Node Influence - Origin of the Google Search Engine

Supervisor:
Dr. Tsui-Wei Weng

Posted:
June 9, 2022

Course:
DSC 291 SP'22 Numerical Linear Algebra

Team:

- Zachary Adler
  PID: A15051602
  Computer Science
- Sai Nikhil Alisetti
  PID: A59004736
  Computer Science

Node Influence

The Origin of the Google Search Engine

1. Introduction

Node influence metrics allows for a better understanding of a graph by quantifying how influential each node in the graph is. This leads to the identification of key nodes in the graph to shape future decisions around.

Node influence metrics can be as simple as the the number of edges, or the degree, of a node. While this does give a way to compare the influences of nodes in the graph, there are many more popular node influence metrics used in practice, which are often motivated by some interpretable theory.

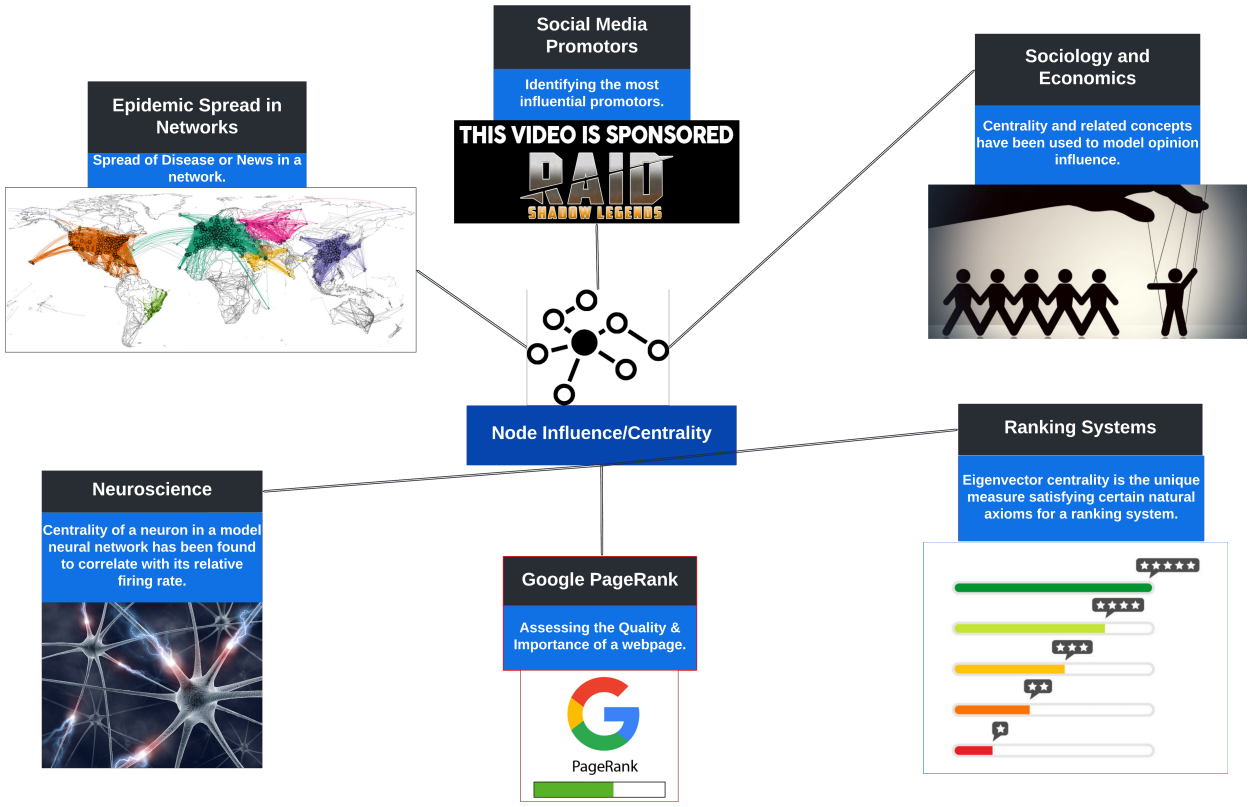
In particular, we will take a look at the theory and practicality of Google's original **PageRank** algorithm, as well as compare it to state of the art node influence algorithms.

1.1 Importance of Node Influence Metrics:

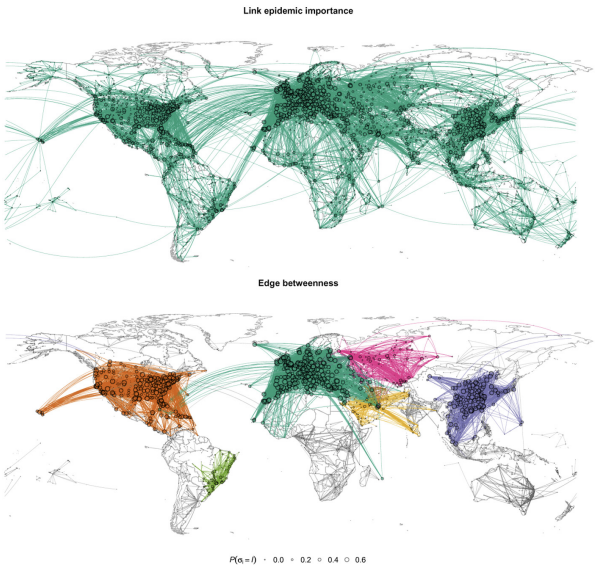
Obtaining influential nodes in a graph and understanding the intuition behind node influence metrics helps in modeling features that account to the steady state of the model and more importantly to estimate conditional quality.

1.2 Applications

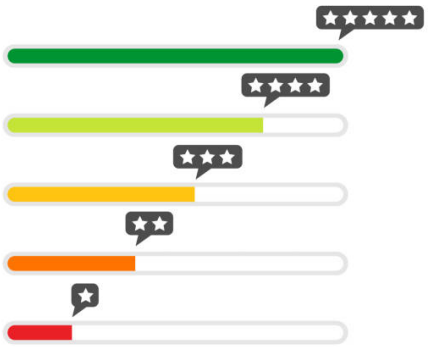
In real world, the concept of node influence and centrality is actually used in a wide range of fields and applications.



- For instance in **Neuroscience**, node influence of a neuron correlates with its rate of relative firing.
- In **Economics & Sociology** it is used to model opinion influence in social groups.
- In **Social media** it is used by businesses to identify the most influential promoters for their marketing.
- In **Search Engine Optimization** to estimate and obtain relative Page Ranking.
- In **Epidemiology** to model the epidemic spread in a network to further perform spectral graph analysis on it.



However It is actually primarily used in **Ranking Systems** or to assess scores of an entity satisfying certain natural axioms/features.

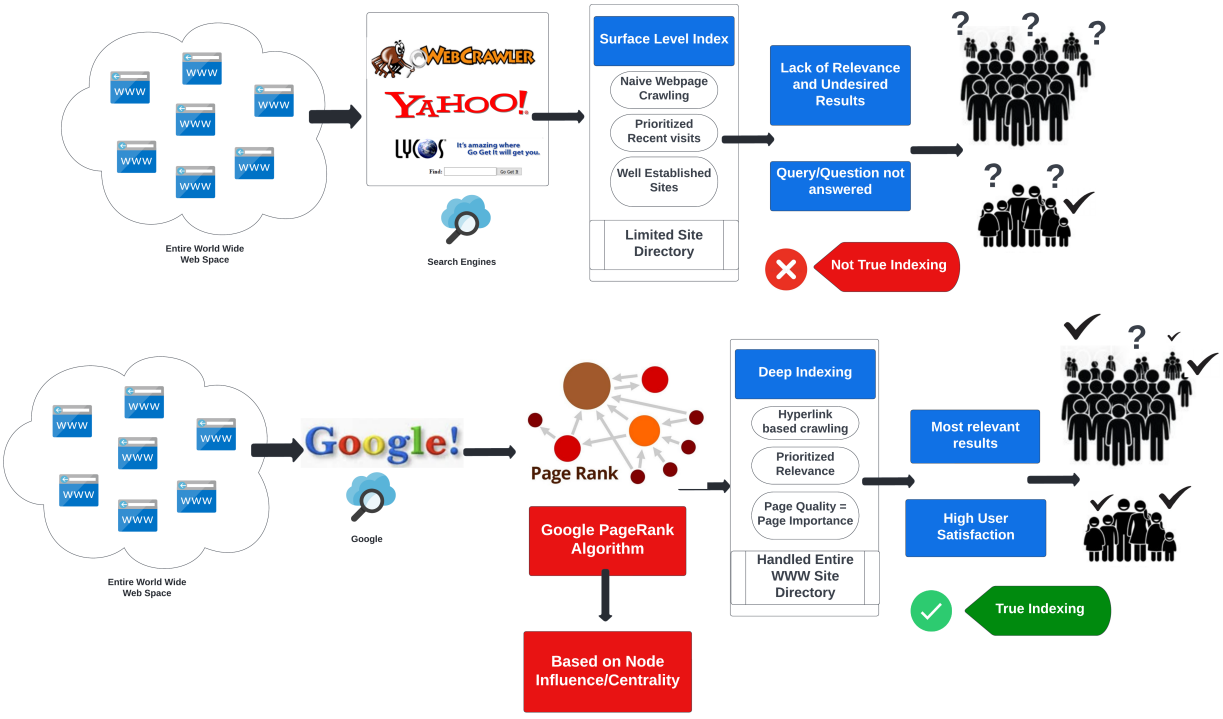


1.3 Google Search Engine and Node Influence - History

Typically a search engine aims to index the web so as to allow us to navigate through the chaotic inter-web with some semblance of order. Prior to google, the search engines usually indexed the web using only the context within the webpage. This naive web crawling approach didn't take into account the relevance of the results or page quality. They primarily incentivized results over most recent visits, established sites and business contract advertisement deals. Having a limited site directory exacerbates the issue at hand and allows room for erroneous data leakage. Earlier search engines deterministically didn’t achieve the goal of Indexing the entire web. This basically left the community with dissatisfied suboptimal results at best, leave with more questions than answers or expend more time to obtain a desired result by hit and trial of keywords.

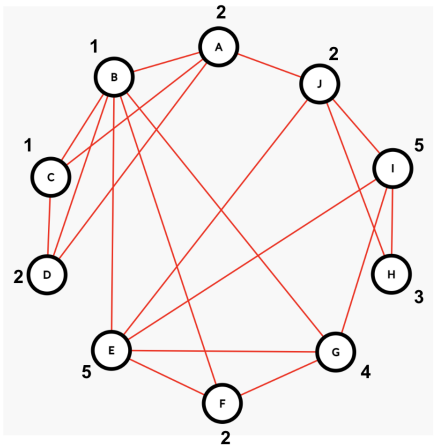
Google tackled this issue with a goal of indexing the entire web but also emphasizing on the relevance of the results. In order to do this they defined the concept of page quality and how to determine page importance using

hyperlink based crawling on top of the standard webpage crawling. They formalized this approach in their 1998 Stanford paper where they established the PageRank Algorithm -> which ranks pages in the entire network based on their node influence/centrality which serves as a pre-requisite for google to further process these results and prioritize per user relevance of the results.



2. Problem Formulation

The goal of node influence metrics are to label each node in a graph with a value of relative influence so as to make decisions based on the most influential nodes. Creating efficient, robust, and interpretable node influence metrics are some of the goals to keep in mind.

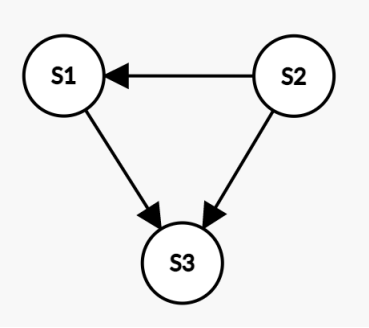


2.1 Relation to numerical linear algebra:

A graph has matrix representations which invite the use of powerful linear algebra techniques. The adjacency matrix, Laplacian, and random walk transition matrix are all ways of viewing a graph in matrix form. The random walk transition matrix, which we denote as , is of particular interest. It is the matrix where T_{ij} is the probability of moving from node j to node i under the context of a random walk, e.g. we travel along an outgoing edge from node j chosen uniformly at random. T is obtained by transposing the adjacency matrix A, dividing each column by that column's sum since each edge has an equal probability of being chosen, and replacing diagonal entries of 0 columns with 1 to define terminal state behavior.

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{transpose}} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \xrightarrow{\text{divide by column sums}} \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \\ 1 & \frac{1}{2} & 0 \end{bmatrix} \xrightarrow{\text{fix diagonal}} \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \\ 1 & \frac{1}{2} & 1 \end{bmatrix} = T$$

T determines the random walk probabilities of the graph corresponding to adjacency matrix A:



From S_1 we take the edge to S_3 with probability 1. From S_2 we take the edge to S_1 with probability $\frac{1}{2}$ and the edge to S_3 with probability $\frac{1}{2}$. Finally from S_3 we stay at S_3 with probability 1 since there are no outgoing edges.

This transition matrix can be related through its eigenvectors to a node influence metric, as we will see in the approach.

2.2 Approach description:

Let $v_t = (p_1, p_2, ..., p_n)$ where p_i is the probability that a random walk is at node i at time step t. If v_t converges to a limiting distribution, i.e. $\lim_{n \to \infty} v_t = v^*$, then v^* is a node influence metric that describes how likely a random walk is to be at each node after many steps. Thus we are interested in finding v^*.

Firstly, the transition matrix T is such that $v_{t+1} = Tv_t$ [insert reference]. That is, multiplying the probability vector by the transition matrix updates the probabilities to 1 step further. We illustrate with the previous example:

$$Tv_0 = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \\ 1 & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 1/6 \\ 0 \\ 5/6 \end{bmatrix} = v_1$$

So if we start at a random node then take one step of a random walk we end up in S_1 with probability $\frac{1}{6}$ and S_3 with probability $\frac{5}{6}$.

Note that $v_{t+1} = Tv_t$ is a recurrence relation which gives $v_t = T^t v_0$. Hence, if we want to find the limiting distribution of a random walk, we may multiply an initial guess vector by T many times until convergence. This is exactly the power method used to find the dominant eigenvector of T. Therefore one may alternatively compute the dominant eigenvector of T to use as the node influence metric.

Google's original search engine utilized **PageRank**, an algorithm that finds influential webpages in a graph connected by edges that represent hyperlinks in nearly the same way. Since hyperlinks are directed however, a random walk may often get stuck in smaller subsets of the graph with no hope of returning, reducing the efficacy of the algorithm. To fix this, the transition matrix is changed such that there is some probability that a step in the random walk goes to one of all the nodes chosen uniformly at random, rather than taking a random edge like normal. That is:

$$P_\alpha = (1-\alpha)T + \alpha\frac{1}{n}J$$

J is the matrix of all 1, so dividing it by n gives the transition matrix that has an equal probability of going to any node from any node. The notation used here is adopted from [7].

So we see the PageRank matrix P_α defines the transitions by taking a random walk step with probability $1-\alpha$ and going to a random node with probability α.

Finally the dominant eigenvector of P_α is found, e.g. through power iteration, and used as the node influence metric.

---

3. State-of-the-art

3.1 SOTA Approach description:

Google employees like John Mueller of SEO in 2020 say that PageRank is still used today in tandem with other indicators. These may include other SOTA approaches like Graph Neural Networks (GNNs).

For our experiment we focus on the SOTA approach proposed by Zhao et al. [2] in 2021, which is Motif-based PageRank or MPR. MPR is PageRank but with a different transition matrix:

$$(P_{M_k})_{ij} = (H_{M_k})_{ij} / \Sigma_j (H_{M_k})_{ij}$$

Here **H** is a linear combination of matrices that depend on the motif M_k, and some example calculations are shown below for the first 7 motifs.

Motif	Matrix Computation	$\mathbf{W}_{M_i} =$
M_1	$\mathbf{C} = (\mathbf{U} \cdot \mathbf{U}) \odot \mathbf{U}^T$	$\mathbf{C} + \mathbf{C}^T$
M_2	$\mathbf{C} = (\mathbf{B} \cdot \mathbf{U}) \odot \mathbf{U}^T + (\mathbf{U} \cdot \mathbf{B}) \odot \mathbf{U}^T + (\mathbf{U} \cdot \mathbf{U}) \odot \mathbf{B}$	$\mathbf{C} + \mathbf{C}^T$
M_3	$\mathbf{C} = (\mathbf{B} \cdot \mathbf{B}) \odot \mathbf{U} + (\mathbf{B} \cdot \mathbf{U}) \odot \mathbf{B} + (\mathbf{U} \cdot \mathbf{B}) \odot \mathbf{B}$	$\mathbf{C} + \mathbf{C}^T$
M_4	$\mathbf{C} = (\mathbf{B} \cdot \mathbf{B}) \odot \mathbf{B}$	$\mathbf{C}$
M_5	$\mathbf{C} = (\mathbf{U} \cdot \mathbf{U}) \odot \mathbf{U} + (\mathbf{U} \cdot \mathbf{U}^T) \odot \mathbf{U} + (\mathbf{U}^T \cdot \mathbf{U}) \odot \mathbf{U}$	$\mathbf{C} + \mathbf{C}^T$
M_6	$\mathbf{C} = (\mathbf{U} \cdot \mathbf{B}) \odot \mathbf{U} + (\mathbf{B} \cdot \mathbf{U}^T) \odot \mathbf{U}^T + (\mathbf{U}^T \cdot \mathbf{U}) \odot \mathbf{B}$	$\mathbf{C}$
M_7	$\mathbf{C} = (\mathbf{U}^T \cdot \mathbf{B}) \odot \mathbf{U}^T + (\mathbf{B} \cdot \mathbf{U}) \odot \mathbf{U} + (\mathbf{U} \cdot \mathbf{U}^T) \odot \mathbf{B}$	$\mathbf{C}$

U is the adjacency matrix of the graph consisting of uni-directional edges, B is the adjacency matrix of the graph consisting of bi-directional edges, and ⊙ is element-wise matrix multiplication.

3.2 Why SOTA?

Motifs encapsulate higher-order structures of the graph and reward strong structures like triangles. This improves upon PageRank which just views direct relations between nodes through edges, hence adapting more information contained in the graph into the resulting node influences.

PageRank is also susceptible to “gaming the system” in that one may purchase dummy websites to link to their website and improve its centrality. MPR deals with this in that the extraneous data added will typically not be part of a higher-order structure, and thus not have as much impact as in PageRank. So we expect MPR to be more robust.

---

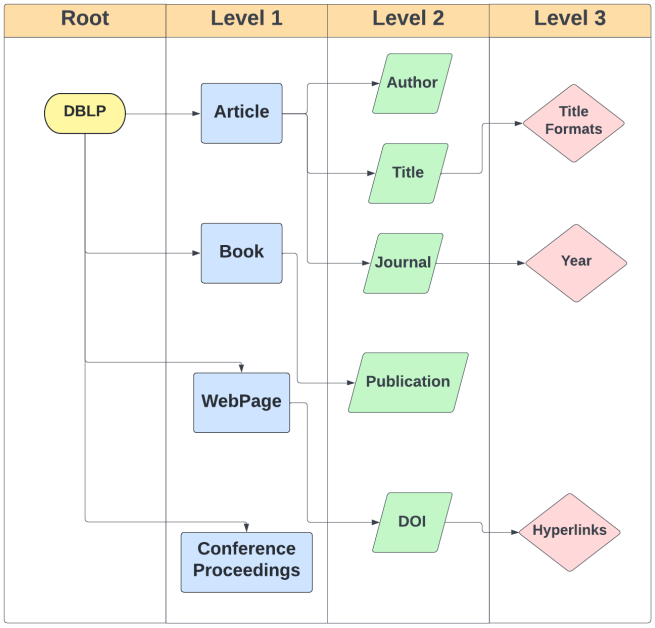
4. Experimental Setup:

Since we cannot test our algorithms over the entire internet space or even a subset of the internet as ignoring nodes would not produce scalable node influence results, we simulate our results over the DBLP dataset which is the dataset consisting of research articles as nodes and edges corresponding citations of that paper.

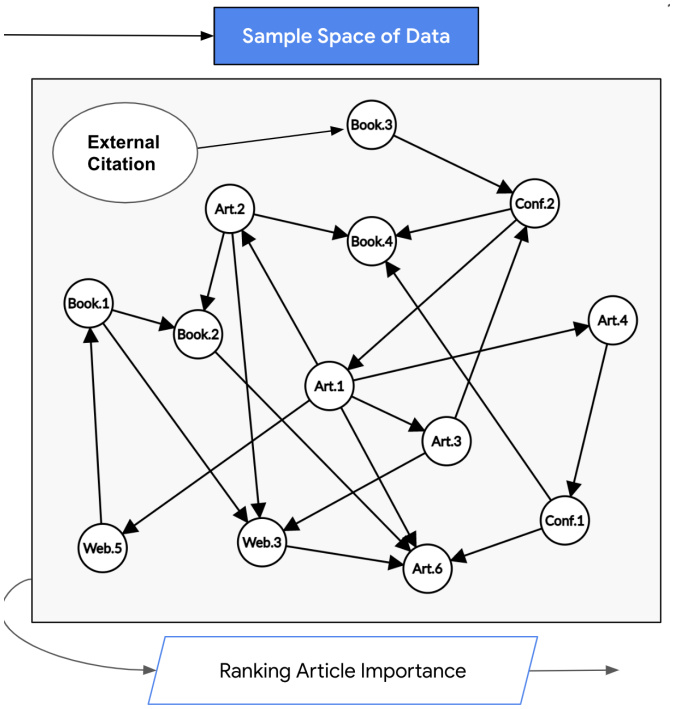
4.1 Dataset

The *dblp computer science bibliography* is a searchable database of bibliographic information on key computer science papers. It has grown from a modest experimental web server to a well-known open-data service for the whole computer science community.

dblp indexes about 4.4 million papers written by over 2.2 million authors as of January 2019. To that purpose, dblp indexes over 40,000 journal volumes, 39,000 conference and workshop proceedings, and 80,000 monographs.



The dblp dataset structure follows 3 levels [10] where the Level 1 distinguishes between the main type of the research publication (article, book, conf, etc...), Level 2 consists of tag data attributes pertaining to paper type in Level 1, and Level 3 contains alternative and extra information pertaining to Level 2.

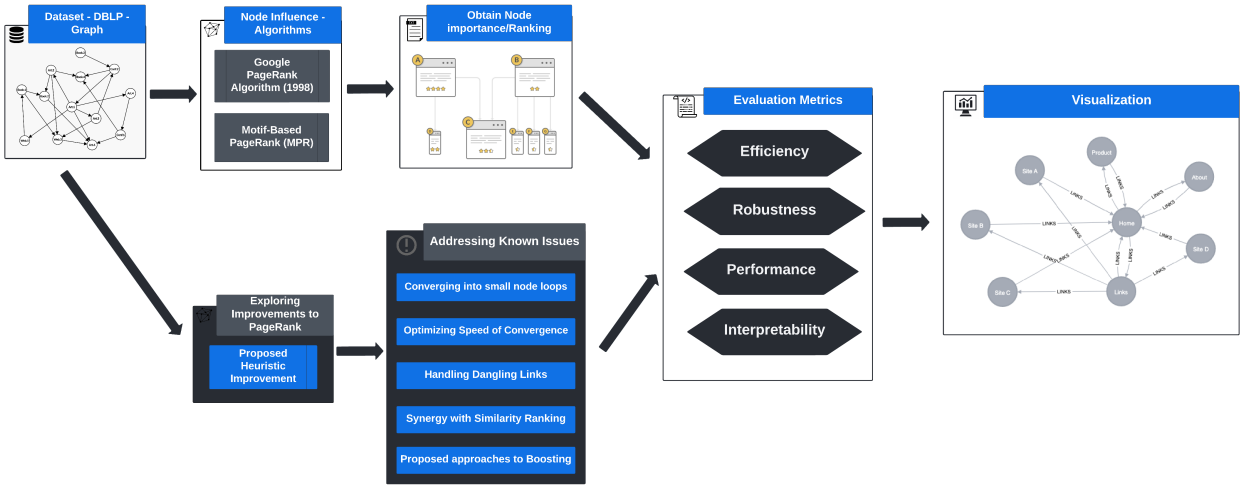


Above is a visualization of what data in DBLP may look like in graph format. The nodes are papers and they are connected by edges that represent citations.

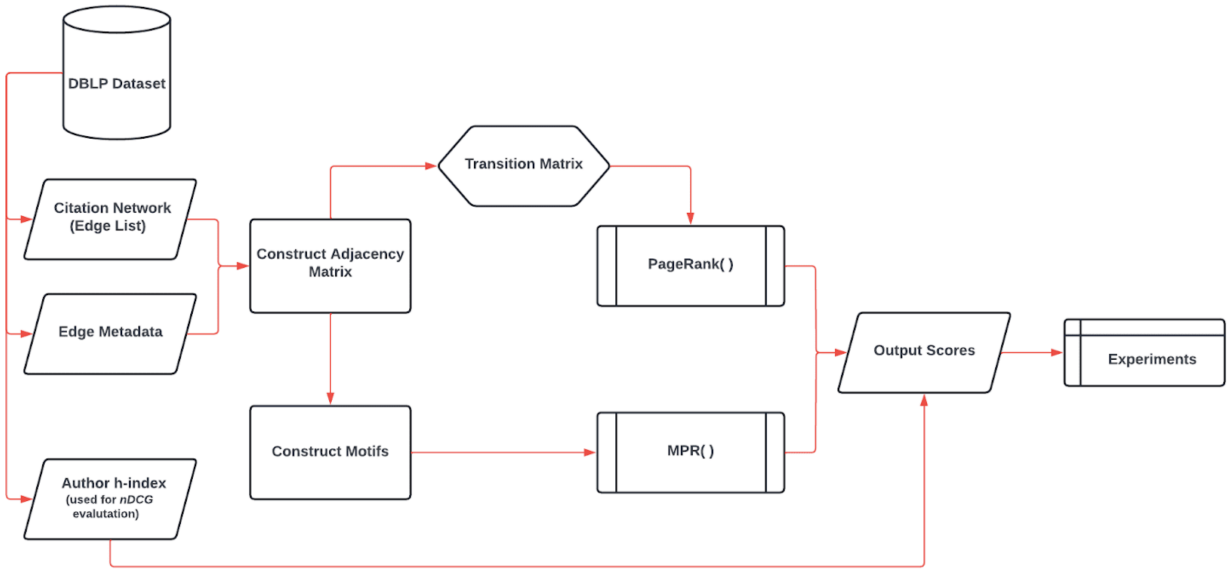
4.2 Experimental Setup Architecture

The primary goal of our experimental setup is to compare the original google page rank algorithm to the Motif-based PageRank algorithm.

We provide the same unaltered data to both algorithms, do the necessary preprocessing, obtain our node influence ranking and then we evaluate both these algorithms against our evaluation metrics. Post evaluation we visualize our results for comparison between each other and also to account for interpretability.



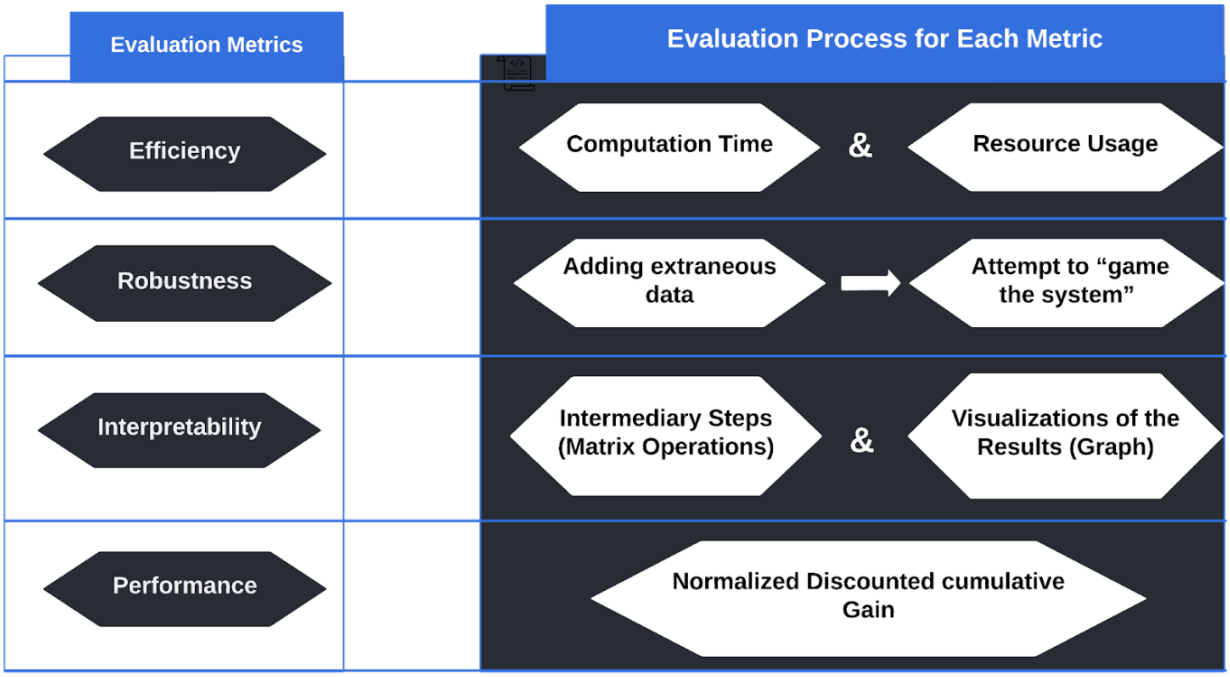
4.3 Experimental Flow



4.4 Evaluation Metrics

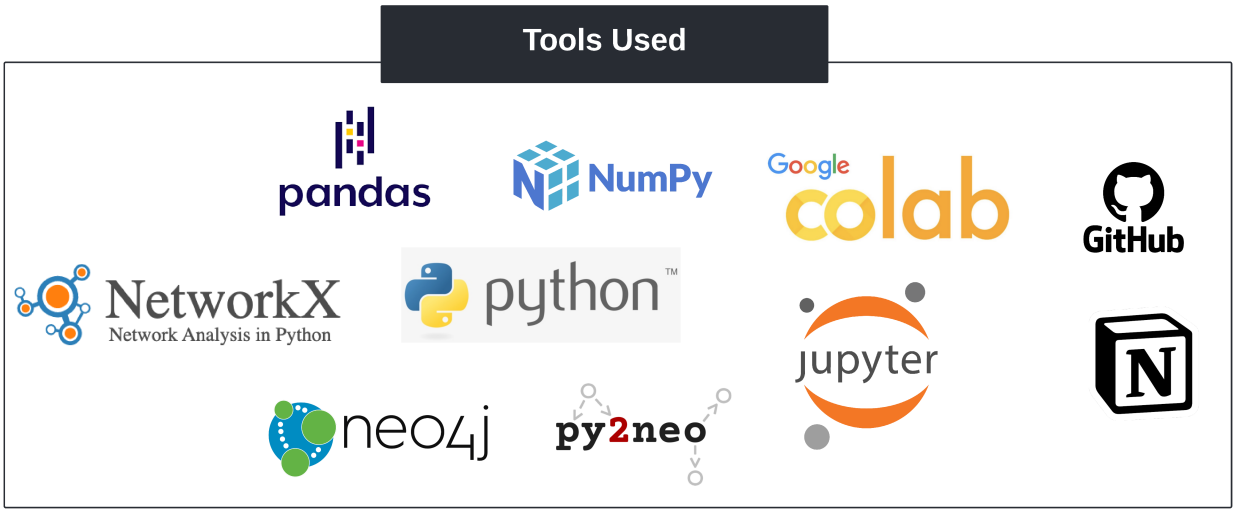
We evaluate efficiency through computation time and resource usage. Robustness is tested by adding extraneous data and seeing whether the results change significantly. For example, if a person in a social network buys bot accounts to subscribe to them then they shouldn't be perceived as more influential in a robust approach. We will determine interpretability through visualizations of the results. Lastly we evaluate performance through calculating

the Normalized Discounted cumulative Gain, or NDCG, which is a popular metric for measuring information retrieval.



4.5 Tools

- We use Python 3.9 as our base kernel.
- python_networkX library to obtain test performance metrics of our networked graphs
- Pandas for data handling.
- Numpy for Linear Algebra.
- Scipy for working with sparse matrices | Linear Algebra
- scikit-network for visualization
- IPykernel, IPywidgets for activating jupyter magics
- psutil, tqdm, multiprocessing for Evaluating Efficiency and memory profiling
- py2neo for porting Neo4j to python
- Google colab cloud VM instance Environment for our Docker Container
- Jupyter as our base code presentation IDE
- GitHub for version control
- Notion for Documentation



5. Results

Now we examine the implementation code.

First import the linear algebra libraries.

```
import numpy as np
from scipy.sparse import csr_matrix
import scipy.sparse as sps
from scipy.sparse import diags
```

Then import the libraries for visualization.

```
from IPython.display import SVG
import sknetwork as skn
from sknetwork.visualization import svg_digraph
import time
```

Next, we must define key functions. First, how to get the Edge List from the citation_network file from DBLP.

```
def edge_list_from_network(network_file):
    f = open(network_file, 'r')
    content = map(lambda x: x[:-1].split(';'), f.readlines())
    cite_net_edges = list(map(lambda x: (int(x[0]), int(x[0])), content))
    f.close()
    return cite_net_edges
```

Then we can use the edge_list to construct the Adjacency Matrix in a CSR - Condensed Sparsed Rows Matrix of type float64.

```
def get_adjacency_matrix(edgeList):
    # edge_list will be a list of tuples of form (src id, dst id)
    main_graph = skn.data.from_edge_list(edge_list=edgeList, directed=True)
    # create the sparse adjacency matrix using the edge data
    adjacency_matrix = main_graph.adjacency.astype(np.float64)
    node_names = main_graph.names
    return adjacency_matrix, node_names
```

Next we follow the transformations described earlier to construct the Random Walk Transition matrix from the Adjacency matrix.

```
def get_transition_matrix(adjMat):
    for number in range(0, np.size(adjMat, 0)):
        row_sum = adjMat.getrow(number).sum()
        if row_sum == 0:
            continue
        else:
            for number_col in adjMat.indices[adjMat.indptr[number]:adjMat.indptr[number+1]]:
                adjMat[number, number_col] = adjMat[number, number_col] / row_sum
    return adjMat.T
```

For the Original PageRank algorithm: Adapted from Zhao et al. [2]

```
MAX_TIME = 30000
# PageRank Algorithm
def pageRank(p, m, v):
    e = np.ones((m.shape[0], 1))
    n = m.shape[0]
    count = 0
    while count <= MAX_TIME:
        v = p * m.dot(v) + ((1 - p) / n) * e
        count = count + 1
    return v

# Initializing PageRank Scores
def firstPr(transitionMatrix):
    pr = np.zeros((transitionMatrix.shape[0], 1), dtype=float)
    for i in range(transitionMatrix.shape[0]):
        pr[i] = float(1) / transitionMatrix.shape[0]
    return pr

#Computing PageRank Scores
def compute_pagerank(adjacency_matrix, alpha):
    T = get_transition_matrix(adjacency_matrix)
    pr = firstPr(T)
    p = 1 - alpha
    return pageRank(p, T, pr)
```

The following is to define motifs M_1 to M_7 as obtained through preliminary transformations of the Adjacency Matrix.

```
def motif_types(adjacency_matrix, motif_order, motif_type):
    if motif_type == 'M1':
```

```
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            U_tran = np.transpose(U_matrix)
            result_1 = U_matrix.dot(U_matrix)
            result_1 = result_1.multiply(U_tran)
            adjacency_matrix = result_1
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
            adjacency_tran = np.transpose(adjacency_matrix)
            adjacency_matrix = adjacency_matrix + adjacency_tran
    elif motif_type == 'M2':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            U_tran = np.transpose(U_matrix)
            result_1 = B_matrix_spare.dot(U_matrix)
            result_1 = result_1.multiply(U_tran)
            result_2 = U_matrix.dot(B_matrix_spare)
            result_2 = result_2.multiply(U_tran)
            result_3 = U_matrix.dot(U_matrix)
            result_3 = result_3.multiply(B_matrix_spare)
            adjacency_matrix = result_1 + result_2
            adjacency_matrix = adjacency_matrix + result_3
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
            adjacency_tran = np.transpose(adjacency_matrix)
            adjacency_matrix = adjacency_matrix + adjacency_tran
    elif motif_type == 'M3':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            result_1 = B_matrix_spare.dot(B_matrix_spare)
            result_1 = result_1.multiply(U_matrix)
            result_2 = B_matrix_spare.dot(U_matrix)
            result_2 = result_2.multiply(B_matrix_spare)
            result_3 = U_matrix.dot(B_matrix_spare)
            result_3 = result_3.multiply(B_matrix_spare)
            adjacency_matrix = result_1 + result_2
            adjacency_matrix = adjacency_matrix + result_3
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
            adjacency_tran = np.transpose(adjacency_matrix)
            adjacency_matrix = adjacency_matrix + adjacency_tran
    elif motif_type == 'M4':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            result_1 = B_matrix_spare.dot(B_matrix_spare)
            result_1 = result_1.multiply(B_matrix_spare)
            adjacency_matrix = result_1
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
    elif motif_type == 'M5':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            U_tran = np.transpose(U_matrix)
            result_1 = U_matrix.dot(U_matrix)
            result_1 = result_1.multiply(U_matrix)
            result_2 = U_matrix.dot(U_tran)
            result_2 = result_2.multiply(U_matrix)
            result_3 = U_tran.dot(U_matrix)
            result_3 = result_3.multiply(U_matrix)
            adjacency_matrix = result_1 + result_2
            adjacency_matrix = adjacency_matrix + result_3
```

```
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
            adjacency_tran = np.transpose(adjacency_matrix)
            adjacency_matrix = adjacency_matrix + adjacency_tran
    elif motif_type == 'M6':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            U_tran = np.transpose(U_matrix)
            result_1 = U_matrix.dot(B_matrix_spare)
            result_1 = result_1.multiply(U_matrix)
            result_2 = B_matrix_spare.dot(U_tran)
            result_2 = result_2.multiply(U_tran)
            result_3 = U_tran.dot(U_matrix)
            result_3 = result_3.multiply(B_matrix_spare)
            adjacency_matrix = result_1 + result_2
            adjacency_matrix = adjacency_matrix + result_3
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
    elif motif_type == 'M7':
        for i in range(motif_order):
            adjacency_tran = np.transpose(adjacency_matrix)
            B_matrix_spare = adjacency_matrix.multiply(adjacency_tran)
            U_matrix = adjacency_matrix - B_matrix_spare
            U_matrix = np.abs(U_matrix)
            U_tran = np.transpose(U_matrix)
            result_1 = U_tran.dot(B_matrix_spare)
            result_1 = result_1.multiply(U_tran)
            result_2 = B_matrix_spare.dot(U_matrix)
            result_2 = result_2.multiply(U_matrix)
            result_3 = U_matrix.dot(U_tran)
            result_3 = result_3.multiply(B_matrix_spare)
            adjacency_matrix = result_1 + result_2
            adjacency_matrix = adjacency_matrix + result_3
            adjacency_matrix = normal_matrix_new(adjacency_matrix)
    return adjacency_matrix

def normal_matrix_new(a):
    # For symmetric matrices, we can use matrix multiplication to normalize, which is faster.
    d = a.sum(0)
    d = np.array(d)
    d = d[0]
    dd = list(map(lambda x: 0 if x==0 else np.power(x, -0.5), d))
    D_matrix = diags(dd, 0)
    C = D_matrix.dot(a)
    C = C.dot(D_matrix)
    a = C
    return a
```

Next we can construct the motifs which is needed for MPR.

```
def construct_motif(adjacency_matrix, motif_order, motif_type, alpha):
    data_array = adjacency_matrix.data
    # adjacency_matrix.data is mainly to be able to build the matrix in the binary case.
    adjacency_matrix.data = np.ones((1, data_array.shape[0]), dtype=np.float64)[0]
    result_B = adjacency_matrix.copy()
    result_B = row_normal_matrix(result_B)
    # result_B is row_normalized adjacency_matrix
    result_C = motif_types(adjacency_matrix, motif_order, motif_type)
    # result_C represents the normalized motif matrix
    # The following uses the parameter alpha to fuse the two matrices
    result_temp1 = result_B.multiply(alpha).tolil()
    result_temp2 = result_C.multiply(1-alpha).tolil()
    result_D = result_temp1 + result_temp2
    MPR_adj = result_D.tocsr()
```

```
    return MPR_adj

def row_normal_matrix(mat):
    for number in range(0, np.size(mat, 0)):
        row_sum = mat.getrow(number).sum()
        if row_sum == 0:
            continue
        else:
            for number_col in mat.indices[mat.indptr[number]:mat.indptr[number+1]]:
                mat[number, number_col] = mat[number, number_col] / row_sum
    return mat
```

Lastly, the MPR algorithm to call is:

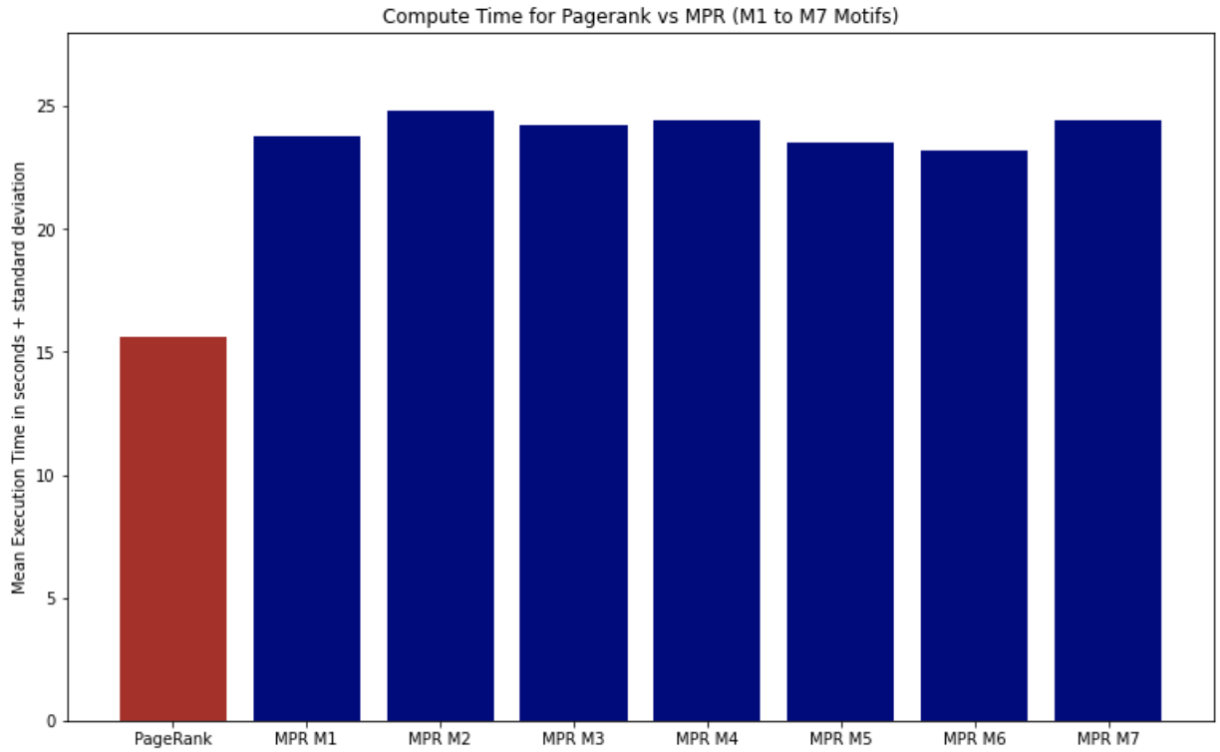
```
def MPR(adjacency_matrix, motif_order, motif_type, alpha):
    mpr_adjacency_matrix = construct_motif(adjacency_matrix, motif_order, motif_type, alpha)
    return compute_pagerank(mpr_adjacency_matrix, alpha)
```

5.1. Evaluating Efficiency

Evaluating Efficiency for α = 0.1

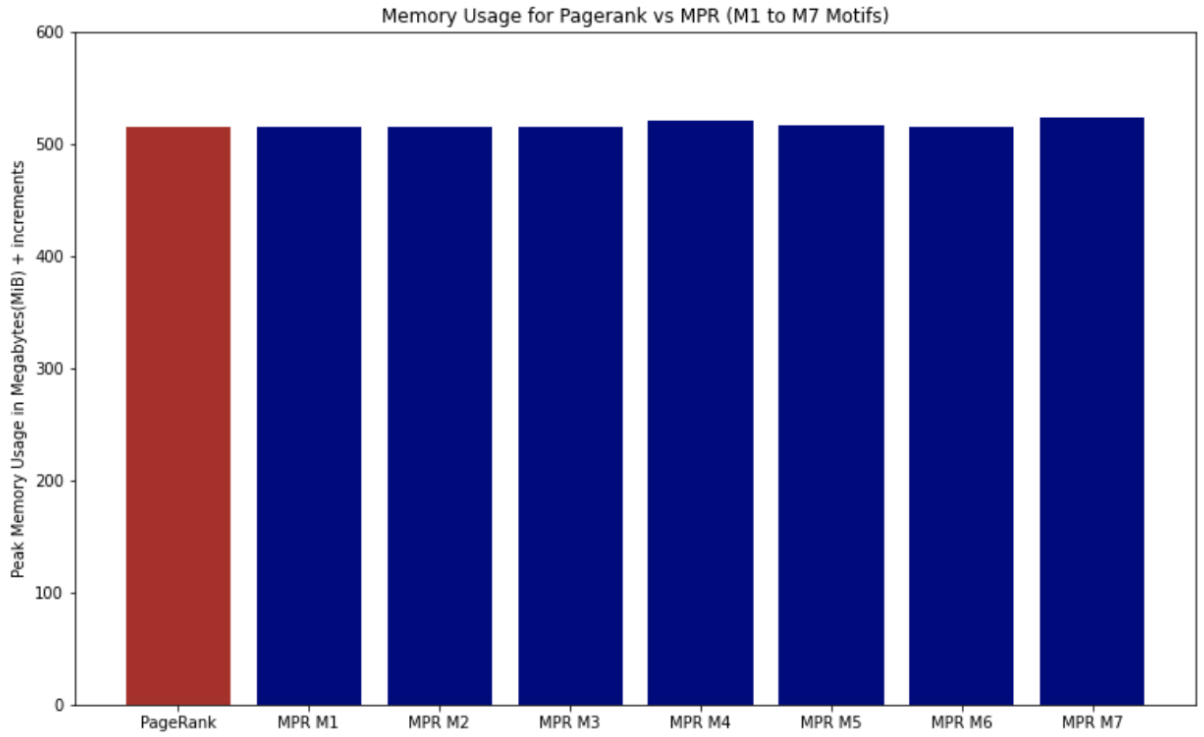
Profiling Compute Time using cell magics

```
%%timeit
cite_net_adjacency_matrix, cite_net_node_names = get_adjacency_matrix(cite_net_edges)
pagerank_original_result = compute_pagerank(cite_net_adjacency_matrix, 0.1)
```



Profiling Memory usage using cell magics

```
%%memit
cite_net_adjacency_matrix, cite_net_node_names = get_adjacency_matrix(cite_net_edges)
pagerank_original_result = compute_pagerank(cite_net_adjacency_matrix, 0.1)
```



Parallel thread profiling CPU and Memory usage using psutil

5.2. Evaluating Robustness

To test robustness we create modified graphs with extraneous nodes and edges.

```
import random

# code adapted from __main__ code in pagerank_motif_direct to instead keep a set of all node ids in the graph
```

```
# network_file: network file path, e.g. 'data/DBLP/citation_network.txt'
def get_ids_and_copy(network_file, out_file):
    ids = set()
    f = open(network_file)
    o = open(out_file, 'w')
    edge_count = 0
    while True:
        line = f.readline()
        o.write(line)
        if line:
            edge_count += 1
            line = line.replace('\n', '')
            line = line.split(';')
            for i in range(len(line)):
                ids.add(int(line[i]))
        else:
            break
    return list(ids), edge_count

def append_extraneous_links(ids, edge_count, out_file, seed_val):
    n = len(ids)
    num_extra = int(0.01 * edge_count)
    id_counter = max(ids)
    out = open(out_file, 'a')
    random.seed(seed_val)
    for i in range(num_extra):
        id_counter += 1
        dst_ind = random.randint(0, n - 1) # choose a random node's index to point to
        # point new dummy citation to the random destination
        line = "%d;%d\n" % (id_counter, ids[dst_ind])
        out.write(line)


# given a network, fill a new network with extra links
def build_modified_network(network_file, out_file, seed_val):
    ids, edge_count = get_ids_and_copy(network_file, out_file)
    append_extraneous_links(ids, edge_count, out_file, seed_val)

if __name__ == "__main__":
    for i in range(30):
        build_modified_network('data/DBLP/citation_network.txt', "data/DBLP/citation_network_modified_%d.txt" % (i), i)
```

Then we can test robustness by comparing PR and MPR on the original citation network to PR and MPR on the modified networks.

```
def compare(networks_list_range, motif, alpha, out_file):
    start = time.time()

    f = open(out_file, 'w')

    network_file = 'data/DBLP/citation_network.txt'
    cite_net_edges = edge_list_from_network(network_file)
    cite_net_adjacency_matrix, cite_net_node_names = get_adjacency_matrix(cite_net_edges)

    PR_old = compute_pagerank(cite_net_adjacency_matrix, alpha)
    MPR_old = MPR(cite_net_adjacency_matrix, 1, motif, alpha)

    for i in networks_list_range:
        network_file_modified = "data/DBLP/citation_network_modified_%d.txt" % (i)
        cite_net_edges_modified = edge_list_from_network(network_file_modified)
        cite_net_adjacency_matrix_modified, cite_net_node_names_modified = get_adjacency_matrix(cite_net_edges_modified)

        PR_new = compute_pagerank(cite_net_adjacency_matrix_modified, alpha)
        MPR_new = MPR(cite_net_adjacency_matrix_modified, 1, motif, alpha)
```

```
        error_PR = np.sum(np.absolute(PR_old - PR_new[:len(PR_old)]))
        error_MPR = np.sum(np.absolute(MPR_old - MPR_new[:len(MPR_old)]))

        f.write("%f;%f\n" % (error_PR, error_MPR))
        print("---compare has taken %f seconds" % (time.time() - start))

if __name__ == '__main__':
    start = time.time()

    motif = 'M7'

    alpha = 0.1
    compare(range(5), motif, alpha, "output/DBLP/%s_alpha%s_errors.txt" % (motif,alpha))
    print("main has taken %f seconds" %(time.time() - start))

    alpha = 0.3
    compare(range(5, 10), motif, alpha, "output/DBLP/%s_alpha%s_errors.txt" % (motif,alpha))
    print("main has taken %f seconds" %(time.time() - start))

    alpha = 0.5
    compare(range(10,15), motif, alpha, "output/DBLP/%s_alpha%s_errors.txt" % (motif,alpha))
    print("main has taken %f seconds" %(time.time() - start))
```

```
For α =  0.1
---comparing modified network 0 has taken 97.861194 seconds
---comparing modified network 1 has taken 154.213963 seconds
---comparing modified network 2 has taken 211.232779 seconds
---comparing modified network 3 has taken 272.446150 seconds
---comparing modified network 4 has taken 327.504060 seconds
Comparing for α = 0.100000  has taken 327.747131 seconds
For α =  0.3
---comparing modified network 5 has taken 94.502688 seconds
---comparing modified network 6 has taken 151.825834 seconds
---comparing modified network 7 has taken 208.500685 seconds
---comparing modified network 8 has taken 264.169082 seconds
---comparing modified network 9 has taken 320.799031 seconds
Comparing for α = 0.300000  has taken 648.815226 seconds
For α =  0.5
---comparing modified network 10 has taken 96.592407 seconds
---comparing modified network 11 has taken 154.380440 seconds
---comparing modified network 12 has taken 211.893048 seconds
---comparing modified network 13 has taken 269.811929 seconds
---comparing modified network 14 has taken 327.910916 seconds
Comparing for α = 0.500000  has taken 976.914358 seconds
```

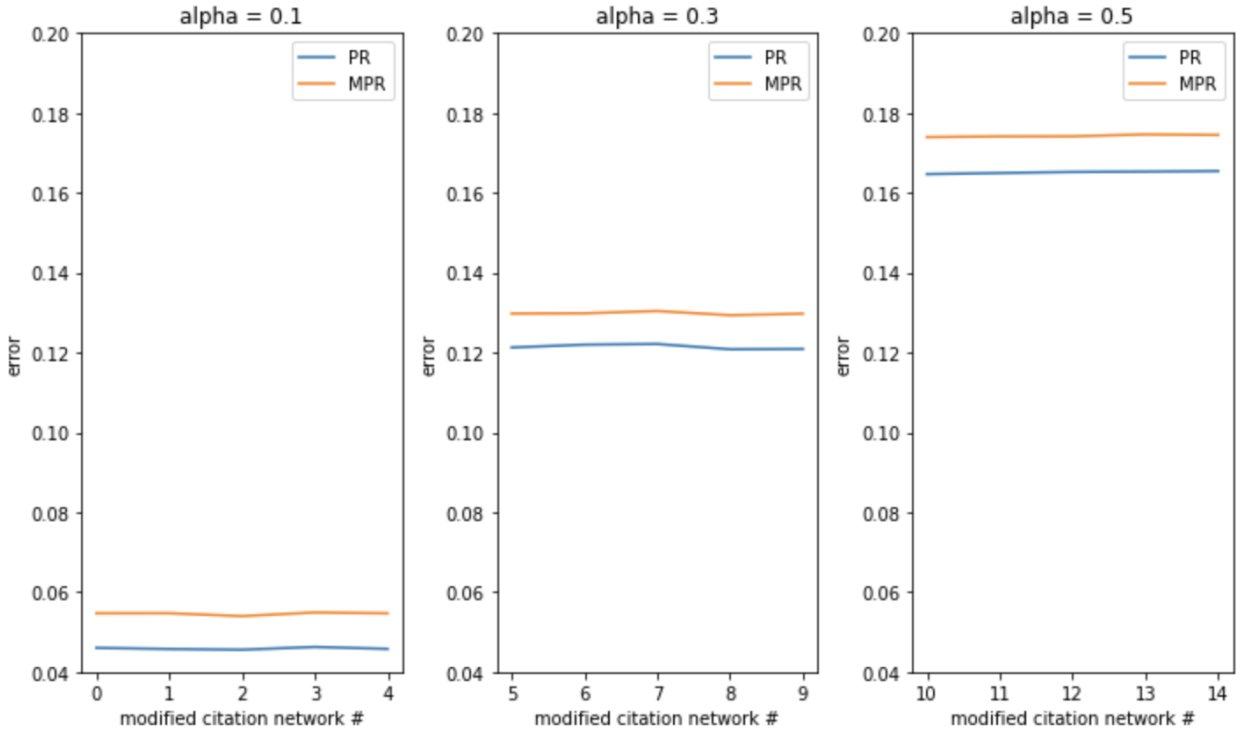
Next we can plot the errors we find from our robustness tests.

```
from matplotlib import pyplot as plt
```

```
if __name__ == '__main__':

    f = open("output/DBLP/M7_alpha0.1_errors.txt")
    pr_1 = []
    mpr_1 = []
    while True:
        line = f.readline()
        if line:
            line = line.strip()
            line = line.split(';')
            pr_1.append(float(line[0]))
            mpr_1.append(float(line[1]))
        else:
            break
    f.close()

    f = open("output/DBLP/M7_alpha0.3_errors.txt")
    pr_3 = []
    mpr_3 = []
    while True:
        line = f.readline()
        if line:
            line = line.strip()
            line = line.split(';')
            pr_3.append(float(line[0]))
            mpr_3.append(float(line[1]))
        else:
            break
    f.close()

    f = open("output/DBLP/M7_alpha0.5_errors.txt")
    pr_5 = []
    mpr_5 = []
    while True:
        line = f.readline()
        if line:
            line = line.strip()
            line = line.split(';')
            pr_5.append(float(line[0]))
            mpr_5.append(float(line[1]))
        else:
            break
    f.close()

    fig, axes = plt.subplots(nrows = 1, ncols = 3, figsize = (10,6))

    axes[0].plot(range(5), pr_1, label = "PR")
    axes[0].plot(range(5), mpr_1, label = "MPR")
    axes[0].set_ylim(0.04, 0.2)
    axes[0].set_xlabel("modified citation network #")
    axes[0].set_ylabel("error")
    axes[0].legend()
    axes[0].set_title("alpha = 0.1")

    axes[1].plot(range(5,10), pr_3, label = "PR")
    axes[1].plot(range(5,10), mpr_3, label = "MPR")
    axes[1].set_ylim(0.04, 0.2)
    axes[1].set_xlabel("modified citation network #")
    axes[1].set_ylabel("error")
    axes[1].legend()
    axes[1].set_title("alpha = 0.3")

    axes[2].plot(range(10,15), pr_5, label = "PR")
    axes[2].plot(range(10,15), mpr_5, label = "MPR")
    axes[2].set_ylim(0.04, 0.2)
    axes[2].set_xlabel("modified citation network #")
    axes[2].set_ylabel("error")
    axes[2].legend()
    axes[2].set_title("alpha = 0.5")
```

```
    fig.tight_layout()
    plt.show()
```

Here are the results of the above code.



5.3. Evaluating Performance

Lastly, we find the Normalized Discounted Cumulative Gain of PageRank and MPR to test performance. Below are some functions we need to read the files and construct data structures that will be used to find the NDCG. The code for computing NDCG is adapted from Zhao et al. [2].

```
import numpy as np
import math

def read_top(txt_name, dict, K):
    f = open(txt_name)
    count = 0
    result_array = []
    name_array = []
    while True:
        line = f.readline()
        if count >= K:
            break
        if line:
            line = line.strip()
            line = line.split(';')
            name = line[1]
            score = dict[name]
            if score < 30:
                continue
```

```
            result_array.append(score)
            name_array.append(name)
            count += 1
        else:
            break
    return result_array, name_array


def construct(txt_name):
    f = open(txt_name)
    user = {}
    count = 0
    while True:
        line = f.readline()
        if line:
            line = line.strip()
            line = line.split(';')
            name = line[0]
            score = int(line[1])
            user[name] = score
            count += 1
        else:
            break
    return user


def get_ndcg(result1, result2):
    sum1 = 0
    sum2 = 0
    for i in range(len(result1)):
        sum1 += result1[i] * 1.0 / (np.log2(i+2))
        sum2 += result2[i] * 1.0 / (np.log2(i+2))
    return sum2 * 1.0 / sum1
```

Then we can compute the NDCGs for different K values. E.g. for $K = 50$, the top 50 results are used to compute the NDCG.

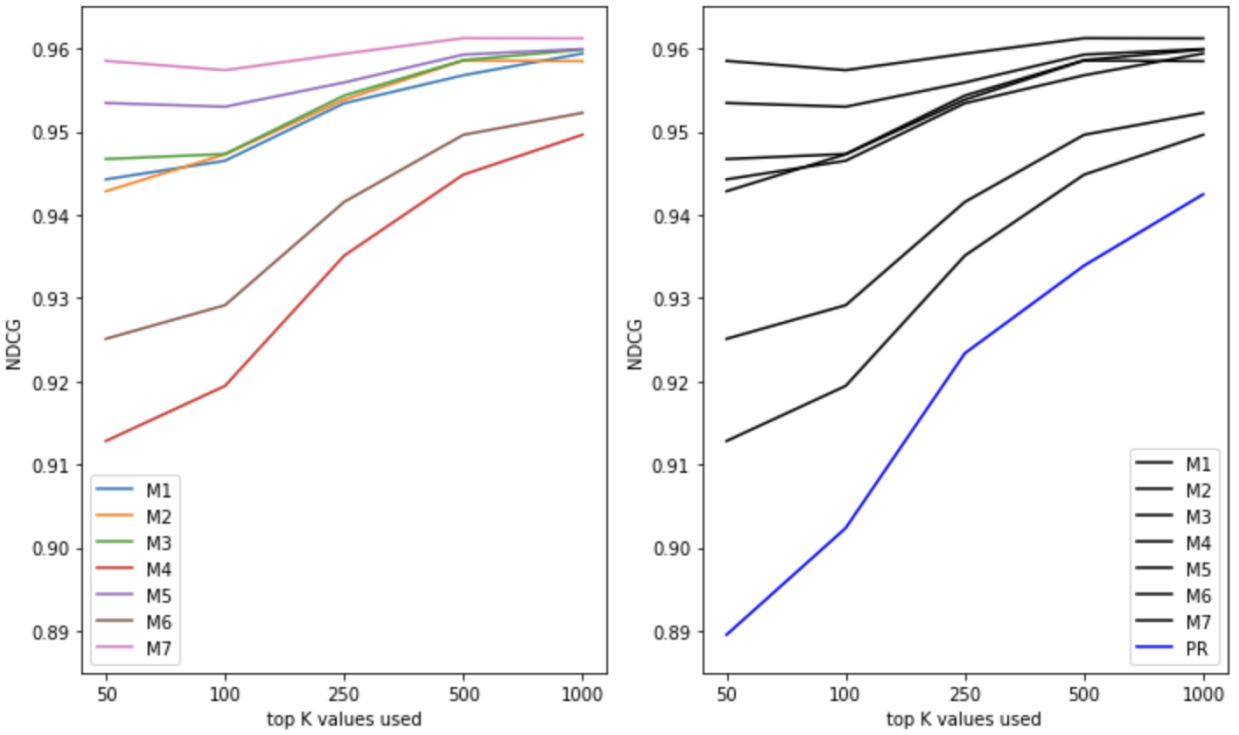
```
def compute_ndcg(results_file, K):
    txt_name3 = 'data/DBLP/h_index_all.txt'
    dict_author = construct(txt_name3)
    txt_name = results_file
    rank_value = K
    rank_array, name_array = read_top(txt_name, dict_author, rank_value)
    result = {}
    for i in range(len(rank_array)):
        score = dict_author[name_array[i]]
        result[name_array[i]] = score
    result_array = []
    dict = sorted(result.items(), key=lambda item: item[1], reverse=True)
    for i in range(len(dict)):
        result_array.append(dict[i][1])

    score = get_ndcg(result_array, rank_array)
    return score

if __name__ == '__main__':
    # for top 10, top 50, top 250, and top 500
    K_list = [50,100,250,500,1000]
    out = open("output/DBLP/ndcg_PR.txt", 'w')

    for K in K_list:
        score = compute_ndcg("output/DBLP/result_PR.txt", K)
        out.write("%d;%f\n" % (K,score))
    out.close()
```

```
    for i in range(1,8):
        motif = "M%d" % (i)
        out = open("output/DBLP/ndcg_%s.txt" % (motif), 'w')
        for K in K_list:
            score = compute_ndcg("output/DBLP/result_%s.txt" % (motif), K)
            out.write("%d;%f\n" % (K,score))
        out.close()
```

Then we can plot the NDCG results.

```
from matplotlib import pyplot as plt

if __name__ == '__main__':
    f = open("output/DBLP/ndcg_PR.txt")

    K_list = []
    ndcg_PR = []
    while True:
        line = f.readline()
        if line:
            line = line.strip()
            line = line.split(';')
            K = line[0]
            val = float(line[1])

            K_list.append(K)
            ndcg_PR.append(val)
        else:
            break
    f.close()

    ndcg_M = []
    for i in range(1,8):
        ndcg_M_i = []
        f = open("output/DBLP/ndcg_M%s.txt" %(i))
        while True:
            line = f.readline()
            if line:
                line = line.strip()
                line = line.split(';')
                K = line[0]
                val = float(line[1])

                ndcg_M_i.append(val)
            else:
                break
        f.close()
        ndcg_M.append(ndcg_M_i)

    fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (10,6))

    for i in range(1,8):
        axes[0].plot(K_list, ndcg_M[i-1], label = "M%s" % (i))

    axes[0].set_ylim(0.885, 0.965)
    axes[0].set_xlabel("top K values used")
    axes[0].set_ylabel("NDCG")
    axes[0].legend()

    for i in range(1,8):
        axes[1].plot(K_list, ndcg_M[i-1], label = "M%s" % (i), color = 'black')
    axes[1].plot(K_list, ndcg_PR, label = "PR", color = 'blue')
    axes[1].set_ylim(0.885, 0.965)
    axes[1].set_xlabel("top K values used")
```

```
    axes[1].set_ylabel("NDCG")
    axes[1].legend()

    fig.tight_layout()
    plt.show()
```



6. Conclusion

Efficiency

We got the expected results where MPR takes longer to compute that PageRank but both MPR and PageRank use nearly identical memory space. These results come about because MPR does the same thing PageRank does except it spends a longer time computing motifs to establish a more technical transition matrix, and MPR uses the same size of transition matrix as in PageRank (with matrix operations being done in place) so the memory cost is effectively the same.

Performance

We found that MPR outperforms PageRank under all motifs from M_1 to M_7 with respect to Normalized Discounted Cumulative Gain. This held for multiple different values for the top K results used in the computation. This is expected, since MPR takes advantage of more information from the graph to make better rankings.

Interpretability

Interpretability is subjective, but we feel that PageRank is more interpretable since the random-walk transition matrix is easily grasped due to its relation to simulating taking steps of a random walk in the graph. MPR similarly

takes transitions with its own transition matrix, but the methods used to compute the motifs are more complicated than the simple transformations used in PageRank. That being said, the relation between MPR and its goal of taking advantage of higher-order structures in the graph, like cycles, helps cement the intuition behind MPR and improves its interpretability.

Robustness

Contrary to our hopes that MPR would be robust to gaming through adding extraneous links, MPR performed slightly worse than PageRank in this area.

Instead, robustness error depended much more on the value of α, our random surfer probability. The dependence on alpha is somewhat expected because the greater α is the greater the probability of transitioning to the added dummy nodes and losing value on actual important nodes. However, the dependence was greater than what we expected and outweighed the difference made by MPR taking advantage of higher-order structures in the graph that extraneous links shouldn't be a part of.

We also suspect that this may be due to the way DBLP is structured as compared to regular links on the internet, as there is a higher change of bi-directional edges between nodes (Authors citing each other, which is common in research articles) and so the higher order features end up overfitting and hence might lead to a slightly lower accuracy, however this is just an interpreted guess rather than a formal proof.

This is an area for further research, where other tests of robustness may reveal more information about Motif-based PageRank.

GitHub Link

https://github.com/alsozatch/PageRank-vs-MPR

---

6. References

[1] Dehmer, Matthias, ed. Structural analysis of complex networks. Springer Science & Business Media, 2010.

[2] H. Zhao, X. Xu, Y. Song, D. L. Lee, Z. Chen and H. Gao, "Ranking Users in Social Networks with Motif-Based PageRank," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 5, pp. 2179-2192, 1 May 2021, doi: 10.1109/TKDE.2019.2953264.

[3] Brin, Sergey, and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." Computer networks and ISDN systems 30.1-7 (1998): 107-117.

[4] Page, Lawrence, et al. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab, 1999.

[5] Altman, Alon; Tennenholtz, Moshe (2005). *Ranking systems*. New York, New York, USA: ACM Press.

[6] Palacios-Huerta, Ignacio; Volij, Oscar (2004). "The Measurement of Intellectual Influence" (PDF). *Econometrica*. The Econometric Society. 72 (3): 963–977.

[7] Martin, Kimball (2014). Graph Theory and Social Networks.

[8] https://www.dalaric.com/how-did-google-win-the-battle-of-the-search-engines/

[9] https://www.forbes.com/sites/forbesagencycouncil/2017/06/05/how-google-came-to-dominate-search-and-what-the-future-holds/

[10] The dblp team: dblp computer science bibliography. Monthly snapshot release of November 2019. https://dblp.org/xml/release/dblp-2022-05-02.xml.gz