

# **Data Mining – Project 3**

## *Link Analysis Report*

**Student ID:** P76124786

**Student Name:** 徐向廷

**Department:** CSIE

### **Increasing Hub & Authority (Node #1)**

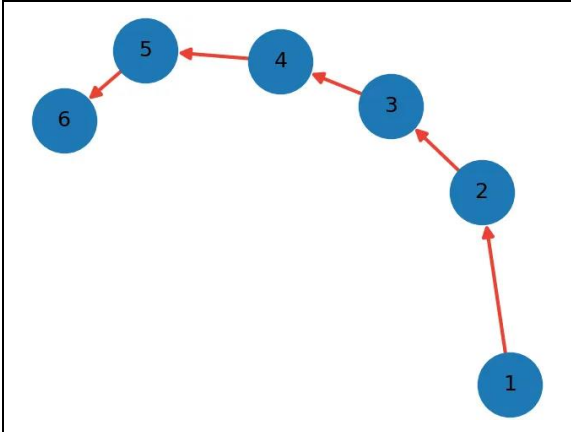
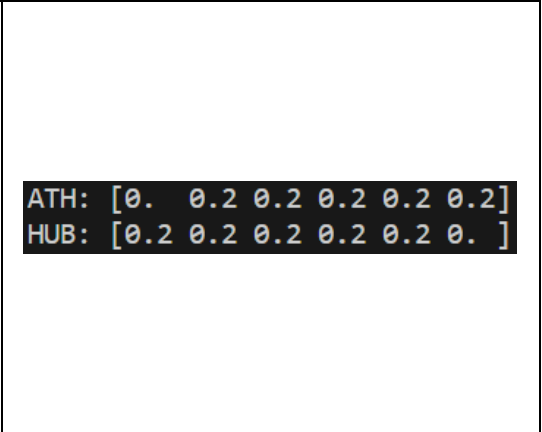
According to the HITS algorithm, “hub” is a measure of how effectively a node connects to other nodes in the network, while "authority" gauges how effectively it is connected to by other nodes in the network. The following equations are used to calculate the hub and authority of nodes through the HITS algorithm.

$$Hub_t(Node_i) = \sum_{\omega \in Childs(Node_i)} Authority_{t-1}(Node_{\omega})$$

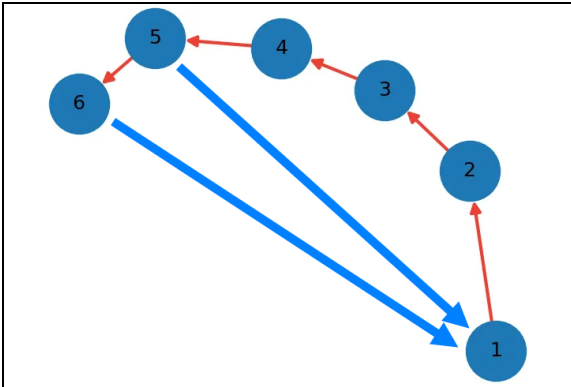
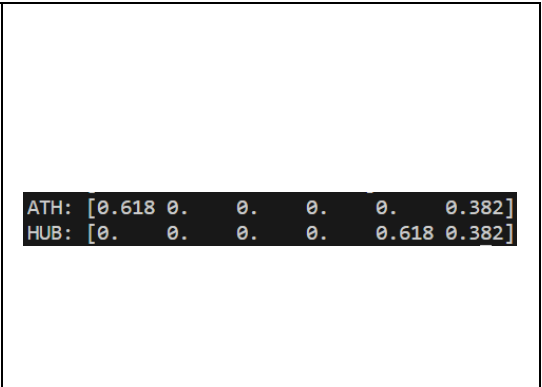
$$Authority_t(Node_i) = \sum_{\omega \in Parents(Node_i)} Hub_{t-1}(Node_{\omega})$$

To significantly boost the hub of node 1, it is crucial to focus on its children nodes with elevated authority values or augment its connectivity by introducing more outgoing edges to other nodes. Regarding the authority of node 1, emphasis should be placed on considering the parent

nodes, achieved by incorporating additional incoming edges leading to node 1. The following images are the visualization of graph 1 and the screenshot of the initial values of authority and hub.

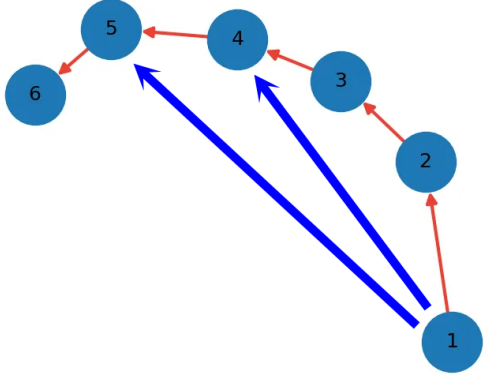
	
<b>Image 1.</b> Graph-1 Visualization	<b>Image 2.</b> Graph-1 Hub/Auth (Init)

As evident from the images provided, node 1 currently lacks incoming edges, resulting in an authority value of 0. To enhance its authority value, we should implement the strategy of adding more incoming edges to node 1. The subsequent pair of images illustrate the added connections and their respective authority values.

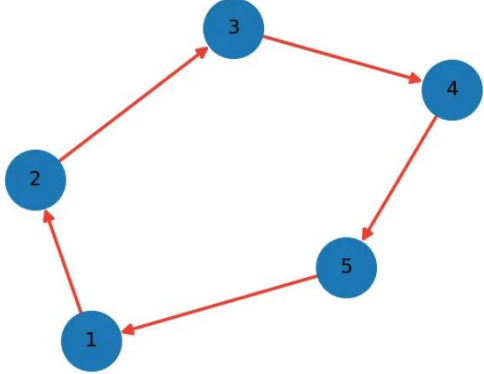
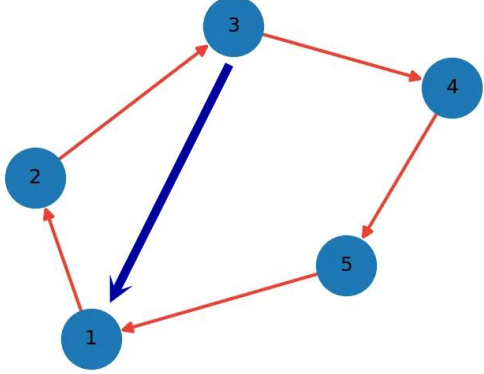
	
<b>Image 3.</b> Graph-1 Vis. (Variant #1)	<b>Image 4.</b> Graph-1 (Var. #1) (Auth ↑)

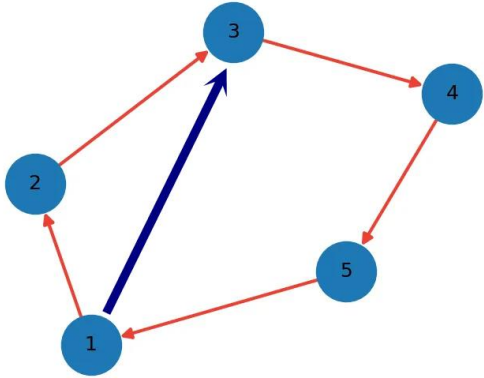
Clearly, the authority value has risen from 0 to 0.618, validating our hypothesis. Let's proceed by introducing more outgoing edges to node 1

in an attempt to elevate its hub score. The ensuing images depict the outcomes of this endeavor.

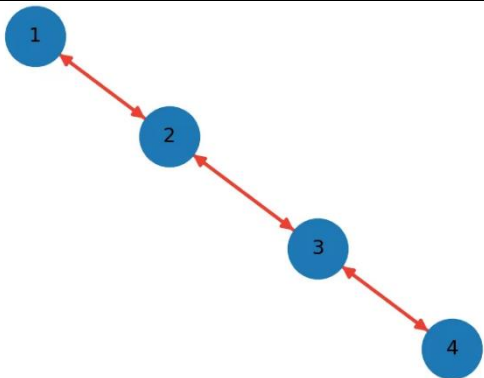
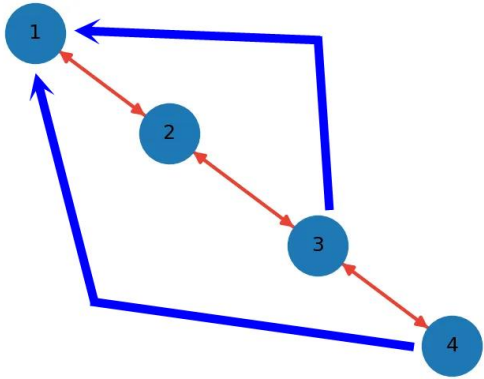
	<pre>ATH: [0.    0.268 0.    0.366 0.366 0. ] HUB: [0.577 0.    0.211 0.211 0.    0. ]</pre>
Image 5. Graph-1 Vis. (Variant #2)	Image 6. Graph-1 (Var. #2) (Hub ↑)

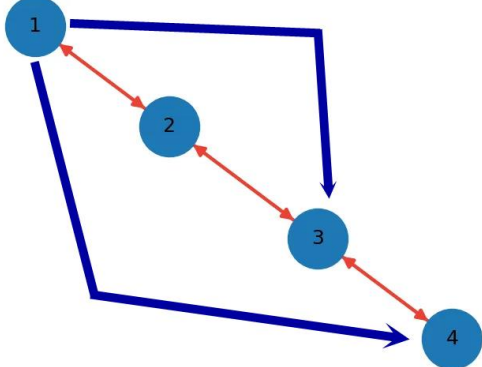
After introducing several outgoing edges to node 1, we noticed a notable increase in its hub value. Let's now delve into the impact that this modification has on Graph 2.

	<pre>ATH: [0.2 0.2 0.2 0.2 0.2] HUB: [0.2 0.2 0.2 0.2 0.2]</pre>
Image 7. Graph-2 Visualization	Image 8. Graph-2 Hub/Auth (Init)
	<pre>ATH: [0.618 0.    0.    0.382 0. ] HUB: [0.    0.    0.618 0.    0.382]</pre>
Image 9. Graph-2 Vis. (Variant #1)	Image 10. Graph-2 (Var. #1) (Auth ↑)

	<div> ATH: <math>\begin{bmatrix} 0. &amp; 0.382 &amp; 0.618 &amp; 0. &amp; 0. \end{bmatrix}</math>  HUB: <math>\begin{bmatrix} 0.618 &amp; 0.382 &amp; 0. &amp; 0. &amp; 0. \end{bmatrix}</math> </div>
<b>Image 11.</b> Graph-2 Vis. (Variant #2)	<b>Image 12.</b> Graph-2 (Var. #2) (Hub ↑ )

In the six images (7-12) provided above, it is evident that augmenting outgoing edges to node 1 leads to an increase in its hub score, while adding incoming edges results in an elevation of its authority score. Let's now apply the same modification to Graph 3 and analyze its effects.

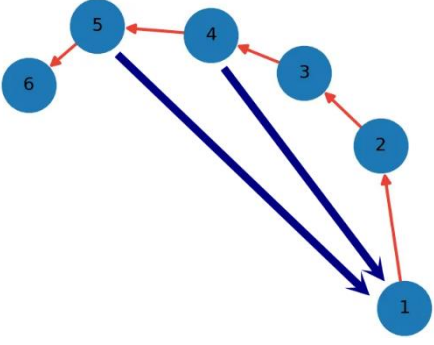
	<div> ATH: <math>\begin{bmatrix} 0.191 &amp; 0.309 &amp; 0.309 &amp; 0.191 \end{bmatrix}</math>  HUB: <math>\begin{bmatrix} 0.191 &amp; 0.309 &amp; 0.309 &amp; 0.191 \end{bmatrix}</math> </div>
<b>Image 13.</b> Graph-3 Visualization	<b>Image 14.</b> Graph-3 Hub/Auth (Init)
	<div> ATH: <math>\begin{bmatrix} 0.41 &amp; 0.18 &amp; 0.266 &amp; 0.144 \end{bmatrix}</math>  HUB: <math>\begin{bmatrix} 0.079 &amp; 0.298 &amp; 0.324 &amp; 0.298 \end{bmatrix}</math> </div>
<b>Image 15.</b> Graph-3 Vis. (Variant #1)	<b>Image 16.</b> Graph-3 (Var. #1) (Auth ↑ )

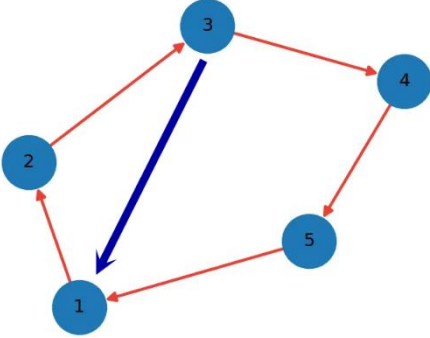
	<div data-bbox="834 371 1331 443" style="background-color: black; color: white; padding: 5px;"> ATH: [0.079 0.298 0.324 0.298]  HUB: [0.41 0.18 0.266 0.144] </div>
<b>Image 17.</b> Graph-3 Vis. (Variant #2)	<b>Image 18.</b> Graph-3 (Var. #2) (Hub ↑ )

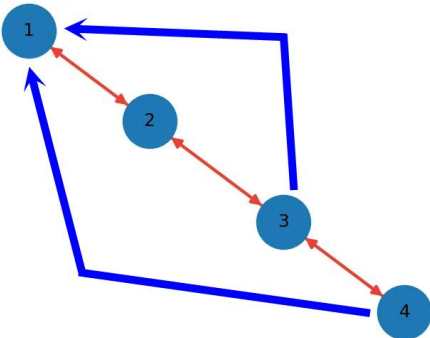
### Increasing PageRank Score (Node #1)

According to the PageRank algorithm, optimizing the inheritance of a higher proportion of the parent's PageRank score involves reducing the number of sibling nodes. Given that both Graph 1 and 2 lack nodes with more than one outgoing edge, a potential strategy is to introduce extra incoming edges from other nodes. The following image illustrates the PageRank equation within the context of a random surfer model.

$$PR_t(Node_i) = \frac{d}{n} + (1 - d) \times \sum_{\omega \in \text{Parents}(Node_i)} \frac{PR_{t-1}(Node_{\omega})}{|Childs(Node_{\omega})|}$$

	<div data-bbox="740 1715 1334 1749" style="background-color: black; color: white; padding: 5px;"> PR (Before): [0.05608622 0.10656382 0.15199366 0.19288053 0.22967869 0.26279706]  PR (After): [0.18131432 0.19268648 0.20296805 0.21227631 0.12503207 0.08572279] </div>
<b>Image 19.</b> Graph-1 Vis. (Var. #3)	<b>Image 20.</b> Graph-1 (Var. #3) (PR ↑ )

	<pre>PR (Before): [0.2 0.2 0.2 0.2 0.2] PR (After): [0.22761787 0.24475871 0.26020145 0.15695642 0.11046558]</pre>
<b>Image 21.</b> Graph-2 Vis. (Var. #1)	<b>Image 22.</b> Graph-2 (Var. #1) (PR ↑ )

	<pre>PR (Before): [0.17241378 0.32758617 0.32758617 0.17241378] PR (After): [0.30343905 0.3681219 0.23341393 0.09502512]</pre>
<b>Image 23.</b> Graph-3 Vis. (Var. #1)	<b>Image 24.</b> Graph-3 (Var. #1) (PR ↑ )

### Algorithm Description (HITS)

In the HITS algorithm, we commence by initializing hub and authority vectors with ones, as per the specifications provided by the TA. Later, over 30 iterations (timesteps), we iteratively accumulate new hub and authority values through calculations and update the vectors at the end of each timestep. The computation of hub values involves summing the authority values of all child vectors associated with a particular vector. Conversely, authority values are determined based on the hub values of a vector's parent vectors.

A technique known as early stopping can be employed in this process. As it has been proven to converge, we calculate the total difference between

timesteps. If the difference (delta) is found to be less than a specified tolerance range (epsilon), we terminate the iteration.

```
def hits_algorithm(adjacency_matrix : np.ndarray, *,
                  epsilon : Optional[ float ] = 0.01,
                  max_iterations : Optional[ int ] = -1) -> Tuple[ np.ndarray, np.ndarray ]:
    # obtaining number of vertices
    num_vertices = len(adjacency_matrix)
    # initialize hub vector with ones
    hubness = np.ones(shape = (num_vertices, ), dtype = np.float32)
    # initialize authority vector with ones
    authority = np.ones(shape = (num_vertices, ), dtype = np.float32)
    # execute for `max_iterations` number of time steps
    while (max_iterations != 0):
        # initialize hub (t+1)
        hubness_tmp = np.zeros_like(hubness)
        # initialize authority (t+1)
        authority_tmp = np.zeros_like(authority)
        # iterate through each vertex
        for vertex in range(num_vertices):
            # calculate vertex hub score with children authority scores
            hubness_tmp[vertex] += np.matmul(authority, adjacency_matrix[vertex].T)
            # calculate vertex authority score with parent hub scores
            authority_tmp[vertex] += np.matmul(hubness, adjacency_matrix[... , vertex].T)
        # normalize authority
        authority_tmp = authority_tmp / np.sum(authority_tmp)
        # normalize hub
        hubness_tmp = hubness_tmp / np.sum(hubness_tmp)

        # calculate delta, hub & authority difference between timesteps
        delta = (
            np.sum(np.sqrt(np.square(hubness_tmp - hubness))) +
            np.sum(np.sqrt(np.square(authority_tmp - authority)))
        )
        # update authority for next timestep
        authority = authority_tmp
        # update hub for next timestep
        hubness = hubness_tmp
        # perform early stopping on minuscule change
        if (delta < epsilon):
            break
        # decrement maximum number of remaining timesteps
        max_iterations -= 1
    # return tuple of authority and hub scores
    return (authority, hubness)
```

## Algorithm Description (PageRank)

In the PageRank algorithm, we initiate the PageRank vector with ones and subsequently normalize it to ensure that the values add up precisely to one. Similarly, vector updates occur only after each iteration (timestep) ends. Within each iteration, for every vertex, we compute the PageRank value for the next timestep. As per the PageRank algorithm, this value is

contingent on the PageRank values of parent vectors in the current timestep. To distribute the inherited PageRank among children nodes, specifically the sibling nodes, we divide by the parent vertex's out-degree. The implementation of early stopping mechanism can enhance efficiency by reducing runtime. It is crucial to emphasize that the PageRank vector undergoes normalization at the end to ensure that the values sum to one.

```
def pagerank_algorithm(adjacency_matrix : np.ndarray,
                      damping_factor : float, *,
                      epsilon : Optional[ float ] = 0.01,
                      max_iterations : Optional[ int ] = -1) -> np.ndarray:
    # obtaining number of vertices in graph
    num_vertices = len(adjacency_matrix)
    # initialize pagerank vector with ones, then normalize it (summation equals to 1)
    pagerank = np.ones(shape = (num_vertices, ), dtype = np.float32) / num_vertices
    # store number of children nodes (outgoing links) for each node
    num_children = np.sum(adjacency_matrix, axis = 1)
    # force min-number of children to 1 => for matrix ops, does not affect result because of mask
    num_children = np.maximum(num_children, np.ones_like(num_children))
    # execute algorithm for maximum number of timesteps
    while (max_iterations != 0):
        # initialize pagerank vector at new timestep (t+1) with zeros
        pagerank_tmp = np.zeros_like(pagerank)
        # iterate through each vertex
        for vertex in range(num_vertices):
            # calculate pagerank value based on random-surfer pagerank equation
            pagerank_tmp[vertex] += [
                damping_factor / num_vertices +
                (1 - damping_factor) * np.sum(pagerank / num_children * adjacency_matrix[... , vertex])
            ]
        # calculate value difference between timesteps
        delta = np.sum(np.sqrt(np.square(pagerank - pagerank_tmp)))

    # update pagerank for next timestep
    pagerank = pagerank_tmp
    # early stopping on minuscule change
    if (delta < epsilon):
        break
    # decrement maximum number of remaining iterations
    max_iterations -= 1
    # normalize pagerank values
    pagerank = pagerank / np.sum(pagerank)
    return pagerank
```

## **Algorithm Description (SimRank)**

In the SimRank algorithm, we begin the process by initializing an identity matrix that serves as the repository for SimRank values. As with the two preceding algorithms, updates to the matrix occur solely at the end of each iteration. Within each iteration, the computation of SimRank values



spans the  $N \times N$  cells in the SimRank matrix, where each cell at row  $i$ , col  $j$  encapsulates the SimRank correlation between node  $i$  and node  $j$ . The derivation of each SimRank value involves the summation of scores across all potential combinations of parent nodes.

For example, if node  $a$  has  $Q$  parents and node  $b$  has  $P$  parents, the computation of  $SimRank(node\_a, node\_b)$  entails aggregating  $Q \times P$  SimRank values. To optimize execution time, a technique can be used—prior identification of all parent nodes. Particularly beneficial for sparse matrices, this approach eliminates the need to iterate through numerous vectors that are not parent nodes of a specific node.

This strategic precomputation of parent nodes proves quite advantageous in scenarios involving sparse matrices, speeding the algorithm's execution time up by negating the necessity to iterate through numerous vectors that do not qualify as parent nodes for a given node. It is important to note that the SimRank value is 0, if there are no parent nodes.

```
def simrank_algorithm(adjacency_matrix : np.ndarray,
                     decay_factor : float, *,
                     max_iterations : Optional[ int ] = 3) -> np.ndarray:
    # initialize simrank matrix with zeros
    simrank = np.zeros_like(adjacency_matrix, dtype = np.float32)
    # obtaining number of vertices
    num_vertices = len(adjacency_matrix)
    # initialize identity matrix
    for vertex in range(num_vertices):
        simrank[vertex][vertex] = 1
    # pre-calculate number of parents for each vertex
    num_parent = np.sum(adjacency_matrix, axis = 0)
    def find_parents() -> Dict[ int, Set[ int ] ]:
        nonlocal num_vertices, adjacency_matrix
        parents = dict()
        # look for parent nodes in adjacency matrix
        for e_vert in range(num_vertices):
            for s_vert in range(num_vertices):
                if (adjacency_matrix[s_vert][e_vert] == 1):
                    parents[e_vert] = parents.get(e_vert, set()).union({ s_vert })
        return parents
    # record all parent nodes for each node
    parents = find_parents()
    def compute_simrank(vertex_a : int, vertex_b : int) -> float:
        nonlocal simrank, simrank_tmp, parents, num_parent, num_vertices, decay_factor, adjacency_matrix
        # same vertex => 1
        if (vertex_a == vertex_b):
            return 1
```

```

# number of parents of vertex a
num_parent_a = num_parent[vertex_a]
# number of parents of vertex b
num_parent_b = num_parent[vertex_b]
# product of both parent numbers equals to 0 => 0
if ((num_parent_a == 0) or (num_parent_b == 0)):
    return 0
# decay factor divided by product of both parent numbers
rescale_factor = decay_factor / num_parent_a / num_parent_b
counter = 0
# aggregate simrank of all parent vertex combinations
for parent_vert_a in parents[vertex_a]:
    for parent_vert_b in parents[vertex_b]:
        counter += simrank[parent_vert_a][parent_vert_b]
#
return counter * rescale_factor
for _ in range(max_iterations):
    # initialize simrank for next timestep (t+1) with zeros
    simrank_tmp = np.zeros_like(simrank)
    # calculate simrank value for each (a,b) pair
    for s_vertex, e_vertex in itertools.product(range(num_vertices), repeat = 2):
        simrank_tmp[s_vertex][e_vertex] = compute_simrank(s_vertex, e_vertex)
    # update simrank for next timestep
    simrank = simrank_tmp.copy()
return simrank

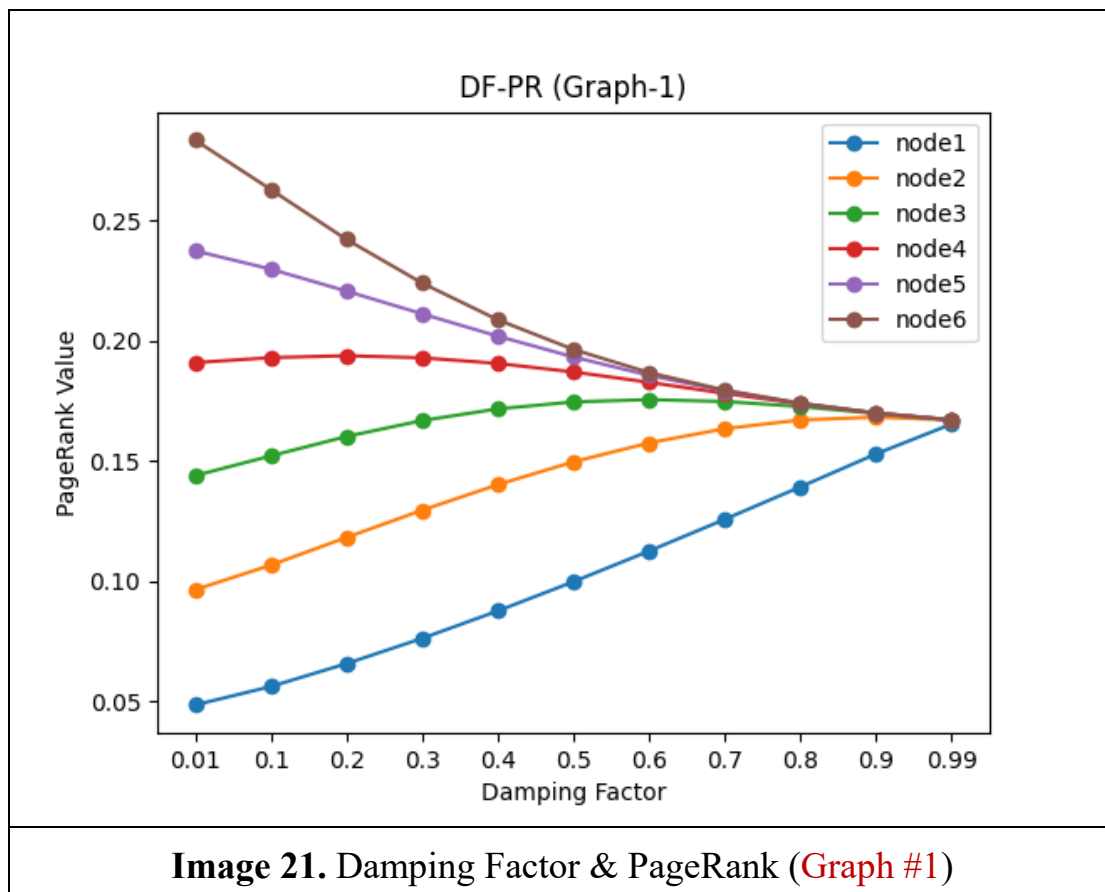
```

$$SimRank_t(Node_A, Node_B)$$

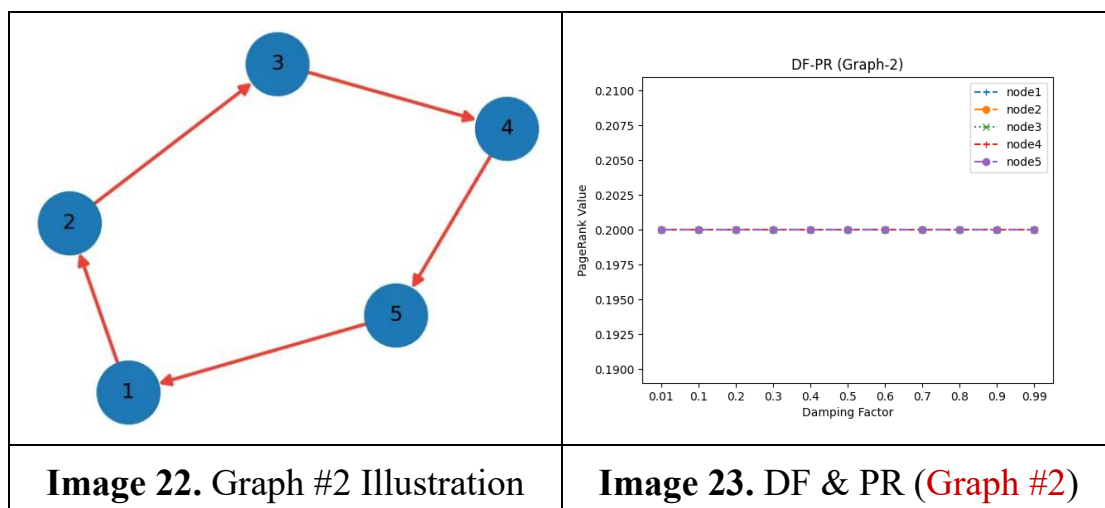
$$\begin{aligned}
 &= \frac{C}{|Parents(Node_A)| \times |Parents(Node_B)|} \\
 &\times \sum_{\omega \in Parents(Node_A)} \sum_{\theta \in Parents(Node_B)} SimRank_{t-1}(Node_{\omega}, Node_{\theta})
 \end{aligned}$$

## **Result Analysis & Discussion**

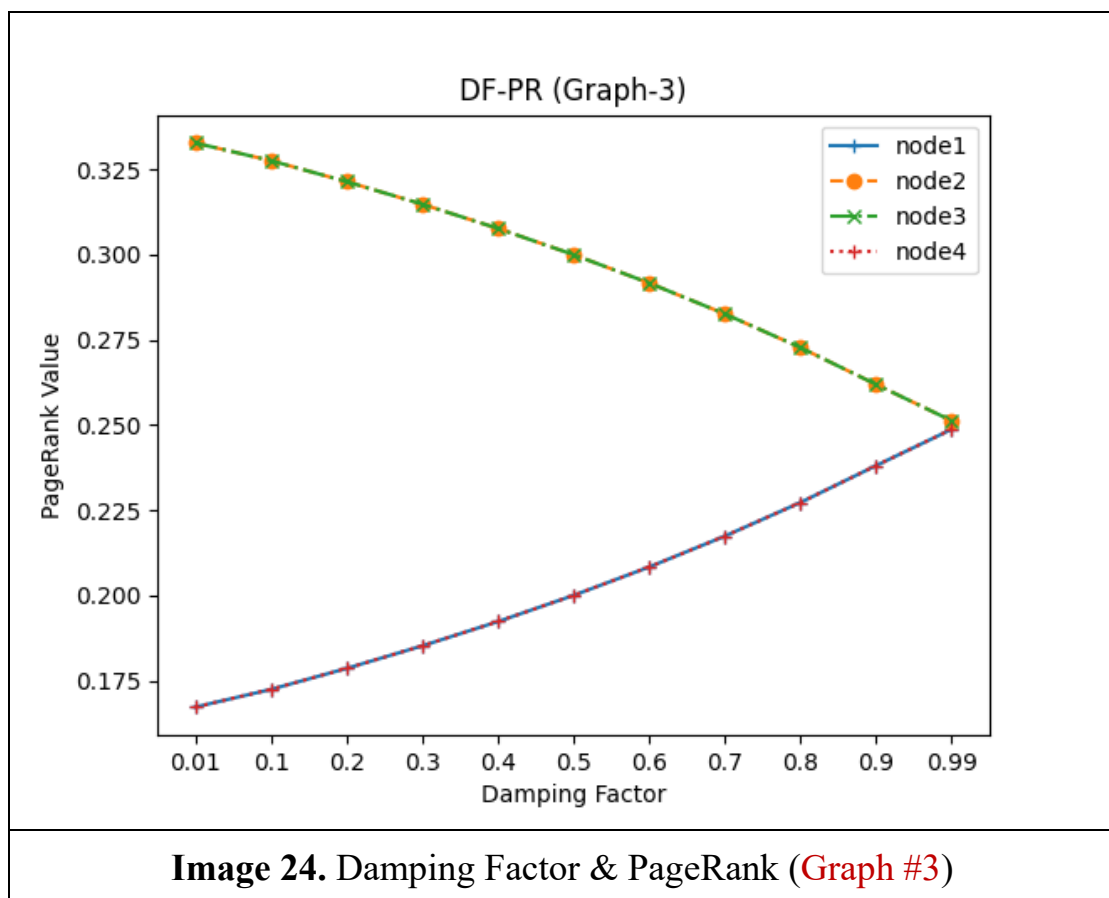
In the first graph, all nodes are connected in a straight line from node 1 to node 6 in one direction. PageRank calculates scores by taking a node's parent scores and dividing them by the parent's outdegree. Consequently, node 1 should have a low PageRank score since it has no parents, while node 6 should have the highest score because all paths lead to it—a kind of "sink." However, the random surfer model prevents the "sink" problem by randomly jumping to nodes. The figure illustrates that increasing the damping factor makes all six nodes converge to the same score, as nodes are chosen randomly without considering their connections.



Regarding graph 2, since it is cyclic, each vector must eventually connect with one another after some timesteps, as shown in the following image. Consequently, the PageRank values should remain constant even after adjusting the damping factor. The subsequent image illustrates the effect of the damping factor on the PageRank values of the second graph.



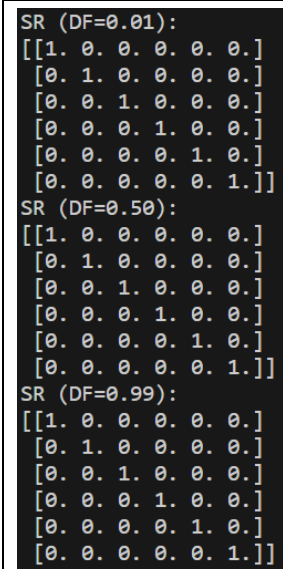
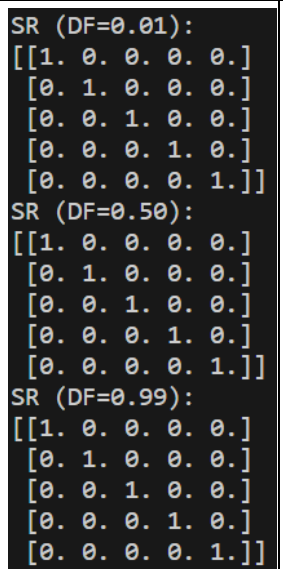
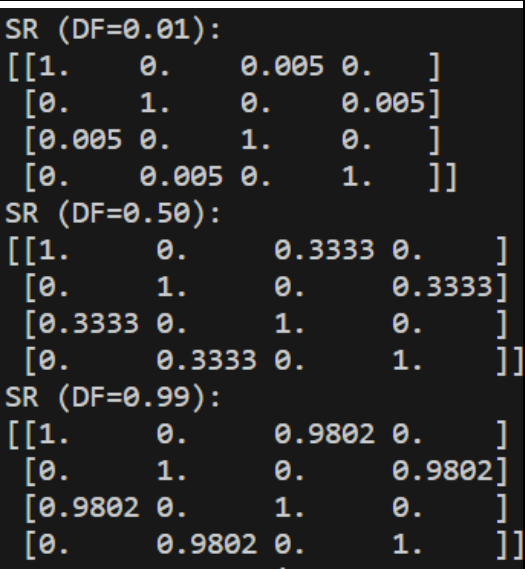
Regarding the third graph, since all vectors are linked by bidirectional edges, the supposed sinks, namely nodes 1 and 4, should have the lowest indegrees. Conversely, nodes 2 and 3, with the highest indegrees, should initially exhibit higher PageRank values. However, as shown in the next image, increasing the damping factor results in the convergence of PageRank values for all vectors to a common value. This occurs because we are primarily selecting vertices to visit at random.



Regarding the results of the HITS algorithm, it's clear that nodes without in-degrees have an authority score of 0, exemplified by Node #1 in Graph #1. Conversely, nodes without out-degrees have a hub score of 0. In our experiment in part 1, where we aimed to raise both the hub and authority scores of Node #1, we noticed that adding more incoming edges increases

authority, while adding more outgoing edges increases hub. This observation aligns with the HITS formula.

Concerning the SimRank values, we employ three distinct decay factor values (0.01, 0.50, 0.99) and examine their influence on SimRank values. For easier visualization, we will only consider the first three graphs. The subsequent three images illustrate the impact of decay factor values on SimRank values.

		
<p><b>Image 25.</b>  <b>SimRank</b>  Graph #1</p>	<p><b>Image 26.</b>  <b>SimRank</b>  Graph #2</p>	<p><b>Image 27. SimRank</b> Graph #3</p>

In the images above, we observe that the decay factor has minimal impact on the SimRank values of graphs 1 & 2. In contrast, for graph #3, there is a noticeable change in the values of certain cells in the SimRank matrix. Increasing the decay factor allows the SimRank value to propagate farther without great loss. This effect is particularly pronounced in bidirectional graphs such as graph #3.

### Execution Time Analysis

The execution times for all seven graphs for the three algorithms (HITS, PageRank, SimRank) are provided in the following table.

	HITS (s)	PageRank (s)	SimRank (s)
<b>Graph #1</b>	<i>0.001</i>	<i>0.000</i>	<i>0.003</i>
<b>Graph #2</b>	<i>0.001</i>	<i>0.001</i>	<i>0.002</i>
<b>Graph #3</b>	<i>0.001</i>	<i>0.001</i>	<i>0.003</i>
<b>Graph #4</b>	<i>0.002</i>	<i>0.001</i>	<i>0.007</i>
<b>Graph #5</b>	<i>0.106</i>	<i>0.074</i>	<i>26.262</i>
<b>Graph #6</b>	<i>0.309</i>	<i>0.213</i>	<i>N/A</i>
<b>IBM-5000</b>	<i>0.142</i>	<i>0.177</i>	<i>N/A</i>

From the table above, there isn't a significant difference in execution time between HITS and PageRank algorithms. However, SimRank particularly exhibits slower performance and greater computational demands, mainly due to its higher time complexity. The increased time complexity results from dealing with two-dimensional data compared to the one-dimensional data handled by the former two algorithms (HITS/PageRank). It's crucial to highlight that the execution time for SimRank has been substantially improved by pre-computing parent nodes from the adjacency matrix and leveraging matrix operations instead of native Python for loops. For HITS and PageRank, the efficient execution could be attributed to the usage of early stopping, where the algorithm comes to a halt when no significant convergence can be further achieved, as specified in the textbook.