pool) make it at least independent of order of aggregation. We should be mindful of inherent property of data before deciding on the operator (associative vs non-associative, proper way of saying order-invariant). For example, chirality is quite important for classifying molecular

properties, hence non-associate aggergators should be used. Note that the same walk embedding $\rightarrow$ graph embedding procedure still applies here as they did in Section and .

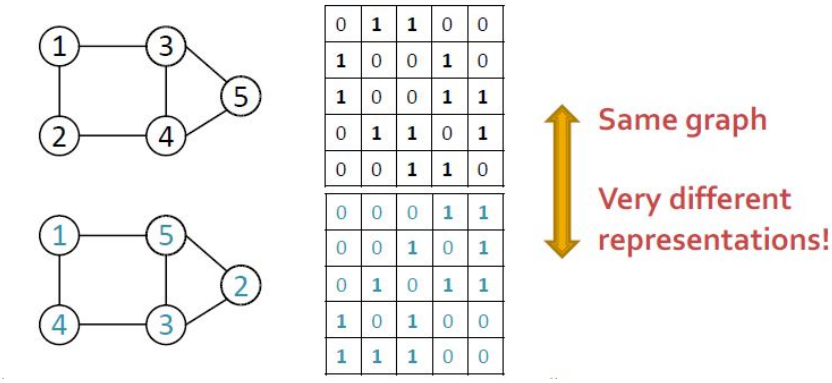


6.2 Recurrent Model for Graph Generation

Graph generation is useful for `(1)` anomaly detection (abnormal behavior, evolution) `(2)` prediction (predicting future from the past) `(3)` simulation (simulation of graph structures) `(4)` graph completion (many graphs are partially observed) `(5)` "what if " scenarios `(6)` insight into the graph formation process itself. Just like any generative model, the generated graph either has to be realistic (social network), or satisfies a certain criteria (drug discovery), or both.

6.2.1 Basics of Generative Models

Any generative problem is hard because the typical setup is generating a large output from limited input. For example, image generation models start with a random vector and outputs a meaningful image. Graph generation problems are typically given a set of known nodes (a number of atoms available for drug molecule) and the output is connection between these nodes. Essentially generating a $n \times n$ adjacency matrix from n nodes. As discussed before, there are also isomorphic graphs that we might have to identify/resolve given properties of the problem. Image below shows that isomorphic graphs can have very different adjacency matrices. Unlike images where pixels are locally correlated and larger features are globally correlated, each edge in a graph can depend on the entire graph due to *long range dependencies*.



We should be familiar with the probabilistic interpretation of generative model. Suppose we have data (X, Y) where X is data, Y is label and θ is the set of model parameters.

- **Discriminative** Find θ to maximize $P(Y = y|X, \theta)$
- **Generative** Find θ to maximize $P(X|Y = y, \theta)$
- **Generative (without class label)** Find θ to maximize $P(X, \theta)$

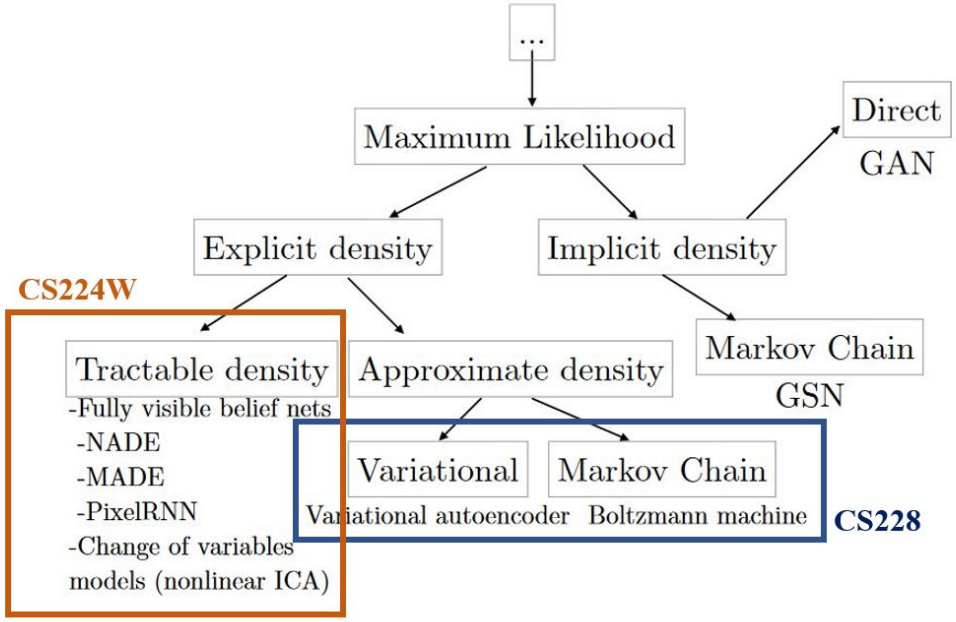
Expressing the without class label case for the entire dataset, then we want to have MLE parameter θ^*

$$\theta^* = \arg\max_{\theta} \sum_{x_i \in X} logP(x_i|\theta) \tag{75}$$

And for generating the actual sample, we simply apply the with class label case and find x_{gen} that maximizes $P(x_i|y, \theta)$. For sequential generators, we express the total generated probability as the following, where t represent one sample generation step.

$$P(\mathbf{x}; \theta) = \prod_{t=1}^{n} P(x_t|x_{t-1}, \ldots x_1; \theta) \tag{76}$$

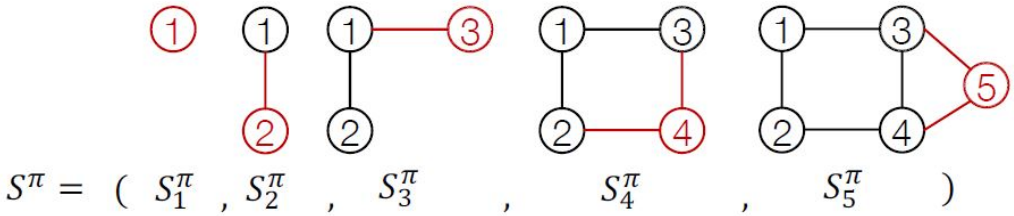
This can of course be tackled using traditional Bayesian methods, as those presented in CS228. An lineage of generative models is presented below as part of Ian Goodfellow's 2016 paper on GAN [link]. GSN here stands for generative stochastic network, first presented [here], serving as a generalized denoising auto-encoder.



The models we discuss here for graph generation serves both density estimation and sampling. This is different from GAN and VAE (variational auto-encoder) where a distinct generator-discriminator separation is built in.

6.2.2 Graph RNN

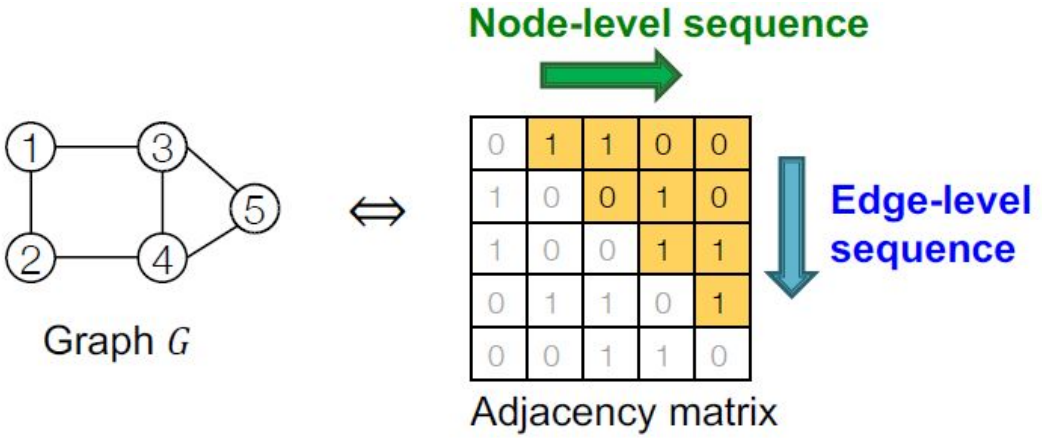
GraphRNN was proposed in 2018 [link] to tackle the issue of graph generation using neural methods. Instead of generating all images at once, we treat graph generation as a sequential problem. That is `(1)` At the beginning, start with 1 node `(2)` For each subsequent step, add 1 additional node, then decide whether edges exist between this added node with all existing nodes.



In the above illustration, S_π is a collection of actions to be taken by the graph generation algorithm. More concretely, as shown in this table.

As shown, graph generation naturally breaks down into node selection (or not, if we process nodes in order, but we still need a node embedding) then edge generation. From the perspective of adjacency matrix, we can produce the following illustration.

Step	Node Action	Edge Action
S_1^π	Add node 1	
S_2^π	Add node 2	Add edge $1-2$. Do not add edge
S_3^π	Add node 3	Add edge $1-2$ Do not add edge $2-3$
S_4^π	Add node 4	Add edge $2-4, 3-4$ Do not add edge $1-4$
S_5^π	Add node 5	Add edge $3-4, 4-5$ Do not add edge $1-5, 2-5$



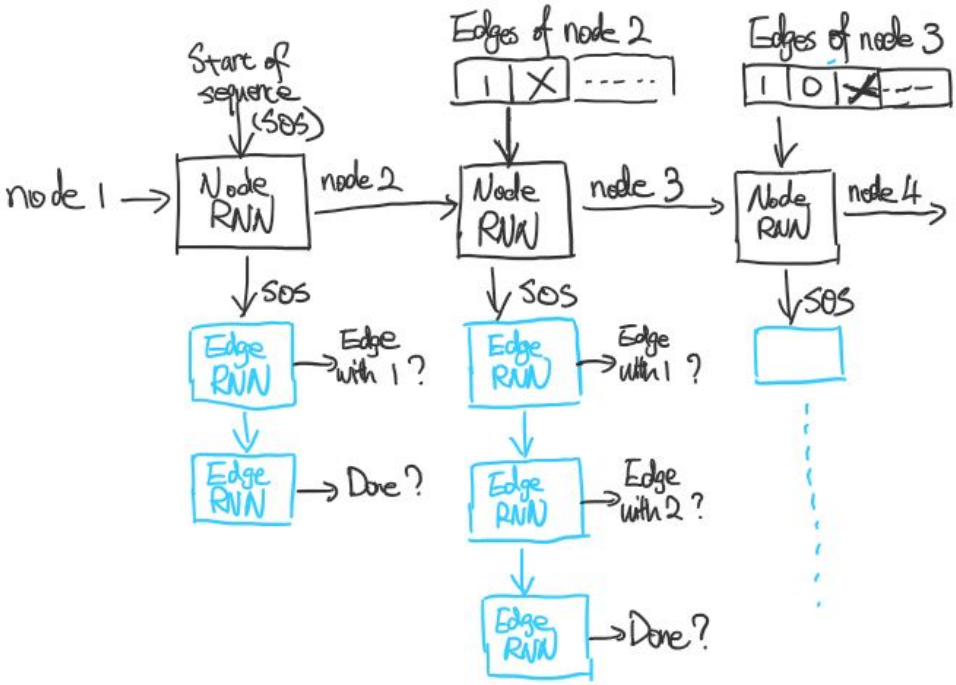
Recall in Section 6.2.1 we raised the complexity/tractability issue of generating n^2 entries in the adjacency matrix given n nodes. The solution we just presented does NOT solve this issue. It merely turned into a $\frac{n^2}{2}$ problem, not decreasing the magnitude of complexity. To solve this, we can claim that the node order (since we are generating node embeddings anyways and we are only given the number of nodes), that we are following BFS order. Let's use the 5 node example again

1. Starting at node 1, we call this layer 0
2. Node $2, 3$ are generated. These connect to node 1, therefore they are in layer 1.
3. Node 4 does not connect to node 1, so neighbour generation (think BFS) for layer 0 is complete. Future nodes can only connect to $2, 3, 4$.
4. When we generate edges for node 5, node 1 is no longer a candidate.
5. *Suppose* that edge $3-4$ does not exist. Then neighbour generation for layer 1 is complete, future nodes can only connect to $4, 5$.

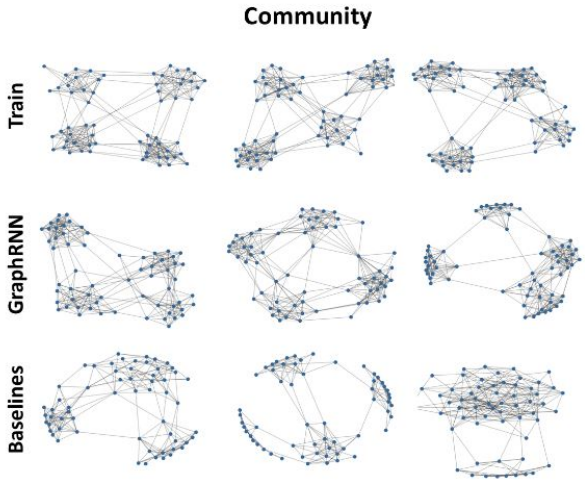
With BFS ordering, we only need to keep track of candidate nodes for 2 layers, instead of the entire graph. This still does not solve the tractability issue, but reduces the average complexity to

$O(n)$ rather than $O(n!)$. This enforced ordering also limits $O(n!)$ possible node orders to number of BFS orders possible for the given graph.

With these in mind, the recurrent structure will be similar to a NLP-oriented structure that has a higher level RNN for words/tokens, then a lower level RNN that generate word embedding directly from characters. The following graph shows *unrolled* RNN for node level and for edge level. Drawing an NLP analogy, the node level RNN provide "context" for edge level RNN. The source of training loss is $0/1$ prediction generated from edge RNN. To speed up training, authors of the graph RNN paper used *teacher-forcing* tecnique. That is, at training time, input to each RNN cell is the correct value from the previous step. See detailed explanation of teacher forcing [here]. Here, For each node, we use the *correct* edge generation vector from its predecessor. For each edge, we use the *correct* individual edge result from the previous edge. *Done/EOS* signal from edge RNN is not necessary because we know exactly how many edges are supposed to be evaluated on. However, adding this additional signal provides more information to the RNN structure.



To evaluate the quality of generated graph, we can use various similarity metrics to compare the generated graph with some in-sample graphs. Similarity metrics can be based on graph statistics (Section 2) and visualization like the comparison below. This image shows that GraphRNN result resembles training samples, whereas baseline results are far from ideal.



Aside from these evaluation metrics, we can also setup the generation loss to include **goals** and **rules**. This is used for drug discovery where we want to generate valid chemicals and achieve certain chemical properties.

Since graph generation is an iterative process, reinforcement learning methods can also be employed, leading to *Graph Convolutional Policy Network* (NIPS 2018) [link] where we setup the RL strategy to target metrics above.

At the moment, researchers are moving toward 3D graph generation where incorporation of positional information and/or representation of point cloud is open to discussion. Moreover, as we have discussed, complexity of generating graphs grow at a not-so-linear rate, thus generating large graphs can be a problem. Another layer of abstraction that have sub-structures as "context" could be used like Kronecker graph generation method. We can also use neural methods for anomaly detection, such as unusual friend request, invalid chemicals, etc.

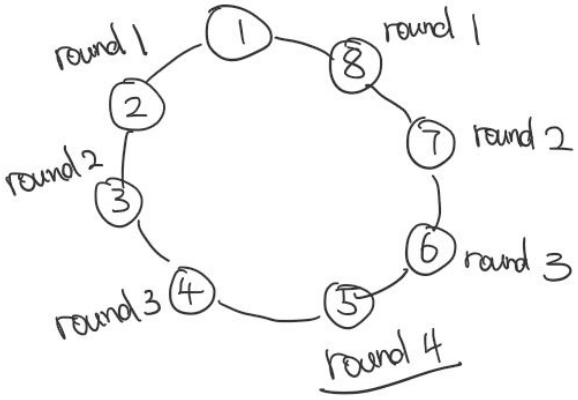
6.3 Limitations of Graph Neural Network

In the previous 2 sections we discussed current development of graph neural networks, describing various state-of-the-art implementation for node classification and graph classification. Built on top of these methods, link prediction for content recommendation has also achieved incredible performance (NIPS 2018) [link]. Despite these achievements, GNN methods have limitations.

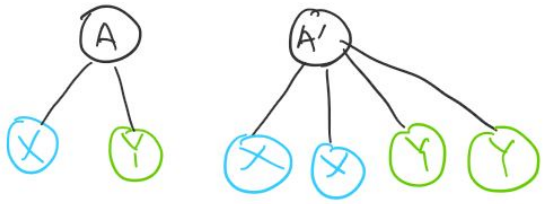
6.3.1 Capturing Graph Structure

Graph convolution is fundamentally a 1-step operation that does not track whether a node has been visited like a path finding algorithm. Therefore, it has limited capability in recognizing difference in not-so-local structure until several layers of propagation.

Detection of circular graph Consider the following graph, where we are trying to check if the graph is circular. Starting a node 1, we need 4 rounds of information propagation before being able to verify that the graph is circular. With fewer than 4 rounds, we have no way of knowing how node $4, 5, 6$ are connected.



Distinction between different neighbourhood setup One possible neighbour aggregation function for vanilla graph convolution is taking the average of all neighbour nodes. Consider the 2 following scenarios, where X, Y are type of nodes. Clearly, simple average does not distinguish between having 2 neighbours and 4 neighbours, which could be important for many applications. Similarly, max pooling would lead to the same issue if embeddings of the same node class are the same.



To solve this issue, we have to design an injective neighbourhood aggregation function. Recall that *injection* is the same as *one-to-one correspondence*, meaning that for function f, $f(a) = f(b)$ if and only if $a = b$. With an injective neighbourhood aggregation function, $\sigma(f_{injective}(X, Y)) \neq \sigma(f_{injective}(X, X, Y))$.

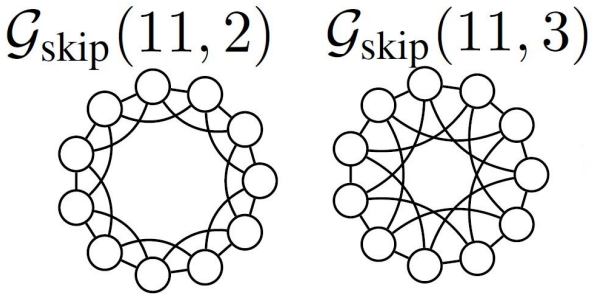
It might sound strange but sum-pooling solves the above issue (ICLR 2019 [link]). And yes, sum-pooling does exactly what you think it does, just element-wise sum. Since we have $tanh$ as activation, we are not concerned with overflowing. $Tanh$ is an injective function because it is monotonically increasing. Also note that nested injective functions are also injective, thus convolution of multiple layers with sum-pooling remain injective, allowing difference in graph structures to be captured (still limited to number of layers, so "detection of circular graph" type of issue cannot be solved).

Weisfeiler-Lehman Isomorphism Test The paper we just mentioned and its proposed sum-pooling method claimes to be comparable in power as Weisfeiler-Leham (WL) Isomorphism test, an iterative method that generates a canonical form for a graph. Unfortunately, WL is only surjective (If $f(a) \neq f(b)$ then $a \neq b$, but the other direction is not enforced). Here is the algorithm [link] for graph $G = (V, E)$ and neighbour function $N(v)$

1. Assign unique representation $C_{0,v}$ to all $v \in V$
2. Iterate the following steps for at least $|V|$ rounds
3. For all $v\ inV$, at iteration i (starting from $i = 1$)
   (a) Get representations from previous step of all its neighbours, form a multiset (where duplicates are allowed, order does not matter) $\{C_{i-1,k} \forall k \in N(v)\}$
   (b) Assign $C_{i,v} = Hash\Big(\{C_{i-1,k} \forall k \in N(v)\}\Big)$

Suppose we want to compare $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, we would run WL on each of these graphs, thus have $multiset\{C_{|V_1|,v} \forall v \in V_1\}$ and $multiset\{C_{|V_2|,v} \forall v \in V_2\}$. If the generated multisets are different, then due to surjectivity of the algorithm, G_1, G_2 must not be isomorphic. Note that we can run the algorithm for both graphs in parallel, and terminate as soon as difference is observed at the end of each of the $|V|$ rounds.

From this algorithm, it is not hard to see that graphs that have similar local structure are likely going to lead to the same multiset output. The following shows one such case, where WL algorithm output is the same for $G_{skip}(11, 2)$ and $G_{skip}(11, 3)$.



6.3.2 Adversarial Attacks

Graph neural network is a relatively new field. Model robustness has not been the top priority for researchers compared to the raw performance of GNN models. Theoretically, attack on GNN models aim to modify target/node features/classifications and/or the existence of edge between nodes. In the real world, we can classify direct attack (those that influence the target directly) as

- Modify the target's features. (Change website content generated based on node embedding)

- Add connections to the target. (Add fake followers and likes)
- Remove connections from the target. (Unfollow, untrust users)

For indirect attacks (the attacker modifies its own features/connections), the attacker can use its own characteristics to skew predictions made by GNN models to achieve its goal.

NEEDS WORK: Read https://arxiv.org/pdf/1805.07984.pdf then write a description of how adversarial attack is done on semi-supervised learning.

Is it just a GNN derivation of https://arxiv.org/pdf/1412.6572.pdf?

At the current stage, we can conclude that GNN methods are not safe to adversarial attack. Noticeable research in terms of model robustness include [link] and [link], but these approaches have only been evaluated on standard datasets with vanilla GNN tasks. Effectiveness of these methods in the real-world has yet to be verified.