Data Mining 2023 Project 3

F74109016 葉惟欣 資訊系大四

1. Find a way [10pts]

(e.g., add/delete some links) to increase hub, authority, and PageRank of Node 1 in first 3 graphs respectively。需要附上圖 (至少說明 Graph 1~3)需要貼上調整後 hub, authority, pagerank 的數值。

Graph 1 : (original)

authority : [0.000 0.200 0.200 0.200 0.200 0.200]

hub : [0.200 0.200 0.200 0.200 0.200 0.000]

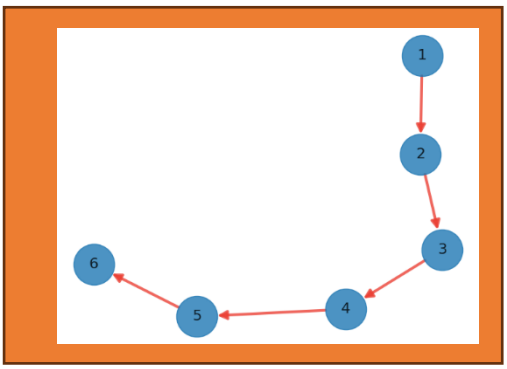
pagerank : [0.025 0.060 0.107 0.171 0.259 0.378]

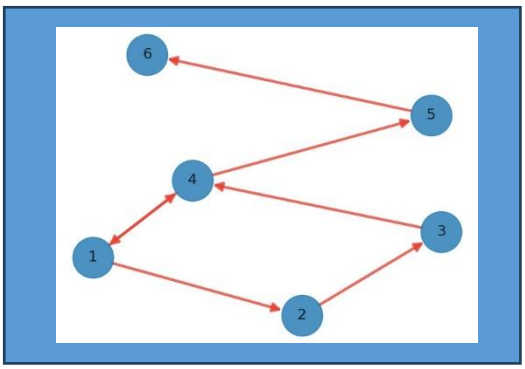
Graph 1 : (add 4->1 link, 1->4 link)

authority : [0.009 0.375 0.000 0.607 0.009 0.000]

hub : [0.612 0.000 0.378 0.009 0.000 0.000]

pagerank : [0.158 0.106 0.135 0.253 0.158 0.191]





Graph 2 : (original)

authority : [0.200 0.200 0.200 0.200 0.200]

hub : [0.200 0.200 0.200 0.200 0.200]

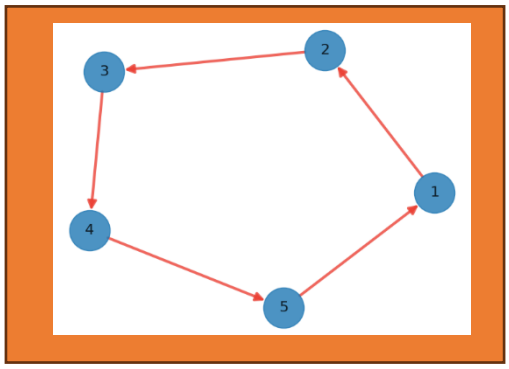
pagerank : [0.200 0.200 0.200 0.200 0.200]

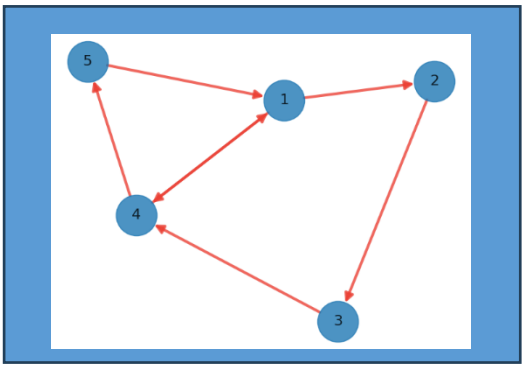
Graph 2 : (add 4->1 link, add 1->4 link)

authority : [0.309 0.191 0.000 0.309 0.191]

hub : [0.309 0.000 0.191 0.309 0.191]

pagerank : [0.278 0.145 0.151 0.280 0.146]





Graph 3 : (original)

authority : [0.191 0.309 0.309 0.191]

hub : [0.191 0.309 0.309 0.191]

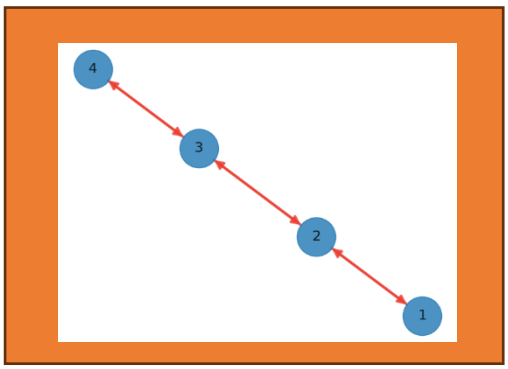
pagerank : [0.172 0.328 0.328 0.172]

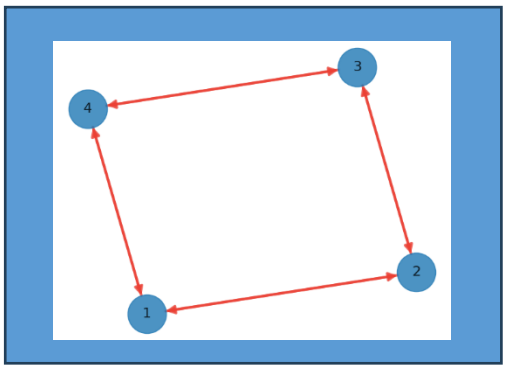
Graph 3 : (add 4->1 link, add 1->4 link)

authority : [0.250 0.250 0.250 0.250]

hub : [0.250 0.250 0.250 0.250]

pagerank : [0.250 0.250 0.250 0.250]





Page-rank and authority 的計算方式都透過 parents node 傳遞過來的，所以可以直接簡單的增加 node1 的 parent 數量來增加 pag-rank 與 authority，把任一個沒有指向 node1 的都加上指向 node1 的 link，增加 node1 其 parent 的數量這樣就可以增加這個 node1 的 page rank and authority。

hub 則是和自己這個 node 的 children 相關，所以跟上方操作相同，把原本 node1 沒有指到的 node，加入一條 directed edge ，使 node 1 多指向一個 node，增加 node1 其 child 的數量，這樣便可以增加這個 node 1 的 hub。

我的作法則是都將加入一個 node 1 -> node 4 的 link 以及 node4 -> node1 的 link，前者是為了增加 hub。後者是為了增加 page rank 與 authority。其也是一個雙向的 link。

2. Algorithm description [10pts] :

解釋每個演算法步驟流程，只貼 code 不說明則不給分。

Hits 實做

```
def UseHit(file_path,iteration=30):
    graph = init_graph(file_path)
    graph = hits(graph, iteration)
    save_auth_hub_file(graph,file_path)
```

1. 建立 graph。我將所有 node 都存在 self.nodes = []
2. 跑 Hit algorithm in function hits

```
def hits(graph, iteration=30):
    for _ in range(iteration):
        auth = []
        hub = []
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            auth.append(cur.update_auth())
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            hub.append(cur.update_hub())
        for node in graph.get_node_list() :
            # 統一更新
            name = node.name
            cur = graph.get_node_id(name)
            cur.auth = auth[graph.get_node_index(name)]
            cur.hub = hub[graph.get_node_index(name)]
        graph.normalize_hits()
    return graph
```

2-1. 將 authority 與 hub 同時做更新。

2-2. 將更新方式將 parent 的 hub 相加成為自己的 authority，將 child 的 authority 相加成為自己的 hub。

```
def update_auth(self):
    auth = 0.
    for parent in self.parents :
        auth += parent.hub
    self.auth = auth

def update_hub(self):
    hub = 0.
    for child in self.children :
        hub += child.auth
    self.hub = hub
```

PageRank 實做

```
def UsePageRank(file_path,iteration=30):
    graph = init_graph(file_path)
    # make every node in graph's pagerank = 1 / N
    for node in graph.get_node_list() :
        node.pagerank = 1. / graph.get_node_num()
    graph = pagerank(graph, iteration)
    save_pagerank_file(graph,file_path)
```

1. 建立 graph。我將所有 node 都存在 self.nodes = []
2. 跑 Page Rank algorithm in function hit

```
def pagerank(graph, iteration=30 ,damping_factor=0.1):
    for _ in range(iteration):
        new_update = []
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            new_update.append(cur.update_pagerank(damping_factor,graph.get_node_num()))

        for node in graph.get_node_list() :
            # 統一更新
            name = node.name
            cur = graph.get_node_id(name)
            cur.pagerank = new_update[graph.get_node_index(name)]

        graph.normalize_pagerank()
    return graph
```

2-1. 將所有 node 的 pagerank 都由舊的算出來後再一起做更新。

2-2. 將更新方式將 parent 的 pagerank 除 parent 的 outdegree。再乘上(1-decay) + decay 除上 num of node.

$$PR(P_i) = \frac{(d)}{n} + (1-d) \times \sum_{l_{j,i} \in E} PR(P_j) / \text{Outdegree}(P_j)$$

```
def update_pagerank(self, damping_factor , numOfNodes):
    pagerank = 0.0
    for parent in self.parents :
        pagerank += parent.pagerank / len(parent.children)
    random_jumping = damping_factor / numOfNodes
    return random_jumping + (1-damping_factor) * pagerank
```

Sim-Rank 實做

```
def UseSimRank(file_path,iteration=30):
    graph = init_graph(file_path)
    sim_matrix = Similaryity(graph)
    sim_matrix = simrank(graph,sim_matrix, iteration)
    save_SimRank(sim_matrix,file_path)
```

1. 建立 graph。我將所有 node 都存在 self.nodes = []
2. 先建立 Similarity 的矩陣

```
class Similaryity:
    def __init__(self,graph,decay_factor=0.7):
        self.graph = graph
        self.decay_factor = decay_factor
        self.new_sim_matrix = np.zeros((self.graph.get_node_num(),self.graph.get_node_num()))
        self.old_sim_matrix = np.zeros((self.graph.get_node_num(),self.graph.get_node_num()))
        self.init_sim_matrix()

    def init_sim_matrix(self):
        for i in range(self.graph.get_node_num()):
            for j in range(self.graph.get_node_num()):
                if i == j : self.old_sim_matrix[i][j] = 1.0
                else :      self.old_sim_matrix[i][j] = 0.0
```

3. 用在 SIMRANK.py 的 simrank 的 function 去更新 Similarity 的矩陣

```
def simrank(graph,sim_matrix,iteration=30,decay_factor=0.7):
    for _ in range(iteration):
        for node1 in graph.get_node_list() :
            for node2 in graph.get_node_list() :
                name1 = node1.name
                name2 = node2.name
                cur1 = graph.get_node_index(name1)
                cur2 = graph.get_node_index(name2)
                new_SimRank = sim_matrix.calculate_simrank(cur1,cur2)
                sim_matrix.update_sim_matrix(cur1,cur2,new_SimRank)

        sim_matrix.old_sim_matrix = sim_matrix.new_sim_matrix.copy()
    return sim_matrix.old_sim_matrix
```

將每個 node 與鄰居的相似度做跟新。

```
def update_sim_matrix(self,node1_index,node2_index,new_SimRank):
    self.new_sim_matrix[node1_index][node2_index] = new_SimRank
```

3-1. 如果兩個點相同則相似度直接為 1。如果兩個點不同，但有一個 node 沒有 parent 則相似度為 0，如果兩個 node 都有 parent，則去看 parent 的相似度為和。再去一併做更新將 old_sim_matrix 換成 new_sim_matrix。這樣在更新各個點的時候就不使用到這一輪剛更新完的值做計算了

```
def calculate_simrank(self,node1,node2):
    if node1 == node2 : return 1.0

    node1_object = self.graph.find(node1)
    node2_object = self.graph.find(node2)
    parents1 = node1_object.get_parents()
    parents2 = node2_object.get_parents()

    if(parents1 == [] or parents2 == []): return 0.0
    SimRank = 0.0

    for parent1 in parents1:
        for parent2 in parents2:
            parent1_index = self.graph.get_node_index(parent1.name)
            parent2_index = self.graph.get_node_index(parent2.name)
            SimRank += self.old_sim_matrix[parent1_index][parent2_index]
    return self.decay_factor * SimRank / (len(parents1) * len(parents2))
```

建造 graph 的方式，裡面有 sort_nodes()是為了讓輸出時請先按照 node 編號順序 sort 輸出，假設 sort 完後 node 編號由小到大[1,3,23,40,307, ...]，則輸出的 array 其對應的 index 0 為 node 1 的數值、index 1 為 node 3 的數值、index 2 為 node 23 的數值...。matrix[1,2] 為 node 3 對 node 23 的數值...。以此類推。

```
class Graph:
    def __init__(self):
        self.nodes = []

    def sort_nodes(self):
        self.nodes.sort(key=lambda x: int(x.name))

    def exist(self, name):
        for node in self.nodes :
            if node.name == name : return node
        new_node = Node(name)
        self.nodes.append(new_node)
        return new_node

    def find(self, name):
        return self.nodes[name]

    def add_node(self, parent, child):
        parent_node = self.exist(parent)
        child_node = self.exist(child)
        parent_node.link_child(child_node)
        child_node.link_parent(parent_node)

    def print_auth(self):
        a = np.array([])
        self.sort_nodes()
        for node in self.nodes : a = np.append(a,round(node.auth,3))
        return a

    def print_hub(self):
        a = np.array([])
        self.sort_nodes()
        for node in self.nodes : a = np.append(a,round(node.hub,3))
        return a
```

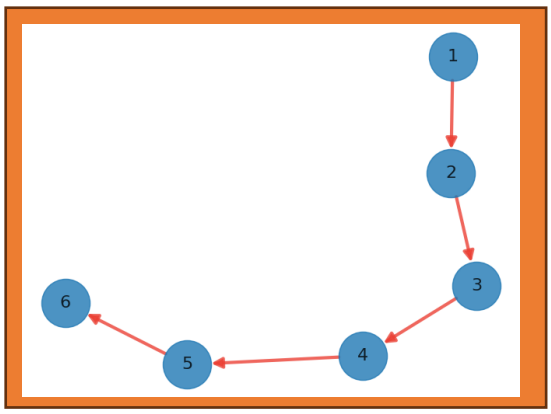
exist 是用來確認 node 是否存在不存在則建立新的。Find 是找出 graph 中 node 的物件。Add node 則是增加 link 的概念，對 child 增加 parent，對 parent 增加 child。

3.Result analysis and discussion [10pts] :

說明每個圖 (至少說明 Graph 1~3) 的結果並討論。針對不同 damping factor 和 decay factor 進行討論。

Page Rank damping factor 是用來模擬用戶在瀏覽網頁(page)時有一定機率點擊 link 到其他 page，而不是繼續瀏覽當前 page。這個機率通常被設置為 0.1，這意味著有 10%的機率用戶會點擊 link 到其他 page，而有 90%的機率用戶停留在當前 page。這樣做的目的是防止在網絡中存在一些節點，雖然沒有被其他 page 連接，但由於沒有跳轉出去的機會，其 PageRank 分數過高。

Graph 1 :



Damping factor = 0.1 (10% user 直接跳去其他 page)

```
graph_1
damping_factor =  0.1
[0.025, 0.06, 0.107, 0.171, 0.259, 0.378]
```

Damping factor = 0.3 (30% user 直接跳去其他 page)

```
graph_1
damping_factor =  0.3
[0.061, 0.112, 0.156, 0.193, 0.225, 0.252]
```

Damping factor = 0.6 (60% user 直接跳去其他 page)

```
graph_1
damping_factor =  0.6
[0.108, 0.155, 0.175, 0.184, 0.188, 0.19]
```

Damping factor = 0.9 (90% user 直接跳去其他 page)

```
graph_1
damping_factor =  0.9
[0.153, 0.168, 0.17, 0.17, 0.17, 0.17]
```

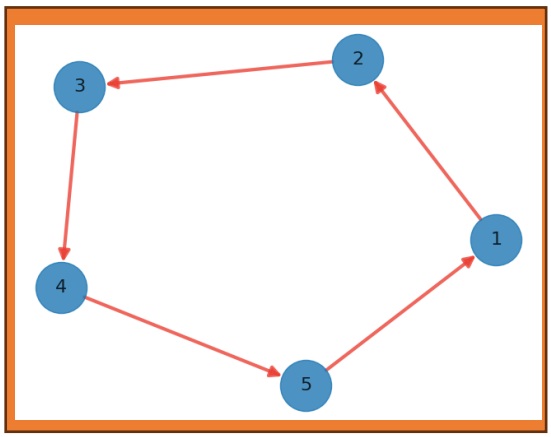
Damping factor = 1 (100% user 直接跳去其他 page)

```
graph_1
damping_factor =  1
[0.167, 0.167, 0.167, 0.167, 0.167, 0.167]
```

分析 ：damping factor 越高代表 user 越高機率會直接跳去其他 page 也就是會越來越傾向隨機機率隨機跳到其他 page。而非遵循原先的 graph 結構。當 damping factor 趨於 0 時，node6 有最高的 page rank。因為所有的節點都可以到 node6。反之 node1 的 page rank 最低。因為除了他自己，以及其他 page 隨機 link 到 node1，不然不會有 page 導向 node1 的 page，因此其 page rank 最低。然而隨著 damping factor 增加，隨機指向的機率越高。因此每個 node 的 page rank 漸趨相同。可以看到如果沒有 damping factor，則沒有 out link 的 page(ex node 6)則獲得高 PageRank，因為 PageRank 值是

一直通過指向的 page 傳遞下去，而不受到減弱的影響。Damping factor 能夠一定程度上避免這種情況，模擬 user 在瀏覽網頁時可能中斷當前頁面瀏覽，轉而點擊鏈接跳轉到其他頁面的行為。這使得 PageRank 更符合實際用戶行為。

Graph 2 :



Damping factor = 0.1 (10% user 直接跳去其他 page)

```
graph_2
damping_factor =  0.1
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Damping factor = 0.3 (30% user 直接跳去其他 page)

```
graph_2
damping_factor =  0.3
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Damping factor = 0.6 (60% user 直接跳去其他 page)

```
 % python main.py
graph_2
damping_factor =  0.6
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Damping factor = 0.9 (90% user 直接跳去其他 page)

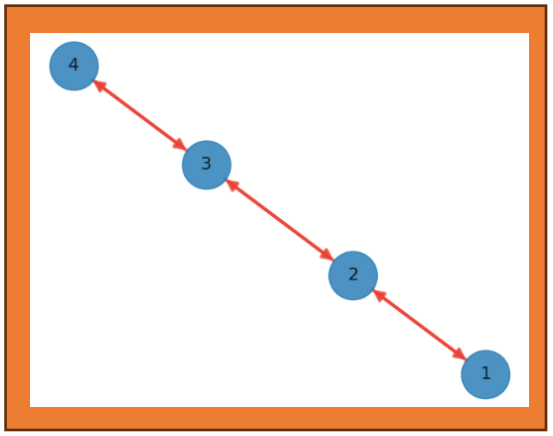
```
 % python main.py
graph_2
damping_factor =  0.9
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Damping factor = 1 (100% user 直接跳去其他 page)

```
 % python main.py
graph_2
damping_factor =  1
[0.2, 0.2, 0.2, 0.2, 0.2]
```

分析：因為每個點在 graph 結構中的地位都相同。因此其 page rank 則為 1/全部的 node 數量 = 平均 0.2

Graph 3 :



Damping factor = 0.1 (10% user 直接跳去其他 page)

```
graph_3
damping_factor =  0.1
[0.172, 0.328, 0.328, 0.172]
```

Damping factor = 0.3 (30% user 直接跳去其他 page)

```
graph_3
damping_factor =  0.3
[0.185, 0.315, 0.315, 0.185]
```

Damping factor = 0.6 (60% user 直接跳去其他 page)

```
graph_3
damping_factor =  0.6
[0.208, 0.292, 0.292, 0.208]
```

Damping factor = 0.9 (90% user 直接跳去其他 page)

```
graph_3
damping_factor =  1
[0.25, 0.25, 0.25, 0.25]
```

Damping factor = 1 (100% user 直接跳去其他 page)

```
graph_3
damping_factor =  0.9
[0.238, 0.262, 0.262, 0.238]
```

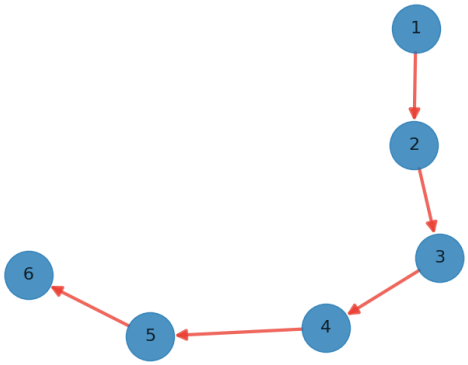
分析：如同在 graph1 所分析的一樣隨著 damping factor 增加，就會讓每個 page 被 visit 的次數越趨近於隨機機率，也因此越接近 0.25。而當 damping factor 較低的時候可以較明顯的發現 node 2 與 node 3 的 page rank 較高。因為從圖的結構來說他們都可以被其他所有節點拜訪。而 node1 與 node4 雖然也可以被其他節點所拜訪。但 node2 與 node3 的 indegree 較多，所以在每次迭代的時候也會有更多 page rank 加入到這兩個點中。比起 node1 與 node4 每次迭代都只會由 node2 與 node3 的 page rank/ 其 outdegree 也就是 2 加入。而 node 1 給 node2 以及 node4 給 node3 的 page rank/ 其 outdegree 都是 1(相當於全給)。

Decay factor 用戶原本的作用是讓距離越遠（距離越多個 hop）的鄰居節點造成的影響越小

Graph 1 :

```
graph_1
decay_factor =  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.]]
```

```
graph_1
decay_factor =  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.]]
```




```
decay_factor =  0.4
[[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.]]
```

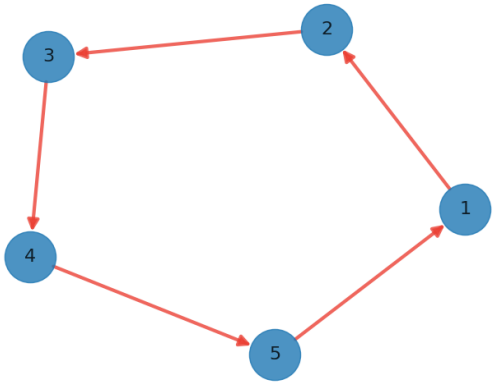
```
graph_1
decay_factor =  0.1
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

```
graph_1
decay_factor =  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. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

Graph 2 :

```
graph_2
decay_factor =  1
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

```
graph_2
decay_factor =  0.7
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

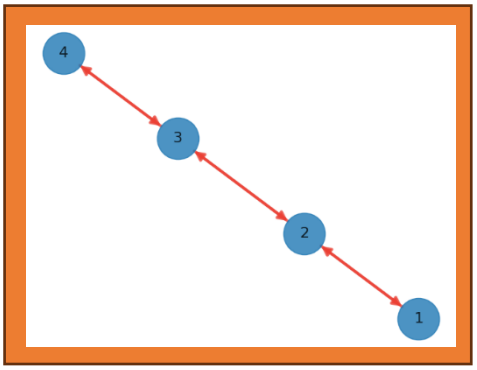


```
decay_factor =  0.4
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

```
graph_2
decay_factor =  0.1
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

```
graph_2
decay_factor =  0
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Graph 3 :




```
graph_3
decay_factor =  1
[[1. 0. 1. 0.]
 [0. 1. 0. 1.]
 [1. 0. 1. 0.]
 [0. 1. 0. 1.]]
```

```
graph_3
decay_factor =  0.7
[[1.         0.         0.53846154 0.        ]
 [0.         1.         0.         0.53846154]
 [0.53846154 0.         1.         0.        ]
 [0.         0.53846154 0.         1.        ]]
```

```
graph_3
decay_factor =  0.4
[[1.   0.   0.25 0.  ]
 [0.   1.   0.   0.25]
 [0.25 0.   1.   0.  ]
 [0.   0.25 0.   1.  ]]
```

```
graph_3
decay_factor =  0.1
[[1.         0.         0.05263158 0.        ]
 [0.         1.         0.         0.05263158]
 [0.05263158 0.         1.         0.        ]
 [0.         0.05263158 0.         1.        ]]
```

```
graph_3
decay_factor =  0
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

由於 Sim-Rank 是透過計算兩個點各自 parent 的相似程度來決定的。也就是看有沒有共同的 parent 來決定的，所以在 graph 1 and graph 2 中，因為 graph1 是線性、另外一個是也是線性連結但成閉迴迴路，所以任意兩點都沒有共同的 parent node，因此除了兩個 node 相同一定相似度為 1 以外，其餘兩個 node 的相似程度為 0。在第三個 graph 中，因為有 node 的 parent 會是一樣的，由於他是雙向 link (舉例 node 1 跟 node3 就有相同的節點 node2)所以 node1 與 node 3 便具有一定的相似性。

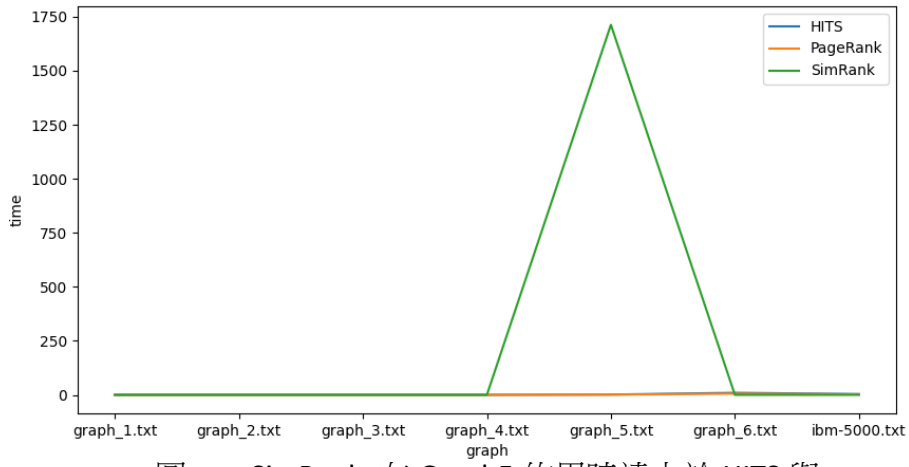
而 decay factor 越大，則 node 的相似性越大。基於公式 decay 為相似度會乘多少給 child node，如果大的話，parent 的相似度越大，child 的相似度也越大，損失的相思度會較少，也就是 child 會大量遺傳 parent。也因此 decay factor 大比 decay factor 小的情況下 node 間的相似度較高，反之亦然

4.Effectiveness analysis [10pts] :

須列出三個演算法在每張圖上的執行時間，並討論其原因。

以下圖都是以秒為單位的，因為我是用 graph 的資料結構去存處每個 node 的資訊，因此每次在做計算 access node 的時候都是 link forward 去存處記憶體。而不是直接像 matrix 直接 access index。所以我的執行時間在 graph5 用 simRank 演算法非常的慢。如下頁圖一。

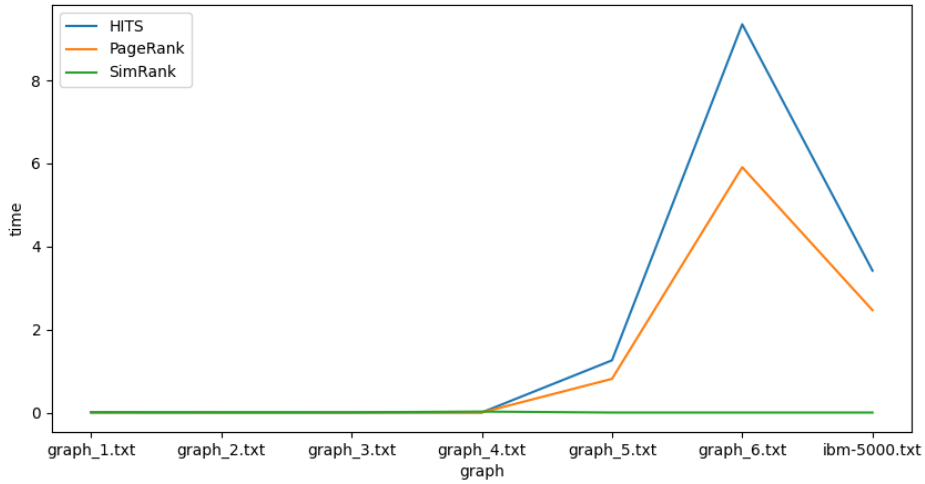
(此圖沒有跑 SimRank 的 graph 6 與 ibm-5000)



圖一 : SimRank 在 Graph5 的用時遠大於 HITS 與 HITS 與 PageRank 在 graph6 與 ibm-5000 的比較。

可以看到 HITS 演算法的用時較 PageRank 高。

(此圖沒有跑 SimRank 的 graph5 與 graph 6 與 ibm-5000)



圖二 : HITS 用時大於 PageRank

SimRank 演算法的時間複雜度每次跑一次遞迴 iterative 的 turn 的複雜度為 : N 個 node，每個 node 的平均 in-degree : 為 d。每次遞迴要對每兩個 node 計算其相似度，所以這樣複雜度為 n^2。而每一對 node 計算其相似度考慮到平均 in-degree，因此平均要計算 d^2。如此一來總體複雜度 k(遞迴次數) * (n^2) * (d^2)。而 HITS 也為遞迴演算法，每一次遞迴時，更新每個 node 的 authority 與 hub。因此每次遞迴會更新所有 node 複雜度為 n，每個 node 透過其 parent 與 child 的值去更新。因此複雜度為 k(遞迴次數)*n*d(平均 out/in degree 數量)。而 pagerank，則為 k(遞迴次數)*n*d(平均 in degree 數量)，因為只考慮 Parent 的數量。因此 hits 與 pagerank 的時間複雜度在同一個數量級。而 SimRank 則遠高於兩者。

以秒為單位

	graph1	graph2	graph3	graph4
HITS	0.001584291	0.001766204833984375	0.001707315444946289	0.0023450851440429688
PageRank	0.0013275146484375	0.0007069110870361328	0.0008246898651123047	0.0015897750854492188
SimRank	0.008971691131591797	0.00662541389465332	0.0047152042388916016	0.03254246711730957

	graph5	graph6	ibm-5000
HITS	1.2839884757995605	10.086986541748047	3.928464651107788
PageRank	0.8719816207885742	5.942318439483643	2.320399284362793
SimRank	1616.7110455036163		

HIT 演算法

```
############### HITS update ###############
def update_auth(self):
    auth = 0.
    for parent in self.parents :
        auth += parent.hub
    return auth

def update_hub(self):
    hub = 0.0
    for child in self.children :
        hub += child.auth
    return hub
```

```
def hits(graph, iteration=30):
    for _ in range(iteration):
        auth = []
        hub = []
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            auth.append(cur.update_auth())
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            hub.append(cur.update_hub())
        for node in graph.get_node_list() :
            # 統一更新
            name = node.name
            cur = graph.get_node_id(name)
            cur.auth = auth[graph.get_node_index(name)]
            cur.hub = hub[graph.get_node_index(name)]
        graph.normalize_hits()
    return graph
```

Page Rank 演算法

```
############### PAGERANK update ###############
def update_pagerank(self, damping_factor , numOfNodes):
    pagerank = 0.0
    for parent in self.parents :
        pagerank += parent.pagerank / len(parent.children)
    random_jumping = damping_factor / numOfNodes
    return random_jumping + (1-damping_factor) * pagerank
```

```
def pagerank(graph, iteration=30 ,damping_factor=0.1):
    for _ in range(iteration):
        new_update = []
        for node in graph.get_node_list() :
            name = node.name
            cur = graph.get_node_id(name)
            new_update.append(cur.update_pagerank(damping_factor,graph.get_node_num()))

        for node in graph.get_node_list() :
            # 統一更新
            name = node.name
            cur = graph.get_node_id(name)
            cur.pagerank = new_update[graph.get_node_index(name)]

        graph.normalize_pagerank()
    return graph
```

分析：因為 page rank 在算每個 node 的 page rank 時候只需要考慮 parent node 的 page rank。而 HITS 由於 hub 與 authority 互相有關係影響，所以要計算 child node 的 authority 與 parent node 的 hub，考慮兩種類型的 node，所以 hits 用時都會較 pageRank 大。