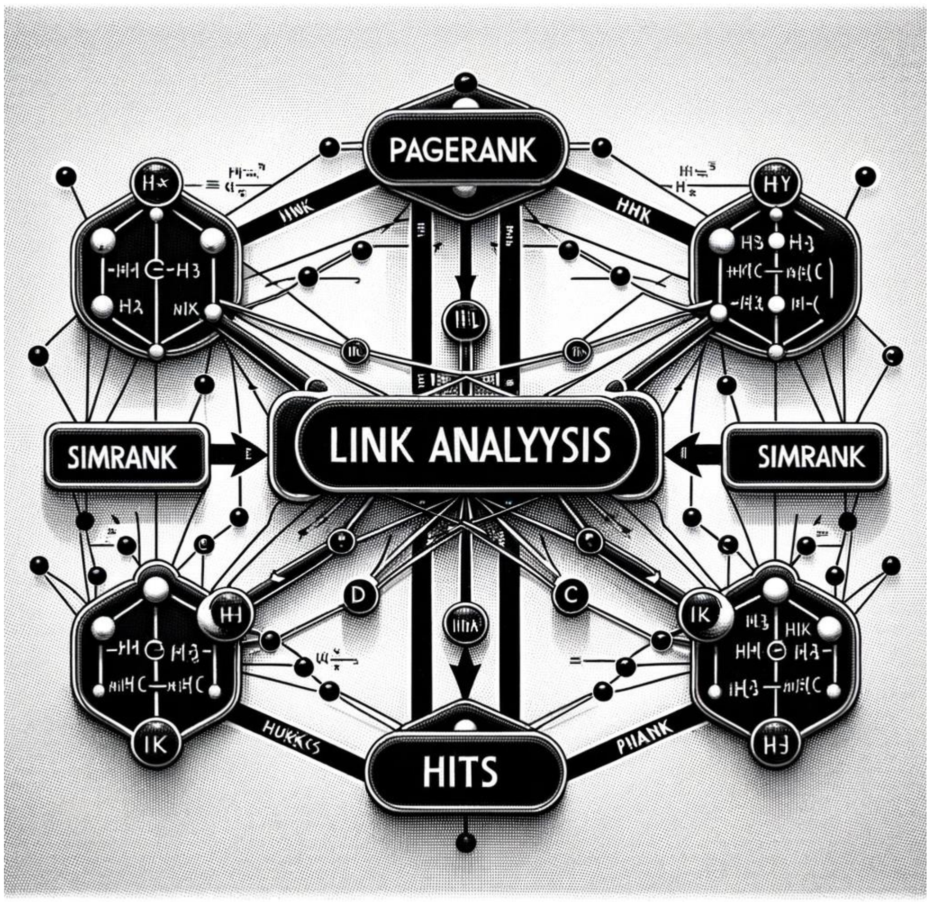
Link Analysis



INDEX

1. Abstract

This report delves into the implementation and analysis of three key link analysis algorithms: PageRank, Hyperlink-Induced Topic Search (HITS), and SimRank. Focused on directed graphs, the study involves custom algorithmic development, ensuring a deep understanding of their theoretical underpinnings. The research uses various directed graphs, including specific IBM data, to investigate these algorithms' behaviors and responses to network topology changes.

Key experiments include modifying network structures to influence specific metrics, such as enhancing Node 1's scores across different graphs. The report also examines the impact of varying algorithmic parameters, like damping and decay factors, on the final results. Execution times across various graphs are analyzed to assess computational efficiency, offering insights into the scalability and practicality of these algorithms in data mining applications.

2. Data

This project utilizes seven directed graph datasets, including graph_1.txt to graph_6.txt for diverse network structures, and ibm-5000.txt, to analyze link analysis algorithms.

- graph_1.txt
- graph_2.txt
- graph_3.txt
- graph_4.txt
- graph_5.txt
- graph_6.txt
- ibm-5000.txt

3. Settings

- **damping_factor** = 0.1
- **decay_factor** = 0.7 → For **SimRank** only.
- **iteration** = 30

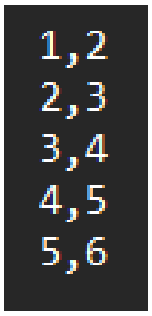
4. Find a way!

Find a way (e.g., add/delete some links) to increase hub, authority, and PageRank of Node 1 in first 3 graphs respectively :

PageRank:

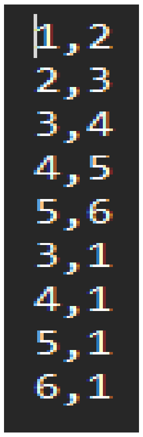
The displayed network from **graph_1.txt, graph_2.txt, graph_3.txt** initially depicts Node 1 with **only an outgoing link to Node 2** and no incoming links, which limits its PageRank. **To boost its importance**, we've added incoming links from Nodes 3 to 6, transforming Node 1 into a prominent hub within the graph. This strategic augmentation of incoming links effectively increases Node 1's PageRank, enhancing its prominence in the network.

Initially



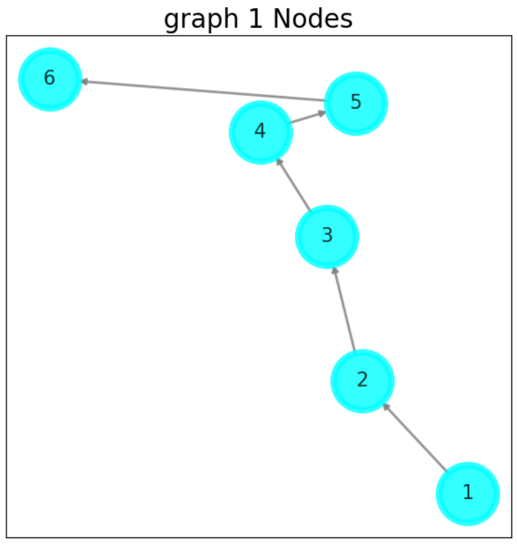
```
1,2
2,3
3,4
4,5
5,6
```

Node 1 with more incoming links

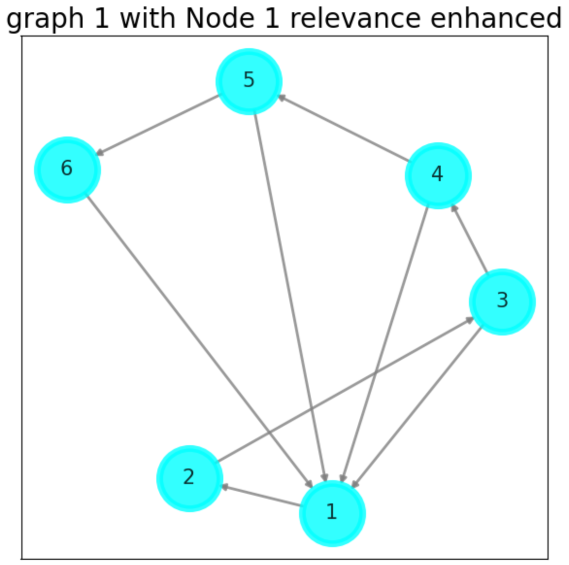


```
1,2
2,3
3,4
4,5
5,6
3,1
4,1
5,1
6,1
```

Initially

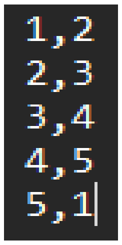


Node 1 with more incoming links



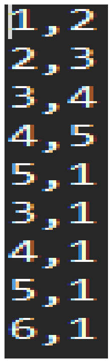
I applied the same strategy for graph_2.txt y graph_3.txt

Initially



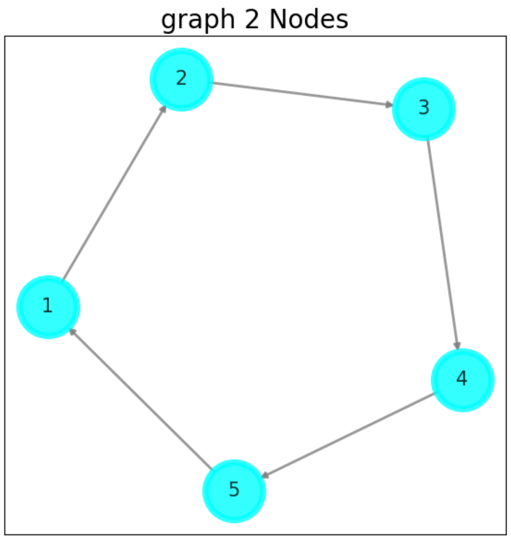
```
1,2
2,3
3,4
4,5
5,1
```

Node 1 with more incoming links

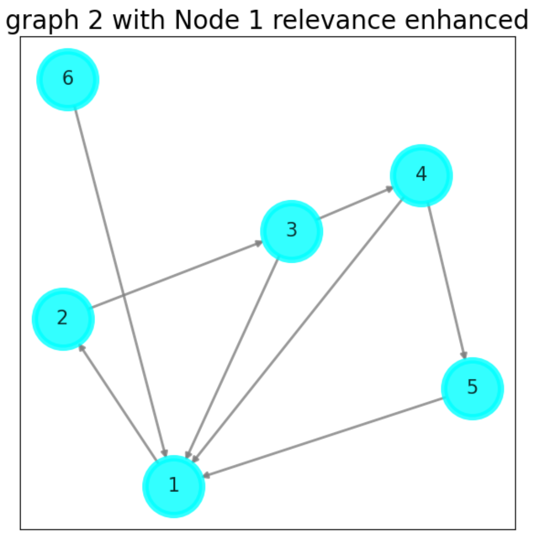


```
1,2
2,3
3,4
4,5
5,1
3,1
4,1
5,1
6,1
```

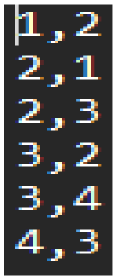
Initially



Node 1 with more incoming links

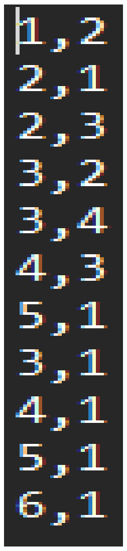


Initially



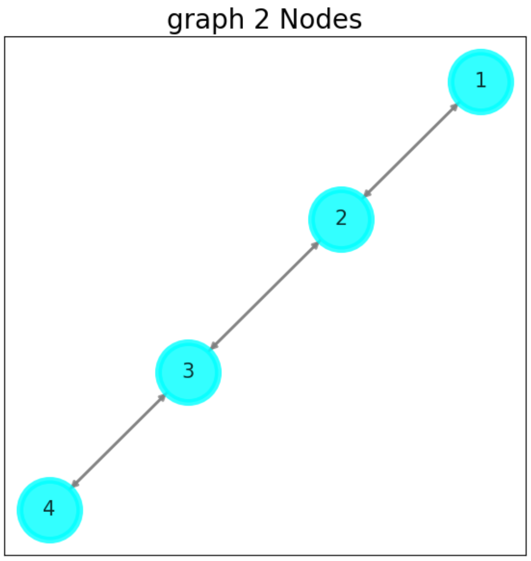
```
1,2
2,1
2,3
3,2
3,4
4,3
```

Node 1 with more incoming links

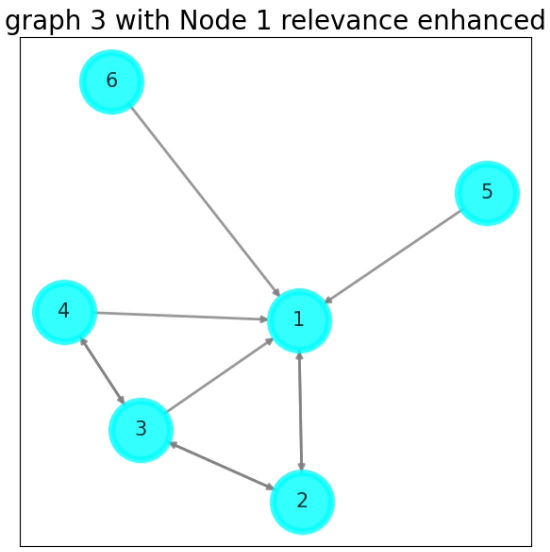


```
1,2
2,1
2,3
3,2
3,4
4,3
5,1
3,1
4,1
5,1
6,1
```

Initially



Node 1 with more incoming links



Strategic Approach for Amplifying Node 1's Hub and Authority

To enhance Node 1's prominence as both a hub and an authority within the network, we will employ a **reciprocal link structure.** This involves:

Outbound Linking: We will initiate outbound links from Node 1 to nodes that already possess high authority scores, positioning Node 1 as a recommender within the network.

Inbound Linking: In turn, we will ensure that these high-authority nodes link back to Node 1, establishing a mutual endorsement that elevates Node 1's authority score.

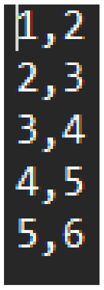
By implementing this **bidirectional linking**, we create a reinforcing feedback loop where Node 1's role as a hub bolsters its authority, and its growing authority enhances its hub status. This synergistic approach is designed to significantly boost Node 1's importance in both metrics after recalculating with the HITS algorithm.

Strategy for graph_1.txt

- Give more authority to the Node 1 , by link every node to it.
- Give more authority to the Node 5,6 and link there as inbound for Node 1, giving even more authority to the Node 1 , making it a good Authority.

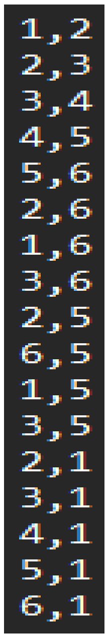
- Outbound link 1 to Node 5,6 , and because they are good authorities too, Node 1 becomes a good Hub for Node 5 and 6.

Initially



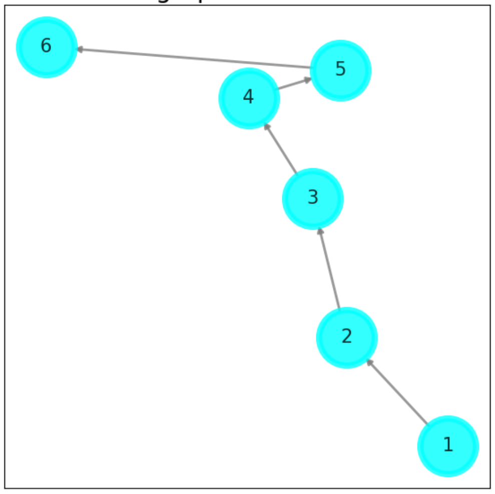
```
1,2
2,3
3,4
4,5
5,6
```

Node 1, 5, 6 with more Authority

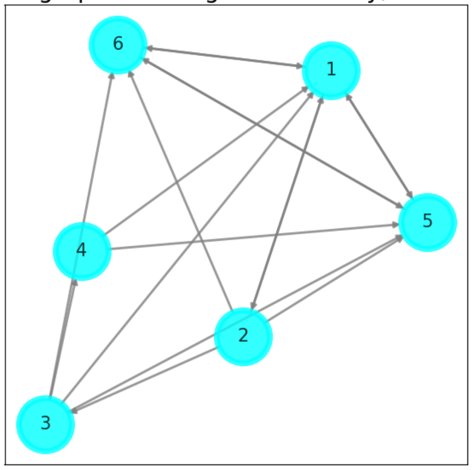


```
1,2
2,3
3,4
4,5
5,6
2,6
1,6
3,6
2,5
6,5
1,5
3,5
2,1
3,1
4,1
5,1
6,1
```

graph 1 Nodes

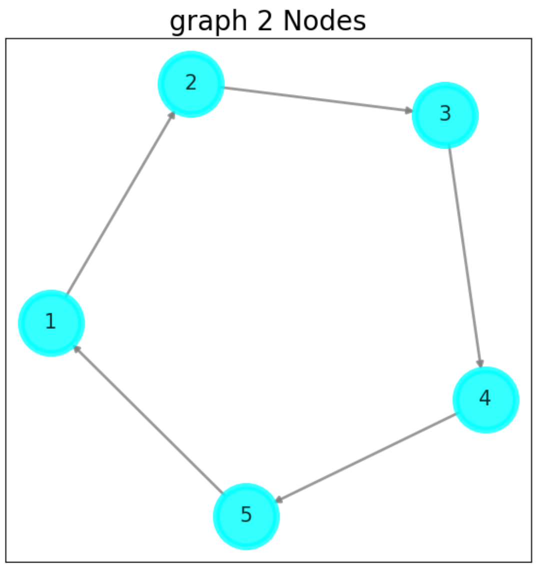


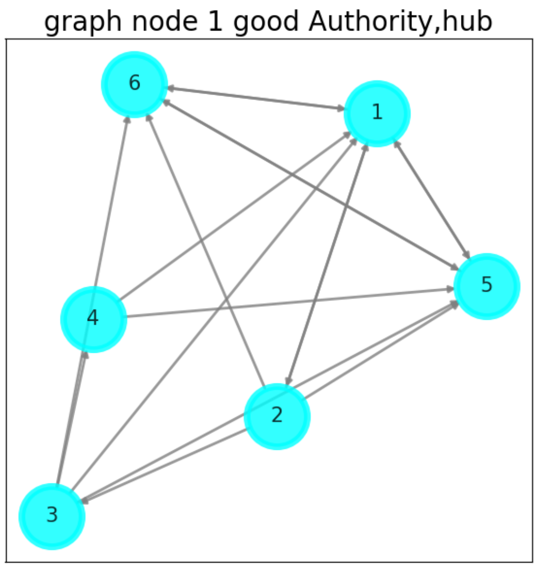
graph node 1 good Authority,hub



Strategy for graph_2.txt

- Because Nodes in this file where similar , each node has the same authority , so I decided to do the same in as in the graph_1.txt too.

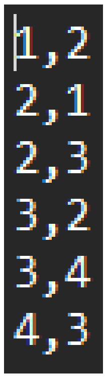




Strategy for graph_3.txt

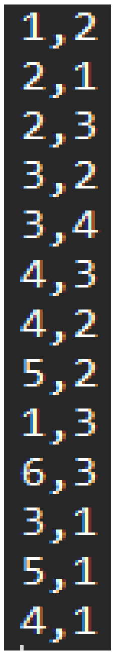
- For Nodes in this file , nodes 2,3 had more authority , so I decided to give them even more , and then inbound them to the Node 1 , so it becomes a good Authority .
- Make Node 1 follow Nodes 2,3 which are good authorities , so Node 1 ,so it becomes a good hub for them.

Initially

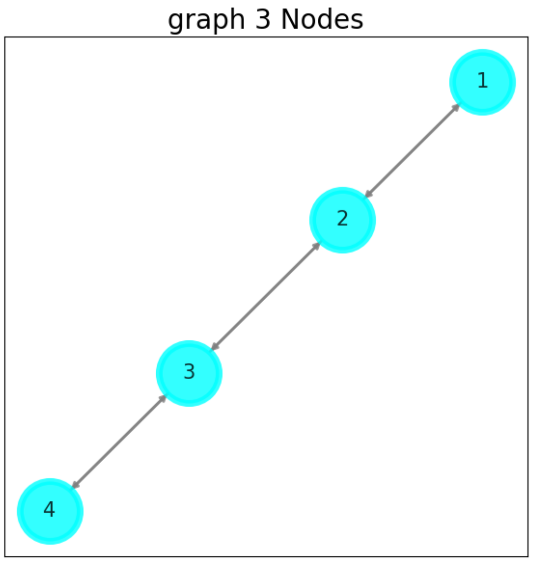


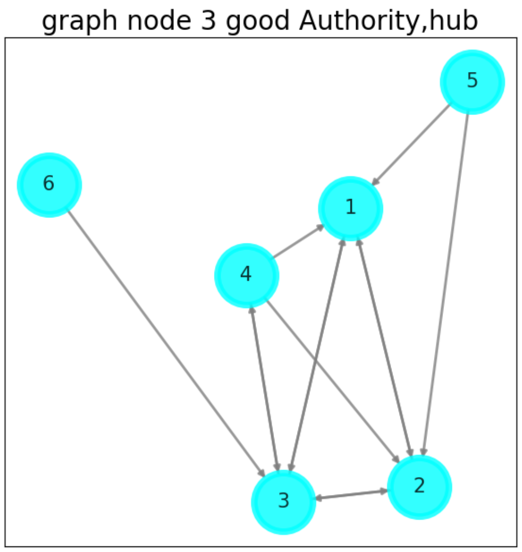
```
1,2
2,1
2,3
3,2
3,4
4,3
```

I created a reciprocal effect between Node 1, 2, 3



```
1,2
2,1
2,3
3,2
3,4
4,3
4,2
5,2
1,3
6,3
3,1
5,1
4,1
```





5. Algorithm description

PageRank Algorithm

The PageRank algorithm is used as a web search engine technology, originally devised to rank web pages based on their importance. It operates on the premise that significant pages are likely to receive more links from other pages. In essence, PageRank interprets a link from page A to page B as a 'vote' by page A for page B's importance.

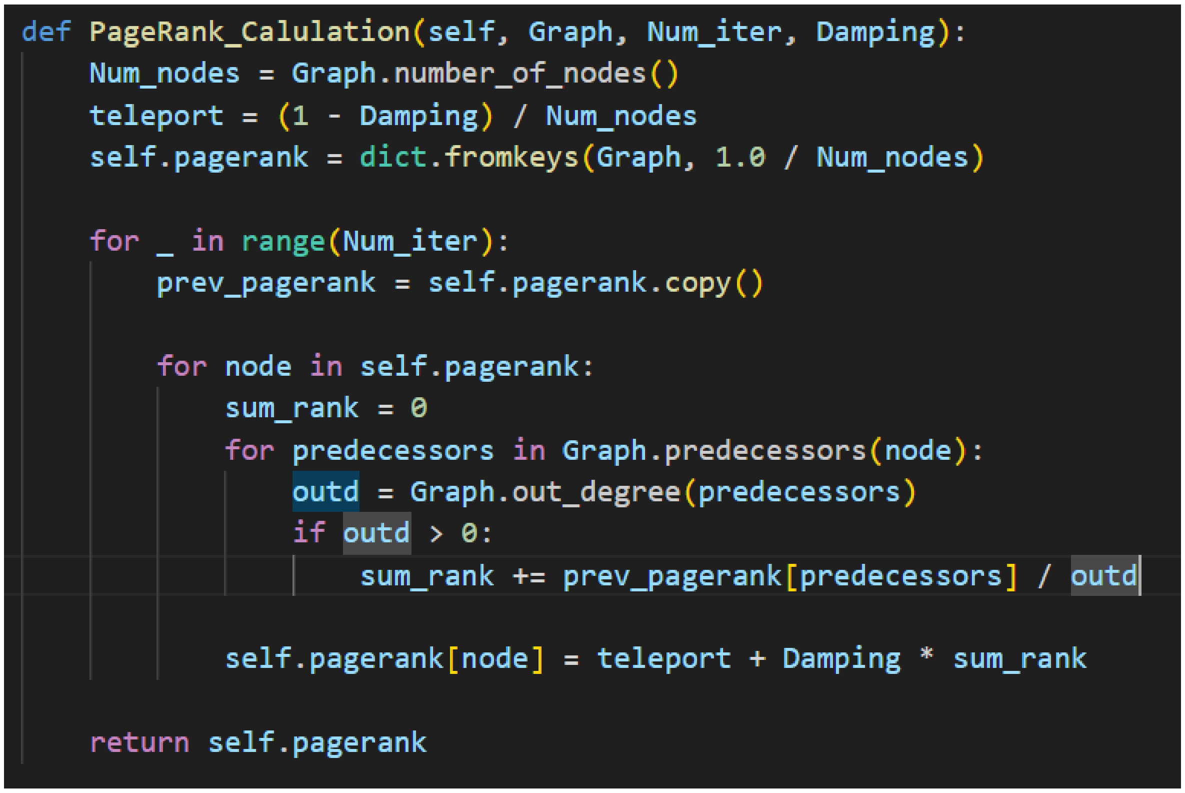
$$PageRank\ of\ site = \sum \frac{PageRank\ of\ inbound\ link}{Number\ of\ links\ on\ that\ page}$$

OR

$$PR(u) = (1 - d) + d \times \sum \frac{PR(v)}{N(v)}$$

Where:

- PR(u) is the PageRank of page u.
- d is the damping factor.
- PR(v) are pages linking to page u.
- N(v) is the number of outbound links on page u.
- (1-d) is the probability of jumping to a page at random(also Known as **Teleport**).



```python
def PageRank_Calulation(self, Graph, Num_iter, Damping):
    Num_nodes = Graph.number_of_nodes()
    teleport = (1 - Damping) / Num_nodes
    self.pagerank = dict.fromkeys(Graph, 1.0 / Num_nodes)

    for _ in range(Num_iter):
        prev_pagerank = self.pagerank.copy()

        for node in self.pagerank:
            sum_rank = 0
            for predecessors in Graph.predecessors(node):
                outd = Graph.out_degree(predecessors)
                if outd > 0:
                    sum_rank += prev_pagerank[predecessors] / outd

            self.pagerank[node] = teleport + Damping * sum_rank

    return self.pagerank
```

PageRank_Calulation function, which takes a directed graph ,**Graph**, **the number of iterations Num_iter**, **and the damping factor Damping** as inputs. Here's how the function works:

Initialization:

The total number of nodes **Num_nodes** in the graph is determined.

A teleport term is calculated, which represents the random jump factor across the graph. It's computed as **(1 - Damping) / Num_nodes,** ensuring that even pages without inbound links have a baseline PageRank value.

The pagerank**(self.pagerank)** dictionary is initialized, assigning an equal initial **PageRank value of 1.0 / Num_nodes** to each node, signifying an equal probability of being on any page at the start.

Iteration Loop:

We enter a loop that will run for **Num_iter iterations**. In each iteration, we perform the following steps:

A temporary copy of the current PageRank values is stored in **prev_pagerank**. This is crucial as the calculation of PageRank for each node should use the values from the previous iteration.

PageRank Calculation for Each Node:

For each node in the graph, we calculate its new PageRank value by summing the PageRank of all its predecessors (nodes that link to it) divided by their respective out-degree, which is the count of their outbound links.

Each node's PageRank is then updated with the new value, which consists of the teleport term plus the aggregated and damped PageRank from the predecessors. **The formula used is self.pagerank[node] = teleport + Damping * sum_rank.**

Result:

After completing the iterations, the function returns the pagerank dictionary containing the PageRank values of all nodes, reflecting their relative importance within the graph.

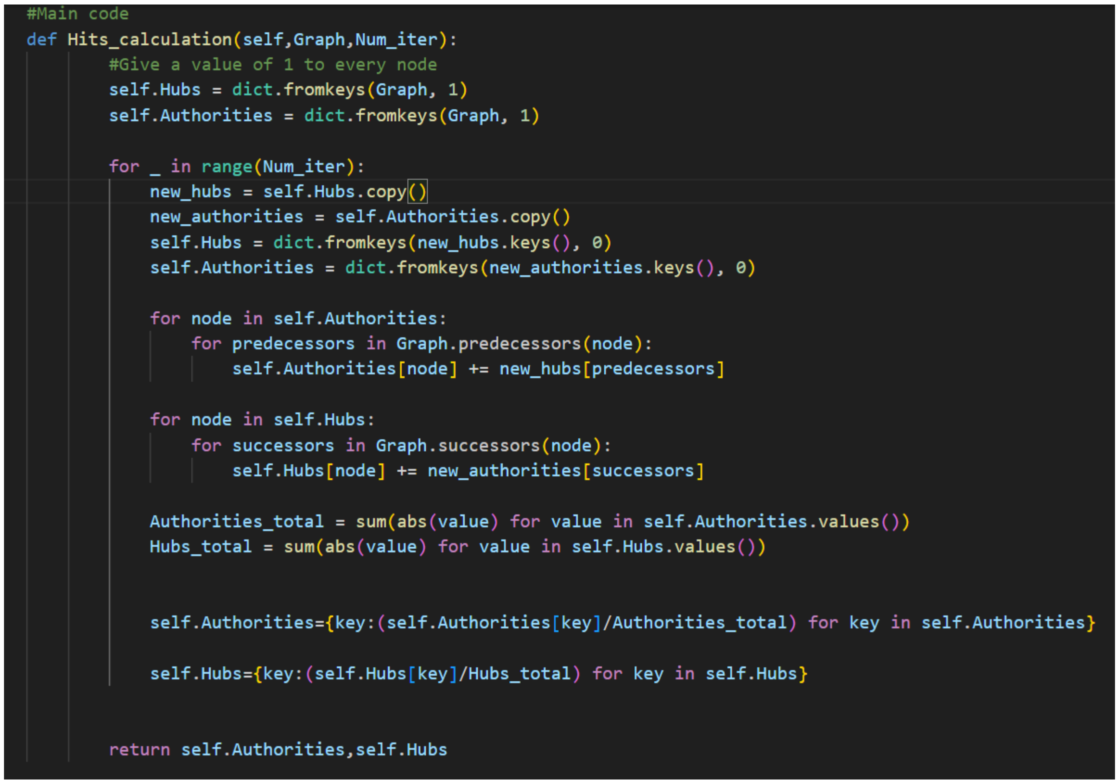
HITS Algorithm

The HITS (Hyperlink-Induced Topic Search) algorithm, also known as "hubs and authorities," is a link analysis algorithm that rates web pages, viewing them as either hubs or authorities. Hubs are pages that link to many other pages, while authorities are the ones with many inbound links from hubs. The key idea is that good hubs are pages that point to good authorities, and good authorities are pages that are linked by good hubs.

The iterative calculation updates the hub and authority scores for each page based on the incoming and outgoing links:

Authority Update Rule: A page's authority score is the sum of the hub scores of the pages that link to it.

Hub Update Rule: A page's hub score is the sum of the authority scores of the pages it links to.



```
#Main code
def Hits_calculation(self,Graph,Num_iter):
        #Give a value of 1 to every node
        self.Hubs = dict.fromkeys(Graph, 1)
        self.Authorities = dict.fromkeys(Graph, 1)

        for _ in range(Num_iter):
            new_hubs = self.Hubs.copy()
            new_authorities = self.Authorities.copy()
            self.Hubs = dict.fromkeys(new_hubs.keys(), 0)
            self.Authorities = dict.fromkeys(new_authorities.keys(), 0)

            for node in self.Authorities:
                for predecessors in Graph.predecessors(node):
                    self.Authorities[node] += new_hubs[predecessors]

            for node in self.Hubs:
                for successors in Graph.successors(node):
                    self.Hubs[node] += new_authorities[successors]

            Authorities_total = sum(abs(value) for value in self.Authorities.values())
            Hubs_total = sum(abs(value) for value in self.Hubs.values())


            self.Authorities={key:(self.Authorities[key]/Authorities_total) for key in self.Authorities}

            self.Hubs={key:(self.Hubs[key]/Hubs_total) for key in self.Hubs}


        return self.Authorities,self.Hubs
```

This implementation of the HITS algorithm **iteratively calculates the hub and authority scores** for each node in a given directed graph. The process is executed within the Hits_calculation function, which accepts a directed graph, **Graph,** and the number of iterations Num_iter as inputs.

Initialization:

We initialize both the Hubs and Authorities dictionaries with all nodes in the graph, **assigning each node an initial score of 1**. **This uniform starting point ensures that all nodes are considered equally in the first iteration.**

Iterative Calculation:

The algorithm enters a loop that **runs for Num_iter iterations.** In each iteration, **the hub and authority scores are updated based** on the following process:

Copy Current Scores: Temporary copies of the current hub **(new_hubs)** and authority **(new_authorities)** scores are created. This ensures that calculations within an iteration are based on the scores from the previous iteration.

Reset Scores: The Hubs and Authorities dictionaries are reset to zero to prepare for the new calculation.

Update Authority Scores: For each node, we sum the hub scores of all its predecessors (nodes that link to it) to calculate its new authority score. This reflects the notion that a good authority is a page that is linked by many good hubs.

Update Hub Scores: Similarly, for each node, we sum the authority scores of all its successors **(nodes it links to)** to calculate its new hub score. A good hub is a page that links to many good authorities.

Normalization:

After each iteration, the total scores for hubs and authorities are calculated. **The individual hub and authority scores are then normalized** by dividing them by these totals. This step ensures that the scores remain comparable across iterations and nodes.

Results:

The function returns the final Authorities and Hubs dictionaries after completing the specified number of iterations. These dictionaries contain the hub and authority scores for each node, indicating their relative importance within the network based on their link structures.

SimRank Algorithm

SimRank is a similarity measure used in graph theory, particularly in the context of link analysis. The core idea behind SimRank is that **two nodes are considered similar if they are linked by similar nodes.** It is an iterative algorithm that evaluates the similarity of nodes based on their structural context in a network.

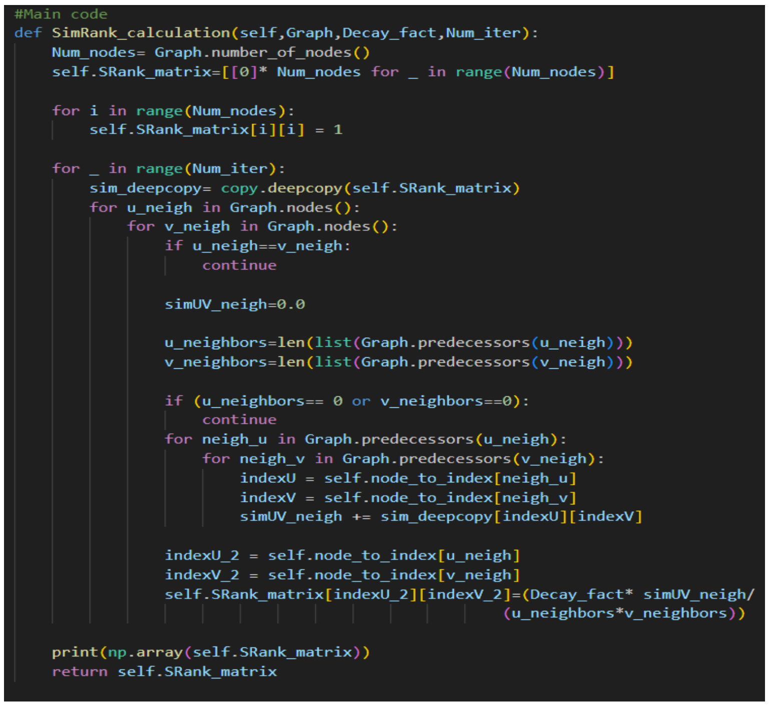
The fundamental principle of SimRank is: "Two objects are similar if they are related to similar objects**." This recursive nature makes it uniquely suited for evaluating similarity in complex networks.**

Formula

$$S(a,b) = \frac{C}{|I(a)| \times |I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} S(I_i(a), I_j(b))$$

Where:

- **S(a,b)** represents the similarity score between nodes **a** and **b**.
- **C** is a constant **decay factor** (usually between 0 and 1) that dampens the similarity score over multiple iterations, preventing it from inflating.
- The denominator, **|I(a)| x |I(b)|** , normalizes the similarity score based on the number of in-neighbors (or incoming links) of each node, ensuring that nodes with more in-links don't unfairly bias the similarity score.



```python
#Main code
def SimRank_calculation(self,Graph,Decay_fact,Num_iter):
    Num_nodes= Graph.number_of_nodes()
    self.SRank_matrix=[[0]* Num_nodes for _ in range(Num_nodes)]

    for i in range(Num_nodes):
        self.SRank_matrix[i][i] = 1

    for _ in range(Num_iter):
        sim_deepcopy= copy.deepcopy(self.SRank_matrix)
        for u_neigh in Graph.nodes():
            for v_neigh in Graph.nodes():
                if u_neigh==v_neigh:
                    continue

                simUV_neigh=0.0

                u_neighbors=len(list(Graph.predecessors(u_neigh)))
                v_neighbors=len(list(Graph.predecessors(v_neigh)))

                if (u_neighbors== 0 or v_neighbors==0):
                    continue
                for neigh_u in Graph.predecessors(u_neigh):
                    for neigh_v in Graph.predecessors(v_neigh):
                        indexU = self.node_to_index[neigh_u]
                        indexV = self.node_to_index[neigh_v]
                        simUV_neigh += sim_deepcopy[indexU][indexV]

                indexU_2 = self.node_to_index[u_neigh]
                indexV_2 = self.node_to_index[v_neigh]
                self.SRank_matrix[indexU_2][indexV_2]=(Decay_fact* simUV_neigh/
                                                        (u_neighbors*v_neighbors))

    print(np.array(self.SRank_matrix))
    return self.SRank_matrix
```

Explanation of the key steps in your **SimRank_calculation** method:

Initialization:

The number of nodes in the graph is determined and a square matrix **SRank_matrix** is initialized **with zeros**. **This matrix will hold the similarity scores between pairs of nodes.**

The diagonal of the matrix is set to **1**, signifying that each node is perfectly similar to itself (as per the SimRank assumption).

Iterative Calculation:

The function enters a loop that runs for a specified number of iterations **Num_iter**. In each iteration, the following operations are performed to update the similarity scores:

A deep copy of the current similarity matrix is created to preserve the values from the previous iteration during the calculation.

For each pair of distinct nodes **(u_neigh and v_neigh),** their similarity score is calculated based on their neighbors.

Similarity Score Computation:

The similarity between two different nodes is initially set to zero.

The algorithm retrieves the **list of predecessors (inbound neighbors)** for both nodes being compared.

If both nodes have predecessors, the algorithm proceeds to calculate the similarity score. If one of the nodes has no predecessors, the score remains zero as they cannot be similar through other nodes.

The similarity score is incremented by the sum of similarity scores of all pairs of predecessors from the previous iteration, scaled by the decay factor **Decay_fact.**

This score is then normalized by the product of the number of predecessors for each node, ensuring the score is averaged over the number of comparisons made.

Updating the Similarity Matrix:

After calculating the similarity score for each pair of nodes, the **SRank_matrix** is updated with the new scores.

The process repeats for the specified number of iterations or until the similarity scores converge.

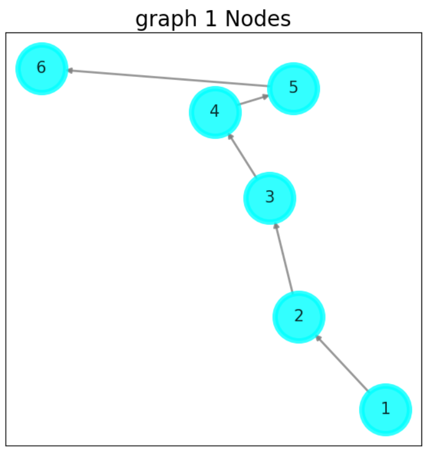
Final Output:

The function prints the final similarity matrix and returns it. This matrix contains the similarity scores between all pairs of nodes, which can be used to determine how similar nodes are to each other based on the graph's topology.

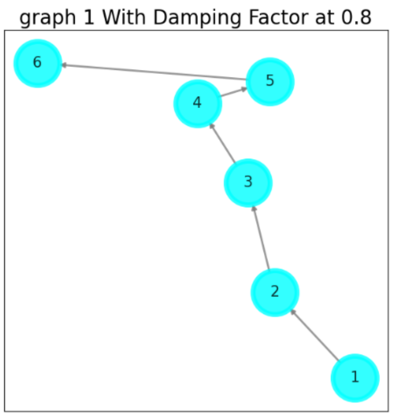
6. Analysis and discussion

PageRank Analysis of Graph 1

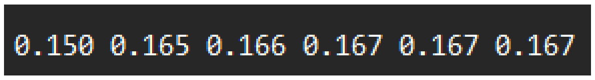
Damping factor at 0.1



Damping factor at 0.1



Graph 1 presents a **simple linear structure** with six nodes connected in a sequence from Node 1 to Node 6. The initial PageRank values indicate that **Node 1 has the lowest score (0.150),** which aligns with its position in the graph as the starting point with a single outbound link and no inbound links.

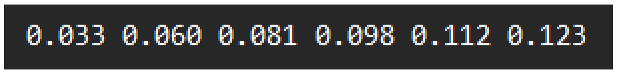


```
0.150 0.165 0.166 0.167 0.167 0.167
```

As we move from Node 1 to Node 6, there's a progressive increase in PageRank scores, with the last three nodes **(Nodes 4, 5, and 6)** all sharing the **highest score (0.167).** This pattern suggests that nodes at the end of the chain benefit from cumulative link equity passed along the chain. Node 1's lower score is a direct result of its lack of incoming links, which in PageRank's eyes, equates to a lack of endorsements from other pages.

Adjusting the **damping factor to 0.8** has resulted in a redistribution of PageRank values across the nodes in Graph 1.

Node 1: The increase in the damping factor has led to a slight decrease in PageRank for Node 1. **This suggests that Node 1, with no incoming links, relies more heavily on the random jump component of the PageRank calculation (the (1 - damping factor)/N term), which is reduced when the damping factor is increased.**



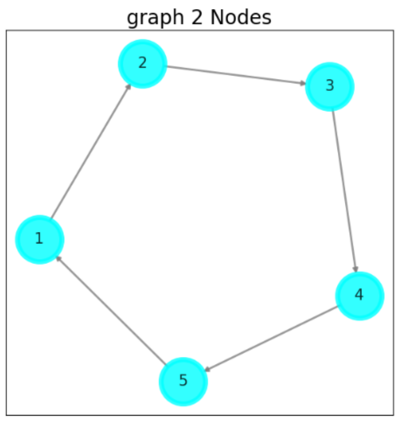
```
0.033 0.060 0.081 0.098 0.112 0.123
```

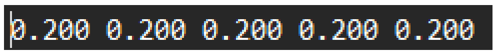
Node 6: Conversely, Node 6, at the end of the chain, now has the **highest PageRank**. This is due to the accumulated link equity from the preceding nodes, which is less dissipated with a higher damping factor. Each node passes on 80% of its PageRank to the next node, leading to Node 6 accruing the most PageRank.

Middle Nodes: Nodes 2 through 5 show a gradual increase in PageRank from Node 1 to Node 6, which is expected in a chain-like graph structure. The higher damping factor emphasizes the direct link equity passed along the chain.

PageRank Analysis of Graph 2

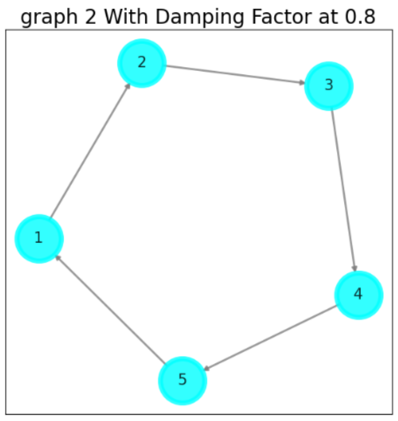
Damping factor at 0.1

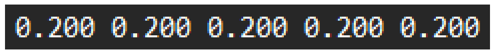




```
0.200 0.200 0.200 0.200 0.200
```

Damping factor at 0.8





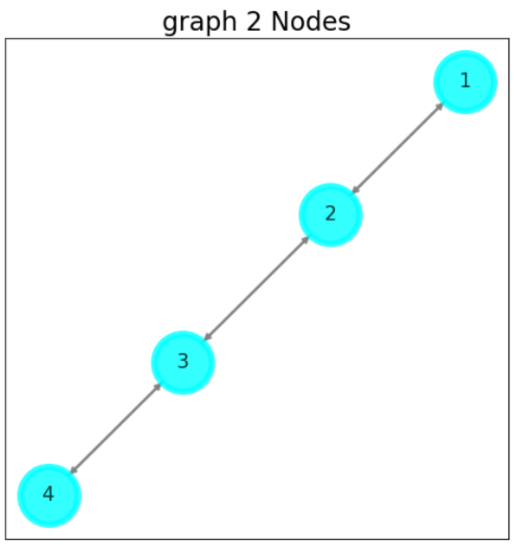
```
0.200 0.200 0.200 0.200 0.200
```

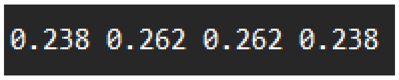
With a **damping factor of 0.1**, the PageRank algorithm **assigns an equal score to all nodes**, reflecting the uniform connectivity in the graph. The low damping factor means that most of the PageRank is distributed evenly across the graph due to the random jump factor.

Considering the uniform PageRank values **(0.200 for all nodes)** for graph 2 with a damping factor of 0.8, it appears that the increase in the damping factor from 0.1 to 0.8 **did not result in any change** in the distribution of PageRank values. Each node maintains an equal score, which is particularly interesting because it suggests that the structure of the graph allows for an even distribution of PageRank regardless of the damping factor used.

PageRank Analysis of Graph 3

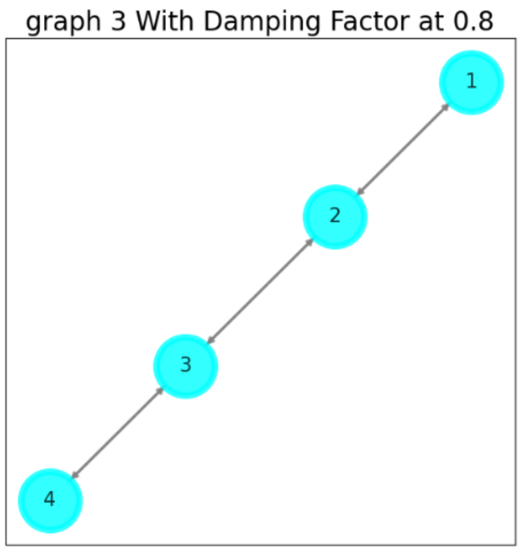
Damping factor at 0.1

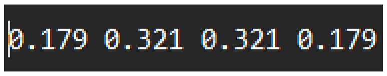




```
0.238 0.262 0.262 0.238
```

Damping factor at 0.8



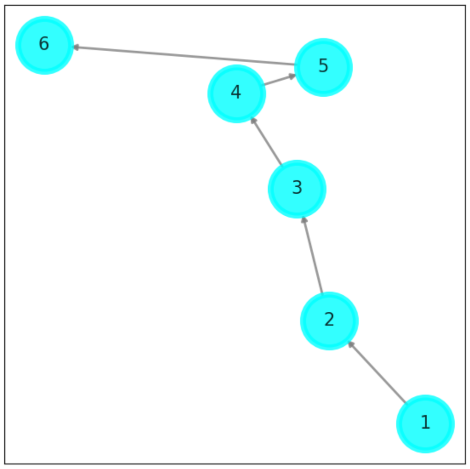


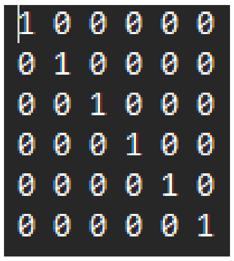
```
0.179 0.321 0.321 0.179
```

Nodes 2 and 3, which are in the middle of the sequence, have identical and the highest PageRank scores **(0.262),** indicating that these nodes benefit from being connected to each other and also receiving links from Nodes 1 and 4.
The terminal nodes, Nodes 1 and 4, have lower PageRank scores **(0.238),** which can be attributed to the fact that they only have one incoming link and one outgoing link.
The distribution of PageRank values **for graph 3** after adjusting the **damping factor to 0.8 (0.179, 0.321, 0.321, 0.179)** shows a significant change from the scenario with a damping factor of 0.1. The central nodes, Nodes 2 and 3, now have significantly higher PageRank scores (0.321) compared to the terminal nodes, Nodes 1 and 4 (0.179). This increase underscores the central nodes' importance in the graph's connectivity, as they act as bridges in the linear structure.

SimRank Analysis of Graph 1

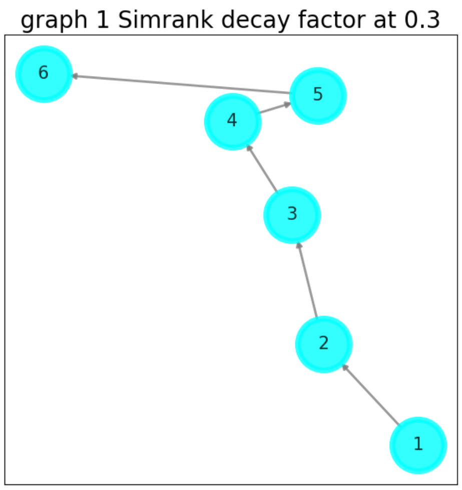
Decay factor at 0.7

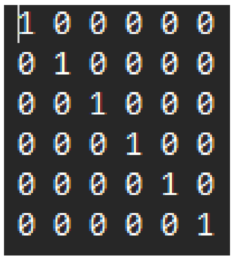




```
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

Decay factor at 0.1





```
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

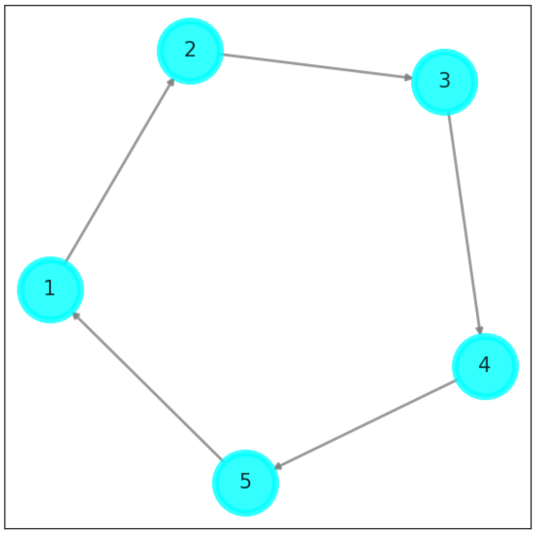
The result of the SimRank algorithm for the graph depicted in the image, with a **decay factor of 0.7**, shows a similarity matrix where all off-diagonal elements are zero and the diagonal elements are one. **This indicates that, according to the SimRank algorithm, each node is only similar to itself**, and no nodes are considered similar to any other nodes.

The SimRank results for Graph 1 with a **decay factor of 0.3** show a similarity matrix identical to the one produced with a decay factor of 0.7, which is an identity matrix. This indicates that the change in the

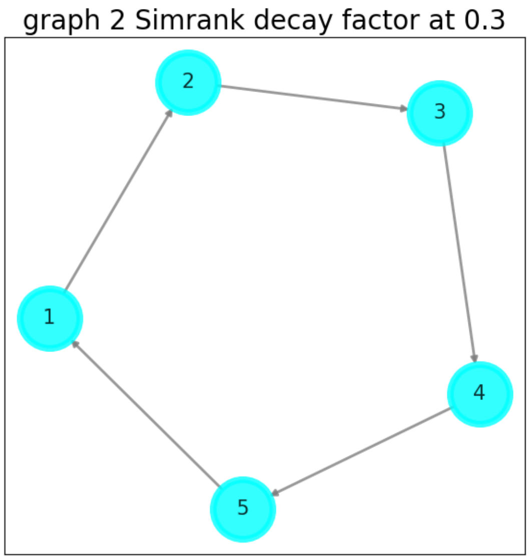
decay factor did not affect the outcome of the SimRank similarities between nodes in this particular graph structure.

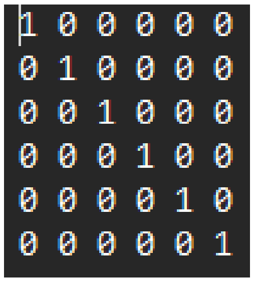
SimRank Analysis of Graph 2

Decay factor at 0.7

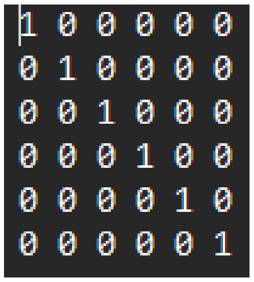


Decay factor at 0.3





```
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```



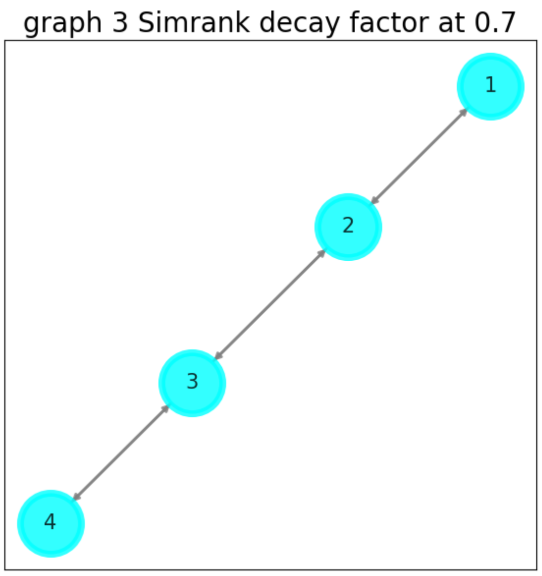
```
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

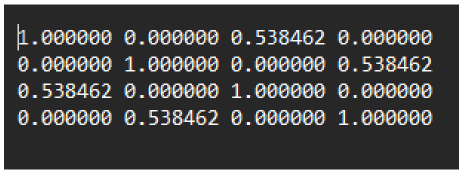
The SimRank results for **Graph 2** with a **decay factor of 0.7** present a similarity matrix with ones along the diagonal and zeros elsewhere. This outcome suggests that **each node is only similar to itself**, and there are no similarities identified between different nodes, even with the decay factor set at a moderately high level.

The similarity scores **remain unchanged** at decay factor 0.3 ,from those with a higher decay factor. Each node is only similar to itself, as evidenced by the 1s on the diagonal of the matrix, and there are no similarities between different nodes, as indicated by the 0s elsewhere.

SimRank Analysis of Graph 3

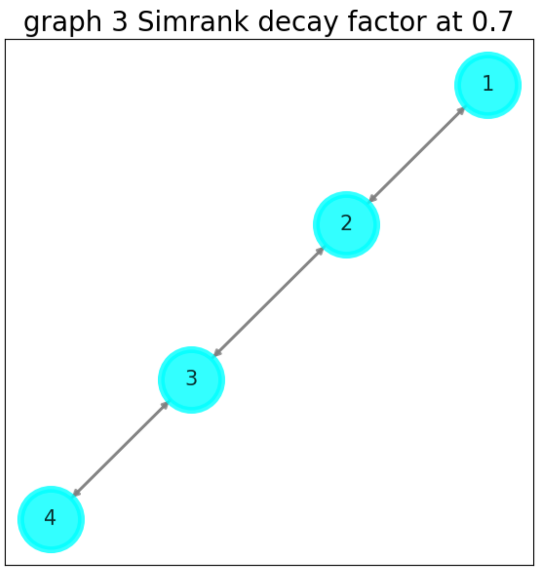
Decay factor at 0.7

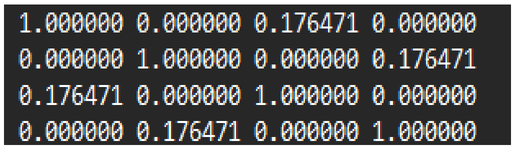




```
1.000000 0.000000 0.538462 0.000000
0.000000 1.000000 0.000000 0.538462
0.538462 0.000000 1.000000 0.000000
0.000000 0.538462 0.000000 1.000000
```

Decay factor at 0.3



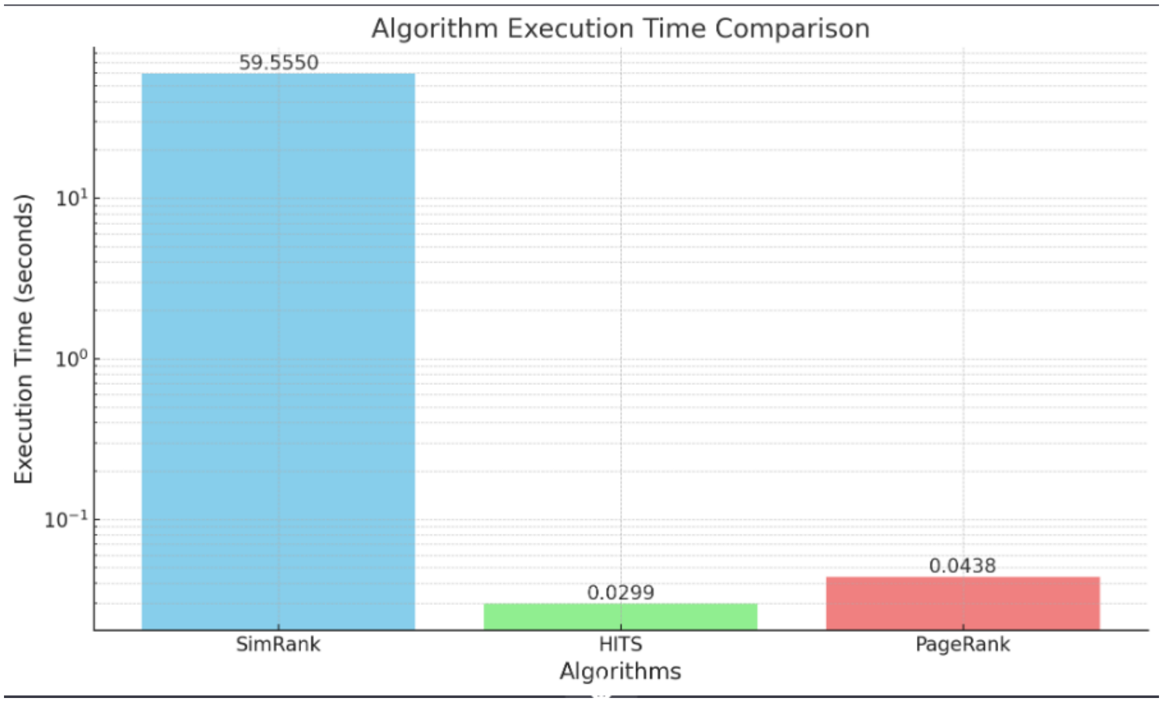


```
1.000000 0.000000 0.176471 0.000000
0.000000 1.000000 0.000000 0.176471
0.176471 0.000000 1.000000 0.000000
0.000000 0.176471 0.000000 1.000000
```

The off-diagonal values of approximately **0.538** between nodes 1 and 3, as well as nodes 2 and 4, indicate a strong similarity according to the SimRank algorithm. **This suggests that these node pairs have similar relationships within the graph's structure.**
The off-diagonal values have decreased to approximately 0.176 when the **decay factor is lowered to 0.3**. This reduction indicates that nodes **1 and 3, as well as 2 and 4, are considered less similar with a lower decay factor.**
Lowering the decay factor reduces the weight of indirect connections in the similarity calculation. This aligns with the concept of SimRank, where a higher decay factor (closer to 1) increases the impact of these indirect connections, and a lower decay factor reduces it.

7. Effectiveness analysis

Time execution Graph



SimRank (59.5550 seconds):

Nature of the Algorithm: SimRank is inherently slower due to its computationally intensive nature. It calculates the similarity between every pair of nodes in a graph, based on the premise that "two nodes are similar if they are connected to similar nodes." This involves a recursive operation that can be costly, especially in larger graphs.

Computational Complexity: The complexity of SimRank grows rapidly with the size and density of the graph, as it requires multiple iterations over all node pairs.

Decay Factor: Additionally, the decay factor influences the convergence of the algorithm, and a higher value may result in more iterations before reaching convergence.

HITS (0.0299 seconds):

Algorithm Efficiency: HITS (Hypertext Induced Topic Search) is notably faster. It calculates two values for each node: hub and authority. This algorithm leverages the graph's link structure, making it more efficient, particularly in graphs with a clear hierarchy or structure.

Lower Complexity: HITS has a lower computational complexity compared to SimRank. It only needs to consider the direct links of each node to compute hub and authority scores, rather than the pairwise comparisons of SimRank.

PageRank (0.0438 seconds):

Balance Between Efficiency and Complexity: PageRank, though slightly slower than HITS in this dataset, is generally efficient and less intensive than SimRank. It computes the importance of each node based on the quantity and quality of inbound links.

Graph’s Link Structure: In graphs where the link structure is not extremely dense, PageRank can quickly compute the relative importance of each node.

Conclusion

This report's analysis of PageRank, HITS, and SimRank across various datasets illuminates their distinct functionalities in link analysis. Key findings include SimRank's depth in node similarity assessment, albeit with high computational demands, HITS' efficiency in identifying key nodes, and PageRank's balance of complexity and effectiveness. The comparison of execution times highlights the practical implications of each algorithm, particularly noting SimRank's computational intensity. Overall, the study affirms the significance of these algorithms in extracting meaningful insights from network data, demonstrating their robustness and adaptability.