Advanced Data Mining

M2177.003000, Seoul National University, Fall 2019

These are lecture notes for "Advanced Data Mining" taught by U Kang at Seoul National University during the fall of 2019. These notes are not official and for personal use, and thus have not been proofread by the instructor for the course and may be of limited use to you. These notes live in my lecture notes respository at

https://github.com/qmdnls/lecture-notes.

If you find any errors, please open a bug report describing the error, and label it with the course identifier, or open a pull request so I can correct it.

Contents

Syllabus

Instructor	Kang U, https://datalab.snu.ac.kr/~ukang/
Lecture	MW 3:30–4:45 PM, 301-101
Textbook	None
Midterm	Wednesday, October 23, 2019
Final	TBA

Data mining attracted much interests as an essential tool for big data analysis. Especially, designing and implementing advanced data mining algorithms and analysis platforms play crucial roles in extracting executable knowledges from big data. This course covers advanced data mining techniques, algorithms, and core platforms for big data analysis. This course also covers the techniques to effectively analyze very large data and high-speed data. A very important aspect of this course is the course project. Students will pick an interesting data mining project, and do data mining researches on the topic. At the end of the course, students will learn how to do interesting researches in data mining, and how to write good papers.

Your final grade for the course will be determined by

$$10\% \text{ attendance} + 20\% \text{ homework} + 20\% \text{ midterm} + 20\% \text{ final} + 30\% \text{ project}.$$

1 Graphs

Starting with a basic recap of graphs. I will not go into much detail here as most of these things should be familiar from previous lectures. Lecture did not have any formal definitions so I will leave them out here as well.

> **Definition** (Graph)**.** A graph $G = (V, E)$ is defined by a set of vertices V and a set of edges $E = V \times V$ between these vertices. *Graph*

Graphs are a way of specifying relationships among a set of items. These relationships are expressed in the forms of edges between vertices (or nodes).

1.1 Types of graphs and basic graph terms

There are different types of graphs:

- directed graph
- undirected graph
- weighted graph
- unweighted graph
- simple and attributed graph

Some examples of real-world graphs include:

- social networks
- citation networks
- world wide web
- protein interactions
- document networks
- computer networks (ArpaNet)

Let's define some basic graph terms: Two vertices are *adjacent* if they share a common edge. Two adjacent vertices are *neighbors*. An edge is *incident* with another edge if they share a vertex. An edge is incident with two vertices. The *degree* of a node is the number of its neighbors. In a directed graph we define the *in-degree* and the *out-degree* based on the number of neighbors with incoming or outgoing edges.

1.2 Small world phenomenon

The *small world phenomenon* describes the results of a series of experiments that have shown that social networks in the real world have a low average path length. Milgram's experiment for instance has shown that there was an average of six degrees of separation in the US. Another popular number: Kevin Bacon number. Describes the distance from Kevin Bacon in the actor-actor collaboration graph. The average Kevin Bacon number of actors is 2.9 which is surprisingly low.

1.3 Densification power law

Describing the growth of real-world graphs has important applications in network simulation and network prediction. But how do these graphs evolve over time? Conventional wisdom was that the number of edges grows linearly with the number of of nodes, so the average degree stays constant, while the diameter slowly grows. However new findings have shown that networks become denser over time, this is called the *densification power law*. Furthermore, the diameter appears to be shrinking as a result of this densification.

Theorem 1.1 (Densification power law)**.** *For the number of edges $E(t)$ and the number of nodes $N(t)$ at time t, we have the following relation: $E(t) \propto N(t)^a$ with $a \in [1, 2]$ where $a = 1$ yields the traditionally assumed constant out-degree (linear growth), while $a = 2$ gives quadratic growth.*

In real world graphs this densification exponent differs and depends on the type of underlying real-world relationships but is typically $a > 1$ which means the average degree is increasing. Further analysis of real-world graphs shows that this densification leads to shorter distances between nodes and thus the diameter shrinks over time.

1.4 Forest fire model

To describe these properties of real-world graphs we introduce a new graph generation model that follows the densification power law and that has a shrinking diameter over time. In the *Forest Fire* model a node randomly chooses an "ambassador" when it arrives. It then starts burning adjacent nodes with probability p and adds a link to burned nodes. This "fire" then spreads recursively. This is very similar to the process of friendship networks in the real world as people will often become friends with their friend's friends. This forest fire model generates graphs that densify and have a shrinking diameter.

1.5 Erdős–Rényi random graph model and preferential attachment

The *Erdős–Rényi random graph model* (ER model) is a graph generation model. Given the desired number of vertices $n \in \mathbb{N}$ and a probability $p \in [0, 1]$ we create n vertices and include an edge between any two vertices with probability p. The degree distribution in this model follows a Poission distribution which means it can be described by $P(k) \sim \frac{e^{-\lambda}\lambda^k}{k!}$ where k is the degree.

However, real graphs show different behavior. We see that the degree distribution typically follows a power law: $P(k) \sim k^{-r}$. To better describe this real-world graph behavior we introduce the *preferential attachment* model. In this model networks continuously expand by the addition of new vertices. These new vertices attach preferentially to other vertices that are already well connected. Specifically, we define the probability that a new vertex is connected to an existing vertex i by

$$P_i = \frac{d(i)}{\sum_j k_k}$$

where $d(i)$ is the degree of vertex i. This then leads to a power law distribution in the resulting graph.

1.6 Graph robustness

How robust are real-world graphs? What happens when we remove nodes from them? We find that preferential attachment is very robust against random failure (low probability of high-degree nodes failing), however it is vulnerable to attacks (attack only high-degree hub nodes). In general, scale-free networks have a high-degree of tolerance against random failures that exponential networks do not have. This observation could explain why many complex systems in the real world manage to function and have high error tolerance.

1.7 Community connection model

A *rebel* is a vertex in a graph that does not belong to its largest connected component (GCC). The rebel probability describes a probability of a given node being a rebel. Models introduced thus far (preferential attachment, forest fire) only have a single connected component. To generate graphs that have many connected components we introduce the *Community connection model*. The model has two parameters: p_{host} and p_{step}. When a new node joins the network in this model, it connects to a new host with p_{host} and then begins a random walk with p_{step} until there are no more "steps". This is repeated for all hosts. We find that this model describes the rebel probability found in real-world graphs very well.

1.8 Power laws in the internet

A few questions that motivate us: What does the internet look like? Are there any topological properties of the internet that remain constant over time? How will it develop in the future? How can we generate internet-like graphs?

There are three main power laws in the internet.

Theorem 1.2 (Power law 1 (rank exponent))**.** *The out-degree d_v of a node v is proportional to the rank of a node r_v to the power of a constant $\mathcal{R}$:*

$$d_v \propto r_v^{\mathcal{R}}$$

Theorem 1.3 (Power law 2 (out-degree exponent))**.** *The frequency f_d of an out-degree d is proportional to the out-degree to the power of a constant $\mathcal{O}$:*

$$f_d \propto d^{\mathcal{O}}$$

Theorem 1.4 (Power law 3 (eigen exponent))**.** *The eigenvalues λ_i of a graph are proportional to the order i to the power of a constant $\mathcal{E}$:*

$$\lambda_i \propto i^{\mathcal{E}}$$

1.9 Generation of power law distributions

Power laws can be used to describe graphs more accurately: the average often falsely implies a uniform or Gaussian distribution when in reality distributions can be skewed. For example, 85% of the nodes in the Int12-98 dataset have an out-degree that is smaller than the average. Besides graph descriptions, power laws can be useful for graph generation, for prediction and extrapolation and to evaluate protocol performance (in the internet for example). Finding power laws: power law distributions are linear in a log-log scale. Thus, we can discover them by plotting data in log-log (but beware: not all distributions that look like power laws at first glance follow power laws).

We can describe a power law distribution in the following way:

$$p(x) = Cx^{-a} \; \forall \; x \geq x_{\min}$$

for a constant C and an exponent a which gives us a way of directly computing $p(x)$ for any x. The exponent a can often be estimated from data like so:

$$a = 1 + n \left(\sum_{i=1}^{n} ln \left(\frac{x_i}{x_{\min}} \right) \right)^{-1}$$

There are three types of power laws: PDF (*probability distribution function*, also frequency-count plot), *Zipf plot* (rank-frequency) and *complementary cumulative distribution function* (CCDF, also true for NCDF $= 1 -$ CDF). We find that, if any one of the three follows a power law, so do the other two.

In short we can say that any distribution which follows a "the rich get richer" principle, typically follows a power law. Examples of generative mechanisms for power law distributions are:

- Chinese restaurant process (Yule distribution)
- Combination of exponentials
- Inverses of quantities
- Random walks

Chinese restaurant process. One such way of generating power law distributions is given by the *Chinese Restaurant Process* (also Yule distribution): any newcomer to a restaurant sits down at an existing table. In his choice the newcomer prefers large groups. She only chooses to start a new table with probability $\frac{1}{m}$. This is also referred to as a *Yule process*.

Combination of exponentials. Combination of exponentials is a very simple technique to generate power law distributions. One example is radioactive decay with half-life $-a$ and $p(y) = e^{ay}$. Another example is Russian roulette where the participants' capital x increases every time they survive with $x \sim e^{by}$. Then their final capital follows a power law distribution. Lastly, imagine a monkey typing on a typewriter. If each letter is equiprobable and the space bar has probability q_s then the frequency of the x-th most frequent word is x^{-a}, i.e. follows a power law.

Random walks. For random walks the number of steps required to arrive at the same position, the *inter-arrival time*, follows a power law.

Random multiplication. Starting with C dollars and a random interest rate $s(t)$ for each year t, we get $C(t) = C(t-1)(1+s(t))$. We have $\log C(t) = \log C + \log.. + \log..$ which is a Gaussian distribution and thus $C(t) = \exp(\text{Gaussian})$ which is a lognormal distribution. The lognormal distribution looks like a power law in its tail distribution (but strictly is not a power law).

Fragmentation. Starting with a stick of length 1, break it at a random point $0 \leq x \leq 1$. Then recursively break the resulting pieces at random. The resulting length distribution is lognormal (analogous to random multiplication).

1.10 Radius plots to describe large networks

Real-world networks often have a giant connected component (GCC). This component is seen to be growing over the years (example: Wikipedia, LinkedIn) while the tail slope remains constant in the distribution (see slides for graphs). But what does the structure of these large networks look like? Which nodes are the most central and how does the structure change over time?

To better describe the structure of a graph we can use its radius: we plot the count of nodes over the radius. Different structures yield different *radius plots*: in a clique all nodes have radius 1, in a star one node has radius 1 and all other nodes have radius 2, similarly a chain and a graph with a near-bipartite core have distinct radius plots (see slides). As a measure of node centrality we can use its radius: to choose a node to advertise in a graph, simply choose the node with the smallest radius for fastest propagation in the network. We can study the change of the structure of networks over time by looking at the radius plot over time.

In real graphs, the radius plot is multi-modal (again, see slides). The LinkedIn and US patent radius plots are bi-modal. This is due to the fact that there is outsiders which have a very small radius, then there is the GCC in which some very-connected nodes have a smaller radius but almost all nodes have a slightly larger radius and then there is the whiskers, which are further outside the core, have a longer path to the core and thus have a larger radius. In other datasets (Yahoo Web) with a multi-modal radius plot there is more than core. Over time, real-world radius plots expand and contract (*expansion-contraction*).

Also consider *radius-degree plots*: describes the degree of nodes of a given radius. We see that high-degree hubs have a lower radius while lower-degree nodes mostly have a larger radius.

2 Spectral analysis

We will see that random walks on graphs and their probabilities are related to electric networks. We will first recap basic restistance theory.

2.1 Resistance theory

For voltage V, current I, resistance R and conductance C we know that

$$C = \frac{1}{R} \qquad\qquad I = CV = \frac{V}{R}$$

and in other words $V = RI$. In series we have the total resistance and total conductance

$$R = R_1 + \ldots + R_m \qquad\qquad C = \frac{1}{\frac{1}{C_1} + \ldots + \frac{1}{C_m}}.$$

Analogous to that, we write the total conductance and total resistance in parallel as

$$R = \frac{1}{\frac{1}{R_1} + \ldots + \frac{1}{R_m}} \qquad\qquad C = C_1 + \ldots + C_m$$

The *effective resistance* between points a, b is defined as $R_{ab} = \frac{V_{ab}}{I_{ab}}$. According to *Kirchhoff's law* the effective resistance can be computed by flow in = flow out. More specifically, given three points V_1, V, V_2 in series with conductors C_1, C_2 between them (see slide for visualization), we have flow from V_1 into V is I_1 and the flow from V into V_2 is I_2 and

$$I_1 = C_1(V_1 - V) \qquad\qquad I_2 = C_2(V - V_2)$$

With Kirchhoff's law $I_1 = I_2$ and thus it follows that

$$V = \frac{C_1}{C}V_1 + \frac{C_2}{C}V_2.$$

Now consider a weighted graph $G = (V, E, C)$ with weights C where C_{ij} is the conductance for the resistor between i, j and $C_i = \sum_{(i,j)\in E} C_{ij}$. The adjacency matrix A is $A_{ij} = C_{ij}$ if $(i, j) \in E$ and o otherwise. The *Laplacian matrix* L is $L = D - A$ where D is the diagonal matrix with entries $D_{ii} = C_i$. Then,

$$L_{ij} = \begin{cases} C_i & \text{if } i = j \\ -C_{ij} & \text{if} (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Lemma 2.1. *Let V be a vector containing the voltages for all nodes. Then $(LV)_i$ is the residual current at node i.*

Assuming $I = 1$ computing effective resistance becomes as easy as solving

$$LV = \begin{bmatrix} 1 \\ 0 \\ \cdots \\ 0 \\ -1 \end{bmatrix}$$

as $R_{1n} = \frac{V}{I} = V = V_1 - V_n$.

2.2 Random walks and electric networks

Given a graph G with edge weights $C_1, C_2, C_3, \ldots$ we can also consider this graph as an electric network with capacities $C_1, C_2, C_3, \ldots$ and show that random walk probabilities on G correspond to voltages in the electric network.

Consider a random walk starting from x and ending at b, then let h_x be the probability of visiting a point a before visiting b from a random walk starting at x. We can easily see that $h_a = 1$ and $h_b = 0$. For all other points we define recursively

$$h_x = \sum_y h_y P_{xy}.$$

where the probability P_{xy} of choosing to go along edge (x, y) is the weight of the edge as a fraction over the sum of the weights of all of x's edges. Assume we set $V_a = 1$ and $V_b = 0$, then

$$V_x = \sum_y V_y \frac{C_{xy}}{C_x} = \sum_y V_y P_{xy}$$

and thus $h = V$ as h and V are harmonic with the same boundary values (see lecture slides for proof). Therefore, if $V_a = 1$ and $V_b = 0$ then V_x is the probability of visiting a before visiting b in a random walk starting at x and can be measured in an electric network (*interpretation of voltage*).

Now consider a random walk starting at at a and ending at b. Let u_x be the expected number of visits to a point x before reaching b. Then $u_b = 0$ and again recursively

$$u_x = \sum_y u_y P_{yx} = \sum_y u_y \frac{C_y}{C_y} P_{yx} = \sum_y u_y \frac{C_x}{C_y} P_{xy}$$

and it follows that

$$\frac{u_x}{C_x} = \sum_y \frac{u_y}{C_y} P_{xy}.$$

Now let $V_a = \frac{u_a}{C_a}$ and $V_b = 0$, then $V_x = \sum_y V_y P_{xy}$ and $V_x = \frac{u_x}{C_x}$. The current i_{xy} is given by

$$i_{xy} = (V_x - V_y) C_{xy} = u_x P_{xy} - u_y P_{yx}$$

and we know that $u_x P_{xy}$ is the expected number of crossings from $x \to y$ and $u_y P_{yx}$ is the expected number of crossings from $y \to x$. Thus i_{xy} is the expected *net* number of crossings from $x \to y$ in a random walk from a and ending at b. It can be measured as the current from x to y in an electric network by setting $V_a = \frac{u_a}{C_a}$ at a and $V_b = 0$ at b.

2.3 Random walks on graphs

Consider a weighted graph $G = (V, E, W)$ with W_{ij} the weight of an edge (i, j). Then the random walk probabilities are defined by the edge weights as follows:

$$W_i = \sum_{(i,j)\in E} W_{ij} \qquad\qquad P_{ij} = \frac{W_{ij}}{W_i}$$

where P_{ij} is the probability of moving along edge (i, j) if we're at node i at time t. The *hitting time* $H(i, j)$ is defined as the expected number of steps before node j is visited starting from i. However, the hitting time is not symmetric, i.e. $H(i, j)$ is not equal $H(j, i)$ for all nodes i, j. To get a symmetric measure we define the *commute time* $k(i, j) = H(i, j) + H(j, i)$. The hitting time can be used for suggestion of queries for example (see slides).

To compute the hitting time we first define $H(x) = H(x, b)$ for a fixed b as the expected number of steps to reach b from a given x. With $H(b) = 0$ we can compute $H(x)$ as

$$H(x) = 1 + \sum_y H_y P_{xy} = 1 + \sum_y H(y)\frac{W_{xy}}{W_x}$$

by solving the linear system. To compute the commute time we would thus have to solve two such linear of equations to compute $k(i, j) = H(i, j)+H(j, i)$. Alternatively we can use electric network and instead compute

$$C = \sum_i C_i, \qquad\qquad k(i, j) = C \times (\text{Effective resistance})_i$$

2.4 HITS algorithm

In this section we will introduce Kleinberg's algorithm also known as the Hyperlink-Induced Topic Search algorithm or HITS for short. We will first quickly review eigenvectors.

> **Definition** (Eigenvector)**.** Given a $n \times n$ matrix A, we call v an eigenvector and λ an eigenvalue of A if $Av = \lambda v$. *Eigenvector*

A matrix multiplication can be interpreted as a vector transformation. An eigenvector is thus a "fixed point" with regards to this transformation – they remain parallel to themselves by definition. We can find an eigenvector-eigenvalue pair by iteration, convergence speed depends on the ratio $\lambda_1 : \lambda_2$.

Lemma 2.2. *A real, symmetric $n \times n$ matrix A has exactly n real eigenvalues. If A is not symmetric, some of its eigenvalues may be complex.*

Kleinberg's algorithm (HITS) finds the most "authoritative" web page for a given query. Given a *root set*, we expand this set to obtain a *base set* by one move forward and backward, i.e. we obtain all pages that point to a page in this set or that are pointed to by a page in this set. On this resulting graph G we give a high score to all nodes that many important nodes point to ("authorities"). Similarly, we give high importance to all nodes that point to good authorities ("hubs").

We can see that this definition is recursive. Each node i has an authority score a_i and a hub score h_i. Let E be the edges and A be the adjacency matrix of G. Let h, a be $n \times 1$ vectors with the hub and authority scores of each nodes in G. Then for the authority scores

$$a_i = \sum_{j:(j,i)\in E} h_j$$

where h_j is the hub score of node j or more simply

$$a = A^T h.$$

Similarly, we can compute for the hub scores

$$h_i = \sum_{j:(i,j)\in E} a_j$$

where a_j is the authority score of node a and more simply

$$h = Aa$$

and thus we are looking for solutions h, a to conditions

$$a = A^T h \qquad\qquad h = Aa.$$

We can formulate

$$a = A^T h = A^T A a$$

and compute a numerically by starting from a random a' and iterating until we converge. More precisely, the solutions to this problem are the left- and right-singular vectors of the adjacency matrix A with the strongest singular values. Singular vectors of A can be computed with singular value decomposition (SVD): $A = U\Sigma V^*$ where the columns of U are left-singular vectors and columns of V^* are right-singular vectors. The singular values are the diagonal entries of Σ.

2.5 PageRank

PageRank is a node ranking algorithm proposed by the Google founders Larry Page and Sergey Brin in 1998. Given a graph G, we want to identify its most important and central nodes. A node is important if it is connected with many other important nodes. The proposed solution to this problem is to consider random walks and spotting the most "popular" node with a high steady state probability (SSP). A node has a high SSP if it connected with many high SSP nodes (again, recursive).

Let A be the adjacency matrix of G and B the column-normalized transition matrix, i.e. $\sum_i b_{ij} = 1 \; \forall \; j$. Note that B is a (column) stochastic matrix. Also note that B is transposed so that the columns represent “from" and the rows represent “to".

Theorem 2.3 (Perron-Frobenius Theorem)**.** *Let M be a positive $n \times n$ matrix. Then there exists a positive Perron-Frobenius eigenvalue r such that for any other eigenvalue λ we have $\lambda < r$. There also exists an eigenvector v of M with eigenvalue r such that all components of v are positive and all other eigenvectors have at least one negative component.*

With the Perron-Frobenius Theorem, there exists a p for B such that $Bp = \lambda p$ where λ is the highest eigenvalue and $\lambda = 1$ since the matrix is column-normalized. We can obtain p through *power iteration*: starting with a vector p_t we can obtain $p_{t+1} = Bp_t$ and it will eventually converge to the eigenvector with the largest eigenvalue which is exactly p.

However, B is not irreducible as not all nodes can be reached by a random walk starting from an arbitrary node. In order to make B irreducible, go to a random node with probability $1 - c$. We can realize this by adding edges to the graph to all other nodes with transition probability $1 - c$. Thus

$$\begin{aligned} p &= cBp + \frac{(1-c)}{n} \, 1 \\ &= \frac{(1-c)}{n} \, (I - cB)^{-1} \, 1 \end{aligned}$$

where I is the unit matrix. Alternatively we can write the modified transition matrix as

$$M = cB + \frac{1-c}{n} \, 1 \, 1^T$$

and compute p through power iteration:

$$p = Mp$$

where p denote the SSP and PageRank scores of M.

2.6 Random walk with restart

Random walk with restart is a sort of “personalized PageRank" algorithm. The goal is to compute proximities of other nodes to a given query node. One application for this could be automatic image captioning where we would like to find the caption for a given image. This works much like PageRank except instead of a random node we jump back to the starting node with probability $1 - c$. Thus, we have

$$p_k = cBp_k + (1-c)e_k.$$

2.7 *Link prediction*

In the link prediction problem, the goal is to infer new interactions or *links* in a given graphs that are likely to occur in the future. This given graph could for instance be a social network graph and these interactions could be friend relations. We would thus be predicting likely friend connections (i.e. recommended friends). To evaluate our measure we choose four timestamps $t_0 < t_0' < t_1 < t_1'$ such that the graph at t_0, t_0' can be used as training data to predict future links and t_1, t_1' can be used as test data to evaluate these predictions.

There are a number of different approaches to link prediction. Typically we rank nodes according to the similarity to a given node x by computing some similarity score $sim(x, y)$. These measures can be based on the graph distance, node neighborhoods or the ensemble of all paths. Higher-level approaches combine different aspects of the above approaches.

Graph distance. The score $sim(x, y)$ is the negated length of the shortest path between x and y. This is a very simple measure.

Common neighbors. The score is computed as the number of common neigbors of x and y so $sim(x, y) = |\Gamma(x) \cap \Gamma(y)|$. Note that a high degree automatically results in a high score with this measure which typically is not what we want.

Jaccard's coefficient. Here the number of common neighbors is normalized for the total number of neighbors. We compute $sim(x, y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}$.

Adamic/Adar. In Adamic/Adar we weigh nodes according to their neighborhood size so nodes with a small degree get a higher score. The similarity measure is computed as

$$sim(x, y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$$

We use log to improve the contribution of low-degree nodes: if x is small then $\frac{1}{\log x} \gg \frac{1}{x}$ and if x is large then $\frac{1}{\log x} \approx \frac{1}{x}$.

Preferential attachment. Preferentially predict links between high degree nodes. We compute $sim(x, y) = |\Gamma(x)| \cdot |\Gamma(y)|$.

Katz proximity. This method is based on the ensemble of all paths. We count the number of all paths from x to y and weigh them according to their length for a constant β as follows:

$$sim(x, y) = \sum_{l=1}^{\infty} \beta^l \cdot |\text{paths}_{x,y}^{<l>}|$$

where $\text{paths}_{x,y}^{<l>}$ is the set of paths from x, y of length l. To compute the Katz proximity we can use the fact that $(M^k)_{ij}$ gives us the number of paths of length k between i, j. Weigh the paths using β like

$$\beta M + \beta M^2 + \beta M^3 + \ldots + \beta M^\infty$$

and then simply compute

$$\begin{aligned} & I + [\beta M + \beta M^2 + \ldots + \beta M^\infty] \\ & = (I - \beta M)^{-1} - I \end{aligned}$$

where I is the unit matrix.

Hitting time/commute time. Use the negative hitting time or commute time as a similarity measure between two points x, y. More precisely, we can compute $sim(x, y)$ as follows:

hitting time	$-H_{x,y}$
hitting time (stationary-normed)	$-H_{x,y} \cdot \pi_y$
commute time	$-(H_{x,y} + H_{y,x})$
commute time (stationary-normed)	$-(H_{x,y} + H_{y,x}) \cdot \pi_y$

where $H_{x,y}$ is the hitting time, i.e. the expected time of a random walk from x to y and pi_y is the stationary distribution weight of y, i.e. the proportion of the time the random walk is at node y.

Rooted PageRank. We adapt PageRank to measure similarity of nodes and define the $sim(x, y)$ to be the stationary probability of y in a random walk that returns to x with probability α at each step, moving to a random neighbor with probability $1 - \alpha$.

SimRank. The SimRank score is the fixed point of the following recursion: two nodes are similar to the extent that they are joined by similar neighbors. Thus we compute

$$sim(x, y) = \begin{cases} 1 & \text{if } x = y \\ \gamma \cdot \frac{\sum_{a \in \Gamma(x)} \sum_{b \in \Gamma(y)} sim(a,b)}{|\Gamma(x)| \cdot |\Gamma(y)|} & \text{otherwise} \end{cases}$$

We can also interpret this as follows: for a random walk on the graph SimRank is the expected value of γ^l where l is a random variable giving the time at which random walks starting from x and y first meet.

Low rank approximation. We can combine previous approaches into a higher level approaches. One such way is to compute a low-rank approximation M_k of the adjacency matrix M as a *noise reduction technique.* Then we can use e.g. Katz measure, common neighbors on M_k.

Unseen bigram. The unseen bigram measure augments the estimated score $sim(x, y)$ using values $sim(z, y)$ for nodes $z \in S_x^{<\delta>}$ similar to x:

$$sim_{\text{weighted}}(x, y) := |\{z : z \in \Gamma(y) \cap S_x^{<\delta>}\}|$$
$$sim_{\text{unweighted}}(x, y) := \sum_{z \in \Gamma(y) \cap S_x^{<\delta>}} sim(x, z)$$

Clustering. Cluster the graph and delete weak edges, then recompute the similarity score $sim(x, y)$ and only new links in the same cluster will be predicted.

When evaluating the above link prediction measures we find that we can group the predictors in two groups which perform similar. Specifically we find that Adamic/Adar, Katz, and low rank inner product (low rank approximation) perform similarly and Jaccard, rooted PageRank and SimRank perform similarly as well. As a result of the small world phenomenon, we find that the graph distance measure does not work very well. Furthermore, as the breadth of data increases (i.e. wider topical focus of dataset) the random predictor worsenes as one would expect.

2.8 Triangle Counting

Given a graph $G = (V, E)$ with $n \times n$ adjacency matrix A we want to count the number of triangles in it. This can be used to e.g. find anomalies in a graph (higher or lower number of triangles than normal behavior). Basic algorithms to accomplish this task do not perform very well. Matrix multiplication needs to compute A^3 which gives a $O(n^3)$ runtime. Fast matrix multiplication is a bit better with $O(n^{2.376})$. Slightly better algorithms are based on listing nodes or edges. The node iterator iterates over all nodes and tests for each pair of if they are connected by an edge. This give a time complexity of

$$\sum_{v \in V} \binom{d(v)}{2} = O(nd_{\text{max}}^2).$$

The edge iterator algorithm similarly iterates over all edges and tests the two adjacent nodes for an edge. This has a time complexity of

$$\sum_{(u,w) \in E} d(u) + d(w) = \sum_{v \in V} d(v)^2$$

A slightly faster variant of the node iterator is the (compact) forward algorithm which utilizes a specific ordering of nodes to get a better time complexity of $\theta(n^{1.5})$ and a space complexity in $\theta(n)$. This time complexity is optimal. However, this algorithm is still slow on large graphs and we would like to find an approximation that allows for faster computation of the number of triangles in G.

Theorem 2.4 (EigenTriangle)**.** *The number of triangles in a graph $G = (V, E)$ with adjacency matrix A is given by its eigenvalues*

$$\Delta(G) = \frac{1}{6} \sum_i \lambda_i^3$$

Proof. The diagonal element α_{ii} of the square matrix A^3 contains the number of paths of lengths 3 that begin and end at the same node i. The only way this can happen is to have a triangle in which node i participates. Therefore the trace of A^3 is three times the total number of triangles (since we are triple counting them because each triangle has 3 participating nodes). Furthermore, since the graph is undirected we are counting each triangle as two (triangle ikj is counted as $i \to k \to j$ and $i \to j \to k$). Therefore the following equality holds: $\Delta(G) = \frac{1}{6}\text{trace}(A^3)$. Furthermore, if λ is an eigenvalue of A then λ^k is an eigenvalue of A^k $(k \geq 1)$. Finally, we know that $\sum_{i=1}^{n} \lambda_i = \text{trace}(A)$. Combining the above equations, we get that $\Delta(G) = \frac{1}{6}\sum_{i=1}^{n} \lambda_i^3$. ■

Theorem 2.5 (EigenTriangleLocal)**.** *The number of triangles in $G = (V, E)$ that node $i \in V$ participates in is given by*

$$\Delta_i(G) = \frac{\sum_j \lambda_j^3 u_{i,j}^2}{2}$$

where $u_{i,j}$ is the i-th entry of the j-th eigenvector.

Proof. Easy extension of 3.1. It follows from the facts that since A_{nxn} is symmetric, $A = U_n \Sigma U_n^{'}$, where Σ is a diagonal matrix with $\text{diag}(\Sigma) = \vec{\Lambda}_n$ (all eigenvalues are real and U_n is an orthogonal matrix and therefore $A^3 = U_n \Sigma^3 U_n^{'}$) and that each triangle is counted twice. ■

Using these theorems we can design an algorithm to approximate the number of triangles in a graph G using the top-k largest eigenvalues of A. We can compute these eigenvalues using Lanczos method. This works because the first few eigenvalues are much stronger than the smaller ones and this effect is further amplified through cubing the eigenvalues. This algorithm is $\geq 1000\times$ faster with $\geq$90% accuracy compared to the node iterator algorithm. The mean required approximation rank (number of eigenvalues) required to achieve $\geq$95% accuracy is 6.2. The speedup over the node iterator algorithm increases as the size of the graph grows.

Another (sampling-based) approach to counting triangles in graphs is *Doulion's algorithm* which constructs a smaller graph G' from G as follows: keep an edge with probability p and discard it with probability $1 - p$. Then count the triangles in G' and multiply the count by $\frac{1}{p^3}$ to get an estimate of the count of triangles in G (since probability of a triangle remaining in G' is p^3 because all three edges have to survive).

3 MapReduce

Many systems need to process large amounts of data and being able to run an algorithm in a distributed fashion can massively speed it up. However, parallel programming is very complex as you need to deal with parallelization, data distribution and handle failures manually. *MapReduce* is an abstraction that allows even people inexperienced with parallel programming to fully utilize the large resources of distributed systems. This abstractions hides the details of the parallelization and takes care of data distribution, load balancing and fault tolerance for the user. MapReduce operates on key-value pairs. The user specifies two functions: a *map* function, which takes an input key-value pair and produces intermediate key-value pairs resulting from some kind of operation on the data, and a *reduce* function which given the intermediate key-value pairs outputs another set of "reduced" key-value pairs (similar to SQL *count(*)*).

3.1 Examples

Let's take a look at some examples of MapReduce applications. Each of these applications can be defined through a map and a reduce function and thus easily utilize a cluster of distributed machines.

Count. We want to count the appearance of different entities, for instance we want to compute a histogram of fruit names. The map function will output (name of fruit, 1) on each machine for its part of the data and the reduce function will *reduce* pairs with the same keys by adding up the values. The result will then be a histogram across the entire dataset.

Distributed grep. Broadcast the supplied pattern to all machines. Map emits a line if it matches the supplied pattern. Reduce is the identity function.

Count of URL access frequency. Analogous to count: Map function outputs (URL, 1) and the reduce function merges same keys to obtain the total count.

Reverse web-link graph. Map outputs a list of (target, source) pairs and reduce function reduces to the same target like (target, list(source)).

Term vector per host. A term vector summarizes the most important words that occur in a document (or in this case in a host). The map function outputs key-value pairs (hostname, term vector) for a given document. Then reduce outputs (hostname, term vector) pairs for each hostname and obtains a list of frequent terms per host.

Inverted index. For each word map outputs a document ID it appears in, so a list of key value pairs (word, document ID). Reduce then reduces the same keys to obtain (word, list(document ID)).

Distributed sort. Map outputs a (key, record) pair and reduce is simply the identity function and outputs the same input.

3.2 Implementation

Worker machines are managed by a master machine which assigns map and reduce tasks to these workers. The master keeps the state (idle, in-progress or completed) and the identity of each worker machine (for non-idle tasks). The master pings every worker periodically and is able to restart tasks (or start backup tasks) in case a worker does not respond or failed. When a MapReduce operation is close to completion the master schedules speculative backup exections of the remaining in-progress tasks. This has been shown to speed up the execution by up to 44%.

The partitioning function decides which output of mappers are assigned to which reducer. The default function for this is $h(k) \bmod R$ for some hash function h, key k and the number of reducers R. Other options are the range partition (i.e. all keys $k < 1000$ to the first reducer etc.) which can be used for sorting, and the identity partition. Within a given partition, the intermediate key/value pairs are processed in increasing key order. Combiner functions allow to combine intermediate key/value pairs before sending them to the reducer to reduce network I/O, i.e. combine all counts of a word to (word, 1000) instead of sending (word, 1) one thousand times.